

Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth

Vishal Sikka
SAP
3412 Hillview Ave
Palo Alto, CA 94304, USA
vishal.sikka@sap.com

Franz Färber
SAP
Dietmar-Hopp-Allee 16
69190, Walldorf, Germany
franz.faeber@sap.com

Wolfgang Lehner
SAP
Dietmar-Hopp-Allee 16
69190, Walldorf, Germany
wolfgang.lehner@sap.com

Sang Kyun Cha
SAP
63-7 Banpo 4-dong, Seocho-ku
137-804, Seoul, Korea
sang.k.cha@sap.com

Thomas Peh
SAP
Dietmar-Hopp-Allee 16
69190, Walldorf, Germany
thomas.peh@sap.com

Christof Bornhövd
SAP
3412 Hillview Ave
Palo Alto, CA 94304, USA
christof.bornhoevd@sap.com

ABSTRACT

The SAP HANA database is the core of SAP's new data management platform. The overall goal of the SAP HANA database is to provide a generic but powerful system for different query scenarios, both transactional and analytical, on the same data representation within a highly scalable execution environment. Within this paper, we highlight the main features that differentiate the SAP HANA database from classical relational database engines. Therefore, we outline the general architecture and design criteria of the SAP HANA in a first step. In a second step, we challenge the common belief that column store data structures are only superior in analytical workloads and not well suited for transactional workloads. We outline the concept of record life cycle management to use different storage formats for the different stages of a record. We not only discuss the general concept but also dive into some of the details of how to efficiently propagate records through their life cycle and moving database entries from write-optimized to read-optimized storage formats. In summary, the paper aims at illustrating how the SAP HANA database is able to efficiently work in analytical as well as transactional workload environments.

Categories and Subject Descriptors

H.2 [Database Management]: Systems; H.2.4 [Systems]: Relational databases; Query processing—*Database Manager*

General Terms

Algorithms, Design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1247-9/12/05 ...\$10.00.

Keywords

SAP HANA, Column store, Transaction processing

1. INTRODUCTION

Data management in modern business applications is one of the most challenging topics in today's software industry. Not only is data driving today's business but also provides the foundation for the development of novel business ideas or business cases. Data management in all the different flavors has become a core asset for every organization. Also, data management has gained significant attention at senior management level as the main tool to drive and develop the current business. On the system side, data management scenarios have become extremely complex and complicated to manage. An efficient, flexible, robust, and cost-effective data management layer is the core for a number of different application scenarios essential in today's business environments.

Initially, classical ERP systems were implemented as the information processing backbone that handles such application scenarios. From the database system perspective, the OLTP workload of ERP systems typically require handling of thousands of concurrent users and transactions with high update load and very selective point queries. On the other hand, data warehouse systems—usually considered as the counterpart to OLTP—either run aggregation queries over a huge volume of data or compute statistical models for the analysis of artifacts stored in the database. Unfortunately, applications like real time analysis to identify anomalies in data streams or ETL/information integration tasks add to the huge variety of different and in some cases absolutely challenging requirements for a data management layer in the context of modern business applications.

As an answer, Stonebraker et al. postulated the “The End of an Architectural Era (It's Time for a Complete Rewrite)” in 2007 [13] by stating the hypothesis that traditional database management systems are no longer able to represent the holistic answer with respect to the variety of different requirements. Specialized systems will emerge for specific problems. In essence, the statement confirms the observation that large data management solutions are usually a zoo of different systems with different capabilities for dif-

ferent application scenarios. For example, classic row-stores are still dominating the OLTP domain. Maintaining a 1:1-relationship between the logical entity and the physical representation in a record seems obvious for entity-based interaction models. Column-organized data structures gained more and more attention in the analytical domain to avoid projection of queried columns and exploit significantly better data compression rates. Key-value stores are making inroads into commercial data management solutions to cope not only with “big data”-volumes but also provide a platform for procedural code to be executed in parallel. In addition, distributed file systems that provide a cheap storage mechanism and a flexible degree of parallelism for cloud-like elasticity made key-value stores a first class citizen in the data management arena. The zoo of systems is completed by triple stores to cope with schema-flexible data and graph-based organization. Since the schema comes with the data, the system provides efficient means to exploit explicitly modeled relationships between entities, run analytical graph algorithms, and exhibit a repository for weakly-typed entities in general. Although specialized systems may be considered a smart move in a first performance-focused shot, the zoo of systems yields tremendous complexity to link different systems, run data replication and propagation jobs, or orchestrate query scenarios over multiple systems. Additionally, setting up and maintaining such an environment is not only complex and error prone but also comes with significantly higher TCO.

Taking a bird’s perspective, we make the following observation of motivations underlying the current situation:

- **Usage perspective:** We consider SQL no longer the only appropriate interaction model for modern business applications. Users are either completely shielded by an application layer or would like to directly interact with their database. In the first case, we see the need to optimally support an application layer with a tight coupling mechanism. In the second case, we see the need for scripting languages with built-in database features for specific application domains like R for statistical processing ([4]) or Pig to work on Hadoop installations. We also see the need for a comprehensive support of domain-specific and proprietary query languages like SAP’s FOX for financial planning scenarios ([7]). Finally we see a huge demand to provide mechanisms to directly enable users considering parallelism from a programming perspective.
- **Cost awareness:** We see the clear demand to provide a lower TCO solution for the complete data management stack ranging from hardware to setup costs to operational and maintenance costs by offering a consolidated solution for different types of workloads and usage patterns.
- **Performance, performance, performance** ([16]): We identify performance still as the main reason to use specialized systems. The challenge is to provide a flexible solution with the ability to use specialized operators or data structures whenever possible and needed.

We would like to point out that different workload characteristics do not fully justify going for the zoo of specialized systems. Our past experience of handling business applica-

tions leads us to support the hypothesis for a need of specialized collections of operators. We are biased against individual systems with separate life cycles and administration set-ups. However, we do not aim at providing a single closed system but a flexible data management platform with common service primitives. The currently available SAP HANA database functionality, which is a core part the SAP HANA appliance ([2]), may be considered as one specific incarnation of such a tool box.

