Merging Differential Updates in In-Memory Column Store

Jens Krueger, Martin Grund, Johannes Wust, Alexander Zeier, Hasso Plattner
Hasso Plattner Institute for IT Systems Engineering
University of Potsdam
Potsdam, Germany
jens.krueger@hpi.uni-potsdam.de, martin.grund@hpi.uni-potsdam.de, johannes.wust@hpi.uni-potsdam.de
alexander.zeier@hpi.uni-potsdam.de, hasso.plattner@hpi.uni-potsdam.de

***Abstract*—To meet the performance requirements of enterprise application for both, transactional application as well as analytical scenarios, data storage of in-memory databases are split into two parts: One optimized for reading and a write-optimized differential buffer. The read-optimized main storage together with the differential buffer for inserts provide the current state of the database. In regular intervals the differential buffer is merged with the main database to maintain compression and query performance. This merge process runs asynchronous to minimize the impact on query performance. However, simple duplication of the data structures prior to the merge process lead to a main memory consumption of at least twice the size of the database. In this paper we propose a differential merge update based on single columns. In typical enterprise application data environments this leads to a significant reduction of memory consumption as this type of applications tend to store transactional data in very large single tables. The Single Column Merge has been implemented in HYRISE and proved in a test scenario based on real enterprise data.**

***Index Terms*—In-Memory Database; Column Store; Merge Process;**

I. INTRODUCTION

Enterprise data management systems currently in use are typically being optimized either for transactional data processing (OLTP) or analytical data processing (OLAP). In order to combine both requirements for mixed workload scenarios the introduction of a write optimized differential buffer together with a read-optimized main storage has been proposed in [4], [8], [14]. The main advantage of this design is that the compression of the read storage does not need to be re-compressed every time a data modification operation is executed as all changes are stored in a differential buffer. However, the main storage and the differential buffer have to be merged at some point of time to maintain the performance in read intensive scenarios, mainly for two reasons:

- Merging the differential buffer into the main relation decreases the memory consumption since better compression techniques can be applied.
- Additionally, merging the buffer allows better read query performance due to an order-preserving value dictionary of the main store.
- Furthermore, the bit compression of valueID's allows better bandwidth utilization which leads to improved read performance since in-memory databases suffer from the bandwidth limitations of todays hardware.

The key requirement for the merge process is to have as little impact as possible on the performance of the database. Therefore it has to run asynchronously to other operations such as query execution. The cost of this process is mainly determined by the performance impact on the other operations and main memory consumption. This paper focuses on the optimization of the memory consumption.

A. Enterprise Application characteristics

We applied the concept of a differential buffer to column-oriented, in-memory databases, as we could show that these databases perform especially well in Enterprise Application scenarios. By analyzing customer applications and customer data we derived typical enterprise application characteristics as shown in [9], [10]. The most important findings based on the customer system analysis and their implications on database design are:

- Enterprise applications typically present data by building a context for a view, modification to the data only happen rarely. Hence column-oriented, in-memory databases that are optimized for reading as proposed in [8], [12] perform especially well in enterprise application scenarios. In fact, over 80% of the workload in an OLTP environment are read operations.
- Tables for transactional data typically consist of 100-300 columns and only a narrow set of attributes is accessed in typical queries. Column-oriented databases benefit significantly from this characteristic as entire columns, rather than entire rows, can be read in sequence.
- Enterprise data is sparse data with a well known value domain and a relatively low number of distinct values. Therefore data of enterprise applications qualifies very well for data compression as these techniques exploit redundancy within data and knowledge about the data domain for optimal results. Abadi et al. have shown in [1] that compression applies particularly well to columnar storages. Since all data within a column a) has the same data type and b) typically has similar semantics and thus low information entropy, i.e. there are few distinct values in many cases.

