Efficient Transaction Processing for Hyrise in Mixed Workload Environments

David Schwalb[1], Martin Faust[1], Johannes Wust[1],
Martin Grund[2], Hasso Plattner[1]

[1]Hasso Plattner Institute, Potsdam, Germany
[2]eXascale Infolab, University of Fribourg, Fribourg, Switzerland

Abstract. Hyrise is an in-memory storage engine designed for mixed enterprise workloads that originally started as a research prototype for hybrid table layouts and with basic transaction processing capabilities. This paper presents our incremental improvements and learnings to better support transactional consistency in mixed workloads.
In particular, the paper addresses a multi-version concurrency control mechanism with lock-free commit steps, tree-based multi-column indices, in-memory optimized logging and recovery mechanisms. Additionally, a mixed workload scheduling mechanism is presented, addressing partitionable transactional workloads in combination with analytical queries.

1 Introduction

Currently, we are observing three different trends in the database community. First, traditional general purpose database systems are evolving and incorporate new technologies [19,12,2]. Second, the separation between transactional processing (OLTP) and analytical processing (OLAP) systems continues. Specialized systems leverage the partition-ability of some transactional workloads and completely serialize the execution on partitions to eliminate the overhead of concurrency control [7,23,17]. However, support for cross-partition queries or analytical queries is poor [24]. Third and in contrast to second, we see a unification of both system types, taking on the challenge of executing a mixed workload of transactional and analytical queries in one system [21,20,18,5,16,15,9]. This unification is based on the characteristics of enterprise databases and builds on the set-based processing of typical business applications and the low number of updates allowing an insert-only approach. The unification provides real time insights on the transactional data and eliminates redundancies.

The in-memory storage engine Hyrise targets a unified transactional and analytical system and is designed to support vertical partitioning of tables to allow for the optimal storage layout for mixed enterprise workloads [5]. It builds on a main-delta-concept leveraging light-weight compression techniques like dictionary encoding and bit-packing. It supports an efficient merge process [10] as well as a balanced execution of mixed enterprise workloads [27].

Contribution. In this paper, we provide an overview of implementation aspects of Hyrise and describe optimizations to better support transactional workloads. In particular, we describe (a) a multi-version concurrency control

mechanism with a lock-free commit step in Section 3, (b) a tree-based multi-column index structure in Section 4, (c) a persistency mechanism optimized for in-memory databases and parallel recovery in Section 5 and (d) an optimized scheduling mechanism for the scheduling of mixed workloads while still leveraging the partition-ability of transactional workloads in Section 6.

2 Architecture

Hyrise is an in-memory storage engine[1] specifically targeted to mixed workload scenarios [5] and the balanced execution of both analytical and transactional workloads at the same time [27]. In this section, we describe the basic architecture of the system.

Although Hyrise supports flexible hybrid storage layouts, we assume a columnar storage of tables. The table data consists of attribute vectors and dictionaries for each column in the table as well as three additional columns used for concurrency control. Hyrise uses multi-version concurrency control to manage transactions, providing snapshot isolation as a default isolation level and allowing for higher isolation levels on request, as described in more detail in Section 3. Additionally, the transaction manager handles a transaction context for each running transaction.

Based on analyses of workloads of productive enterprise applications, Hyrise is optimized for read-only queries in order to optimally support the dominant query types based on the set processing nature of business applications [10]. Data modifications follow the insert-only approach and updates are always modeled as new inserts and deletes. Deletes only invalidate rows. We keep the insertion order of tuples and only the most recently inserted version is valid. The insert-only approach in combination with multi-versioning allows Hryise to process writers without stalling readers. Additionally, keeping the history of tables provides the ability of time-travel queries [8] or to keep the full history due to legal requirements [18]. Furthermore, tables are always stored physically as collections of attributes and meta-data and each attribute consists of two partitions: main and delta partition.