Contribution and Outline

Giving a holistic view of the SAP HANA database platform on the one hand and diving into details of some specific aspects in order to cope with transactional workloads on the other hand is the overall goal of this paper. We first give an overview of the overall architecture and the design guidelines to emphasize the differences to classical relational database management systems. In particular, we want to stress the following main distinguishing features of the SAP HANA database for the scope of typical business applications.

- The HANA database comprises a multi-engine query processing environment that offers different data abstractions supporting data of different degrees of structure—from well-structured relational data to irregularly structured data graphs to unstructured text data. This full spectrum of processing engines is based on a common table abstraction as the underlying physical data representation to allow for interoperability and the combination of data of different types.
- It supports the representation of application-specific business objects (like OLAP cubes) and logic (domain-specific function libraries) directly inside the database engine. This permits the exchange of application semantics with the underlying data management platform that can be exploited to increase the query expressiveness and to reduce the number of individual application-to-database roundtrips and the amount of data transferred between database and application.
- HANA database is optimized to efficiently communicate between the data management and the application layer. For example, the HANA database natively supports the data types of the SAP application server. Furthermore, plans are to integrate novel application server technology directly into the SAP HANA database cluster infrastructure to enable an interleaved execution of application logic and database management functionality.
- Last not least, the SAP HANA database supports the efficient processing of both transactional and analytical workloads on the same physical database leveraging a highly-optimized column-oriented data representation. This is achieved through a sophisticated multi-step record life cycle management approach.

While the first three features are addressed in [2] the last one is discussed in the second part of this paper. Here we outline some details related to the unified table structure and the propagation mechanism to move records through the system within a controlled life cycle management process. More specifically, we first motivate the different data

structures to hold the data in different phases of their life cycle. Thereafter, we focus on the efficient implementation of the propagation steps between different data structures to move data from a write-optimized store to handle insert/update/deletes extremely well to finally highly compressed main memory structures to efficiently answer OLAP-style query patterns. Furthermore, we would like to mention that the SAP HANA database also provides techniques required to run enterprise-critical applications like an SAP ERP system. For example, the SAP HANA database shows fault tolerance of individual nodes within a SAP HANA cluster installation, i.e. other nodes are taking over the load if a node goes down. In such a situation, query processing is neither blocked nor stopped but is transparently rerouted by the distributed query execution framework. Other features like backup and recovery are available as well. From a transactional behavior, the SAP HANA database uses multi-version concurrency control (MVCC) to implement different transaction isolation levels. The SAP HANA database supports both transaction level snapshot isolation and statement level snapshot isolation. However, due to space restrictions, we are not able to dive into details at this point.

2. LAYERED ARCHITECTURE OF THE SAP HANA DATABASE

As already outlined in [2], the SAP HANA product which is already commercially available consists of an appliance model with different components to yield a ready-to-go package for data analytics scenarios. In a first step, the product was aiming at data mart scenarios to get customers accustomed to the new philosophy of the column-oriented, main-memory-centric SAP HANA database. As of now, SAP is providing native support for the SAP Business Warehouse product to significantly speed up query and transformation scenarios but also allows to completely skip individual materialization steps. In order to provide this capability, SAP HANA has data loading and transformation tools plus a modeling studio to create and maintain complex data flows in and out of SAP HANA. The SAP HANA database is the core component of the SAP HANA product and takes care of the efficient and flexible data storage and data querying scenarios ([11, 12]).

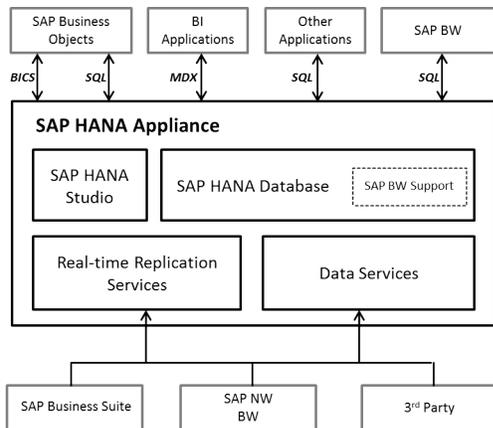


Figure 1: Overview of the SAP HANA appliance

The SAP HANA database itself follows a strict layered architecture as outlined in figure 2. Similar to classical systems, the SAP HANA database distinguishes between compile time and run time of a database request. Also, although not shown in the figure, multiple components like transaction manager, authorization manager, meta data manager etc. complement the overall architecture.

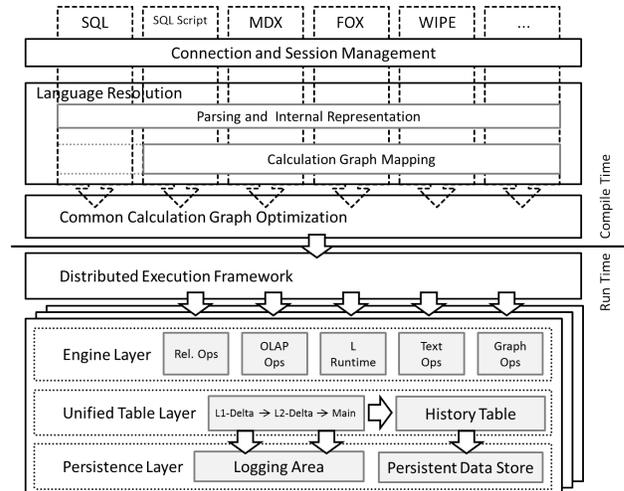


Figure 2: Overview of the HANA database layered architecture

In addition to pure data processing performance, we also identified the lack of an appropriate coupling mechanism between the application layer and the data management layer as one of the main deficits of state-of-the-art systems. This analysis was one of the main drivers to design the SAP HANA database as an extensible platform for different query languages. As can be seen in figure 2, different query languages can enter the system via a common connection and session management layer performing all infrastructural tasks with the outside world (JDBC, ODBC connectors etc.). In a first step, a query string is translated into an internal optimized representation (similar to an abstract syntax tree), which is local for every domain-specific language. In a second step, the query expression is mapped to a “Calculation Graph” (calc graph for short), which forms the heart of the logical query processing framework.