 ISBN:978-1-61208-115-1

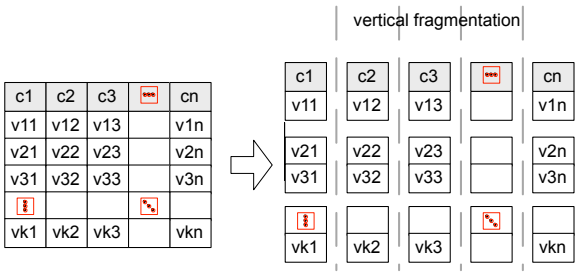

Fig. 1. Column-oriented storage paradigm

- Enterprise applications typically reveal a mix of OLAP and OLTP characteristics [10]. To support both, the data storage of in-memory databases are split into two parts, one optimized for reading and one for writing.
- Data growth in enterprise systems has not shown the same growth rate as for example social networks. Despite the fact of a growing number of captured events in enterprise environments all events are based on actual events related to the business which have an inherent processing limit by the size of the company.

Given that findings on enterprise application our approach to build an application-specific data management is focused on in-memory data processing with data compression and column-wise data representation in order to utilize todays hardware as best as possible.

B. Structure of the paper

The remainder of the paper is structured as the following: First we introduce HYRISE, our prototypical column-oriented, in-memory database prototype used to empirically validate our findings. The next section gives an overview of the traditional merge process of main storage and differential buffer. In Section IV, we propose a modified merge algorithm, the Single Column Merge, that reduces additional memory consumption during the merge process. Section V gives an overview of related work while Section VI concludes this work.

II. Overview of HYRISE

A. HYRISE architecture

The following section describes the architecture of the HYRISE prototype, including the storage manager and query executor.

The storage manager maintains the physically stored data in main memory and provides access methods for accessing data while organizing data along columns with applied dictionary compression. Consequently, all relations are fully decomposed while a surrogate identifier allows the reconstruction of tuples of the column partitions. Figure 1 shows the vertical fragmentation of a table as used in HYRISE and depicts that attribute focused read operation can exploit sequential memory access while tuple reconstruction requires random access to each column. By choosing to optimize this database prototype for an online mixed workload (OLXP) as described in [10] the reconstruction of complete relation in a timely manner gets equally important as data modifications and scans over large sets of data.

In case of HYRISE the row or surrogate identifier is implicit and can be extracted from the position of a value in a column. Therefore, fast access due to offsetting is made possible which can also be leveraged in positional joins algorithms. Unlike other lightweight compression techniques the implemented dictionary encoding enables this positional access since it facilitates the change of variable-length fields into fixed-length data types on each column.

In order to speed up read access by as late as possible decompression of the actual value the dictionary of the encoding in HYRISE is sorted leading to order-preserving values in the actual column. Considering this, predicates can be applied on the attribute vector and ranges can be looked up without decompressing every single value. Besides, the sortation enables fast binary search on the dictionary.

Furthermore, the storage bit-compresses the values pointing from the dictionary to the attribute vector by using only the amount of bits necessary to represent the cardinality of distinct values of each column. Especially in enterprise applications the attributes are characterized by a limited domain. Hence, bit compressing value identifiers is very effective and improves the compression factor even more. Besides the additional compression bit compressed value identifiers support better bandwidth utilization in late materialized query executions.

While this extended dictionary compression technique offers both good compression ratio and optimized read access modifications of data are almost impossible due to fact that the data would have to be re-compressed every time modification operation would be executed. For example, if a new value would change the sort order of the existing dictionary or the cardinality of distinct values changes in a way that the already used bits are not sufficient the complete attribute vector has to be modified. Consequently, all modifications are handled by a dedicated differential buffer for each table to postpone re-compression cost to later point of time to distribute the re-compression cost over all data modifications stored in the buffer. This re-compression is done by merging differential buffer and main storage. The buffer implements a vertical partitioning as well but leaves out both the order-preserving and value bit-compressing optimizations in order to allow fast appends to the table. This architecture is based on the fact that decomposition of relations in main memory with the lookup or extension of the dictionary is way faster than writing the log to disk that has to happen in in-memory databases to assure durability.