The main partition is typically dictionary compressed using an ordered dictionary, replacing values in the tuples with encoded values from the dictionary. In order to minimize the overhead of maintaining the sort order, incoming updates are accumulated in the write-optimized delta partition as described in [22,10]. In contrast to the main partition, data in the write-optimized delta partition is stored using an unsorted dictionary. In addition, a tree-based index with all the unique uncompressed values of the delta partition is maintained per column. The index on top of the dictionary allows for fast value searches on the dictionary and also speeds up value insertions into the column, as inserting a value into a dictionary encoded column requires to search the dictionary [20]. The attribute vectors of both partitions, storing the dictionary encoded values, are further compressed using bit-packing mechanisms [25,3].

[1] Source code available at `https://github.com/hyrise/hyrise`

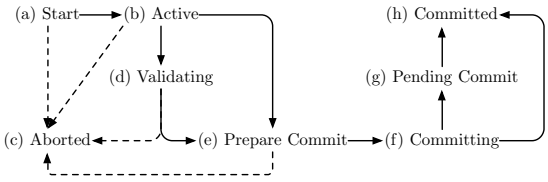


Fig. 1. Internal Hyrise transaction states. Once a transaction entered phase (f) no more logical transaction aborts are possible. Validation phase (d) is optional depending on additional validation steps for serializability.

To ensure a constantly small size of the delta partition, Hyrise executes a periodic merge process. A merge process combines all data from the main partition as well as the delta partition to create a new main partition that then serves as the primary data store [10].

3 Concurrency Control

The choice between an optimistic or pessimistic concurrency control approach highly depends on the expected workload [1,11]. Hyrise uses a multi-version concurrency control (MVCC) mechanism to provide snapshot isolation. This optimistic approach fits well with the targeted mixed workload enterprise environment, as the number of expected conflicts is low and long running analytical queries can run on a consistent snapshot of the database [18]. This section describes our concurrency control implementation that is based on known MVCC mechanisms and focuses on the parallel commit of transactions.

In Hyrise, the transaction manager is responsible for tracking a monotonically increasing next transaction id *ntid* and the last visible commit id *lcid*, as well as maintaining a commit context list *ccl*. Each transaction keeps local information in a *transaction context* containing a local last visible commit id $lcid_T$, its own transaction id tid_T and two lists referencing inserted and deleted rows plus a reference to a commit context in case the transaction is in the commit phase. Each table maintains three additional vectors: a transaction id vector *vtid* used to lock rows for deletion and two commit id vectors *vbeg* and *vend* indicating the validity of rows.

Transactions can be in 8 different phases: (a) transaction start, (b) active processing, (c) transaction aborted, (d) validating, (e) preparing commit, (f) transaction committing, (g) pending commit and (h) transaction committed. Figure 1 shows the actual states and how transactions can change between them.

3.1 Start Transaction Phase

When a new transaction is started, it enters the start phase and is assigned a unique transaction id tid_T by the transaction manager. Additionally, the transaction copies the global last visible transaction id *lcid* to the local transaction

	Own? $vtid = tid_T$	Activated? $vbeg \leq lcid_T$	Invalidated? $vend \leq lcid_T$	Row visible?
Past Delete	yes	yes	yes	no
Past Delete	no	yes	yes	no
Impossible*	yes	no	yes	no
Dirty Own Delete	yes	yes	no	no
Impossible*	no	no	yes	no
Own Insert	yes	no	no	yes
Past Insert/Future delete	no	yes	no	yes
Dirty Insert/Future Insert	no	no	no	no

Table 1. Evaluation rules determining the visibility of rows for a transaction T. Not yet committed inserts and deletes are listed as 'dirty'. *Impossible combination as rows are always activated before they are invalidated.

context as $lcid_T$. After the transaction context is successfully prepared, the transaction enters the active state. During processing and validation, write-write conflicts might occur leaving the transaction in the aborted state. Once a transaction enters the commit phase, the transaction is guaranteed to commit successfully and to reach the committed state.