2.1 Calculation Graph Model

The “Calculation Graph Model” follows the classical data flow graph principle. Source nodes represent either persistent table structures or the outcome of other calc graphs. Inner nodes reflect logical operators consuming either one or multiple incoming data flows and produce any arbitrary number of outgoing data flows. Moreover, the set of calc graph operators can be split into two groups of operator types. On the one side, the calc model defines a set of intrinsic operators, e.g. aggregation, projection, joins, union etc. SQL for example can be completely mapped to this class of operators. On the other side, the calc model provides operators which implement core business algorithms like currency conversion or calendar functionality. In addition the calc model supports the following types of operators:

- **Dynamic SQL nodes:** A calc model operator may execute a complete SQL statement on the incoming data flow. The statement can be a parameter and compiled and executed at runtime of the calc graph, resulting in a form of “nested calc” models.
- **Custom nodes:** A custom node may be used to implement domain-specific operators in C++ for performance reasons. For example, the planning scenario with the SAP proprietary language FOX [6] can exploit a special “disaggregate” operator to natively support financial planning situations [7]. Another example are optimized operations for graph traversal and analysis in data graphs via the proprietary WIPE graph language.
- **R nodes:** An R node ([4]) can be used to forward incoming data sets to an R execution environment. The R script, given as a parameter, will then be executed outside of the SAP HANA database and results are moved back into the calc graph for further processing.
- **L nodes:** The language L represents the internal runtime language of the SAP HANA database. L is designed as a safe subset of the C language and usually not directly accessible for end users or application designers. Instead, L is the target language for all constructs of domain-specific languages which cannot be directly mapped to data-flow graphs, i.e. all sorts of imperative control logic.

In addition to the set of functional operators, the calc model provides “split” and “combine” operators to dynamically define and re-distribute partitions of data flows as a base construct to enable application-defined data parallelization.

The individual compilers of the different domain-specific languages try to optimize the mapping from a given query script to a calc graph. For SQL, the mapping is based on the well-defined logical representation of a query expression. In the general case, the mapping may be based either on heuristics or cost-based, depending on the estimated size of the input data etc. For example, the compiler may decide to unroll a loop into a regular data flow graph or generate L code for the specific expression [6]. In the case of regular SQL, which is by far the largest and most complex part and taken from the SAP P*Time¹ system ([1]), the internal representation is directly mapped to a relational operators to capture the intent of the SQL statement.

A sample calc model graph is depicted in figure 3. Calc models are either created indirectly via the compiler of an individual domain-specific language, or can be visually modeled in the SAP HANA Studio and registered as calc views in an application-level content repository of the SAP HANA database. The overall idea behind this process is to customize specific fragments of a complex business logic scenario, which can be fine-tuned and re-used in multiple database scenarios, independent of the actual query language, i.e. calc models can be consumed from any domain-specific language stack in the form of a virtual table. The collection of calc

¹P*Time is a main-memory row-oriented relational database system, acquired by SAP in 2005 and optimized for SAP’s applications.

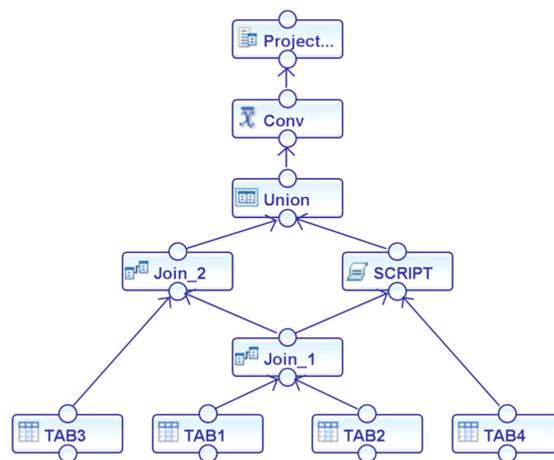


Figure 3: Example of a SAP HANA calc model graph

models is also referred to as SAP HANA content and undergoes a separate product life cycle process. The calc model shown in figure 3 outlines some of the differences with respect to regular query plans in standard relational database systems. For example, the result of an operator may have multiple consumers to optimize for shared common subexpressions already from an application point of view. Secondly, the node labeled “script” wraps imperative language snippets coming either from the calc model designer or being generated by a domain-specific query compiler. Finally, the node “conv” shows the use of a built-in business functions to perform application-specific conversion routines, e.g. for currency conversion or unit conversion.

2.2 Calc Graph Compilation and Execution

Once the user-defined query expressions or query scripts are mapped to a data flow graph in the calc model, the optimizer runs classical rule and cost-based optimization procedures to restructure and transform the logical plan into a physical plan which can then be executed by the distributed execution framework. The execution framework, which was inherited from prior SAP products SAP BWA (Business Warehouse Accelerator) and SAP Enterprise Search, orchestrates the actual data flow and the distributed execution of physical operators. During optimization, the fragments of the logical data-flow graph are mapped to physical operators provided by the “Engine Layer”. The Engine layer itself consists of a collection of different physical operators with some local optimization logic to adapt the fragment of the global plan to the specifics of the actual physical operator. In particular, the SAP HANA database provides the following set of operators:

- **Relational Operators:** The collection of relational operators handles classic relational query graph processing. As outlined in subsection 3.1 in more detail, relational operators show different characteristics, e.g. some of the operators like equi-join ([5]) directly leverage existing dictionaries of a unified table.
- **OLAP operators:** OLAP operators are optimized for star-join scenarios with fact and dimension tables.

Once the optimizer recognizes this type of scenarios, mapping of the corresponding query plan fragment to OLAP operators is enumerated as a feasible physical plan with corresponding cost estimation ([8]).

- **L runtime:** The runtime for the internal language L reflects the building block to execute L code represented in the L nodes of a given calc graph. Using the “split and combine” operator pair, the L runtime can be invoked in parallel working on the pre-defined partitions.
- **Text operators:** The set of text search analysis operators comprises the set of functionality already available in the SAP Enterprise Search product to deliver comprehensive text analysis features ranging from similarity measures to entity resolution capabilities ([14]).
- **Graph operators:** Graph operators finally provide support for graph-based algorithms to efficiently implement complex resource planning scenarios or social network analysis tasks.

Since a data flow graph is distributed not only between multiple server instances (usually running on different physical nodes) but also between different types of operators, the system provides a toolbox for the optimal data transfer and exchange format. Although all operators are required to implement a standard data transfer protocol, individual operators within or beyond different “collections” may have a highly specialized communication protocol. For example, the relational and OLAP operators are exchanging data in a highly compressed and proprietary format. Also, the R node provides a mapping to the R internal data frame format ([4]).