Given the fact of a dedicated buffer to handle all data changes, updates have to be implemented as an insert followed by an invalidation of the to be updated record. The invalidation is maintained by a two bit vectors, which keep track of updates and deletes in the compressed storage and the corresponding differential buffer. The storage manager is in charge of keeping data consistent what in this case means the main storage and delta storage have to be kept in sync and corresponding merge

 ISBN:978-1-61208-115-1

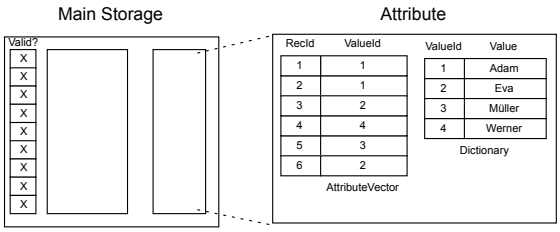

Fig. 2. Example of a main storage

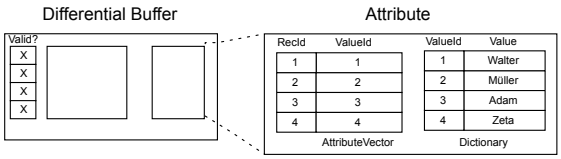

Fig. 3. Example of a differential buffer

processes have to run asynchronously to avoid conflicts with running queries during the merge process.

The query executor is responsible for executing a given query plan, including loading the necessary meta data and materializing results with regards of late materialization strategies that are a result of the column-wise data representation with applied dictionary encoding. The current state of the system provides no direct access using a query language like SQL but focuses on the implementation of the plan operators. It implements the necessary relational algebra operators and leaves the query plan design up to the user of the prototype. Hence the query plans used are written by hand and than executed by the execution engine while assuming that this written query plans are optimal and no further optimization takes place.

For the purpose of this study, some features of a conventional database such as multi-threading, transactions, or recovery are not implemented to avoid the related overhead. We omit these features because we believe they are orthogonal to the question of how to compact data using a merge process in an in-memory column store. For the same reason the process of loading data from a storage system at startup time is not taken into account.

B. HYRISE data structures

In the following we illustrate the data structures for main storage and differential buffer. Figure 2 shows an illustration of a main storage. The data structures for one column are illustrated in detail. The table *AttributeVector* shows the vector holding the values for each record of a particular attribute. The values are dictionary compressed; therefore the stored *ValueId*s are references to the table *Dictionary* containing the actual values. The *Valid?BitVector* indicates whether this record is still valid or has been invalidated by an update or delete in the differential buffer.

New entries are stored in a write optimized differential storage as shown in Figure 3. The example shows 4 newly added entries in the *AttributeVector*. Similar to the main storage, the differential buffer has a *Dictionary*. The main difference between both storages is the implementation of the dictionary as discussed in the section above. All dictionaries used by the main storage need to be sorted in order to allow binary search and are bit-compressed. In contrast, the dictionaries in the differential buffer are unsorted and not bit-compressed to allow fast appends.

III. THE MERGE PROCESS

A. Description of the merge process

The merge process and its complexity is described in detail in [8]; we give a brief overview here. The process can be separated into three phases - *prepare merge*, *attribute merge* and *commit merge*.

The prepare merge phase locks the differential buffer and main storage and creates a new empty differential buffer storage for all new inserts, updates, and deletes which occur during the merge process. Additionally the current valid vector of the old buffer and main storage at merge time are copied to be used throughout the merge process, as these may be changed by concurrent updates or deletes applied during the merge while affecting records involved in this process. In the attribute merge phase the following steps are executed for each attribute: the first step is merging the dictionaries of the differential buffer and main storage. Next, the value ids of main storage and write buffer are copied to a new main storage - thereby changes in the dictionary have to be applied to the new value ids; invalidated values of the original main storage are not copied and can be transferred to a history log. To ensure persistency, the merge result is written to secondary storage.