3.2 Active Processing Phase

During active processing, a transaction T might read or write rows and needs to guarantee the required isolation level. Whenever a set of rows is retrieved from a table through either a table scan operation or an index lookup, the set of rows is validated based on the respective *vbeg*, *vend* and *vtid* values of a row in combination with $lcid_T$ and tid_T. Table 3.2 outlines the different combinations and if T sees them as visible. Some combinations are impossible based on the design of the commit mechanism but listed for completeness. Not yet committed inserts and deletes are listed as dirty. In case transactions need a higher isolation level, serializability can be requested to enforce read stability and phantom avoidance through additional checks before the commit step [11].

Inserts are straight forward, appending a new row to the delta partition of a table with $vtid = tid_T$. As *vbeg* and *vend* are initialized to ∞, the new row is only visible to T and no other transaction can read the in-flight row before T successfully commits. Deletes only invalidate rows by setting *vend*. However, as a transaction does not have a commit id in the active phase, it only deletes the row locally in the transaction context and marking the row by setting *vtid* to tid_T with an atomic compare-and-swap operation. This blocks any subsequent transaction from deleting the same row, resulting in the detection of write- write conflicts. Updates are realized as an insert of the new version with an invalidation of the old version.

Algorithm 3.1: finishCommit(c)

$c.pending \leftarrow True$
while c **and** $c.pending$
do $\begin{cases} \textbf{if } atomic\_cas(lcid, c.cid - 1, c.cid) \begin{cases} send\_response(c) \\ c \leftarrow c.next \end{cases} \\ \textbf{else } \{\textbf{return } (0) \end{cases}$

3.3 Lock-free Commit Phase

Multiple transactions can enter the commit phase in parallel and synchronization is handled by the following lock-free mechanism. Although transactions can process their commit step in parallel, cascading commits realized by using commit dependencies guarantee the correct ordering of the final step of incrementing *lcid*.

Once a transaction T is ready to commit, it enters the prepare commit phase and is assigned a commit context c. Through an atomic insertion of c into the global commit context list *ccl*, a unique commit id cid_T is implicitly assigned to the committing transaction by incrementing the id of the predecessor. Each commit context contains the transaction's commit id cid_T, connection information to send a response to the client and a next pointer to the next commit context in the list. The insertion into *ccl* is performed by executing a compare and swap operation on the next pointer of the last commit context *lcx* to c. Although this mechanism is not wait-free, it provides a lock-free way of creating a linked list of commit contexts with sequentially increasing commit ids. T is guaranteed to proceed to the actual commit phase after successfully inserting c and can not enter the abort state anymore. During the commit phase, T traverses all its changes by iterating through the list of inserted and deleted rows and writing the commit id. Inserted rows are committed by setting *vbeg* to cid_T and all deleted rows are committed by setting *vend* to cid_T.

Finally, T determines if it can directly enter the committed state or if it needs to enter the pending commit state. As multiple transactions can enter the commit phase concurrently, it is possible that transactions T_1 and T_2 commit concurrently and $cid_{T2} > cid_{T1}$. If T_2 enters the committed state first, it would set the global *lcid* to cid_{T2}. However, due to the implemented visibility mechanism through one single last visible commit id, this would allow newly starting transaction to see the in-flight changes of T_1, which is still not fully committed. A pessimistic approach might serialize the commit phases of transactions and avoid this problem. However, if a large number of rows is touched leading to longer commit phases, this quickly turns into a bottleneck. Therefore, Hyrise supports parallel commits that allow transactions to commit in any order except for the last step of incrementing the global *lcid*. Instead, commit dependencies take care of incrementing *lcid* at the correct point in time and only then the respective transactions are returned as committed to the client. This allows parallel and

