High-Performance Transaction Processing in SAP HANA

Juchang Lee[1], Michael Muehle[1], Norman May[1], Franz Faerber[1], Vishal Sikka[1],
Hasso Plattner[2], Jens Krueger[2], Martin Grund[3]
[1]SAP AG
[2]Hasso Plattner Insitute, Potsdam, Germany,
[3]eXascale Infolab, University of Fribourg, Switzerland

Abstract

Modern enterprise applications are currently undergoing a complete paradigm shift away from traditional transactional processing to combined analytical and transactional processing. This challenge of combining two opposing query types in a single database management system results in additional requirements for transaction management as well. In this paper, we discuss our approach to achieve high throughput for transactional query processing while allowing concurrent analytical queries. We present our approach to distributed snapshot isolation and optimized two-phase commit protocols.

1 Introduction

An efficient and holistic data management infrastructure is one of the key requirements for making the right decisions at an operational, tactical, and strategic level and core to support all kinds of enterprise applications[12]. In contrast to traditional architectures of database systems, the SAP HANA database takes a different approach to provide support for a wide range of data management tasks. The system is organized in a main-memory centric fashion to reflect the shift within the memory hierarchy[2] and to consistently provide high performance without prohibitively slow disk interactions. Completely transparent for the application, data is organized along its life cycle either in column or row format, providing the best performance for different workload characteristics[11, 1]. Transactional workloads with a high update rate and point queries can be routed against a row store; analytical workloads with range scans over large datasets are supported by column oriented data structures. However, the DBA is free to choose an appropriate layout and to apply automated partitioning strategies as presented in [2]. In addition to a high scan performance over columns, the column-oriented representation offers an extremely high potential for compression making it possible to store even large datasets within the main memory of the database server. However, the main challenge is to support the new kind of mixed workload that require a rethinking of the storage layer for modern database systems. In such a mixed workload for enterprise-scale applications the aggregated size of all tables will no longer fit on a single node, emphasizing the need for optimized distributed query processing. As a consequence, we have to enhance the traditional transactional query processing with workload specific information to achieve the best possible throughput. The overall goal of SAP HANA as a distributed in-memory mixed workload database is to exploit transactional locality as often as possible. In this paper, we present insights into the distributed transaction and session management in SAP

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

HANA that allow to combine transactional and analytical query processing inside a single DBMS. Therefore we will discuss our approach to distributed snapshot isolation and our optimizations to our in-memory optimized two-phase commit protocol.

2 Distributed Transaction Management

In contrast to other scale-out solutions like Hadoop, SAP HANA follows the traditional semantics of databases by providing full ACID support. In order to deliver on the promise of supporting both OLTP and OLAP-style query processing within a single platform, the SAP HANA database relaxes neither any consistency constraints nor any degree of atomicity or durability. To achieve this the SAP HANA database applies reduced locking with MVCC and logging schemes for distributed scenarios with some very specific optimizations like optimizing the two-phase commit protocol or providing sophisticated mechanisms for session management. Based on the deployment scheme for the given use case the framework provided by SAP HANA allows to minimize the cross-node cost for transaction processing in the first place. This allows to use for example one big server for transaction processing of the hot tables and use a multitude of smaller servers for analytical processing on the rest of the data partitions. In general, SAP HANA relies on Multi-Version-Concurrency-Control (MVCC) as the underlying concurrency control mechanism, the system provides distributed snapshot isolation and distributed locking to synchronize multiple writers. Therefore, the system implements on a distributed locking scheme with a global deadlock detection mechanism avoiding a centralized lock server as a potential single point of failure. Based on this general description we will now explain our optimizations for handling distributed snapshot isolation and the optimized two-phase commit protocol for distributed transactions.

2.1 General Concurrency Control Mechanism

Isolation of concurrent transactions is enforced by a central transaction manager maintaining information about all write transactions and the *consistent view manager* deciding on visibility of records per table. A so-called *transaction token* is generated by the transaction manager for each transaction and encodes what transactions are open and committed at the point in time the transaction starts. The transaction token contains all information needed to construct the consistent view for a transaction or a statement. It is passed as an additional context information to all operations and engines that are involved in the execution of a statement.

