The VLDB Journal (2018) 27:421–444
https://doi.org/10.1007/s00778-018-0503-z

REGULAR PAPER

Parallel replication across formats for scaling out mixed OLTP/OLAP workloads in main-memory databases

Juchang Lee[1,2] · Wook-Shin Han[3] · Hyoung Jun Na[1] · Chang Gyoo Park[1] · Kyu Hwan Kim[1] · Deok Hoe Kim[1] · Joo Yeon Lee[1] · Sang Kyun Cha[2] · SeungHyun Moon[3]

Received: 12 September 2017 / Revised: 31 January 2018 / Accepted: 23 March 2018 / Published online: 16 April 2018
© Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract
Modern in-memory database systems are facing the need of efficiently supporting mixed workloads of OLTP and OLAP. A conventional approach to this requirement is to rely on ETL-style, application-driven data replication between two very different OLTP and OLAP systems, sacrificing real-time reporting on operational data. An alternative approach is to run OLTP and OLAP workloads in a single machine, which eventually limits the maximum scalability. In order to tackle this challenging problem, we propose a novel database replication architecture called HANA Asynchronous Parallel Table Replication (ATR). ATR supports OLTP workloads in one primary machine, while it supports heavy OLAP workloads in replicas. Here, row store formats can be used for OLTP transactions at the primary, while column store formats are used for OLAP analytical queries at the replicas. ATR is designed to support elastic scalability of OLAP query performance, while it minimizes the overhead for transaction processing at the primary and minimizes CPU consumption for replayed transactions at the replicas. ATR employs a novel optimistic lock-free parallel log replay scheme which exploits characteristics of multi-version concurrency control (MVCC) to enable real-time reporting by minimizing the propagation delay between the primary and replicas. It supports adaptive query routing depending on its predefined acceptable staleness range. Through extensive experiments with a concrete implementation available in a commercial product, we demonstrate that ATR achieves sub-second visibility delay even for update-intensive workloads, providing scalable OLAP performance without notable overhead to the primary. In addition, with extension of ATR to eager parallel replication, we demonstrate how the parallel log replay and its log-less replica recovery mechanisms improve run-time transaction performance under eager replication.

Keywords Database replication · In-memory database · Scaling out · SAP HANA

1 Introduction

Modern database systems need to support mixed workloads of online transaction processing (OLTP) and online analytical processing (OLAP) workloads [18,36,37]. OLTP workloads contain short-lived, light transactions which read or update small portions of data, while OLAP workloads contain long-running, heavy transactions which reads large portions of data. That is, transactional and analytical behaviors are mixed in today's workloads. Note that row store formats are typically used for handling OLTP workloads, while column store formats are typically used for handling OLAP workloads.

A conventional approach to support such mixed workloads is to isolate OLTP and OLAP workloads into separate, specialized database systems, periodically replicating operational data into a data warehouse for analytics. Here, we can rely on an external database tool, such as extraction–transformation–loading (ETL) [41,42]. However, this ETL-style, application-driven data replication between two different OLTP and OLAP systems is inherently unable to achieve real-time reporting. Note that we may run OLTP and OLAP workloads in a single machine. However, this approach requires an extremely expensive hardware. Previous work such as Hyper [18,37] focuses on scaling up mixed workloads in a single hardware host, which eventually limits the maximum scalability of analytical query processing.

✉ Juchang Lee
juc.lee@sap.com

1 SAP Labs Korea, Seoul, Korea

2 Seoul National University, Seoul, Korea

3 Pohang University of Science and Technology, Pohang, Korea

From analysis of our various customer workloads, we notice that one modern server machine can sufficiently handle OLTP workloads, while heavy OLAP workloads need to be processed in different machines. This architecture can be realized through database replication. In this situation, we need to support (1) real-time and (2) scalable reporting on operational data. In order to support real-time reporting, we need to minimize the propagation delay between OLTP transactions and reporting OLAP queries. In order to support scalable reporting, query processing throughput should be able to increase accordingly with the increasing number of replicas, elastically depending on the volume of the incoming workloads.

Data replication is a widely studied and popular mechanism for achieving higher availability and higher performance [2–6,9,10,13,16,17,33,35]. However, to the best of our knowledge, there is little work on replication from row store to column store for enhancing scalability of analytical query processing. Middleware-based replication [4], which is typically used for replication across different (and heterogeneous) DBMS instances, is not directly comparable to our proposed architecture where both the primary and replicas belong to the same database schema and common transaction domain. We also notice that the state-of-the-art parallel log replayer [16] is not scalable due to the contention at the inter-transaction dependency checking.

In this paper, we propose a novel database replication architecture called HANA Asynchronous Parallel Table Replication (ATR). ATR is designed to incur low overhead to transaction processing at the primary site, while it supports scalability of the analytical query performance and shows less CPU consumption for replayed transactions. In addition, with novel parallel log replay and early log shipping mechanisms, ATR can minimize the propagation (or snapshot) delay between the primary and replicas under lazy replication, while ATR improves the primary transaction performance under eager replication.

The contributions included in our earlier paper [24] are summarized as follows:

- Through deep analysis in design requirements and decisions, we propose a novel database replication architecture for real-time analytical queries on operational data.
- In order to reduce the propagation delay, we propose a novel optimistic lock-free parallel log replay scheme which exploits so-called RVID (record version ID) to apply record-wise partial ordering.
- We propose a novel log-less replica recovery scheme which exploits characteristics of in-memory column stores in order to simplify replica recovery and to reduce logging overhead.
- We propose a framework for adaptive query routing depending on its predefined max acceptable staleness range.
- Through extensive experiments with a concrete implementation available in a commercial product, SAP HANA [36], we show that ATR provides sub-second visibility delay even for update-intensive workloads, achieving scalable, OLAP performance without notable overhead to the primary.

In addition to the above contributions, we have newly added the following contributions to this extended paper:

- In Sect. 3.4, we propose a novel optimization, called *optimistic interleaving*, for maximizing parallelism of log replay even for high-conflict workloads where multiple concurrent transactions try to update the same records. With the additional experiments in Sect. 6.4 (Fig. 12), we demonstrate the benefit of the proposed optimistic interleaving scheme.
- In Sect. 7, we propose a novel *eager parallel replication* mechanism that exploits the ATR's parallel log replay, early log shipping, and log-less replica recovery. With the eager replication implementation already available in the productive versions of SAP HANA, we demonstrate how the proposed parallel log replay contributes to high-performance transaction processing at the primary under eager replication (Figs. 15, 16).
- In Sects. 3.5.1 and 3.5.2, we describe how we make ATR log replayer light-weight. With the additional experiment in Sect. 6.5 (Fig. 13), we show how the proposed light-weight replayer contributes to lower CPU consumption at the replicas and thus to increase the CPU capacity for more OLAP workloads.
- In Sect. 4, we elaborate the implementation of our novel log-less replica recovery protocol with additional discussion on the recovery performance.
- We provide more complete description on practical issues that were addressed while implementing the proposed replication mechanism in the commercial product. It includes MVCC and garbage collection at the replicas (in Sect. 3.5.3), adaptive query routing protocol proposed to gracefully handle replica-side errors (in Sect. 4.3), and wait-and-forward scheme proposed to deal with transactional consistency issues arising at lazy replication (Sect. 5).
- In Sect. 8, we present potential future extensions of ATR. It includes (1) sub-table replication, (2) various forms of cross-format replication, (3) semi-multi-master replication, (4) log forwarding for efficiently handling log serialization error, (5) advanced replication log buffer management for reducing contention at the primary, (6) log compression and (7) online non-disruptive replica

addition protocol for elastic scaling in cloud environment.
- Finally, in Sect. 9, we extend the scope of our survey of related work in order to emphasize the uniqueness and novelty of the proposed replication mechanism.

The rest of this paper is organized as follows. Section 2 shows the proposed architecture of ATR and its design choices. Section 3 presents how logs are generated at the primary and replayed at replicas. In Sect. 4, we present a post-failure replica recovery mechanism and ATR's various implementation issues. Section 5 discusses three particular transaction consistency issues arising by the nature of lazy replication architecture. Section 6 presents the results of performance evaluations. Section 7 presents eager replication implementation. Section 8 presents potential future optimizations of ATR. Section 9 gives an overview of related work. Finally, Sect. 10 concludes the paper.

2 Architecture and design choices

2.1 Overall architecture

Figure 1 shows the overall architecture of ATR. The ATR system consists of the primary and one or more replica servers, each of which can be connected with another by a commodity network interconnect without any shared storage necessarily. All write requests are automatically directed to the primary server by the database client library, embedded in the application process. During the course of processing a received write request, the primary server generates a replication log entry if the write request makes any change to a replication-enabled table. Note that ATR can be applied to only a selected list of tables, not necessarily replicating the entire database. The generated replication log entry is shipped to the replicas via the network interconnect and then replayed at the replicas. By replaying the propagated replication log entries, the in-memory database copies of the replicas are maintained in a queryable and transactionally consistent state. The database client library transparently and dynamically routes read-only queries to the replicas if the replica database state meets the given freshness requirements of the queries.