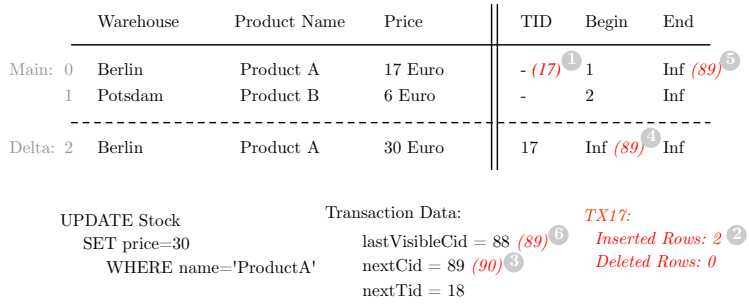


Fig. 2. Example outlining the implemented multi-version concurrency algorithm.

lock-free commit phases and although the final commit step might be deferred, worker threads are already freed and can process other queries.

Algorithm 3.1 outlines the process of the final commit step that allows workers to finish processing of a transaction by adding a commit dependency although the final last step of incrementing *lcid* might not yet be possible. First, a committing transaction T_1 with commit context c sets its commit context status to pending and indicates that it is trying to increment the *lcid*. Then, T_1 tries to atomically increment the *lcid*. In case the increment failed, T_1 depends on another currently committing transaction T_2 with $cid_{T2} < cid_{T1}$ to commit T_1. The processing worker thread is then freed and can process new queries. The atomic incrementation of *lcid* ensures that only one thread succeeds even if multiple threads are concurrently incrementing *lcid*. When T_2 finally commits, it checks if pending transactions exist by following the list of commit contexts. As long as there are pending commits, T_2 proceeds and increments the *lcid*.

The fact that the *lcid* is only updated after all commit ids for *vbeg* and *vend* have been written, ensures that all changes during the commit phase appear to other transactions as future operations, leaving the affected records untouched from the viewpoint of other transactions. Until the global *lcid* is set to cid_T of a committing transaction and makes all changes visible for subsequent transactions in one atomic step.

3.4 Aborts

Transactions can only abort before they enter the commit phase. Therefore, aborting transactions do not yet have an assigned commit id and have only inserted new rows which are still invisible or have marked a row locally for deletion. This means that an aborting transaction only has to clear potentially locked rows by removing their id from *vtid* using the lists of inserted and deleted rows from the transaction context.

3.5 Example

Figure 2 shows an example of an update query with $tid_T = 17$ setting the price of a product A from 17 to 30. The image shows a logical view of a table separated

into main and delta partitions. (1) Row 0 is locked by setting $vtid = 17$, (2) the new version of the row is inserted into the delta and added to the local list of inserted rows, (3) the commit phase starts, assigning T the commit id $cid_T = 89$, (4) *vbeg* of the newly inserted row is set to 89 though it is still invisible to other running transactions as the *lcid* is still 88, (5) the old row gets invalidated by setting $vend = 89$ and added to the local list of deleted rows, (6) the *lcid* gets incremented making all changes visible to other transactions.

4 Index Structures

Hyrise allows the definition of indices to efficiently support transactional queries which select only a few tuples. Index data structures are maintained separately for the main and delta partition of columns to account for their different characteristics. The following describes a read-only Group-Key Index for the main partition of a single column [4], a tree-based index structure for the delta partition and index structures on multiple columns.

4.1 Single Column Indices

A single-column index on the main partition leverages the read-only nature of the main partition to reduce the storage footprint by creating an immutable structure for the mapping of values to positions during the merge process. The main index consists of two bit-packed vectors that map dictionary entries to position lists.