In addition to the “horizontal” communication between different physical operators, they also exploit a common interface to the unified table layer. As we will outline in more detail in the following section, the SAP HANA database provides an abstract tabular view with a variety of access methods for the different operators. The common tabular structure implements a complete life cycle of a data entity and basically consists of a combination of row- and column-store to capture the effects of the most recent modification operations. Since a table in the SAP HANA database can be marked as “historic”, the table layer also provides the implementation of a history table capturing the past values of an active entity and provides access methods for time travel queries.

Finally, SAP HANA relies on a persistence layer to provide recoverability in case of loss of the database state captured in main memory. The persistence layer is based on a virtual file concept with visible page limits of configurable size. Adapting the concepts of the SAP MaxDB system, the persistence layer relies on frequent savepointing to provide a consistent snapshot with very low resource overhead. The following section will provide more details.

2.3 Summary

In contrast to classical database systems, the SAP HANA database aims at playing the role of a flexible platform to support multiple (proprietary) domain-specific languages. A flexible data flow model (calc graph model) provides the conceptual core of the system: On the one side, query expressions or query scripts are mapped to an instance of the

model. On the other side, all different physical operators are using the same table layer interface implementing a complete life cycle management for individual records. Logging and data area are used to maintain a transactionally consistent copy of the main memory database in persistent storage.

3. LIFECYCLE MANAGEMENT OF DATABASE RECORDS

As shown in figure 4, a unified table structure provides data access for all applicable physical operators. Looking behind the facade of the notion of a unified table structure, the system provides life cycle management for an individual database record. We see the technique of the unified table not only as the key to provide excellent performance for both scan-based aggregation queries but also for highly selective point queries. This provides a key differentiator to classical (row-based) database architectures. While a record conceptually remains at the same location throughout its lifetime in update-in-place-style database systems, the SAP HANA conceptually propagates records through different stages of a physical representation. Although designed as a general concept, the most usual setup consists of the following three stages for records within a regular table (Figure 4).

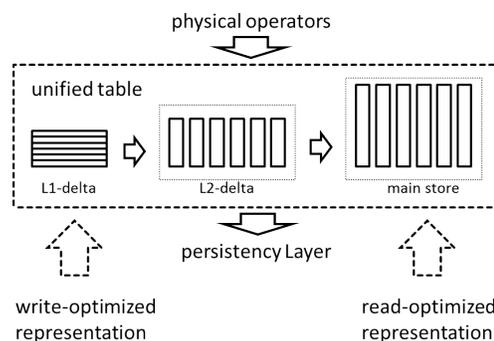


Figure 4: Overview of the unified table concept

- **L1-delta:** The L1-delta structure accepts all incoming data requests and stores them in a write-optimized manner, i.e. the L1-delta preserves the logical row format of the record. The data structure is optimized for fast insert and delete, field update, and record projection. Moreover, the L1-delta structure does not perform any data compression. As a rule of thumb, the L1-delta structure may hold 10,000 to 100,000 rows per single-node database instance depending on the workload characteristics and the amount of available memory.
- **L2-delta:** The L2-delta structure represents the second stage of the record life cycle and is organized in the column store format. In contrast to the L1-delta, the L2-delta employs dictionary encoding to achieve better memory usage. However, for performance reasons, the dictionary is unsorted requiring secondary index structures to optimally support point query access patterns, e.g. fast execution of unique constraint checks. The L2-delta is well suited to store up to 10 millions of rows.

- **Main store:** The main store finally represents the core data format with the highest compression rate exploiting a variety of different compression schemes. By default, all values within a column are represented via the position in a sorted dictionary and stored in a bit-packed manner to have a tight packing of the individual values ([15]). While the dictionary is always compressed using a variety of prefix-coding schemes, a combination of different compression techniques—ranging from simple run-length coding schemes to more complex compression techniques—are applied to further reduce the main memory footprint ([9, 10]).

Since the SAP HANA database was originally designed for OLAP-heavy use-cases with complex and high-volume loading scenarios, the system provides a special treatment for efficient bulk insertions, which may directly go into the L2-delta, bypassing the L1-delta. Independent of the place of entry, the RowId for any incoming record will be generated when entering the system. Also, logging happens at the first appearance of a row, be it within the L1-delta for regular update/insert/delete operations or for the L2-delta in case of bulk load operations.

3.1 Unified Table Access

The different data structures share a set of common data types. The access is exposed through a common abstract interface with row and column iterator, both optionally dictionary-based. Moreover, some of the physical operators may pull record-by-record or in a vectorized way (i.e. block-by-block) following the classical ONC-protocol [3] to enable pipelined operation and reduce the memory requirements for intermediate results as much as possible. Other physical operators implement the “materialize all”-strategy to avoid operator switching costs during query execution. The optimizer decides on a mixture of the different types of operators depending on the logical calc model, i.e. the different types of physical operators are seamlessly integrated within a final query execution plan.

For the operators leveraging sorted dictionaries, the unified table access interface also exposes the table content via a global sorted dictionary. Dictionaries of two delta structures are computed (only for L1-delta) and sorted (for both L1-delta and L2-delta) and merged with the main dictionary on the fly. In order to implement efficient validations of uniqueness constraints, the unified table provides inverted indexes for the delta and main structures.

The record life cycle is organized in a way to asynchronously propagate individual records through the system without interfering with currently running database operations within their transactional sphere of control. The current SAP HANA database system provides two transformations, called “merge steps”:

- **L1-to-L2-delta Merge:** The transformation from L1-delta to L2-delta implies a pivoting step from row to column organization. Rows of the L1-delta are split into their corresponding columnar values and column-by-column inserted into the L2-delta structure. At the receiving side, the system performs a lookup to identify potentially missing values in the dictionary structure and optionally inserts new entries at the end of the dictionary to avoid any major restructuring operations

within the dictionary. In the second step, the corresponding column values are added to the value vector using the dictionary encodings (append-only structure). Both steps can be performed in parallel, because the number of tuples to be moved is known in advance enabling the reservation of encodings in the new dictionary before actually inserting them. In a third step, the propagated entries are removed from the L1-delta. All running operations either see the full L1-delta and the old end-of-delta border or the truncated version of the L1-delta structure with the expanded version of the L2-delta. By design, the transition from L1-delta to L2-delta is incremental in nature, i.e. the transition of records does not have any impact in terms of reorganizing the data of the target structure.

