The SAP HANA Database – An Architecture Overview

Franz Färber Norman May Wolfgang Lehner Philipp Große Ingo Müller
Hannes Rauhe Jonathan Dees
SAP AG

Abstract

Requirements of enterprise applications have become much more demanding. They require the computation of complex reports on transactional data while thousands of users may read or update records of the same data. The goal of the SAP HANA database is the integration of transactional and analytical workload within the same database management system. To achieve this, a columnar engine exploits modern hardware (multiple CPU cores, large main memory, and caches), compression of database content, maximum parallelization in the database kernel, and database extensions required by enterprise applications, e.g., specialized data structures for hierarchies or support for domain specific languages. In this paper we highlight the architectural concepts employed in the SAP HANA database. We also report on insights gathered with the SAP HANA database in real-world enterprise application scenarios.

1 Introduction

A holistic view on enterprise data has become a core asset for every organization. Data is entered in batches or by-record via multiple channels, such as enterprise resource planning systems (e.g., SAP ERP), sensors used in production environments, or web-based interfaces. For example in a sales process, orders are created, modified, and deleted. These orders are the basis for production planning and delivery. Hence, during the sales process records are looked up, inserted, and updated. This kind of data processing is typically referred to as Online Transactional Processing (OLTP). OLTP has been the strength of current disk-based and row-oriented database systems.

Upon closer inspection, a supposedly simple sales process exhibits a significant amount of complex analytical processing. For example, checking the availability of a product for delivery as part of a sales process requires aggregating expected sales, expected delivery, and completion of production lots, as well as comparing the resulting inventory with the customer demand. Similarly, a sales organization would be interested in profitability measures for planning based on most recent sales and cost information. This kind of workload is considered Online Analytical Processing (OLAP). Periodical tasks, such as quarter-end closing or customer segmentation, are executed by replicating data into a read-optimized data warehouse. For those types of reporting, column-stores have become more and more popular [3].

Additionally, analytical applications require procedural logic, which cannot be expressed with plain SQL, e.g., clustering sales number of different products or classifying customer behavior. The natural approach is to transfer all the data needed from the database to the application and process it there. Therefore, optimized data

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

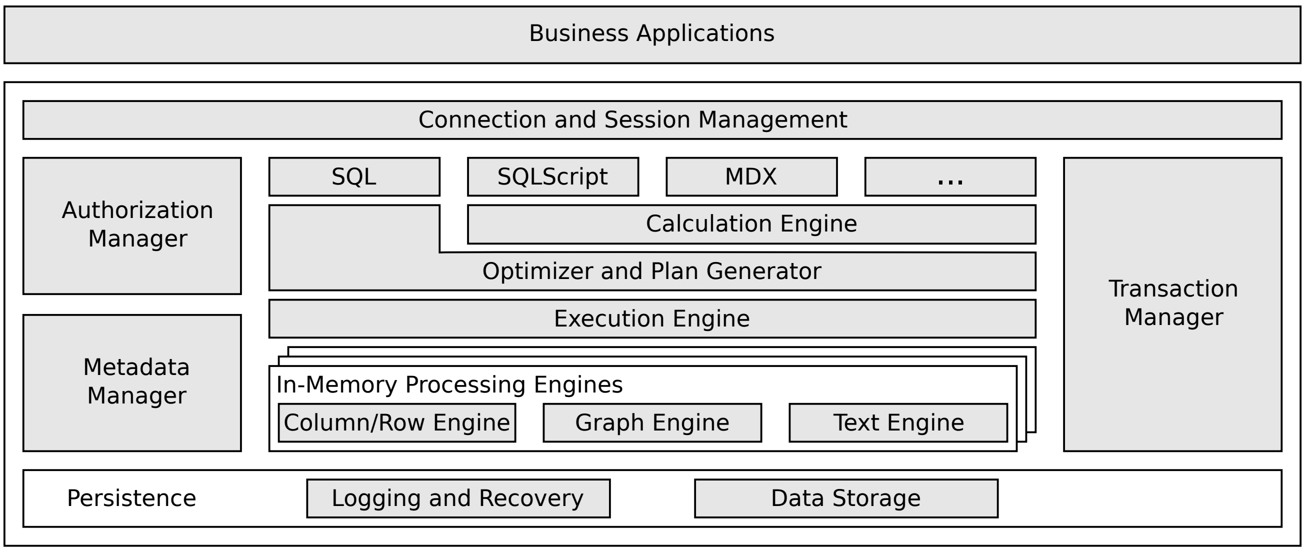

Figure 1: Overview of the SAP HANA DB architecture

structures and metadata cannot be used and intermediate results have to be transferred back to the database if they are needed in the following business process steps.