For token generation, the transaction manager keeps track of the following information for all write transaction: (i) unique transaction IDs (TID), (ii) the state of each transaction, i.e., *open*, *aborted*, or *committed*, and (iii) once the transaction is committed, a commit ID (CID) (see Figure 1). While CIDs define the global commit order, TIDs identify a single transaction globally without any order. However, there are reasons to avoid storing the TID and CID per modified record. In the compressed column store, for example, the write sets of the transactions are directly stored in the modifiable delta buffer of the table. To achieve fine granular transaction control and high performance it becomes necessary to translate the information associated with the global CID into an representation that only uses TIDs, to avoid revisiting all modified records during the CID association at the end of the commit. The data consolidation is achieved using the transaction token. It contains the global reference to the last committed transaction at this point in time and the necessary information to translate this state into a set of predicates based on TIDs. The transaction token contains the following fields: `maxCID`, `minWriteTID`, `closedTIDs`, and the `TID`. Now, we will explain how the state information of the `maxCID` is translated into a set of `TID` filter predicates. The `minWriteTID` attribute of the token identifies *all* records that are visible to *all* transactions that are currently running hereby exploiting the assumption that the number of concurrently running transactions is limited and the order of the TIDs is similar to the order of the CIDs. This means that for all transactions T_j all records R with $TID_R < minWriteTID$ are visible. Since there can be committed transactions between `minWriteTID` and `TID` the attribute `closedTIDs`

identifies all subsequently closed transactions. Thus, the for the transaction T_j all records R are visible where $TID_R < minWriteTID \vee TID_R \in closedTIDs \vee TID_R = TID_{T_j}$. This expression identifies all TIDs that match a certain transactional state.

Rows inserted into the differential store by an open transaction are not immediately visible to any concurrent transaction. New and invalidated rows are announced to the consistent view manager as soon as the transaction commits. The consistent view manager keeps track of all added and invalidated rows to determine the visibility of records for a transaction. Using the TID expression from above and these two lists, all visible records are identified. Furthermore it is important to mention that for the column-store tables in HANA the write sets are not stored separately but directly in the differential buffer. If transactions are aborted this may result in a little overhead, but only until the merge process[8] recompresses the complete table and discards all invalid records.

To generalize, for every transaction T_j, the consistent view manager maintains two lists: one list with the rows added by T_j and a second list of row IDs invalidated by T_j. For a transaction T_k, we apply the previously mentioned predicate expression on this list of added and invalidated records. For a compact representation of change information, the consistent view manager consolidates the added row list and invalidated row lists of all transactions with a TID smaller than the *maxWriteTID*. This allows a fast consolidation of the first part of the predicate. For all other TIDs individual change information is kept requiring to scan the list of added or invalidated records.

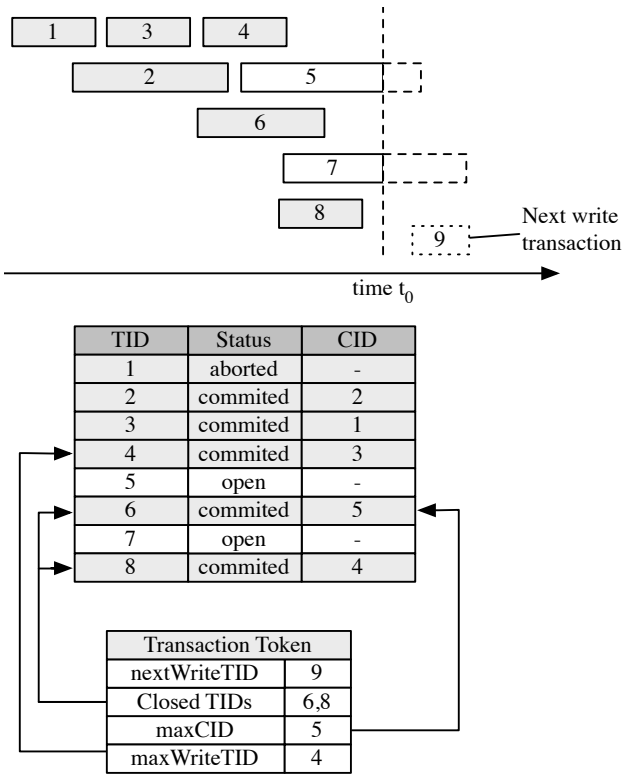

Figure 1: Data Consolidation with Transaction Token

2.2 Distributed Snapshot Isolation

For distributed snapshot isolation reducing the overhead of synchronizing transaction ID or commit timestamp across multiple servers belonging to a same transaction domain has been a challenging problem [10, 4, 5]. Therefore, the primary goal of all our optimizations is to increase the performance of short-running local-only transactions. The motivation for this case is that typically only long-running analytical queries will require data access to multiple nodes, where short-running queries will be mostly single-node queries. This observation is close in spirit to fixed partitioning approaches recently presented in [6, 7]. To avoid costly short-running multi-node queries the DBA is supported with partitioning tools for automatic partitioning. For long-running analytical queries spanning across multiple nodes, the incurred overhead incurred for communicating this transactional information will be comparable small in comparison to the overall execution time of the query.