- **L2-delta-to-main Merge:** A new main structure is created out of the L2-delta and the existing main. While the L1-to-L2-delta Merge is minimally invasive with respect to running transactions, an L2-delta-to-main merge is a resource-intensive task which has to be carefully scheduled and highly optimized on a physical level. As soon as an L2-delta-to-main merge is started, the current L2-delta is closed for updates and a new empty L2-delta structure is created serving as the new target for the L1-to-L2-delta merge. If a merge fails, the system still operates with the new L2-delta and retries the merge with the previous versions of L2-delta and existing main. Section 4 will detail the core algorithms and give some more details of different optimization techniques such as column-wise (subsection 4.2) or partial merge (subsection 4.3).

Both merge operations do not directly affect persistent storage and are independent of restart or backup log replay.

3.2 Persistency Mapping

Although the SAP HANA database is a main-memory-centric database system, its full ACID support guarantees durability as well as atomicity and recovery in case of a system restart after regular shutdown or system failure. Persistency of the SAP HANA database is based on multiple persistency concepts. In general no fine-grained UNDO mechanisms with respect to persistent storage are necessary, because only bulk changes like a new version of a main structure are propagated to persistent storage and have to be rolled back in case of a system failure. As can be seen in figure 5, the persistency is based on a combination of temporary REDO logs and save pointing for short-term recovery or long-term backup.

Logging for the REDO purpose is performed only once when new data is entering the system, either within the L1-delta or for bulk inserts within the L2-delta. New versions of a record are logged when entering the L1-delta. Changes which occur during the incremental propagation from the L1- to the L2-delta are NOT subject of REDO logging. Instead, changes in the dictionary as well as in the value index are added to the data structures residing in individual data pages, which are eventually moved to persistent storage within the next savepoint. Obviously the event of the merge is written to the log to ensure a consistent database state after restart.

Figure 6 outlines some of the details. Both structures, the dictionary and the value index are based on a paged storage

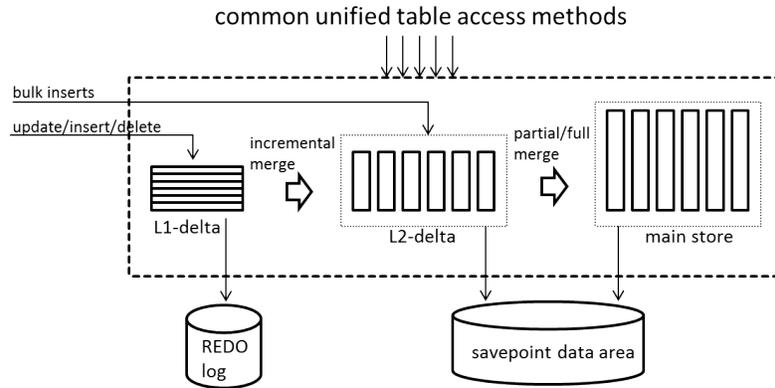


Figure 5: Overview of the persistency mechanisms of the unified table

layout managed by the underlying storage subsystem. Dirty pages—either existing pages with additional entries or new pages—are flushed out by the storage subsystem under the control of the savepointing infrastructure. Although the L2-delta structure is organized per column, the system may store fragments of multiple L2-deltas within a single page in order to optimize for memory consumption. Especially for small but wide tables, storing multiple L2-deltas within the same page can be very reasonable.

3.3 Summary

In summary, the physical representation of a table within the SAP HANA database consists of three levels—a row store (L1-delta) to efficiently capture incoming inserts as well as update and delete requests, an intermediate structure in column format (L2-delta) to decouple the write-optimized from a read-optimized store, the main store structure. This third structure is extremely well suited for OLAP-like queries, but is also well tuned to answer point queries efficiently by using inverted index structures. During the lifetime, a record will be asynchronously propagated through the storage structures to land in the most update efficient store at the beginning and stay in the most read-efficient store for the rest of its lifetime.

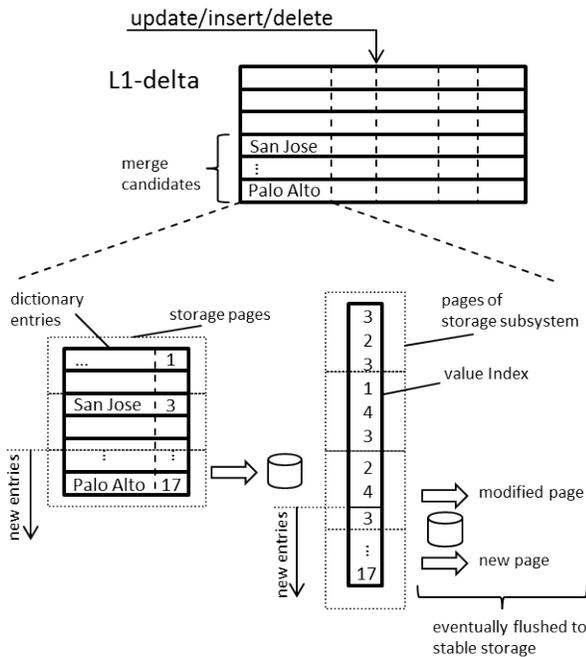


Figure 6: Details of the L1-to-L2-delta merge

After the savepoint, the REDO log can be truncated. During recovery, the system reloads the last snapshot of the L2-delta. Similarly, a new version of the main will be persisted on stable storage and can be used to reload the main store of the unified table. In summary, neither changes at the L2-delta nor changes of the main are recorded in a log because the image of the previous version still exists. Classical logging schemes are only employed for the L1-delta.

4. MERGE OPTIMIZATION

The basic idea of the Unified Table approach is to provide a transparent record propagation from write-optimized storage structure to read-optimized storage structures with the L2-delta index to de-couple both extreme worlds. While the transition from the L1-delta to the L2-delta can be conducted without major disruption of the existing data structures, the merge of L2-delta and main requires a major reorganization of the table's content.