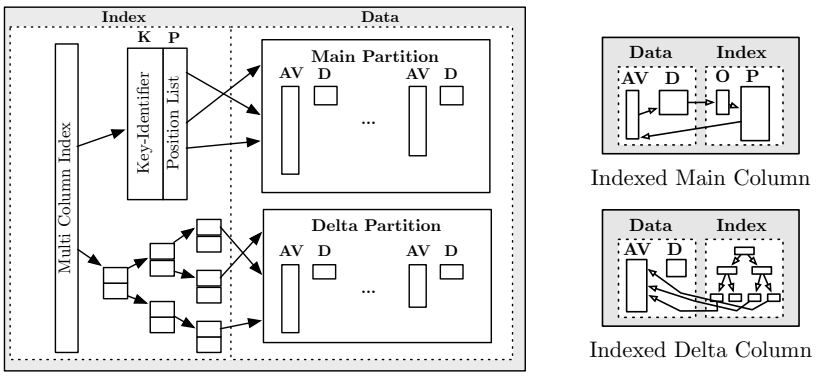
It consists of an offset vector O and a position vector P. O is parallel to the dictionary D of a column and contains the start of the list of values in P for each value in D, in other words the offset which is used to jump into P. P is parallel to the attribute vector AV and contains all row positions sorted by their value. Thereby all rows for a distinct value can be retrieved with only two direct reads at the respective position in the two vectors O and P.

In contrast to the main index, the delta index needs to efficiently handle newly inserted values and is implemented as a multi-map of actual values and positions using a tree-based data structure. Entries are kept in ascending order so that the list of positions for a single value is always sorted. Figure 3(b) shows a schematic overview of the used index structures.

4.2 Multi Column Indices

Hyrise supports the indexing of multiple columns through the usage of *Composite Group-Keys* on the main partition and tuple-indexing on the delta partition. The challenge for our column-oriented in-memory store is to efficiently obtain a unique identifier from the composite key, as the parts of the key are encoded and not co-located. In Hyrise, Composite Group-Key Indices store the concatenation of a key's value- ids in a key-identifier list K, as shown in Figure 3(a).

This leads to an additional dictionary lookup for each part of the key before the index lookup, since all values of the predicate have to be transformed into



(a) Multiple Columns (b) Single Column

Fig. 3. Overview of index data structures on single columns and multiple columns for main and delta partitions. Main: Value lookup in dictionary D, jumping from offset vector O into positions vector P, referencing values in attribute vector AV. Delta: Tree-based index on values referencing attribute vector.

value-ids prior to a binary search on K. The offset of the found key-identifier can be used to directly obtain the row-id from the position list P. In the delta partition, where the storage footprint is not as important as in the main partition, we concatenate the actual values in the index. We use transformations similar to Leis et al. [13] to obtain binary-comparable keys.

Internally, Hyrise uses different strongly-typed data types. To allow the flexible definition and querying of multi-column indices at runtime, the system provides key-builder objects that accept any internal data type. Multiple calls to a key builder object can be executed with different data types, which allows to conveniently and efficiently support composite keys with mixed data types.

Indices are unaware of the visibility of records. Hence, the delta index is used in an append-only manner and retrieved records need to be validated using the defined visibility mechanism. In case of primary key lookups, the index is traversed backwards to find the first valid version of a key. While this increases the lookup overhead moderately, it allows to maintain the visibility information at one single location and to have transaction-agnostic index structures.

5 Persistency: Logging, Recovery and Checkpointing

Although in-memory databases keep their primary copy of the data in main memory, they still require logging mechanisms to achieve durability. The persistency mechanisms applied in Hyrise differ from traditional disk-based mechanisms due to the lack of a paging mechanism and the used multi-version concurrency control. In this section, we describe the implemented logging, checkpointing and recovery mechanisms.

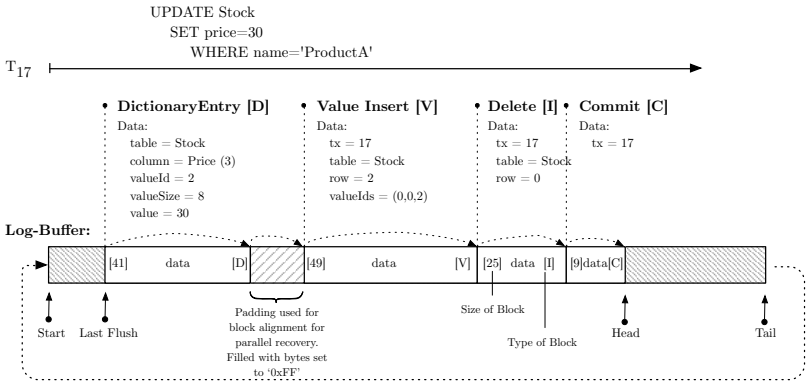


Fig. 4. Delta Log Format. Last flush marks entries already flushed to disk. Padding is used to align log-file for parallel recovery, log entries are fixed sized based on their type, only dictionary entries have a variable length.