Ideally, a database shall be able to process all of the above-mentioned workloads and application-specific logic in a single system [13]. This observation sparked the development of the SAP HANA database (SAP HANA DB). Historically, the in-memory columnar storage of the SAP HANA DB is based on the SAP TREX text engine [15] and the SAP BI Accelerator (SAP BIA) [10], which allows for fast processing of OLAP queries. The high-performance in-memory row-store of the SAP HANA DB is derived from P*Time [2] and specially designed to address OLTP workload. The persistence of the SAP HANA DB originated from the proven technology of SAP's MaxDB database system providing logging, recovery, and durable storage. As of today, the SAP HANA DB is commercially available as part of the SAP HANA appliance.

In the next section, we give a brief overview about the architectural components of the SAP HANA DB. Section 3 discusses the ability to execute analytical application-specific logic. In section 4 we outline how the SAP HANA DB accelerates traditional data warehouse workloads. We discuss how we address challenges on transactional workloads in enterprise resource planning systems in section 5 and summarize our work on the SAP HANA DB in section 6.

2 SAP HANA DB Architecture

The general goal of the SAP HANA DB is to provide a main-memory centric data management platform to support pure SQL for traditional applications as well as a more expressive interaction model specialized to the needs of SAP applications [4, 14]. Moreover, the system is designed to provide full transactional behavior in order to support interactive business applications. Finally, the SAP HANA DB is designed with special emphasis on parallelization ranging from thread and core level up to highly distributed setups over multiple machines.

Figure 1 provides an overview of the general SAP HANA DB architecture. The heart of the SAP HANA DB consists of a set of in-memory processing engines. Relational data resides in tables in column or row layout in the combined column and row engine, and can be converted from one layout to the other to allow query expressions with tables in both layouts. Graph data and text data reside in the graph engine and the text engine respectively; more engines are possible due to the extensible architecture [4]. All engines keep all data in main memory as long as there is enough space available. As one of the main distinctive features, all data structures are optimized for cache-efficiency instead of being optimized for organization in traditional disk blocks. Furthermore, the engines compress the data using a variety of compression schemes. When the limit of available main memory is reached, entire data objects, e.g., tables or partitions, are unloaded from main memory under control of application semantic and reloaded into main memory when it is required again.

From an application perspective, the SAP HANA DB provides multiple interfaces, such as standard SQL for generic data management functionality or more specialized languages as SQLScript (see section 3) and MDX. SQL queries are translated into an execution plan by the plan generator, which is then optimized and executed by the execution engine. Queries from other interfaces are eventually transformed into the same type of execution plan and executed in the same engine, but are first described by a more expressive abstract data flow model in the calculation engine. Irrespective of the external interface, the execution engine can use all processing engines and handles the distribution of the execution over several nodes.

As in traditional database systems, the SAP HANA DB has components to manage the execution of queries. The session manager controls the individual connections between the database layer and the application layer, while the authorization manager governs the user's permissions. The transaction manager implements snapshot isolation or weaker isolation levels – even in a distributed environment. The metadata manager is a repository of data describing the tables and other data structures, and, like the transaction manager, consists of a local and a global part in case of distribution.

While virtually all data is kept in main memory by the processing engines for performance reasons, data has also to be stored by the persistence layer for backup and recovery in case of a system restart after an explicit shutdown or a failure. Updates are logged as required for recovery to the last committed state of the database and entire data objects are persisted into the data storage regularly (during savepoints and merge operations, see section 4).

3 Support for Analytical Applications

A key asset of the SAP HANA DB is its capability to execute business and application logic inside the database kernel. For this purpose the calculation engine provides an abstraction of logical execution plans, called calculation models. For example SQLScript, a declarative and optimizable language for expressing application logic as data flows or using procedural logic, is compiled into calculation models. Following this route, multiple domain-specific languages can be supported as long as a compiler generates the intermediate calculation model representation.