4.1 The Classic Merge

In a first step of a classic merge operation, the dictionary entries of the L2-delta are compiled into the dictionary of the main lexicographically to yield a sorted new main dictionary for the specific column. The new dictionary contains only valid entries of the new main structure, discarding entries of all deleted or modified records. The sort order of the dictionary not only provides the prerequisite for optimal compression but also is the base for special operators working directly on dictionary encoded columns.

Figure 7 shows the principal phases of a merge step. Based on the L2-delta with an unsorted dictionary and the old main with a sorted dictionary, the first phase generates the new sorted dictionary and preserves the mapping information from the new positions (which are obviously not explicitly stored) and the old positions within the main and L2-delta. As can be seen in the figure, some entries show positions in both dictionaries (e.g. “Los Gatos”) or they only

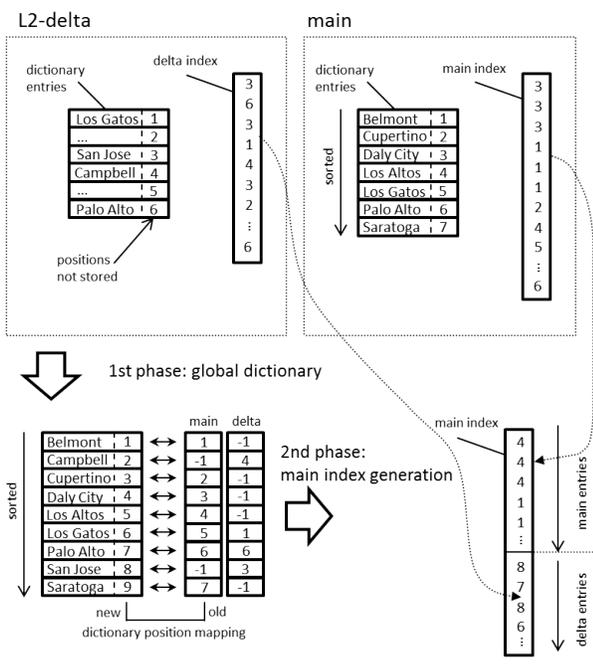


Figure 7: Details of the L2-delta-to-main merge

appear in the main or L2-delta dictionary (e.g. “Campbell” with value 4 in the delta and a value of -1 at the main side of the dictionary position mapping table). In a second phase, the new main index is constructed with the positions referring to the new dictionary for existing and newly added entries. For example-referring to figure 7—the entries for “Daily City” are transferred to the new main with the new position value 4. Entries for “Los Gatos” are also mapped to the new position (now 6) from position 1 in the L2-delta and position 5 in the old main structure. The new main (dictionary and value index) is written to disk and the old data structures are released. In any case the system has to keep the old and the new versions of a column (dictionary and main index) in main memory until all database operations of open transaction still referring to the old version have finished their execution.

Since the naive version of the merge is very resource-intensive, the SAP HANA database implements a number of different optimizations. For example, if the dictionary of the L2-delta is a subset of the main dictionary, the first phase of a dictionary generation is skipped resulting in stable positions of the main entries. Another special case exists if the values of the L2-delta dictionary are greater than the values in the main dictionary, e.g. in the presence of increasing timestamps. In this situation, the dictionary of the L2-delta can be directly added to the main dictionary, if the number of bits to encode the dictionary values are sufficient to cope with the extended cardinality.

More complex optimizations can be seen in the orthogonal techniques of re-sorting merge and partial merge strategies—both techniques will be outlined in more detail below.

4.2 Re-Sorting Merge

The classic version of the merge (subsection 4.1) between the L2-delta and the main requires a mapping of the previous

positions of the dictionary entries to the new positions of the new dictionary. The positions then encode the real values within the bit-packed value index, i.e. with C as the number of distinct values of a column, the system spends $\lceil \log_2(C) \rceil$ -many bits to encode the positions. The merge maps the old main values to new dictionary positions (with the same or an increased number of bits) and adds the entries of the L2-delta at the end of the new value index.

An extended version of the merge aims at reorganizing the content of the full table to yield a data layout which provides higher compression potential with respect to the data distribution of ALL columns. Since the SAP HANA database column store exploits a positional addressing scheme, the values of the k -th record have to be at the k -th position in every column. Re-sorting one column to gain an optimal compression scheme therefore directly affects the compression potential of all other columns within the table. Applying the concepts discussed in [9], the system computes the “best” sort order of the columns based on statistics from main and L2-delta structures before creating the new main.

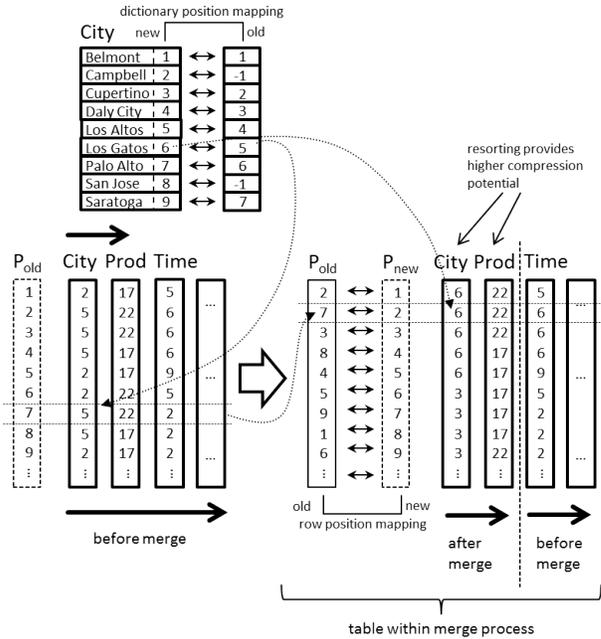


Figure 8: Delta-to-main merge with reordering

Figure 8 shows the necessary data structures. In addition to the mapping table for the dictionary to translate old dictionary positions to the positions in the new dictionary, the version of the re-sorting merge additionally creates a mapping table of the row positions to be able to reconstruct the row after merging and re-sorting individual columns. The figure shows columns of the same table before and within a merge process where columns “City” and “Prod” are already merged, the remaining columns (e.g. “Time” etc.) still reflect the status before the merge. Therefore, the entries of the old version of the main correspond to positions in the old dictionary, e.g. the entry “Los Gatos” of the “City” column is encoded with value 5 in the old dictionary and 6 in the version after the merge. Thus in general, after applying the merge to the “City” column, the new main index shows the dictionary positions of the new dictionary as well as a

re-sorting of the rows. As highlighted, the 7th row can now be found at the second position. The “Prod”-column was also merged without building a new dictionary, e.g. the dictionary positional values are preserved. The “Time”-column however was not yet merged and still refers to the old dictionary and the old sort order. Any access to not yet merged columns is required to take an additional indirection step via the row position mapping table if a row construction with already merged columns is required. The row position mapping table can be eliminated after the merge of all columns has been completed. Although the system may conceptually delay the merge of infrequently accessed columns by “stacking” row position mapping tables, the system always completely finishes a merge operation for the full table before starting a new merge generation.