In isolation mode Read-Committed the boundary of the lifetime of the MVCC snapshot is the query itself. Thus, the isolation requirements can be determined at compile time to identify which partitions of the tables are accessed. The entire transaction context that is needed for a query to execute is cached on a local node. For the *Repeatable Read* or *Serializable* isolation level, the transaction token can be reused for the queries belonging to the same transaction, which means that its communication cost with the coordinator node is less important. Whenever a transaction (in a transaction-level snapshot isolation mode) or a statement (in a statement-level

snapshot isolation mode) starts, it copies the current transaction token into its context (called snapshot token) for visibility detection.

In a distributed environment, the transaction protocol defines that for every started transaction a worker node should access the transaction coordinator to retrieve its snapshot transaction token. This could cause (1) a throughput bottleneck at the transaction coordinator and (2) additional network delay to the worker-side local transactions. To avoid these situations we use three techniques: First, enabling local read-only transactions to run without accessing the global coordinator, second, enabling local read or write transactions to run without accessing the global coordinator, and third, using *Write-TID-Buffering* to enable multi-node write transactions to run with only a single access to the global coordinator.

Optimization for worker-node local read transactions or statements In general, every update transaction accesses the transaction coordinator to access and update the global transaction token. However, read-only statements can reuse the cached local transaction token ensuring a consistent view on the data, as it contains an upper limit for visible records. This cached local transaction token is refreshed in two cases: (i) by the transaction token of an update transaction when it commits on the node, or (ii) by the transaction token of a *global statement* when it comes in to (or started at) the node. If the statement did not need to access any other node, it can just finish with the cached transaction token, i.e., without any access to the transaction coordinator. If it detects that the statement should also be executed in another node, however, it is promoted to a *global statement* type, after which the current statement is retried with the global transaction token obtained from the coordinator. Single-node read-only statements/transactions do not need to access the transaction coordinator at all, since they reuse the locally-cached transaction token. This is significant to avoid the performance bottleneck at the coordinator and to reduce the performance overhead of single-node statements (or transactions).

Optimization for worker-side local write transactions Each node manages its own local transaction token independent of the global transaction token. In case of a single-node update transaction the transaction token of the node is only refreshed locally. In contrast to traditional approaches, each database record in the delta partition has two TID (or Transaction ID) columns for MVCC version filtering: one for global TID and another for local TID. In case of a local transaction the local TID is updated, in case of a global transaction, it reads either global or local TID (reads a global TID if there is a value in its global TID column; otherwise, reads a local TID) and updates both global and local TIDs. As a consequence, the global transactions carry two snapshot transaction tokens: one for global transaction token and second one for the current worker node's local transaction token. In the log record both global and local TIDs are also recorded if it is a global transaction. On recovery a local transaction's commit can be decided by its own local commit log record. Again, only global transactions are required to check with the global coordinator.

Optimization for multi-node write transactions A multi-node write transaction needs a global write TID as all multi-node transactions do. In a HANA scale-out system one of the server nodes becomes the transaction coordinator which manages the distributed transaction and coordinates two phase commit. Executing a distributed transaction involves multiple network communications between the coordinator node and worker nodes. Each write transaction is assigned a globally unique write transaction ID. A worker-node-first transaction, one that starts at a worker node first, should be assigned a TID from the coordinator node as well, which causes an additional network communication. Such a communication might significantly affect the performance of distributed transaction execution. This extra network communication is eliminated to improve the worker-node-first write transaction performance. We solve this problem by buffering such global write TIDs in a worker node. Buffering TIDs will not incur ordering conflicts as the TID is only used as a unique transaction identifier and has no explicit ordering criterion. When a request for a TID assignment is made the very first time, the coordinator node returns a range of TIDs which gets buffered in the worker node. The next transaction which needs a TID gets it from

the local buffer thus eliminating the extra network communication with the coordinator node. For short-running transactions this can increase the throughput by $\approx 30\%$. The key performance parameters are the size of the range and the time-frame when unused TIDs are discarded. These parameters can be calibrated and adjusted at runtime to achieve the best throughput. Discarded TIDs are not reused to avoid unnecessary communications with the coordinator for consolidation of TIDs. It is important to mention that the above described techniques allow improving the overall transactional performance without losing or mitigating transactional consistency.

2.3 Optimizing Two-Phase Commit Protocol

Two-phase commit (2PC) protocol is widely used to ensure atomicity of distributed multi-node update transactions. In HANA, we use a series of optimization techniques that attempt to reduce the network and log I/O delays during a two-phase commit to increase throughput.