Although ATR can also be extended for high availability or disaster recovery purposes, the main purpose of ATR is to offload OLAP-style analytical workloads from the primary server which is reserved for handling OLTP-style transactional workloads. Additionally, by having multiple replicas for the same primary table, ATR can elastically scale out the affordable volume of the OLAP-style analytical workloads. Moreover, by configuring the primary table as an OLTP-favored in-memory row store while configuring its replicas as OLAP-favored in-memory column stores in SAP HANA, ATR can maximize the processing capability of OLTP/OLAP mixed workloads under the common database schema and the single transaction domain.

2.2 Design choices

Under the overall architecture and design goals, Table 1 shows the practical design decisions during the development of ATR for the SAP HANA commercial enterprise in-memory database system. We group these design deci-

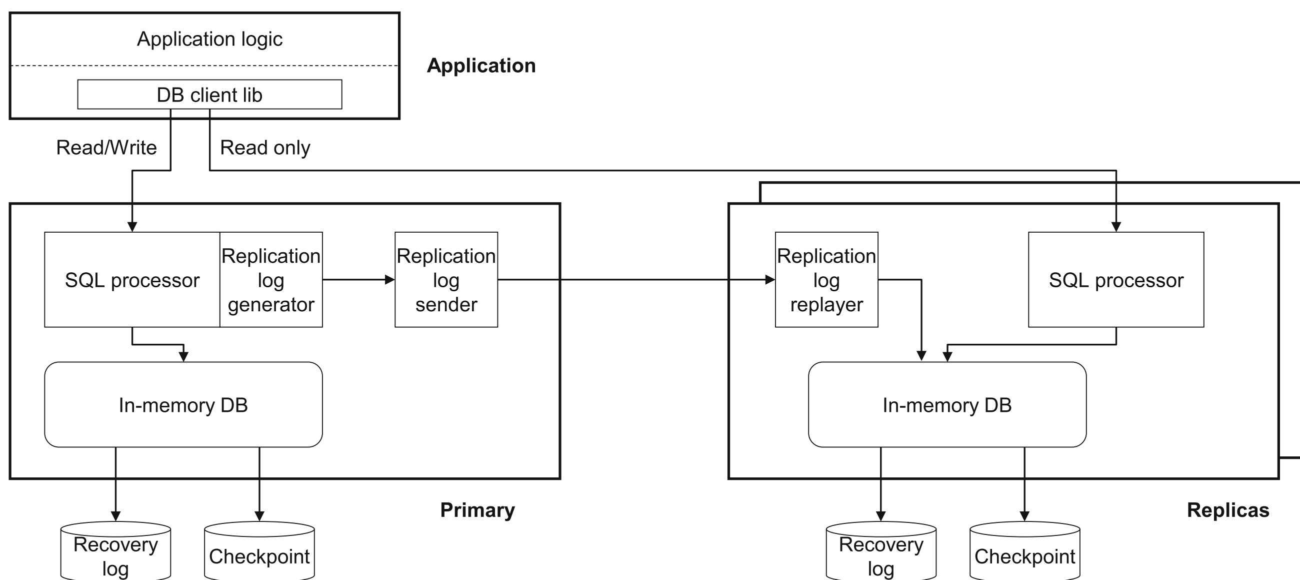

Fig. 1 Overall architecture

Table 1 Summary of ATR design decisions

(a) Common	
D1.1	Replicate across formats (including across row store format and column store format)
D1.2	Decouple and separate the replication log from the storage-level recovery log
(b) Primary server	
D2.1	Tightly couple the replication log generator and sender within the DBMS engine
D2.2	Log the record-level SQL execution result to avoid non-deterministic behaviors and the potential conflict during parallel log replay
D2.3	Ship generated replication log entries as soon as execution of its DML statement is completed
(c) Replicas	
D3.1	Perform parallel log replay in replicas to minimize visibility delay
D3.2	Enable adaptive query routing depending on its predefined max acceptable staleness range
D3.3	Make replayer transactions light-weight to spare more CPU resource for OLAP processing at the replicas
D3.4	Support efficient post-failure replica recovery

sions into three categories depending on where each decision is affected to either both primary and replicas (Table 1a), primary only (Table 1b), and replicas only (Table 1c).

Now, we explain each decision in detail and elaborate its rationale. First, ATR replicates across different table formats (D1.1). Given that SAP HANA provides both the OLTP-favored in-memory row store and the OLAP-favored the in-memory column store, replicating from a row store to a column store could be an interesting option for the cases that require higher OLTP and OLAP performance together. Note that replication from a column store to a row store is not yet implemented in SAP HANA because there has been no specific need for this combination.

Second, we have decoupled and separated the *replication log* from the storage-level *recovery log* that is generated basically for the purpose of database recovery (D1.2). Because it has been an important goal to make ATR work across different table formats, it is almost impossible to rely on the existing SAP HANA recovery log, which is tightly coupled with the physical format of the target table type (for example, *differential logging* for the row store [21]). There are also many application cases where replicating only a selected list of tables is sufficient and efficient, instead of replicating all the tables in the database. Since the storage-level recovery log is organized as a single ordered stream for the entire database, it could generate an additional overhead to extract the redo logs of a few particular tables from the global log stream. Moreover, in order to minimize any disruptive change in the underlying storage engine of SAP HANA, a practical design decision was made to decouple the newly developed replication engine from the existing underlying storage engines.

Third, we have decided to log the record-level SQL execution result (called *record-level result logging*) instead of logging the executed SQL operation itself (called *operation logging*) (D2.2). If we log the executed SQL string as it is, it becomes very difficult to keep the replica database state consistent with the primary because of the non-deterministic SQL functions or because of the dependency on the database state at the time of log replay. For example, the execution order of the following two update statements is important depending on the parameter value of the first statement, but it will require a more complicated comparison method to infer that these two statements have a dependency with each other or will lead to restrictive parallelism during log replay. In contrast to the operation logging, the record-level result logging is free from such non-deterministic behaviors, and the potential conflict between two different log entries is easily detected by using RVID, which will be explained in more detail in Sect. 3.3.

```
update table1 set col1 = ? where col2 = 'B';
update table1 set col3 = 'C' where col1 = 'A';
```

Fourth, although ATR supports both *lazy (or asynchronous) replication* and *eager (or synchronous) replication* [9], we have chosen the lazy replication as the default mode in order to minimize the latency overhead to the write transactions running at the primary. In the lazy replication, a transaction can commit without waiting for its replication log propagation to the replicas. As a side effect, it could happen that a query executed at the replicas may refer to an outdated database state. Although such a *visibility delay* is unavoidable under the lazy replication, we have made additional design decisions to minimize the visibility delay at the lazy replicas especially for the OLAP applications which require the real-time reporting for operational data.

- *In-database replication* The replication log generator and sender are tightly embedded inside the DBMS engine (D2.1) instead of relying on an external application-driven replicator like ETL tool [42] or middleware-based database replication [4] that can involve an additional network round trip to replicate from one database to another.
- *Early log shipping* ATR early ships the generated replication log entry as soon as its DML statement is completed (D2.3) even before the transaction is completed, differently from [16]. As illustrated in Fig. 2, this is especially important for reducing the visibility delay of multi-statement transactions. Note that, under the early log shipping, if the primary transaction is aborted later, then the replica changes made by the replication log entries should be rolled back as well. However, compared to

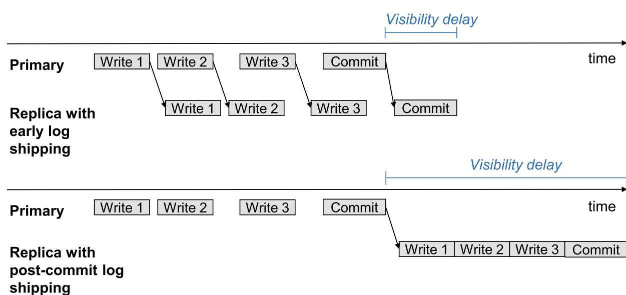

Fig. 2 Early log shipping versus post-commit log shipping

database systems employing the Optimistic Concurrency Control [45], SAP HANA can show relatively lower abort ratios because it relies on pessimistic write locks for concurrency control among the write transactions. Notice that the read queries in SAP HANA do not require any lock based on the MVCC implementation [26].

– *Parallel log replay* ATR performs parallel log replay in replicas to minimize visibility delay (D3.1). As SAP HANA is typically deployed to the shared-memory multiprocessor architecture, the replication log entries can be generated from multiple CPU cores at the primary. Therefore, without the parallel log replayer, the replicas may not catch up with the log generation speed of the primary which can eventually lead to high visibility delay. To achieve full parallelism during the log replay, we propose a novel lock-free parallel log replay scheme which is explained in Sect. 3.

Fifth, together with the above approaches for reducing the visibility delay, ATR allows users to specify the maximum acceptable staleness requirements of individual queries by using a query hint like "select … with result lag (x seconds)" (D3.2). When a commit log is generated at the primary, the current time is stored in the commit log entry which is propagated to the replicas. Additionally, at the replica side, when the commit log is replayed, the stored primary commit time is recorded as the *last commit-replay time*. Based on the last commit-replay time maintained at the replica and the staleness requirement specified in the executed query, it is determined whether or not the query is referring to a database snapshot that is too old. If it is, then the query is automatically re-routed to the primary in order to meet the given visibility requirements. While the primary is idle, a simple dummy transaction is periodically created and propagated to replicas to maintain the last commit-replay time more up to date.