The primitives of a calculation model constitute a logical execution plan consisting of an acyclic data flow graph with nodes representing operators (plan operations) and edges reflecting the data flow (plan data). One class of operators implements the standard relational operators like join and selection. In addition, the SAP HANA DB supports a huge variety of special operators for implementing application-specific components in the database kernel. Almost all these operators are only able to accelerate data processing because they exploit the columnar data layout. By implementing special operators in the calculation engine, several application domains can be supported:

Statistical Algorithms can be attached to calculation models to perform complex statistical computations inside an associated R runtime. This includes different statistical methods, such as linear and nonlinear models, statistical tests, time series analyses, classification, and clustering. At the same time, the calculation model allows to leverage the capabilities to pre- and post-process large data in the database kernel and thereby interweave the statistical algorithms with database operations [5].

Planning provides a set of commonly used and generic planning functions that allow to model and execute complex planning scenarios directly in the database. Planning logic is expressed using data flow operators of the calculation engine. In addition, special operators perform specific planning algorithms such as disaggregation and custom formulas [7, 8].

Other Special Operators provided within the calculation engine include business logic which requires complex operations that are hard to implement efficiently using SQL, e.g. currency conversion. Another example are large hierarchies describing, e.g., relations between employees and associated information in the human capital management of an enterprise. Here, the SAP HANA DB provides application-specific operators to return query results on these hierarchies almost instantaneously exploiting alternative internal data structures.

A specific calculation model or logical execution plan—once submitted to SAP HANA DB (e.g., by using SQLScript)—can be accessed in the same way as a database view, making the calculation model a kind of parameterized view. A query consuming such a view invokes the database plan execution to process an execution plan. This plan is derived from the logical data flow description provided by the calculation model. If the calculation model contains independent data flow paths, the derived execution plan implicitly contains inter-operator parallel execution. This is explored by SQLScript and the domain-specific languages compiled into calculation models.

4 Analytical Query Processing

As generally agreed, column-stores are well suited for analytical queries on massive amounts of data [1]. For high read performance the SAP HANA DB's column-store uses efficient compression schemes in combination with cache-aware and parallel algorithms. Every column is compressed with the help of a sorted dictionary, i.e., each value is mapped to an integer value (the valueID). These valueIDs are further bit-packed and compressed. By resorting the rows in a table, the most beneficial compression (e.g., run-length encoding (RLE), sparse coding, or cluster coding) for the columns of this table can be used [11, 12]. Compressing data does not only allow to keep more data on a single node, but it also allows for faster query processing, e.g., by exploiting the RLE to compute aggregates. Scans are accelerated by excessively using SIMD algorithms working directly on the compressed data [16].

Since single updates are expensive in the described layout, every table has a delta storage, which is designed to balance between high update rates and good read performance. Dictionary compression is used here as well, but the dictionary is stored in a Cache Sensitive B+-Tree (CSB+-Tree). The delta storage is merged periodically into the main data storage. To minimize the period of time where tables are locked, write operations are redirected to a new delta storage when the delta merge process starts. Until it is finished, read operations access new and old delta storage as well as the old main storage [9].

Query execution exploits the increasing number of available execution threads within a node by using intra-operator parallelism. For example, grouping operations scale almost linearly with the number of threads until the CPU is saturated. Additionally the SAP HANA DB also exploits parallelism inside a query execution plan and across many cores and nodes. Large tables can be partitioned using various partitioning criteria. These parts or complete tables can then be assigned to different nodes in the landscape [10]. The execution engine schedules operators in parallel if they can be processed independently and—if possible—executes them on the node that holds the data. In case of changing workload, the partitioning scheme and assignment of tables to nodes can be adapted while the database is available for queries and updates. Joins involving tables distributed across multiple nodes are processed using semi-join reduction [6].

5 Transactional Query Processing

While it is clear that column-stores work well for OLAP workloads, we also argue that there are several reasons to consider the column-store for OLTP workloads, especially in ERP systems [9]:

1) OLTP scenarios can greatly benefit from the compression schemes available in column-stores: In a highly customizable system like SAP ERP, many columns are not used, and thus only contain default values or no values at all. Similarly, some columns typically have a small domain, e.g., status flags. In both cases, compression is very efficient, which can be a decisive advantage for OLTP scenarios: By reducing the memory consumption, the necessary landscape size becomes smaller, so either fewer or smaller nodes are required. Moreover, compression also leads to lower memory bandwidth utilization.

2) Real-world transactional workloads have larger portions of read operations than standard benchmarks like TPC-C define. Hence, the read-optimized column-oriented storage layout may be more appropriate for OLTP workloads than suggested by the benchmarks.

3) Column-stores usually follow the simple "append-only" scheme: When an existing row is updated, the current version is invalidated and a new version is appended. This scheme is simpler than in-place updates as it neither requires reordering nor encoding the values.