Applying a re-sorting merge is therefore a cost-based decision to balance the overhead of the additional position mapping for column accesses during the merge for all columns and the resulting potential for a higher compression rate. The sort criterion for applying the merge to the individual columns also depends on multiple factors, e.g. ratio of point versus range access, improvement in compression potential etc.

4.3 Partial Merge

The major drawback of the classic or the re-sort merge consists in the overall overhead to create a new version of the main. For large tables or partitions, computing a new dictionary and re-generating the main index does have a negative impact on available CPU and disk resources. The partial merge tries to soften this problem by generalizing the previous algorithms. The partial merge strategy shows the best potential for saturated columns, i.e. in situations when the number of new entries in the dictionary is small.

The core idea of the partial merge is to split the main into two (or even more) independent main structures:

- **Passive main:** The passive main reflects a stable part of the main store which is in general not part of the merge process.
- **Active main:** The active main is the part of the column which grows/shrinks dynamically and takes part of the merge process with the L2-delta.

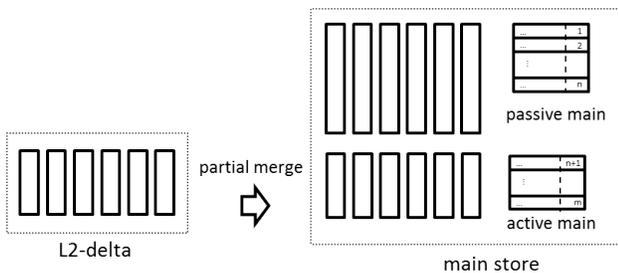


Figure 9: Overview of partial merge

Conceptually, a merge interval within the partial merge strategy starts with an empty active main. The passive main reflects the regular main structure with a sorted dictionary and a corresponding values index. Whenever a merge operation is scheduled, the L2-delta merges with the (still

empty) active main; the passive main remains untouched. Compared to the full merge, the partial merge shows one small exception. The dictionary of the active main starts with a dictionary position value of $n + 1$ with n as the cardinality of the passive main dictionary. Although the system now has two main structures with locally sorted dictionaries, the encodings of the individual main value index structures are not overlapping. Obviously, the dictionary of the active main only holds new values not yet present in the passive main’s dictionary.

Figure 10 shows a sample situation with a passive and an active main after a partial merge. The dictionary codes of the active main start with the encoding value $n + 1 = 6$, such that it continues the encoding scheme of the passive main. While the corresponding value index structure of the passive main only holds references to entries in the passive main dictionary, the value index of the active main also may exhibit encoding values of the passive main making the active main dictionary dependent on the passive main dictionary.

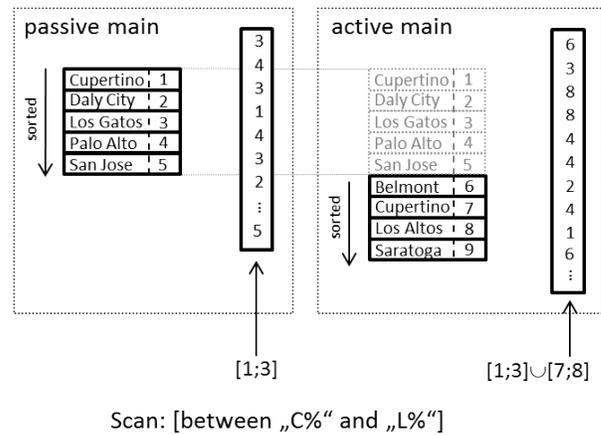


Figure 10: Range query execution for active and passive main

A point access is resolved within the passive dictionary. If the requested value was found, the corresponding position is used as the encoding value for both, the passive and the active main value index. Parallel scans are executed to find the corresponding entries. However, if the requested value was not found, the dictionary of the active main is consulted. If the value is present, only the active main value index is scanned to identify the resulting row positions. For a range access, the ranges are resolved in both dictionaries and the range scan is performed on both structures. For the active main, the scan is broken into two partial ranges, one for the encoded range value of the passive dictionary and one for the encoded range value of the active main dictionary. Figure 10 illustrates this behavior for a range query with values between $C\%$ and $L\%$. In order to guarantee transaction consistency, the query processing additionally requires similar merges with the L1- and L2-delta.

While the system is operating, the active main may dynamically shrink and grow until a full merge is scheduled. The major advantage of the concept is to delay a full merge to situations with low processing load and reduce the cost of the L2-to-(active-)main merge. Also, the optimization strategy may be deployed as a classical merge scheme by