The commit merge phase starts by acquiring a write lock of the table. This ensures that all running queries are finished prior to the switch to the new main storage including the updated value IDs. Then, the valid vector copied in the first phase is compared to the actual vector to mark potentially invalidated rows - they are eventually deleted in the next merge process. As last step the new main storage replaces the original differential buffer and main storage and the latter ones are unloaded from memory.

Figure 4 shows the result of the merge process based on the differential buffer and main storage shown in figures 2 and 3. *New AttributeVector* now holds all value records of the original main storage, as well as the differential buffer. Note that the new dictionary includes all values from the main and differential buffer and is resorted to allow binary search and late materializing range queries. Therefore the *ValueId* of single value records has changed compared to the original entry in the main storage and differential buffer.

B. Memory consumption of the merge process

As discussed in [8] prior to the commit merge phase the complete new main storage is kept inside main memory. Hence, at this point double the size of the original main storage

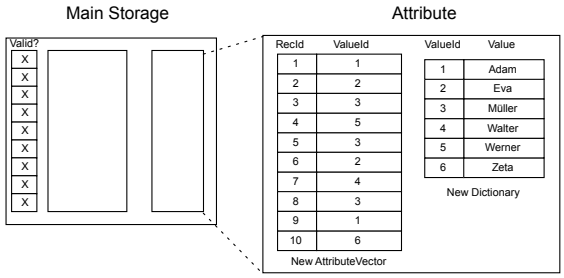

Fig. 4. Example of a delta storage

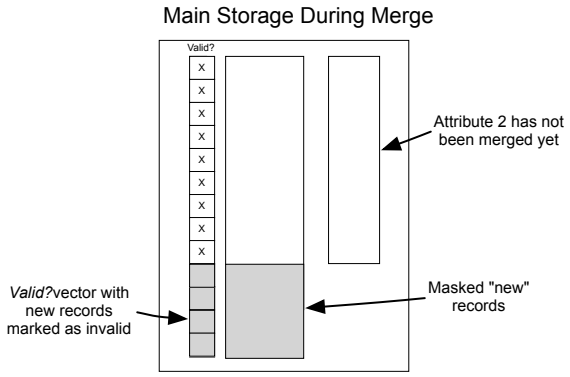

Fig. 5. During the merge: abstract view on the main storage with a single merged attribute

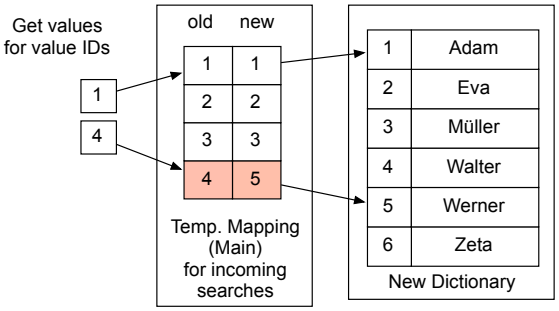

Fig. 6. Example of a remapped lookup for values 1 and 4

plus differential buffer is required in main memory to execute the proposed merge process. In the subsequent section we propose a modification of the algorithm to decrease the overall additional memory consumption.

IV. Single Column Merge

In this section we describe a modified merge process called *Single Column Merge* with the objective of reducing the size of memory consumption throughout the merge process. By merging single columns independently from the delta into the main storage the algorithm reduces the additional memory consumption to the size in memory of the largest column. In order to use this technique the insert only strategy has to be used otherwise records would be physically deleted what could lead to inconsistent surrogate identifiers if merged columns are applied independently. So far deleted records are kept as invalid in the storage system but could be removed by a dedicated garbage collection run.