Sixth, we have also paid attention to reducing CPU consumption of the replayer transactions (D3.3). If the replayer transactions just repeat the same amount of work as the primary write transaction, then the same amount of CPU resource will be needed to replay the write transaction. However, ATR reduces the CPU consumption of the replayer transactions by the following design decisions.

– To find the target record at the replica for a given replication log entry, ATR log replayer performs a simple hash operation with the 8-byte RVID, instead of involving the primary key search operation or any other predicate evaluation. It is also another benefit of the record-level result logging (D2.1).
– ATR log replayer skips locking and unlocking operations based on its lock-free parallel log replay scheme (D3.1). More detail is explained in Sects. 3.3 and 3.4.
– ATR log replayer skips constraint checks because it was already done at the primary. More detail is explained in Sect. 3.5.1.
– ATR log replayer also performs light-weight commit operations, which is explained in Sect. 3.5.2.

Such saved CPU resources at the replicas can eventually lead to more capacity for more OLAP workloads at the replicas.

Seventh and finally, as a consequence of lazy replication, if a failure is involved during replication, a number of replication log entries could be lost before they are successfully applied to replicas. In order to deal with this situation, ATR supports a post-failure replica recovery with an optimization especially leveraging the characteristics of in-memory column store (D3.4), which will be explained in Sect. 4.1.

3 Log generation and replay

After describing the structure of the replication log entries (Sect. 3.1), this section presents how they are generated by the primary server (Sect. 3.2) and then replayed by the replica server in parallel (Sect. 3.3).

3.1 Log records

Each replication log entry has the following common fields.

– *Log type* Indicates whether this is a DML log entry or a transaction log entry. The transaction log is again classified into a pre-commit log entry, a commit log entry, or an abort log entry.
– *Transaction ID* Identifier of the transaction that writes the log entry. This is used to ensure the atomicity of replayed operations in the same transaction.
– *Session ID* Identifier of the session to which the log generator transaction is bound. Transactions are executed in order within the same session sharing the same context. Session ID is used to more efficiently distribute the repli-

cation log entries to the parallel log replayers, which will be explained in more detail in Sect. 3.3.

In particular, the DML log entries have the following additional fields.

- *Operation type* Indicates whether this is an insert, update, or delete log entry.
- *Table ID* Identifier of the database table to which the write operation is applied.
- *Before-update* RVID Identifier of the database record to which the write operation is applied. In SAP HANA employing MVCC, even when a part of a record is updated, a new record version is created instead of overwriting the existing database record. Whenever a new record version is created, a new RVID value, which is unique within the belonging table, is assigned to the created record version. Since RVID has 8 bytes of length, its increment operation can be efficiently implemented by an atomic CAS (compare-and-swap) instruction without requiring any lock or latch. Note that the insert log entry does not require Before-update RVID.
- *After-update* RVID While Before-update RVID is used to quickly locate the target database record at replica, After-update RVID is applied to keep the RVID values identical across the primary and the replicas for the same record version. Then, on the next DML log replay for the record, the record version can be found again by using the Before-update RVID of the DML log entry. For this, RVID fields of the replica-side record versions are not determined by the replica itself but filled by After-update RVID of the replayed log entries. Note that the delete log entry does not require its own After-update RVID.
- *Data* Concatenation of the pairs of the changed column ID and its new value. Note that the column values have a neutral format that can be applied to either of the HANA row store or the HANA column store so that, for example, a DML log entry generated from a row store table can be consumed by the corresponding column store table replica.

3.2 Log generation

Figure 3 shows the architecture of the replication log generator and sender. After a DML statement is successfully executed, the corresponding DML log entries are generated from the record-level change results together with their Before-update RVID and After-update RVID values. The generated DML log entries are directly appended to a shared log buffer without waiting for the completion of the transaction. There can exist multiple threads which are trying to append to the single shared log buffer, but the log buffer can

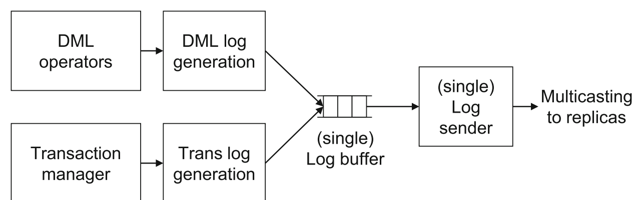

Fig. 3 Log generator and sender

be efficiently implemented by a lock-free structure using an atomic CAS instruction.

The transaction log entries are generated after the corresponding transaction's commit or abort is decided, but before their acquired transaction locks are released. Such generated transaction log entries are also appended to the same log buffer as DML log entries. Together with the single log sender thread which multicasts the appended log entries to the corresponding replicas in order, it can be concluded that all the generated replication log entries are ordered into a single log stream in the log buffer and delivered to each of the replicas, ensuring the following properties.

- The transaction log entries are placed after their preceding DML log entries in the replication log stream.
- A later committed transaction's commit log is placed after its earlier committed transaction's commit log in the replication log stream.

Note that, in the current ATR implementation, the multicast operation is implemented by using repeated network *send* calls to different target hosts, but we do not exclude the option of using faster network-level multicast operation in the future.

3.3 Parallel log replay

The basic idea of the ATR parallel log replayer is to parallelize the DML log replay while performing the transaction commit log replay in the same order with the primary. Here, in order to reduce unnecessary conflict and minimize the visibility delay, we propose the novel concepts of the *SessionID-based log dispatch* method and the *RVID-based dynamic detection of serialization error*, which will be detailed below.

As illustrated in Fig. 4, after receiving a chunk of replication log entries, the log dispatcher dispatches the received log entries depending on their log type. If the encountered log entry is a commit log, then it is dispatched to the global transaction log queue. If the encountered log entry is a DML log, a pre-commit log, or an abort log entry, then it is dispatched to one of DML log queues basically by the modulo operation with Session ID stored in the log entry. Since a

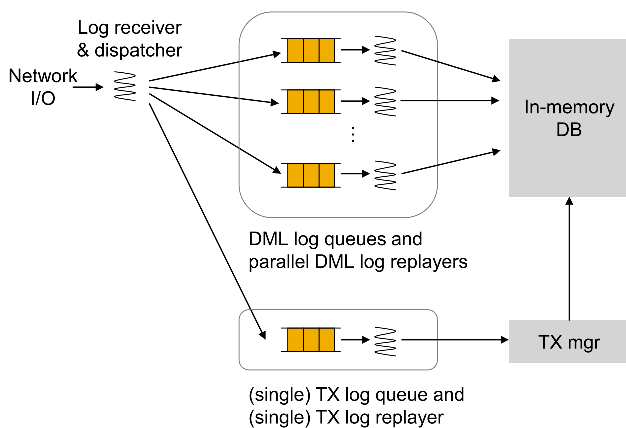

Fig. 4 Parallel log replay

transaction is bound to a single session, all the log entries generated from the same transaction are dispatched to the same DML log queue. For the session which repeatedly accesses the same set of database objects with different transactions, the SessionID-based log dispatch method can reduce unnecessary conflict among the parallel replayers than a plain TransactionID-based dispatch method. We have not chosen the TableID-based dispatch method because it can limit the parallelism for the skewed update workloads to a particular table. Note that, although it is not yet fully implemented, it is also considered to combine the SessionID-based dispatch method with a dynamic adjustment method for better load balancing across the available queues by monitoring the length of each queue.

The log entries distributed to multiple queues are dequeued and replayed by the log replayer dedicated to each log queue. The replay algorithm for each log type is presented in Algorithms 1, 2, 3 and 4. The trickiest part in the log replay algorithm is how to ensure replaying DML log entries in their generation order on the same database records while replaying the transactions in parallel by multiple DML log replayers. For example, in case of the parallel log replay algorithm suggested in [16], the transaction replay order is determined by using a central run-time inter-transaction dependency tracker which may subsequently become a global contention point. Unlike the pessimistic approach in [16], ATR does not maintain any run-time inter-transaction dependency graph nor any additional lock table. Instead, ATR follows an *optimistic lock-free* protocol. After finding the target database record for the log replay, the ATR replayer checks whether or not the database change happened before the current log entry is already applied. If not, we call it a *log serialization error* and retry the log replay with re-reading the target database record (lines 9–15 and 17–23 in Algorithm 1).

In order to correctly detect the log serialization error, ATR exploits the characteristics of the MVCC implementation of

Algorithm 1 Replay a DML log entry (α, β, and τ denote After-update RVID, Before-update RVID, and TableID, respectively.)