The main partition of a table is always stored as a binary dump on disk after a merge process. Therefore, only changes to the delta are written to a log file, which uses group commits to hide the latency of disks or SSDs. Checkpoints create a consistent snapshot of the database by also dumping the delta partitions as binary dumps. In a recovery case, existing dumps for main and delta are restored from a checkpoint and an eventually existing delta log is replayed to restore the latest consistent state of tables. Binary dumps are a snapshot of a table persisted onto disk in the form of binary files directly storing the respective data structures. Separate files for the table meta-data containing the number and name of columns, attribute vectors, dictionaries and indexes are created. Using this information, the system is able to recreate the complete table by loading the respective files.

5.1 Delta Log

In contrast to ARIES style logging techniques [14], logging in Hyrise leverages the applied dictionary compression [26] and only writes redo information to the log. This to reduce the overall log size by writing dictionary-compressed values and parallel recovery as log entries can be replayed in any order.

The actual log entries that are written to the log-file are of the following 8 types: (1) dictionary entries indicate a newly inserted value with its value id, (2) value entries indicate a newly inserted row in a table, (3) invalidations invalidate an existing row, (4) commit entries indicate a successfully committed transaction, (5) rollback entries indicate that a transaction performed a rollback and aborted, (6) skip entries are padding entries used for alignment, (7) checkpoint start entries indicate the start of a checkpoint, (8) checkpoint end entries indicate the end of a checkpoint. Dictionary Entries do not include the inserting transaction's TID, as this information is irrelevant to the recovery process. Even if a transaction that inserted a value into the dictionary needs to be rolled

back, the value can stay in the dictionary without compromising functionality. If a log entry is to be written and its size would overlap into another block, the remaining space is filled with a Skip Entry and the log entry is written to the beginning of the next block in order to align the log-file to a specified block-size. Thereby, it is guaranteed that log entries to not span across block boundaries which allows easy parallel recovery as thread can start reading the log entries at block boundaries. Skip entries consist only out of bytes set to $0xFF$ and introduce only a minimal overhead as block sizes for the alignment are typically in the range of multiple megabytes.

Figure 4 outlines the used format for writing the log file. New log entries are buffered in a ring-buffer before they are flushed to the log-file. Similarly to recent work, buffer fill operations are only synchronized while acquiring buffer regions and threads can fill their regions in parallel [6]. Each entry in the buffer starts with a character specifying the size of the entry, followed by its data and closed by the type of the entry. This design allows to forward iterate through the list of entries by skipping the respective sizes of entries and to read the log entries backwards in case of recovery by processing each entry based on its type. Entries do have a fixed length based on their type, except variable length dictionary entries which contain a dedicated value length in the log entry.

5.2 Checkpointing

Checkpoints create a consistent snapshot of the database as a binary dump on disk in order to speed up recovery. They are periodically initiated by a checkpoint daemon running in the background. In a recovery case, only the binary dumps from the last checkpoint need to be loaded and only the part starting at the last checkpoint time from the delta log needs to be replayed. In contrast to disk based database systems where a buffer manager only needs to flush all dirty pages in order to create a snapshot, Hyrise needs to persist the complete delta partition of all tables including *vbeg* and *vend*.