4) Column-stores greatly reduce the need for indices: As a matter of fact, the high scan performance of column-stores on modern hardware permits us to have indices only for primary keys, columns with unique constraints and frequent join columns. In all other cases, scan performance is good enough without indices, especially in small tables or small partitions with up to a few hundred thousand rows. The advantages are a significantly simplified physical database design, reduced main memory consumption, and eliminated effort in the maintenance of indices, which in turn speed up the overall query throughput.

Beside these intrinsic advantages of column-stores for OLTP, there are several challenges we have discovered in this context. One challenge emerges directly from the column data layout. Although it allows for a more fine-grained data access pattern, it can result in a significant performance overhead to allocate the memory per columns to handle a large number of columns, for example when constructing a single result row consisting of 100 columns or more. As of now, the SAP HANA DB combines memory allocations for multiple columns into a single one whenever it helps to reduce the performance overhead.

As a major challenge, we see that in ERP applications, a substantial number of updates is performed concurrently. In contrast to data warehouses, where updates are executed in batches, frequent updates of single records constantly add new entities into the delta storage, implying frequent merges from the delta into the main data storage. As the merge operation is a CPU and memory intensive operation, the challenge is to minimize its impact on the requests that are processed concurrently. The SAP HANA faces this problem by careful scheduling and parallelization [9].

6 Summary

In this paper we summarize the principles guiding the design and implementation of the SAP HANA DB. Our analysis shows that in-memory processing using a columnar engine is the most promising approach to cope with analytical and transactional workloads at the same time. The strong support of business application requirements and both kinds of workloads differentiate the SAP HANA DB from other column-stores. After all, the majority of OLAP and OLTP operations are read operations, which benefit from column-wise compression. Moreover, fewer indices are required leading to a simplified physical database design and reduced memory consumption. To further hold the massive amount of data produced by today's enterprise applications in memory, SAP HANA DB allows to distribute it in a cluster of nodes. As a result, the analysis of large data sets is orders of magnitude faster than on conventional database systems.

However, an in-memory column-store supporting distribution raises a number of challenges including the need for partitioning, support for distributed transactions and a carefully designed process for merging updates into the read-optimized storage layout. But these challenges are just the tip of the iceberg because as we consider data-intensive applications in the cloud, elasticity requirements, multi-tenancy, or scientific computing further challenges have to be addressed.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proc. SIGMOD*, pages 967–980, 2008.

[2] S. K. Cha and C. Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In *Proc. VLDB*, pages 1033–1044, 2004.

[3] S. Chaudhuri, U. Dayal, and V. Narasayya. An Overview of Business Intelligence Technology. *CACM*, 54(8):88–98, 2011.

[4] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. *SIGMOD Record*, 40(4):45–51, 2011.

[5] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging Two Worlds with RICE – Integrating R into the SAP In-Memory Computing Engine. *Proc. VLDB*, 4(12):1307–1317, 2011.

[6] G. Hill and A. Ross. Reducing outer joins. *VLDB Journal*, 18(3):599–610, 2009.

[7] B. Jäcksch, F. Färber, and W. Lehner. Cherry Picking in Database Languages. In *Proc. IDEAS*, pages 117–122, 2010.

[8] B. Jäcksch, W. Lehner, and F. Färber. A Plan for OLAP. In *Proc. EDBT*, pages 681–686, 2010.

[9] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, P. Dubey, H. Plattner, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proc. VLDB*, 5(1):61–72, 2011.

[10] T. Legler, W. Lehner, and A. Ross. Data Mining with the SAP NetWeaver BI Accelerator. In *Proc. VLDB*, pages 1059–1068, 2006.

[11] C. Lemke, K.-U. Sattler, F. Färber, and A. Zeier. Speeding Up Queries in Column Stores – A Case for Compression. In *Proc. DaWak*, pages 117–129, 2010.

[12] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krüger. How to Juggle Columns: An Entropy-Based Approach for Table Compression. In *Proc. IDEAS*, pages 205–215, 2010.

[13] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proc. SIGMOD*, pages 1–2, 2009.

[14] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, Berlin Heidelberg, 2011.

[15] F. Transier and P. Sanders. Engineering Basic Algorithms of an In-Memory Text Search Engine. *ACM TOIS*, 29(1):2, 2010.

[16] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on- Chip Vector Processing Units. *Proc. VLDB*, 2(1):385–394, 2009.