Require: A DML log entry L.
1: Find the transaction object T for $L.TransactionID$.
2: **if** T is empty **then**
3: Create a transaction object for $L.TransactionID$.
4: **end if**
5: **if** $L.OperationType = Insert$ **then**
6: Insert $L.Data$ into the table $L.\tau$.
7: Set the inserted record's $RVID$ as $L.\alpha$.
8: **else if** $L.OperationType = Delete$ **then**
9: **while** true **do**
10: Find the record version R whose $RVID$ equals
11: to $L.\beta$ in the table $L.\tau$.
12: **if** R is not empty **then**
13: Delete R. **return**
14: **end if**
15: **end while**
16: **else if** $L.OperationType = Update$ **then**
17: **while** true **do**
18: Find the record version R whose $RVID$ equals
19: to $L.\beta$ in the table $L.\tau$.
20: **if** R is not empty **then**
21: Update R with $L.Data$ and $L.\alpha$. **return**
22: **end if**
23: **end while**
24: **end if**

Algorithm 2 Replay a pre-commit log entry

Require: A pre-commit log entry L.
1: Find the transaction object T for $L.TransactionID$.
2: Mark T's state as $pre-committed$.

Algorithm 3 Replay a commit log entry

Require: A commit log entry L.
1: Find the transaction object T for $L.TransactionID$.
2: Wait until T's state becomes $precommitted$.
3: Increment the transaction commit timestamp of the replica server by marking the T's generated record versions with a new commit timestamp value.

Algorithm 4 Replay an abort log entry

Require: An abort log entry L.
1: Find the transaction object T for $L.TransactionID$.
2: Abort T with undoing the changes made by the transaction T.

SAP HANA. The update and delete log entries check whether there exists a record version whose RVID equals to *Before-update* RVID. If such a record version is not yet visible to the replaying transaction (that is, when R is empty in line 12 or 20 of Algorithm 1), it means that the preceding DML operation for the same record has not yet been replayed. For example, imagine that there are three transactions which have inserted or updated the same database record in order, as illustrated in Fig. 5 (T_1 inserted, T_2 updated, and then T_3 updated the same record). Then, the version space at the primary and the corresponding log entries can be populated as in Fig. 5. Under

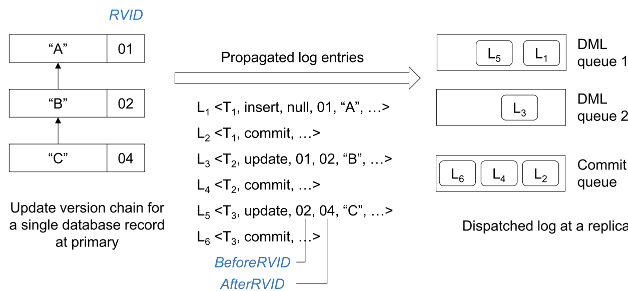

Fig. 5 Parallel log replay example

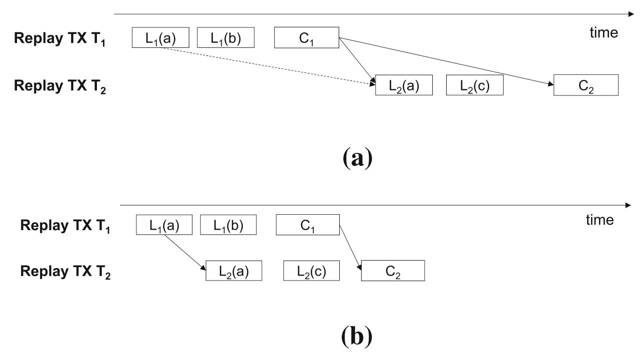

Fig. 6 Replay of two inter-conflicting transactions ($L_i(a)$ denotes a DML log entry for a database record a which is replayed by transaction T_i. C_i denotes the commit log entry of transaction T_i. The arrows denote the inter-operation dependency implicitly imposed by the given replay algorithm). **a** When optimistic interleaving is not applied. **b** When optimistic interleaving is applied

this scenario, after replaying log entries L_1 and L_2, L_5 can be encountered by a DML replayer before L_3 is replayed. However, while trying to replay L_5, the DML replayer recognizes that there is no record version whose RVID is equals to L_5's *Before-update* RVID, 02 and thus, it will encounter the log serialization error and retry the DML replay operation (after some idle time, if necessary).

By this proposed RVID-based dynamic detection of serialization error, the DML log entries can be dispatched and replayed to multiple queues freely without restriction (for example, without TableID-based dispatch), and it is one of the key reasons why ATR can significantly accelerate the log replay and thus minimize the visibility delay between the primary and the replicas.

3.4 Optimistic interleaving for high-conflict workloads

By the parallel log replay scheme explained in Sect. 3.3, DML log entries are replayed in parallel without having any central run-time inter-transaction dependency tracker which could be a potential contention point. However, the scheme itself does not parallelize the transactions which updated the same database record at the primary. For example, if replayer transactions follow the two-phase locking protocol, then the log entry L_3 can be replayed only after T_1 finally commits and releases its acquired record lock in the scenario of Fig. 5. In order to overcome such a parallelism limitation under high-conflict workloads, we propose the so-called *Optimistic Interleaving* scheme which is seamlessly combined with the parallel log replay scheme proposed in Sect. 3.3.

Our optimistic interleaving scheme consists of two parts. One is about handling write-write collisions, and the other is about handling read-write collisions. First, regarding the write-write collision, unlike any locking protocol, the DML replay operation does not acquire any record lock. It is possible because there is no other concurrent write transaction in the replica except the other DML log replayers, and the transaction serialization is ensured by checking the RVID visibility among the DML log replayers as explained in Sect. 3.3. Second, regarding the read-write collision, when the DML replayer reads the RVID value of the target database record at replica, it directly reads not-yet-committed changes instead of following the read-committed semantics. Note that, in our initial implementation of ATR, the replay transactions still followed the read-committed semantics of the snapshot isolation. As a result, even though they do not acquire any record locks, lines 10 and 18 of Algorithm 1 have to wait until the preceding DML operation's transaction finally commits. On the other hand, in our latest implementation, lines 10 and 18 immediately read the RVID values of not-yet-committed changes instead of following the conventional read-committed semantics. Remark that, to enable this optimization, it is ensured that the RVID value is updated at the last step of the DML replay operation.

Optimistic Interleaving brings the benefit of further increasing the parallelism of replayed transactions even for high-conflict workloads. Figure 6 illustrates the benefit of this optimistic interleaving implementation. In Fig. 6a, $L_2(a)$ can start its replay operation after waiting for C_1's replay operation. On the other hand, in Fig. 6b where the optimization is in place, $L_2(a)$ can start right after waiting for $L_1(a)$. Note that, in Fig. 6, ordering between C_1 and C_2 is ensured by the single commit-replay queue, as explained in Sect. 3.3. In addition to the benefit of increased parallelism for high-conflict workloads, Optimistic Interleaving also helps reduce the CPU cost of each replay transaction since replayer transactions skip record lock and unlock operations. Note that the benefit of Optimistic Interleaving is also shown experimentally in Sect. 6.4.

Optimistic Interleaving does not break any data consistency at the replica tables because (1) for the same database record, DML operations are still replayed in their initial execution order at the primary system (by the RVID-based parallel log replay scheme), and thus the record versions are created in their initial execution order; (2) the commit log

replay is performed in the same order as the primary based on single commit log queue even for the inter-conflicting transactions; (3) following MVCC, each DML replay creates its own record versions which become visible only when the replay transaction finally commits; and (4) the to-be-aborted replay transaction does not affect any log serialization dependency at the replicas because any After-update RVID value of an aborted transaction cannot be referred to as a Before-update RVID value of the next executed write transaction at the primary side.

3.5 Further optimizations and implementation issues

3.5.1 DML replay with skipped constraint checks

In Algorithm 1, the DML replay operation skips the integrity constraint check because it was already done at the primary. Due to the skipped integrity check and the skipped locking during parallel log replay, it is possible that uncommitted duplicate records that have the same unique key values co-exist tentatively (for example, when a record at the primary is inserted, deleted and then inserted again by transactions T_1, T_2, and T_3, replaying their DML log entries in the order of T_1, T_3, and T_2 at a replica can lead to such a situation). However, this does not lead to any real problem because the result of DML replay is not directly visible to the queries executed at the replica but visible only after the corresponding commit replay is completed and also because the commit log entries are replayed strictly in the same order as the primary.

3.5.2 Light-weight commit replay

We have paid special attention to the implementation of the commit log replay not to make it as a bottleneck point in the ATR parallel log replay scheme. The key idea is rather to break down the transaction commit work into three parts, *pre-commit*, *commit*, and *post-commit*, and then delegate the pre-commit work to the parallel DML log replayers by using the pre-commit log entry and delegates the post-commit work to asynchronous background threads. As a result, the serialized part of the transaction commit operation is made short and light-weight.

The pre-commit log entry plays the role of marking that all DML log entries of the transaction have been successfully replayed and of informing the commit log replayer by using the transaction state information maintained in the transaction object, as shown in Algorithm 2. The important role of the commit log replay is to mark the generated record versions by the transaction's DML replay as committed and thus to make the record versions visible to the queries executed at the replica server, as shown in Algorithm 3. Right after finishing the commit operation of a commit log entry, the commit log replayer processes the next commit log entry in the queue while delegating the remaining post-commit work of the transaction to other background threads.

3.5.3 MVCC at replicas

The insert (line 6), delete (line 13) and update (line 21) operations in Algorithm 1 create their own records versions instead of performing in-place updates. The record versions created by the same transaction are associated as a group by pointing to the same so-called TransContext object. At the time of replaying its transaction commit operation (Algorithm 3), the commit timestamp value is determined for the committing transaction, and the value is written to the TransContext object. Then, the commit timestamp value becomes immediately visible to all related record versions of the committing transaction.