A checkpoint is created in three steps: (1) prepare checkpoint, (2) write checkpoint and (3) finish checkpoint. In the first step, the checkpoint is assigned a unique id and the global log file is switched from the current file A to a new empty log file B, redirecting all subsequent log entries into the new file. The first entry in the new log file is the checkpoint start entry. Additionally, the necessary folder structure is created with a file indicating that the checkpoint is in progress. The transaction manager then waits for all currently running transactions to finish before the checkpoint enters the next phase. This guarantees that log file B contains all relevant information to roll forward to the latest consistent state during recovery. This mechanism adds a delay to the checkpoint process, but does not block any transactions from executing. In the second phase, the actual checkpoint is written and all delta tables are written in a binary format to disk, including eventually existing index structures. Additionally, the *vbeg* and *vend* of all tables are persisted to disk, as the delta potentially contains updated versions of rows from the main. In the third and final checkpoint phase, a checkpoint end entry is written to the log and a file is created indicating that the checkpoint as finished successfully. This makes the checkpoint the latest available checkpoint in

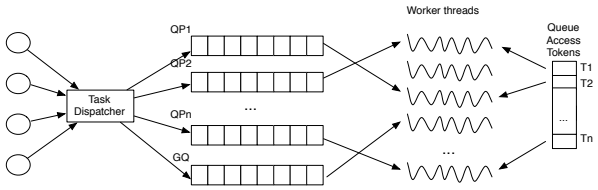


Fig. 5. Task queues for partitionable transactions

the system so that it will be used in case of a recovery. Due to the applied insert-only approach, the checkpoint mechanism can operate concurrently as writing transactions are executed.

5.3 Recovery Process

The recovery process is executed in two steps: (1) load checkpoint and (2) replay delta log. The checkpoint contains binary dumps of the main and delta partition and is loaded in parallel. The delta replay step can be easily distributed across multiple threads based on the layout of the log-file in blocks.

Each thread reads its assigned blocks from the back and replays all successfully committed transactions which are identified by a commit entry as their first log entry. This can be executed in parallel without any synchronization as the log entry replay is independent of the replay order. The only requirement is some upfront meta-data about table sizes, dictionary sizes and transaction numbers in order to preallocate the data structures. In case a thread does not read a commit entry for one transaction, it needs to make sure that no other thread has processed the respective commit entry before ultimately discarding the changes of this transaction. This synchronization between threads is handled by setting a field in a global bit- vector based on the transaction id for each processed commit entry and parking all log entries that are not preceded by a commit entry for later evaluation. After the processing of all blocks, the threads are synchronized through a barrier and reevaluate all discarded transactions by checking if another thread read a commit entry by looking up the transaction id in the bit-vector and replaying the changes if necessary.

Both steps are reasonably optimized and implemented distributing the work across all available cores to fully leverage the available parallelism and bandwidth on modern systems to provide the fastest possible delta log replay.

6 Scheduling

To execute mixed database workloads, Hyrise leverages a task-based query execution model. The main advantages of this execution model are (1) almost perfect load balancing on multi-core CPUs, (2) efficient workload management based on a non-preemptive priority task scheduling policy.

The general idea of this execution model is to partition a query into smaller, non-preemptive units of work, so called tasks, and map these tasks dynamically to a pool of worker threads by a user-level scheduler. Short running OLTP queries are executed as a single task, complex OLAP style queries are transformed into a graph of fine granular tasks by applying data parallelism on operator level. The granularity of tasks is controlled by a system parameter for the maximum task size. Each partitionable operator is split dynamically at runtime into tasks, based on the size of the input data and the maximum task size [27].

The task-based execution model achieves almost perfect load balancing, as the actual degree of parallelism for executing complex queries can vary dynamically throughout execution depending on the current workload. Assuming a complex query is executed as the only query on a multi-core machine, it can leverage all worker threads for execution. Once another query enters the system, tasks of both queries are distributed over the available worker threads taking query priorities or predefined resource shares into account [29,28].