A. Description of the Single Column Merge

In the merge process described in section III-A the merge result for single columns is calculated independently in the respective attribute merge phases. The merge result is kept in main memory until all attributes are merged to ensure an instant switch to the new main storage in the commit merge phase. The basic idea of Single Column Merge is to switch to an updated main storage after every attribute has been merged while maintaining a consistent view on the data.

Partial hiding of merge results: Switching already merged columns leads to a problem: Some attributes are already merged while others are not. Those finished attributes typically have a longer attribute vector since new rows could have been inserted into the differential buffer. And as this buffer is not updated throughout the merge process value entries for newly created rows are duplicated in the update main storage and original differential buffer. To resolve this issue all newly created rows are marked as invalid until all columns are merged as shown in Figure 5.

Remapping old value IDs: After one attribute is merged, its state differs from the rest of the index that has yet to be merged. Some values potentially have new value IDs if the merge process has changed the value IDs. Incoming queries might still rely on old value IDs, e.g. in case they have been cached by queries started prior to the merge process. To avoid locking of the table for each attribute a mapping table from the old value IDs to the new ones is provided throughout the merge process until all attributes are merged into the main store. This mapping table from old to new values is created in the attribute merge phase of the merge process described in section III-A when merging the dictionaries of differential buffer and main store. Figure 6 shows an example for a remapped lookup of the cached old value IDs 1 and 4.

Modifications of the traditional merge process: To implement the Single Column Merge as described we have to make the following changes to the merge process as described in section III-A:

- *prepare merge*
  - The valid vector of the main store has to be enlarged by the number of rows that are currently in the differential buffer. This is required to hide the newly created merge results in the main storage until all attributes are merged.
  - The newly created valid record entries are initialized with *false* to deactivate those rows.
- *attribute merge*: For each attribute the following changes have to be made:

  - Keep the mapping tables from old to new value IDs in memory. These tables have to be provided to functions in the query executor that might be called while a merge is running to have a consistent view on the data.
  - Switch the attribute data structure of the old main storage to the merge result right after merging the attribute.
- *commit merge*
  - activate the newly merged rows by setting the valid vector entries to *true*.
  - Unload mapping tables from old to new value IDs after the lock on the table is acquired.

B. Evaluation of memory consumption

Applying Single Column Merge eliminates the need to additionally hold the newly created main storage of the size of the original main storage and differential buffer in main memory. As only one attribute is merged at a time the additional amount of main memory needed for the merge process is the size of the attribute data structure currently merged plus the size of the mapping tables from old value IDs to new value IDs for the dictionaries as described in section IV-A. Assuming that the main storage is significantly larger in size than the differential buffer, the overall additional memory consumption for the merge process is driven by the size of the largest data structure of all attributes.

To test how large the savings in additional memory consumption are, we compared the traditional merge process described in section III-A and the Single Column Merge using live customer data. The two major tables in the database consist of 28 million rows with 310 columns and 11 million rows with 111 columns. The main memory usage during the test is shown in Figure 7. The graph shows the additional memory consumption during a merge process for both merge strategies. The column that consumes the most memory can be seen in both test series. The main memory usage during the Single Column Merge clearly peaks at around the size of the largest column, as opposed to the steadily increasing memory usage during the traditional merge.

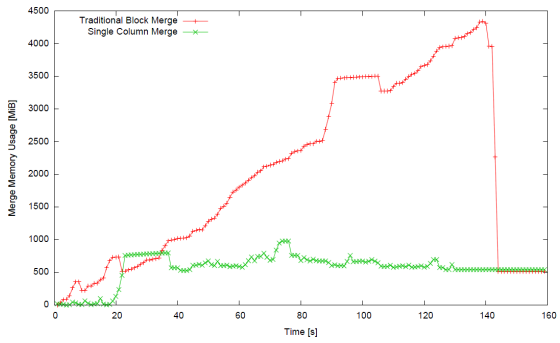

Fig. 7. Main memory usage during traditional merge process and single column merge