The garbage collection [26] at replicas can be performed independently of the primary's garbage collection because it is not allowed that a single query accesses both of the primary and its replica during its execution. Also, because a single query is not allowed to access multiple replicas of the same table during its execution, the garbage collection operations of the replicas do not need to synchronize with each other.

Overall, the replicas also follow the same MVCC protocol with the primary, which is described in more detail in [26].

3.5.4 Query processing at replicas

Queries running at the replicas just follows the existing visibility rule of MVCC in SAP HANA. When a query starts at a replica, it takes its snapshot timestamp (or read timestamp) from the replica commit timestamp which is incremented by the commit log replayer as in Algorithm 3. Then, during its query processing, the query judges which record versions should be visible to itself by comparing the record versions' creation timestamp values with the query's snapshot timestamp. Again, the visibility decision protocol for the queries is also described in more detail in [26].

3.5.5 Handling DDL operations

Following the SAP HANA distributed system architecture [23], the replica server does not maintain its own metadata persistency but caches the needed metadata entities on demand by reading from the primary. Therefore, if a DDL transaction is executed at the primary, it does not generate a separate DDL log entry but it invalidates the corresponding metadata entities at the replicas. This invalidation operation is performed at the time when the DDL transaction commits after waiting until its preceding DML log entries for the table are replayed.

4 Replica recovery

4.1 Log-less replica recovery

By the nature of the lazy replication, if a failure is involved during log propagation or log replay, a series of replication log entries could be lost before they are successfully applied to the replica database. In order to deal with this problem, a typical approach under the lazy replication is the so-called *store-and-forward* method. The generated log entries are stored persistently within the primary transaction boundary and then propagated to the replicas lazily. Then, by maintaining a watermark at the replayer side, the lost log entries can be easily identified and resent from the persistent store. Although we do not exclude the store-and-forward approach, we propose a novel efficient replica recovery method that does not rely on the persistent replication log store, in order to further reduce the overhead to the primary transaction execution and simplify the replica recovery protocol.

The key idea is to detect the discrepancy between the primary table and its replica table by comparing the RVID columns of the two tables, as presented in Algorithm 5. Two sets of the RVID values are collected from the latest record versions of the primary and the corresponding replica tables. And then, depending on the result of the relative complements of the two sets, the database records existing only in the primary table are re-inserted to the replica and the records existing only in the replica table are deleted.

Algorithm 5 Recover a replica table

Require: P, a set of RVID values from the primary table.
Require: R, a set of RVID values from the replica table.
1: Delete the records $R \setminus P$ from the replica.
2: Insert the records $P \setminus R$ into the replica

In the example of Fig. 7, $P = \{r_1, r_3, r_5, r_9\}$ is collected from the primary table and $R = \{r_1, r_2, r_4, r_8\}$ from the replica table. Then, since $R \setminus P = \{r_2, r_4, r_8\}$ and $P \setminus R = \{r_3, r_5, r_9\}$, the replica records matching with $\{r_2, r_4, r_8\}$ are deleted, and the primary records matching with $\{r_3, r_5, r_9\}$ are re-inserted to the replica.

Comparison of two RVID columns is implemented by a merge-join-style algorithm where two RVID columns are compared after being collected from each table in a sorted order. Although comparing the entire RVID columns of two tables looks expensive at first glance, we took this approach especially considering the characteristics of SAP HANA in-memory column store. Since the RVID column values of the entire table are stored on a contiguous memory in a compressed form [36] in SAP HANA column store, scanning the entire RVID column values of a column store table can be done rapidly. Moreover, the column scan performance can

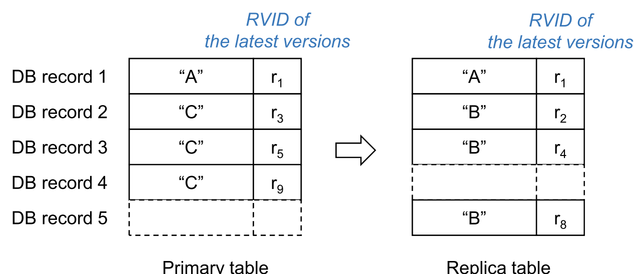

Fig. 7 Post-failure replica recovery

be further accelerated by exploiting SIMD-based vectorization and parallelization as explained in more detail in [46]. For example, Figure 16 of [46] shows 2.4 GB/s of full table scan performance even without any parallelization, which means that 1 billion RVID column values can be scanned in a few seconds in the tested hardware configuration. Note that the core part of SAP Business Warehouse Accelerator (BWA) used for the experiment of [46] is incorporated in SAP HANA column store [30].

4.2 Redo and undo logging at replicas for recovery and transaction abort

During the DML log replay, the recovery redo log entries are generated for the recovery of the replica server. They are asynchronously flushed to the persistent log storage, and even the commit replay does not wait for the log flush completion because the lost write transactions on any failure at a replica can be re-collected from the primary database as explained in Sect. 4.1. The undo log entries are also generated during the DML log replay because the not-yet-committed replication log entries can be replayed for *early log shipping*, as explained in Sect. 2.2. When a transaction is aborted and its replication log entries has already been shipped to any of its replicas, then its abort log entry is generated and shipped. At the replica side, the change made by the transaction's DML replay is rolled back as in Algorithm 4.

4.3 Adaptive query routing for handling replica errors

When a particular replica becomes unavailable due to crash or run-time error such as an out-of-memory exception, it is desirable to continue the overall query service without disrupting or throwing errors to the end users. For this, ATR incorporates an adaptive query routing scheme which consists of the following query routing rules.

– If a replica-side query encounters an error, then the query is implicitly aborted and retries. And, if it turns out that the replica is not available after checking the process sta-

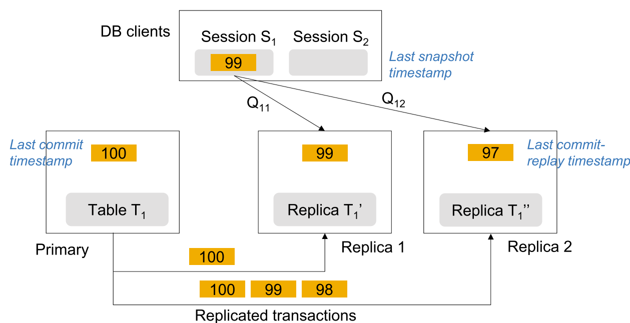

Fig. 8 A simplified view of a multi-replica replication scenario

tus or after waiting for a predefined time period, then the query is retried with being forwarded to the primary or another available replica. Because the forwarding mechanism happens automatically and implicitly by the client library, this type of error does not need to be handled at the application or user side.

– If a replica is marked as unavailable, then the replica status is also notified to the primary metadata. Then, the next incoming queries exclude the replica during their query compilations until it is marked as available again.

5 Handling additional transaction consistency issues

In this section, we discuss three particular transaction consistency issues arising from the nature of lazy replication architecture and present a practical way of dealing with them.

5.1 Ensuring transactional consistency among multiple replicas with wait-and-forward scheme

When a table has more than one replica, it is an important issue to ensure transactional consistency among multiple replicas of the same table. A simple solution is to perform an atomic multi-node commit operation, like two-phase commit, for the replayer transaction of the multiple replicas in order to keep the replicas in the same database state. However, it is not a desirable approach in terms of the replay performance because each commit log replay may involve a cross-node synchronization overhead. To avoid this problem, we propose the so-called *wait-and-forward* scheme for ensuring transactional consistency across multiple replicas of the same table. In this wait-and-forward scheme, each replica commits independently with each other but applies the following special query routing rules.

– First, the query plan generator does not allow a single query to access more than one replica of the same table in its generated query plan. That is, a single query can access at most only one replica for a given table during its query execution.

– Second, each database session maintains the *last commit-replay timestamp* of the last accessed replica node in the session and stores it as the *last snapshot timestamp* of the session. And then, when a newly accessed replica node has an older commit-replay timestamp than the last snapshot timestamp of the current session, then the query execution at the replica node is postponed until the last commit-replay timestamp of the replica node becomes equivalent to or higher than the last snapshot timestamp of the session. If the waiting time at the replica becomes larger than a predefined time threshold, the query is automatically forwarded to the primary node or other available replica nodes.

In the example of Fig. 8, a database session S_1 is executing two queries Q_{11} and Q_{12} in order. After executing Q_{11} at the replica 1, S_1 caches 99 as its last snapshot timestamp. And then, when executing Q_{12} at the replica 2, it detects that the replica 2 has an older database state than its last executed replica by comparing its last snapshot timestamp (99) with the replica 2's last commit-replay timestamp (97). Then, Q_{12}'s execution is postponed until the replica 2 gets a sufficiently high last commit-replay timestamp.

Note that, in the wait-and-forward scheme, read queries that are executed in a database session established only to a particular replica do not need to involve any waiting or forwarding.