To optimize scheduling for transactional throughput, we extend the task-based execution model by introducing specific queues for partitionable transactions. Note that we still apply the concurrency control mechanism described in Section 3 to enable transaction safe read access for analytical queries based on snapshot isolation. Figure 5 gives an overview of the concept of transaction specific queues. Queries that modify data of a particular data partition n are placed in one of the corresponding queues shown as QPn in Figure 5. Analytical queries are placed in the general queue GQ. Each worker thread tries to pull tasks from the partitionable queues with priority and only takes tasks from the general queue, if no tasks from transactional query is available. Tasks of one partition are serialized through a token mechanism to ensure that only one transactional query per partition is executed at a time. This mechanism avoids the execution of multiple tasks of one partition and therefore eliminates possible write conflicts.

7 Conclusion

In this paper, we presented implementation specific design choices for the in-memory storage engine Hyrise to optimize transaction processing in a mixed enterprise workload setting. We outlined the main architectural design choices and addressed the following parts in particular: (1) a multi-version concurrency control mechanism with lock-free commit steps, (2) tree-based multi-column indices, (3) in-memory optimized logging and recovery mechanisms and (4) a mixed workload scheduling mechanism addressing partition-able transactional workloads in combination with analytical queries.

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

2. C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. *SIGMOD*, 2013.
3. M. Faust, M. Grund, T. Berning, D. Schwalb, and H. Plattner. Vertical Bit-Packing: Optimizing Operations on Bit-Packed Vectors Leveraging SIMD Instructions. *BDMA in conjunction with DASFAA,*, 2014.
4. M. Faust, D. Schwalb, J. Krueger, and H. Plattner. Fast lookups for in-memory column stores: group-key indices, lookup and maintenance. *ADMS in Conjunction with VLDB*, 2012.
5. M. Grund, J. Krueger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE—A Main Memory Hybrid Storage Engine. *VLDB*, 2010.
6. R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *VLDB*, 2010.
7. R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, and Y. Zhang. H-store: a high-performance, distributed main memory transaction processing system. *VLDB*, 2008.
8. M. Kaufmann, P. Vagenas, P. M. Fischer, D. Kossmann, and F. Färber. Comprehensive and interactive temporal query processing with SAP HANA. *VLDB*, 2013.
9. A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *ICDE*, 2011.
10. J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *VLDB*, 2011.
11. P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 2011.
12. P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL server column store indexes. *SIGMOD*, 2011.
13. V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. *ICDE*, 2013.
14. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 1998.
15. H. Mühe, A. Kemper, and T. Neumann. Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems. *CIDR*, 2013.
16. T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics. *BTW*, 2013.
17. I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *VLDB*, 2010.
18. H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. *SIGMOD*, 2009.
19. V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and

L. Zhang. DB2 with BLU acceleration: so much more than just a column store. *VLDB*, 2013.
20. D. Schwalb, M. Faust, J. Krueger, and H. Plattner. Physical Column Organization in In-Memory Column Stores. *DASFAA*, 2013.
21. V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient Transaction Processing in SAP HANA Database - The End of a Column Store Myth. *SIGMOD*, 2012.
22. M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, and E. O'Neil. C-store: A Column-oriented DBMS. *VLDB*, 2005.
23. M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2), 2013.
24. S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, 2013.
25. 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. *VLDB*, 2009.
26. J. Wust, J.-H. Boese, F. Renkes, S. Blessing, J. Krueger, and H. Plattner. Efficient logging for enterprise workloads on column-oriented in-memory databases. In *CIKM*, 2012.
27. J. Wust, M. Grund, K. Hoewelmeyer, and D. Schwalb. Concurrent Execution of Mixed Enterprise Workloads on In-Memory Databases. *DASFAA*, 2014.
28. J. Wust, M. Grund, and H. Plattner. Dynamic query prioritization for in-memory databases. *IMDM in conjunction with VLDB*, 2013.
29. J. Wust, M. Grund, and H. Plattner. Tamex: a task-based query execution framework for mixed enterprise workloads on in-memory databases. In *GI-Jahrestagung*, 2013.