V. Related Work

Vertical partitioned databases as HYRISE have been researched from the very first conferences on database systems [2], [11], [11], [15] while focusing on read-intensive environments. Pure vertical partitioning into a “column-store” has been a recent topic of interest in the literature. Copeland and Khoshafian [5] introduced the concept of a Decomposition Storage Model (DSM) as a complete vertical, attribute-wise partitioned schema, which has been the foundation for multiple commercial and non-commercial column store implementations such as MonetDB/X100 [4], C-Store [14] or Sybase IQ [7]. All of those examples has shown ability to outperform conventional databases in read-mostly analytic-style scenarios with low selectivity. However, unlike HYRISE, most of the column-store implementations are pure disk based approaches and focus to improve the overall performance by reducing the number of disk seeks by decomposing relations. Consequently, data modifications must be propagated to multiple files on disk, which leads to the fact that this implementation variant is inappropriate for workloads combining transactional- and analytical-style queries, because updates and inserts are spread across different disk locations.

As in HYRISE, data compression can limit the applicability to scenarios with frequent updates leading to dedicated delta structures to improve the performance of inserts, updates and deletes. The authors of [4] and [13] describe a concept of treating vertical fragments as immutable objects, using a separate list for deleted tuples and uncompressed delta columns for appended data while using a combination of both for updates. In contrast, HYRISE maintains all data modification of a table in one differential buffer and keeps track of invalidation with a valid bit-vector. However, none of before mentioned work describes in detail how the merge process works.

In contrast to this disk based research, HYRISE builds up on in-memory data processing, which has been influenced in the last decade by the work around MonetDB [3]. The widening gap between the growth rate of CPU speed and memory access speed leads to the usage of compression techniques requiring higher effort for de-compression. Besides the direct effect of storage savings, less physical data has to be transferred from main memory traded for higher CPU costs at the de-compression of the data as described for instance in [16] or [6]. All works on compression on databases systems focus on the data amount reductions and at the same time on query optimizations.

VI. Conclusion

Optimized for main memory consumption, the Single Column Merge removes the need to keep a complete copy of the table during the merge process. Instead the main memory consumption can be reduced to a copy of each attribute. The maximum table size increases from half of the total available main memory to the total available main memory minus the

 ISBN:978-1-61208-115-1

largest columns size. As the merge process is a background task for an operational system queries can still process the data, and lookup information in both the attribute vector and the value dictionary. This concurrency is a requirement for reengineering the merge process in an online mixed workload environment. The Single Column Merge solves concurrency issues by storing an additional mapping table for each column. Every value dictionary lookup during the merge has to access the mapping table first, before it can access the value dictionary. Consequently, this remapping results in one additional random memory access for every value ID lookup but only in case the merge process has not been finished.

REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, New York, NY, USA, 2006. ACM.

[2] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD*, 2004.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, 1999.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[5] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[6] G. V. Cormack. Data Compression on a Database System. *Commun. ACM*, 28(12):1336–1342, 1985.

[7] C. D. French. "One Size Fits All" Database Architectures Do Not Work for DDS. In *SIGMOD*, 1995.

[8] J. Krueger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber. Optimizing Write Performance for Read Optimized Databases. In *DASFAA*, 2010.

[9] J. Krueger, M. Grund, A. Zeier, and H. Plattner. Enterprise Application-specific Data Management. In *EDOC 2010*, 2010.

[10] J. Krueger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner. A case for online mixed workload processing. In *DBTest*, 2010.

[11] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Trans. Database Syst.*, 9(4), 1984.

[12] H. Plattner. A common database approach for oltp and olap using an in-memory column database. In *SIGMOD*, 2009.

[13] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[14] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[15] P. J. Titman. An Experimental Data Base System Using Binary Relations. In *IFIP Working Conference Data Base Management*, 1974.

[16] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3), 2000.

 ISBN:978-1-61208-115-1