5.2 Ensuring read-your-write consistency for read queries in a write transaction

If a transaction tries to read its own earlier DML result and the read operation is routed to the replica, then the replica-routed query may not see its own change result yet. In the terminology of [44], it corresponds to so-called *read-your-writes consistency*. In order to guarantee the read-your-writes consistency with ATR, we have considered two practical solutions. First, the primary maintains additional watermarks incremented on every DML, and then the replica-routed query checks whether the sufficient number of DML logs are already replayed at the replica. In the second solution, each database session maintains the changed table list for the currently active transaction and then directly routes such detected read-your-write queries to the primary. In the current production version of ATR, the second approach is available for the simplicity of implementation. In the example of Fig. 9, $R_1(a)$ is routed to the primary.

Note that the read-your-write consistency issue can arise for already committed changes. In the example of Fig. 9, if $W_1(a)$ or C_1 is not yet replayed at a replica, a replica-routed query $R_2(a)$ is unable to see the latest updated result

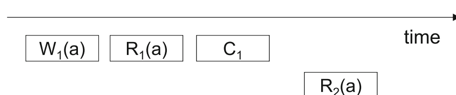

Fig. 9 Read-your-writes example (($W_i(a)$ denotes a write operation for a database object a by transaction T_i. $R_i(a)$ denotes a read operation for a database object a by transaction T_i. C_i denotes the commit operation of transaction T_i)

of the same session at the replica yet. The read-your-write consistency for already committed changes can be ensured by extending the *wait-and-forward* scheme of Sect. 5.1. When a transaction makes changes at the primary, the transaction's primary commit timestamp is also stored as the *last snapshot timestamp* of the session. Then, in the example of Fig. 9, $R_2(a)$ is executed after waiting until C_1 is replayed at the replica.

5.3 Ensuring monotonic read consistency for consecutive read queries

Let us say that a database session performs two read queries in order where the first one is routed to the primary but the second one to a replica. In this scenario, some database state that was visible to the first query may not be visible to the second replica-routed query by the nature of the lazy replication of ATR. Again, in the terminology of [44], it corresponds to so-called *monotonic read consistency*.

The monotonic read consistency for repeated read queries in a database session is also achieved by further extending the *wait-and-forward* scheme of Sect. 5.1. When a query is executed at the primary, the query's snapshot timestamp value is stored as the *last snapshot timestamp* of the session and then compares it with the replica's last-replay timestamp value on the next query execution in the session.

6 Experiments

In this section, with the following experiment goals, we evaluate the performance of ATR implemented in SAP HANA:

- The optimistic parallel log replay scheme of ATR shows superior multi-core scalability over the primary-side transaction processing or another pessimistic parallel log replay scheme which relies on a run-time inter-transaction dependency tracker (Sect. 6.2).
- Based on its optimistic parallel log replay, ATR shows sub-second visibility delay in the given update-intensive benchmark (Sect. 6.3).
- Regardless of transaction conflict ratio, ATR log replayer constantly shows higher throughput than the primary or a pessimistic algorithm (Sect. 6.4).
- The proposed optimistic interleaving optimization contributes higher parallelism under high contention situation (Sect. 6.4).
- The overhead of ATR at the primary is not significant in terms of primary-side write transaction throughput and CPU consumption (Sect. 6.5).
- ATR log replayer consumes fewer CPU resources than the primary-side transaction processing for the same amount of workloads, which results in higher capacity for OLAP workloads at the replicas (Sect. 6.5).
- Finally, with the increasing number of replicas, ATR shows scalable OLAP performance without notable overhead to the OLTP side (Sect. 6.6).

6.1 Experimental setup

ATR has been successfully incorporated in the SAP HANA production version since its SPS 10 (released in July 2015) [22,25] and under continuous evolution. For the comparative experiments in this paper, we have used the most recent development version of SAP HANA at the time of writing and modified it especially to make the log replayer switchable between the original ATR optimistic parallel replayer and another pessimistic parallel log replayer [16].

To generate a OLTP and OLAP mixed workload, we used the same TPC-CH benchmark program as the one used in [37]. The benchmark program runs both TPC-C and TPC-H workloads simultaneously over the same data set, after initially populating 100 warehouses as in [37]. Whenever a transaction starts, each client randomly chooses its warehouse ID from the populated 100 warehouses. Depending on the purpose of the experiments in this section, we also used only a subset of the TPC-CH benchmark which will be explained in more detail in the next subsections. All the tables used in the TPC-CH benchmark are defined as in-memory column store tables. Due to legal reasons as in [37], the absolute numbers for the TPC-CH benchmark are not disclosed but normalized by undisclosed constants, except for the micro-benchmark results conducted in Sects. 6.4 and 6.5.

We have used up to six independent machines which are connected to each other via the same network switch. Each machine has four 10Gbit NICs which are bonded to a single logical channel aggregating the network bandwidth up to 40 Gbit/s. Each machine has 1TB of main memory, 60 physical CPU cores (120 logical cores with hyper-threading), and local SSD devices for storing the HANA recovery log and checkpoint files. In the experiment of Sect. 6.6, we scale up to four replica servers with one primary server and one client machine, while the other experiments focus on single-replica configuration.

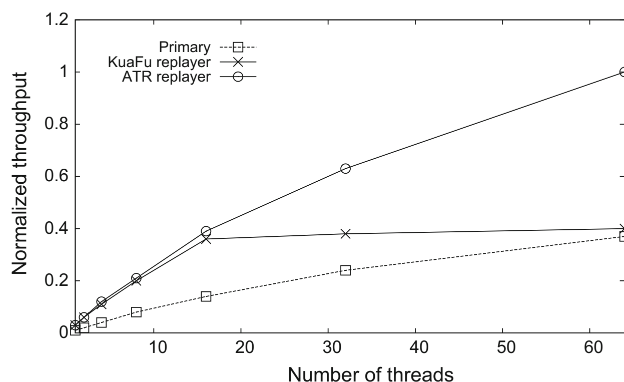

Fig. 10 TPC-C throughput over the number of threads (normalized by the 64-thread ATR replayer throughput)

6.2 Multi-core scalability with parallel log replay

To see multi-core scalability of the ATR parallel log replayer, we first generated the ATR log entries from the primary while running TPC-C benchmark for one minute of the warm-up phase and five minutes of the high-load phase. Then, after loading all the pre-generated ATR log entries into main memory of a replica, we measured the elapsed time for the ATR log replayer to process all the pre-generated and pre-loaded log entries with varying the number of replayer threads at the replica. To compare the log replay throughput of the replica with the log generation throughput of the primary, we also measured TPC-C throughput at the primary with varying the number of TPC-C clients.

Figure 10 shows the experimental results. The normalized throughput was calculated by dividing the number of transactions included in the pre-generated log by the elapsed time, and then normalized by the 64-thread ATR replayer throughput. ATR showed scalable throughput with the increasing number of replayer threads and constantly higher throughput than the primary transaction throughput. This means that the log received from the primary could be processed at the replica without any queuing delay.

Furthermore, to compare the optimistic parallel log replay algorithm of ATR with a pessimistic parallel replay algorithm that relies on an inter-transaction run-time dependency tracker, we have implemented a KuaFu-style parallel replayer based on our best understanding of their paper [16]. For fair comparison, we used the same ATR log format for the KuaFu implementation. At the primary side, the generated log entries are accumulated until the transaction's commit time (as explained in Fig. 2) since the KuaFu replayer assumes that log entries generated from the same transaction appear consecutively in the log stream. In KuaFu, the so-called *barrier*[16] plays the role of synchronizing the parallel log replayers to provide a consistent database snapshot to the replica queries, but we avoid using the barrier in order to see the theoretically maximum replay throughput of KuaFu.

The experiment result with the KuaFu implementation is included in Fig. 10. The KuaFu-style replayer also showed higher throughput than the primary but its throughput was saturated when the number of replayer threads is higher than 16. According to our profiling analysis, the critical section used in the global inter-transaction dependency tracker turned out to be a dominant bottleneck point as the number of the replayer threads increases. Note that [16] also describes that the log replay throughput under a TPC-C-like workload is saturated at 16 CPU cores "due to the high cost of inter-cpu-socket locks". Compared to a pessimistic parallel replayer like KuaFu, ATR does not require any global dependency tracker which could be a single point of contention. This is an important reason why ATR shows better multi-core scalability and can outperform the other approach against higher workloads at the primary.

6.3 Visibility delay

To determine whether ATR can achieve real-time replication with the proposed optimistic parallel log replay algorithm under the early log shipping protocol, we measure the commit-to-commit visibility delay at the replica side. While running the TPC-C benchmark at the primary side, the replayer periodically measures the average visibility delay every 10 s. After synchronizing the machine clocks between the primary and the replica, the replayer calculates the visibility delay by subtracting the primary transaction commit time recorded in the replayed commit log entry from the current time at the time of the commit log replay. Note that this visibility delay measurement method is also used when we enable the adaptive query routing based on its acceptable staleness range, as described in Sect. 2. We also measure the visibility delay with different number of concurrent TPC-C connections to see the impact of the volume of the primary transaction workloads. Note that the number of replayers is dynamically configured to be the same number as the number of TPC-C clients.