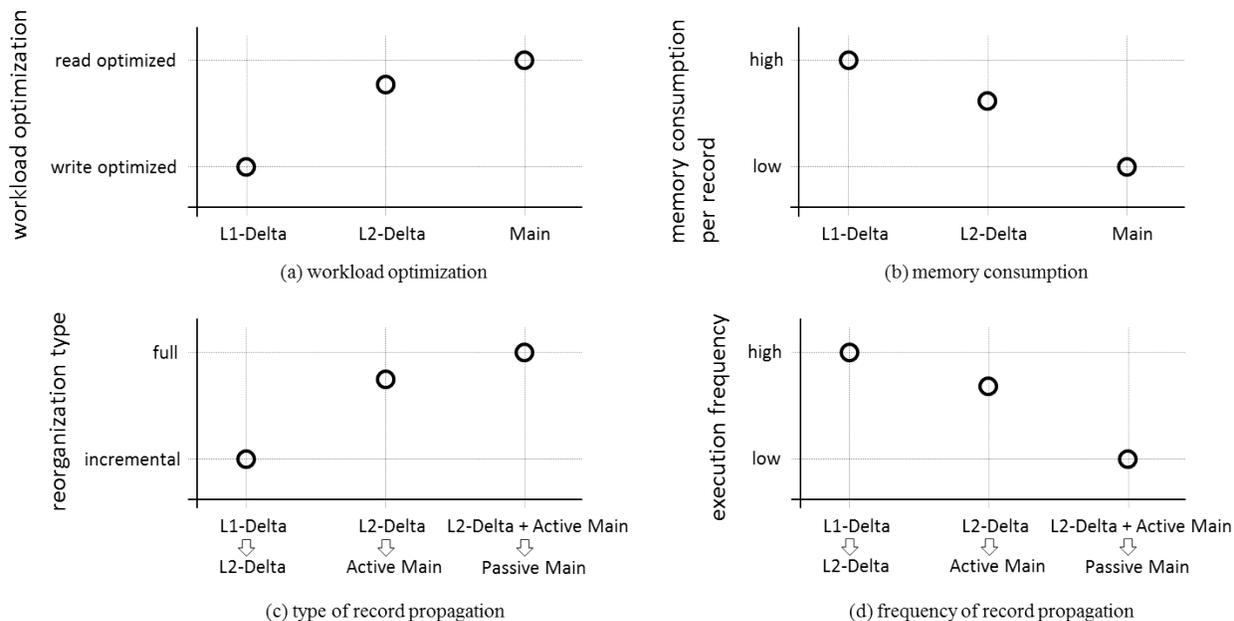


Figure 11: Characteristics of the SAP HANA database record life cycle

setting the maximal size of the active main to 0 forcing a (classical) full merge in every step. Obviously, the procedure can be easily extended to multiple passive main structures forming a logical chain with respect to the dependencies of the local dictionaries. This configuration is extremely worthwhile for columns with slowly changing or stable dictionaries (e.g. “Country” column in a “Customer”-table). However, for most of the columns, the system will hold only one passive main.

From a conceptual perspective, the partial merge optimization strategy implements an additional step in the general record life cycle of the SAP HANA database unified table concept. The closer to the end of the pipeline, the more complex, time- and resource consuming re-organizations are applied to the records to finally end in the highly compressed and read-optimized format of the traditional column store. In addition, the SAP HANA database provides the concept of historic tables to transparently move previous versions of a record into a separate table construct. Therefore, a table has to be defined of type “historic” during creation time. Furthermore, the partitioning concept can be used to separate recent data sets from more stable data sets from an application point of view.

4.4 Summary

As outlined, the SAP HANA database exploits the idea of a record life cycle management to provide efficient access for transactional and analytical workloads. Figure 11 highlights the different characteristics of the discussed storage formats and propagation steps. The L1-delta is optimized for update intensive workloads and can be incrementally and frequently merged into the L2-delta structure. The L2-delta structure is already well-tuned for read operations but requires a larger memory footprint compared to the highly read-optimized main structure. However, L2-delta serves particularly well as a target of the L1-delta rows or bulk

insertions. As previously discussed, the main, optionally split into an active and passive part, exhibits the highest compression rate and is optimized for scan-based query patterns. Due to the resource-intensive re-organization tasks, merges into the active main and especially full merges to create a new main structure are scheduled with a very low frequency. The merge of L1- to L2-delta, in contrast, can be performed incrementally by appending data to the L2-delta dictionary and value index.

5. CONCLUSION

Column store systems are well known to provide superb performance for OLAP-style workload. Typically aggregation queries touching only a few columns of 100s of millions of rows benefit from a column-oriented data layout. Unfortunately, the separation of OLAP-style and OLTP-style queries is no longer reflecting state-of-the-art requirements from modern business applications. On the one hand, operational systems embed more and more statistical operations for the on-the-fly business decision into the individual business process. On the other hand, classical data-warehouse infrastructures are required to capture transactions feeds for real-time analytics. Based on the classical column store architecture of the SAP HANA database, we outline the query processing environment showing query transformation, plan generation, and the interaction model of different specialized engines. Additionally, we explained in more detail the common unified table data structure consisting of different states on the one hand but providing a common interface to the consuming query engines on the other hand. The overall goal is to demonstrate some of the optimizations which are implemented within the SAP HANA database to make a column store suitable for high-scale transactional processing and ending the myth to use columnar technique only for OLAP-style workloads.

6. ACKNOWLEDGMENTS

We would like to express our sincere thanks to the SAP HANA database team in Walldorf, Seoul, Berlin, and Palo Alto for making the HANA story a reality.

7. REFERENCES

- [1] S. K. Cha and C. Song. P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload. In *VLDB*, pages 1033–1044, 2004.
- [2] 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.
- [3] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.
- [4] 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. *PVLDB*, 4(12):1307–1317, 2011.
- [5] G. Hill and A. Ross. Reducing outer joins. *VLDB J.*, 18(3):599–610, 2009.
- [6] B. Jaeksch, F. Färber, F. Rosenthal, and W. Lehner. Hybrid Data-Flow Graphs for Procedural Domain-Specific Query Languages. In *SSDBM Conference*, pages 577–578, 2011.
- [7] B. Jaeksch, W. Lehner, and F. Färber. A plan for OLAP. In *EDBT conference*, pages 681–686, 2010.
- [8] T. Legler, W. Lehner, and A. Ross. Data mining with the sap netweaver bi accelerator. In *VLDB*, pages 1059–1068, 2006.
- [9] C. Lemke, K.-U. Sattler, F. Färber, and A. Zeier. Speeding up queries in column stores - a case for compression. In *DaWak*, pages 117–129, 2010.
- [10] 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 *IDEAS*, pages 205–215, 2010.
- [11] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD Conference*, pages 1–2, 2009.
- [12] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, Berlin Heidelberg, 2011.
- [13] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it’s time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
- [14] F. Transier and P. Sanders. Engineering basic algorithms of an in-memory text search engine. *ACM Trans. Inf. Syst.*, 29(1):2, 2010.
- [15] 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*, 2:385–394, August 2009.
- [16] M. Winslett. Bruce Lindsay Speaks Out. *SIGMOD Record*, 34:71, June 2005.