Early commit acknowledgment after the first commit phase Our first optimization is to return the commit acknowledgment early after the first commit phase[9, 3] as shown in Figure 2. Right after the first commit phase and the commit log is written to the disk, the commit acknowledgment can be returned to the client. And then, the second commit phase can be done asynchronously.

For this optimization, we consider three main points: Writing the commit log entries on the worker nodes can be done asynchronously. During crash recovery, some committed transactions will be classified as in-doubt transactions, which are resolved as committed by checking the transaction's status in the coordinator. If transaction tokens are cached asynchronously on the worker nodes, the data is visible by a transaction but not by the next (local-only) transaction in the same session. This situation can be detected by storing the last transaction token information for each session at the client side. And then, until the second commit phase of the previous transaction is done, the next query can be stalled. If transactional locks are released after sending a commit acknowledgment to the client, a "false lock conflict" may arise by the next transaction in the same session. This situation can however be detected and resolved by the worker and coordinator. If this is detected, the transaction waits for a short time period until the commit notification arrives to the worker node.

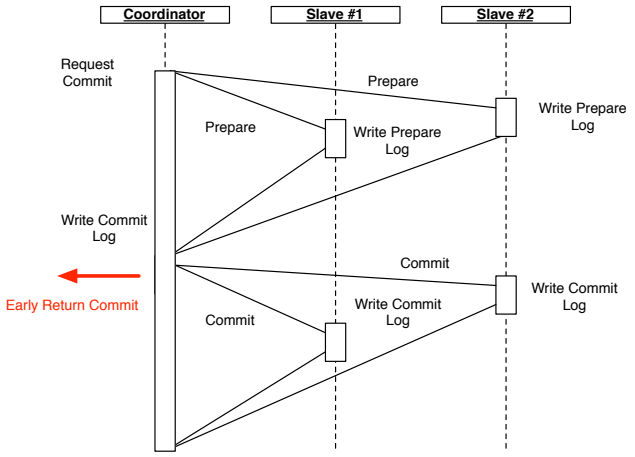

Figure 2: Returning commit ack early after first commit phase

Skipping writes of prepare logs The second optimization is to eliminate log I/Os for writing prepare-commit log entries. In a typical 2-phase-commit the prepare-commit log entry is used to ensure that the transaction's previous update logs are written to disk and to identify in-doubt transactions at the recovery time. Writing the transaction's previous update logs to disk can be ensured by only comparing the transaction-last-LSN (log sequence number) with the log-disk-last-LSN. If the log-disk-last-LSN is larger or equal than the transaction-last-LSN, it means that the transaction's update logs are already flushed to disk. If we do not write the prepare-commit log entry, we handle all the uncommitted transactions at recovery time as in-doubt transactions. Their commit status can be decided by checking with the transaction coordinator. The goal is to trade transactional throughput with recovery time, as the size of in-doubt transaction list can increase, but reduces the run-time overhead for log I/Os.

Group two-phase commit protocol This is a similar idea to the one described earlier in this Section, but instead of sending commit requests to the coordinator node individually (i.e., one for each write transaction), we can group multiple concurrent commit requests into one and send it to the coordinator node in one shot. Also, when the coordinator node multicasts a "prepare-commit" request to multiple-related worker nodes of a transaction, we can group multiple "prepare-commit" requests of multiple concurrent transactions which will go to the same worker node. By this optimization we can increase throughput of concurrent transactions. Two-phase commit itself cannot be avoided fundamentally for ensuring global atomicity to multi-node write transactions. However, by combination of the optimizations described in this subsection, we can reduce its overhead significantly.

3 Summary

In this paper, we presented details about the transaction processing system in SAP HANA. The main challenge for our system is to be able to support two opposing workloads in a single database system. Therefore, we carefully optimized our system to be able to separate transactional from analytical queries and to guarantee the best possible throughput for transactional queries.

References

[1] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[2] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[3] Ramesh Gupta, Jayant R. Haritsa, and Krithi Ramamritham. Revisiting Commit Processing in Distributed Database Systems. In *SIGMOD Conference*, pages 486–497. ACM Press, 1997.

[4] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Scalable Versioning in Distributed Databases with Commuting Updates. In *ICDE*, pages 520–531. IEEE Computer Society, 1997.

[5] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Asynchronous Version Advancement in a Distributed Three-Version Database. In *ICDE*, pages 424–435. IEEE Computer Society, 1998.

[6] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD Conference*, pages 603–614. ACM, 2010.

[7] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[8] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[9] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[10] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *SIGMOD Conference*, pages 124–133. ACM Press, 1992.

[11] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD Conference*, pages 1–2. ACM, 2009.

[12] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD Conference*, pages 731–742. ACM, 2012.