Figure 11 shows the result. When the ATR parallel log replayer is used, the visibility delay is maintained mostly under 1 millisecond over time regardless of the volume of concurrent TPC-C workloads at the primary. On the other hand, the KuaFu-style parallel log replayer shows higher visibility delay and, especially when the number of concurrent TPC-C workloads increases to 64 to see the impact of more update-intensive workload, the length of the log replayer queue started growing up and eventually ended up with high visibility delay (more than 10 s) due to the performance mismatch between the primary log generation and the replica log replay, as also indicated by Fig. 10.

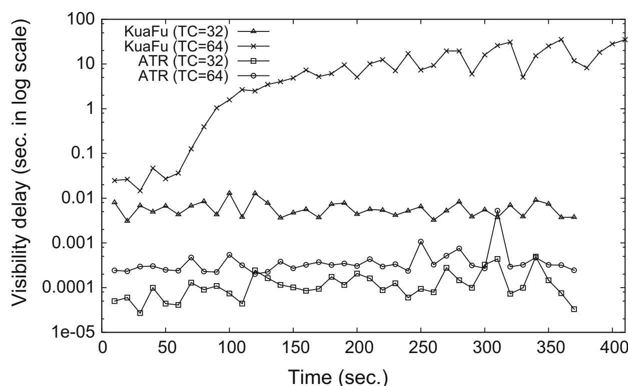

Fig. 11 Visibility delay at the replica in log scale while running TPC-C benchmark at the primary (TC denotes the number of TPC-C clients)

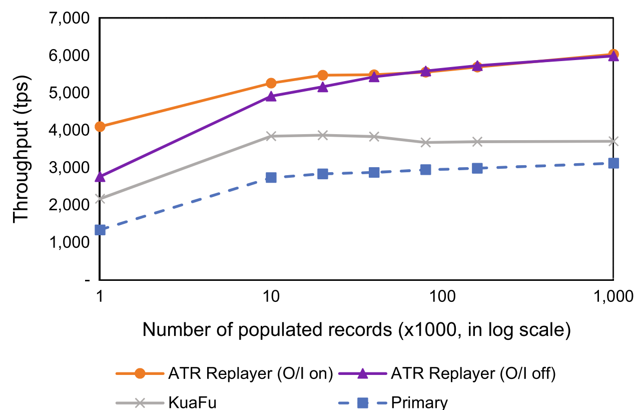

Fig. 12 Micro-benchmark throughput over the conflict ratio (*O/I* denotes optimistic interleaving)

6.4 Impact of inter-transaction conflict

To see whether the superior throughput of ATR over the primary is sustained regardless of the inter-transaction conflict ratio, we have measured the log replay throughput with varying the conflict ratio. To emulate the conflict ratio, we have chosen the ORDERLINE table from the TPC-CH benchmark, and let 100 clients concurrently run update transactions on top of the table while varying the initial table size from 1000 to 1 million records. Consistently with the other experiments, the number of replayers is configured to be the same as the number of the primary-side clients, which is 100 in this case. Each update transaction commits after repeating the following update statement 10 times.

```
UPDATE ORDERLINE SET OL_DELIVERY_D=?
WHERE OL_W_ID=? and OL_D_ID=? and OL_O_ID=?;
```

The 10 primary keys used for each transaction are picked up randomly from the key range of the initially populated data and then assigned in a monotonic order within the transaction to avoid any unnecessary deadlock. In ORDERLINE table, OL_W_ID, OL_D_ID, and OL_O_ID comprise the primary key. Note that we have used this single-table micro-benchmark to generate more severe inter-transaction conflict situation since the performance variation is not notable when we varied the conflict ratio by changing the number of warehouses in the original TPC-CH benchmark. Remark that, in [24], we performed this experiment with 40 clients and 40 replayers, but we increased these numbers to make the inter-transaction conflict situation more severe and thus to clearly see the gain of the optimistic interleaving optimization.

Figure 12 shows the experimental results. ATR constantly shows higher throughput than the primary or our KuaFu implementation regardless of the conflict ratio. In addition, to see the benefit of the optimistic interleaving optimization, we have additionally measured the throughput of the ATR replayer with turning off the optimistic interleaving. The result shows that the proposed optimistic interleaving brings notable gain as the conflict ratio gets higher (for example, when the populated table size is less than 40K records).

In Fig. 12, compared to Fig. 10, the gap between the ATR replayer and the primary is smaller since Fig. 10 was measured with the TPC-C benchmark consisting of read/write workloads, while Fig. 12 was measured with the write-only micro-benchmark. Because only the write statements are propagated to the replica, the replica in the TPC-C benchmark handles fewer replay workloads compared to the designed write-only micro-benchmark.

6.5 Replication overhead

To evaluate the overhead incurred by ATR at the primary side, we have measured the primary transaction throughput while replicating the generated log entries to its replica. To highlight the overhead, we have run the same update-only single-table micro-benchmark as Sect. 6.4 while populating 1 million records initially with 40 clients and 40 ATR replayers. Also, differently from Sect. 6.4 where the replayers run with the pre-generated replication log, we measured the actual performance with the log replicated from the primary online.

Table 2 shows the result. When the replication is turned off, the primary processed 3046 transactions per second while showing 25.76% CPU consumption at the primary. When the replication is turned on, the primary processed 2948 transactions per second while showing 26.19% CPU consumption at primary. It means that the primary throughput dropped by only 3.2% with ATR enablement. The CPU consumption at the primary increased by 1.6% (the third column in the table) or by 5.0% in terms of the normalized CPU consumption by the throughput (the fifth column in the table). According to our CPU profiling analysis, the additional CPU consumption was mainly contributed to by replication log generation, log buffer management, and network operations, as expected. Note, however, that most of the replication operations at

Table 2 Micro-benchmark throughput and CPU consumption of each site

	Primary throughput (tps)	Primary CPU (%)	Replica CPU (%)	Primary CPU normalized by throughput	Replica CPU normalized by throughput
Repl. Off	3046	25.76	N/A	8.46	N/A
Repl. On	2948	26.19	15.60	8.88	5.29

the primary (except the log generation itself) are executed asynchronously by background threads without delaying the primary transaction execution, and thus the impact to the primary transaction throughput is negligible.

In addition to the primary overhead analysis, we have also measured the replica-side CPU consumption as in Table 2. The replica showed only 60.6% of CPU consumption compared to the primary-side execution of the same transaction (= 15.60/25.76) or 62.5% in terms of the normalized CPU consumption (= 5.29/8.46).

To explain the low CPU consumption at the replica, we profiled the CPU consumption of the primary and the replica, as shown in Fig. 13. According to this CPU profiling analysis, the following three factors contributed to the low CPU consumption at the replica, compared to the CPU consumption at the primary.

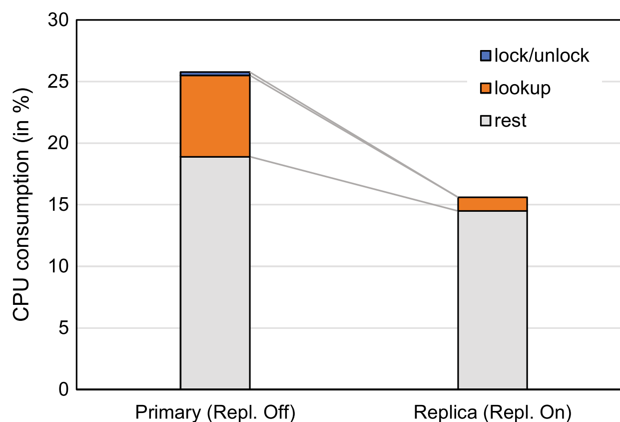

Fig. 13 CPU consumption breakdown from the same experiment with Table 2

Such saved CPU resources at the replicas can eventually lead to more capacity for OLAP processing at the replicas, which will be shown in more detail in Sect. 6.6.

- "lookup": RVID-based record lookup at the replica is the major contributor to such CPU cost savings. While the target record at the primary is searched by the primary key value consisting of OL_W_ID, OL_D_ID, and OL_O_ID in this update-only micro-benchmark, the target record at the replica is found by a simple hash index lookup with the 8-byte *Before-Update RVID* value.
- "lock/unlock": The skipped record locking/unlocking during the log replay (Sect. 3.4) contributed to some extent, albeit small. Note that the gain from the skipped integrity constraint check is not visible at all in Fig. 13 because this experiment is performed with the non-key-field update workloads which will not involve any constraint check even at the primary. However, depending on the workload type and the involved integrity constraints at the primary, there is a high chance that the CPU cost savings at the replica gets bigger.
- "rest": In Fig. 13, the rest of CPU consumption is simply classified to "rest" because it is not our focus. Nevertheless, beside "lookup" and "lock/unlock", we observed that the reduced depths of function call stacks during the log replay also contribute to the CPU cost savings in the "rest" part. In addition, the cost relevant to the session management for remote database clients at the primary is replaced by simpler network channel management at the replica because it is sufficient to handle only the replication log stream.

6.6 Multi-replica scalability under mixed OLTP/OLAP workload

Finally, we show the performance scalability of OLAP queries under OLTP/OLAP mixed workload by using the TPC-CH benchmark. We measured both TPC-C throughput (in terms of transactions per second) and TPC-H throughput (in terms of queries per second) varying the number of replicas from 0 to 4. As the number of replicas increases, we have increased the number of TPC-H clients proportionally since the overall OLAP capacity increases with the number of replicas. While the number of TPC-C client is fixed to 32, 120 TPC-H clients are added per replica server. The number of clients has been chosen so that a single HANA database server can be fully loaded in terms of CPU consumption. Note that SAP HANA provides so-called intra-statement parallelism for OLAP-style queries, where a single OLAP query execution is parallelized by using multiple available CPU cores at the time of its execution. However, throughout this experiment, the intra-statement parallelism was disabled to see more deterministic behavior with the varying number of TPC-H clients. All the tables in the TPC-CH schema have been replicated to all the available replica servers. All the TPC-C transactions are directly routed to the primary

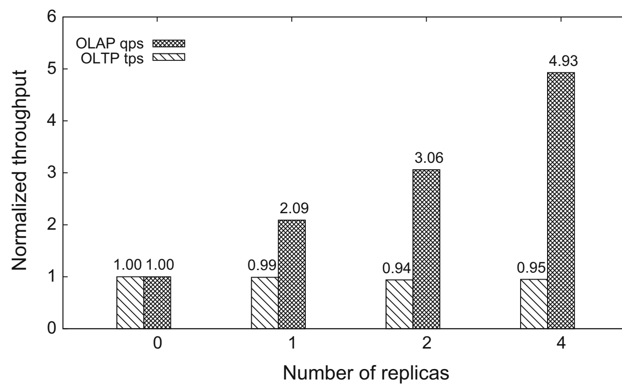

Fig. 14 TPC-CH throughput for varying the number of the replicas and TPC-H clients (normalized by the 0-replica throughput numbers respectively; N-replica configuration means that there are in total $N+1$ database servers including the primary)

while the TPC-H queries are evenly routed across all the available HANA database servers including the primary server. Note that each TPC-H client communicates with a designated database server and thus the delay by *wait-and-forward*, explained in Sect. 5.1, is not involved in this experiment.

Figure 14 shows the normalized throughput of OLTP and OLAP with the different number of replicas. N replicas denote that there are $N+1$ database servers including the primary. The normalized throughput was calculated by dividing the measured throughput by the throughput without replication. Although the OLTP throughput decreases slightly with the increasing number of replicas, the OLAP throughput increases almost linearly. This result confirms that ATR can offer scalable OLAP performance without creating notable performance overhead to OLTP workloads. Note that the OLAP throughput also shows slightly super-linear scalability when the number of replicas is 1 or 2. This is because the replayed transaction consumes less CPU compared to its original execution at the primary by the proposed lightweight log replay mechanism while the number of TPC-CH clients are configured so that the system is over-loaded. As a result, each replica has a larger OLAP capacity than the primary in terms of available CPU resources.

7 Eager parallel replication

7.1 Implementation

ATR can be extended to perform eager replication where it is ensured that the primary and its replicas have the same database state at the time of a query execution. However, under the eager replication, it is inevitable to pay additional performance overhead either at the write transaction side (*writer-pays-cost* approach) or at the replica-executed query side (*reader-pays-cost* approach).

In the *writer-pays-cost* approach, the primary write transaction commits after all of its changes are successfully applied to its replicas and thus the writer transaction's commit processing time can increase. In the *reader-pays-cost* approach, we let the read queries coming to the replicas pay the cost. The primary write transactions commits without waiting for its log application to replicas, following the commit protocol of the lazy replication. However, when a query is dispatched to a replica, it first reads its transaction snapshot timestamp at the primary by making an additional network round trip and then executes the query processing at the replica after waiting until the replica's last commit-replay timestamp equals to or becomes higher than the transaction snapshot timestamp read at the primary. In this approach, while the writer transaction's commit processing time does not increase, the replica-routed queries' execution time can increase due to the additional network round trip to acquire the primary-side transaction snapshot timestamp. In the current productive versions of SAP HANA (since HANA SPS 12 which was officially released in May 2016), the writer-pays-cost approach is incorporated but we do not exclude offering the reader-pays-cost approach alternatively in the near future.

Our writer-pays-cost eager replication inherits most of the ATR lazy replication implementation with adjusting its transaction commit protocol so that the primary transaction can be committed only after it is ensured that all of its DML changes are applied to the replica. With this approach, the eager replication implementation inherits the benefits of the proposed parallel log replay and the early log shipping mechanisms. Also, we avoided the expensive two-phase commit protocol by exploiting the proposed log-less replica recovery mechanism. That is, since we can still recover the eager replica contents by referring to their primary copies by using the algorithm presented in Sect. 4.1, the primary write transaction does not need to wait for the redo-logging at the replicas, which can be performed asynchronously in the background. Remark that the eager replication can co-exist in the same system with the lazy replication, meaning that some replica for a table can be defined as a lazy replica and another replica for the same table can be defined as a eager replica.

7.2 Experimental evaluation

To demonstrate the impact of the proposed parallel log replay under eager replication, we have compared (1) the primary running without any replica (labeled as *no-replica*), (2) the primary running with a lazy replica (labeled as *lazy*), and (3) the primary running with a eager replica (labeled as *eager*), in terms of TPC-C transaction throughput, as in Fig. 15. All

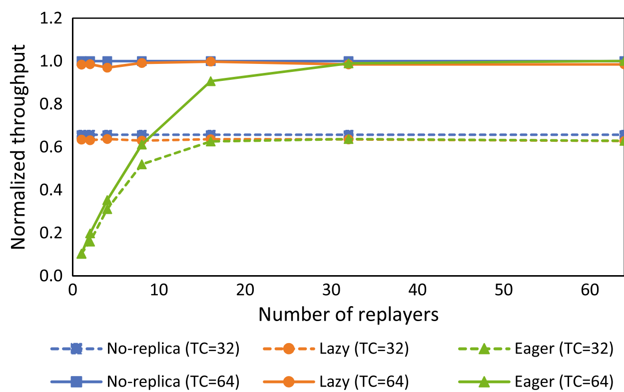

Fig. 15 TPC-C throughput with varying the number of the replayer threads and TPC-C clients (normalized by the 64-client no-replica throughput number)

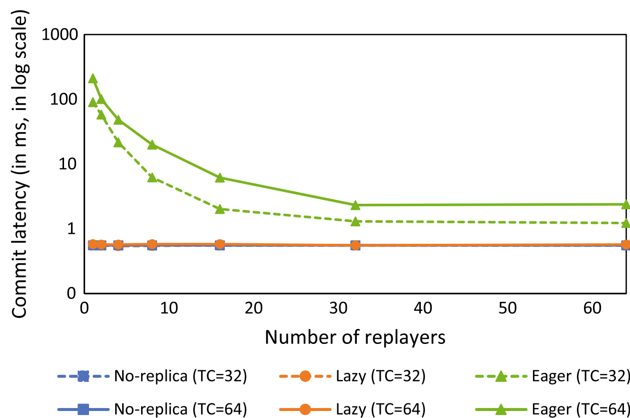

Fig. 16 Transaction commit latency measured during the same experiment with Fig. 15

the hardware and client configurations are identical to the ones described in Sect. 6.1.

The result shows that the primary transaction throughput is not affected by the replayer performance under the lazy replication, while the primary transaction throughput under the eager replication significantly drops as the degree of the replayer parallelism decreases. However, when sufficient number of replayer threads is assigned (16 or higher for TC = 32; 32 or higher for TC = 64), the eager replication does not show any notable performance drop compared to the lazy replication. According to our analysis, the following three factors contributes to such good performance of our eager replication implementation.

- First and most importantly, the *parallel log replay* mechanism proposed in this paper plays the key role as indicated by Fig. 15. If a replica is not able to catch up with the transaction processing throughput of the primary, then the replication requests will keep being delayed and it will end up with higher transaction commit time because a transaction cannot commit until its preceding DML operations are fully applied to the replicas.
- Second, the *eager log shipping* mechanism proposed in this paper is another important contribution factor. Since the DML log replay operations become overlapped with the next operations of the same transaction (as illustrated in Fig. 2), the amount of work that needs to be waited at the time of transaction commit processing can be significantly reduced.
- Third, with the *log-less replica recovery* mechanism proposed in this paper, the implemented eager replication involves only a single network round trip with its eager replica at the time of transaction commit differently from the expensive two-phase commit.

To see the internal behavior of the eager replication, we also measured the transaction commit latency during the same experiment. Figure 16 shows the result measured at the client side. With eager replication, the transaction commit time itself significantly increased, compared to lazy replication: for example, 549us at no-replica (TC = 32), 562us at lazy (TC = 32), and 1223us at eager (TC = 32). However, considering that a single transaction in our TPC-C benchmark consists of multiple client-server interactions, where each database query involves a network round trip between a client and the primary server, such increase at the commit time did not significantly affect the overall transaction processing throughput.

Even though our client-server benchmark configuration is to emulate a typical application-to-database configuration in SAP systems, remark that the increase of the commit latency by eager replication can turn into more notable performance drop particularly for the transactions written in stored procedures, which will be executed fully inside the database engine. In addition, note that the primary transaction performance under eager replication can be affected by the network latency between the primary and its replicas, differently from lazy replication. In order to further reduce the inevitable network delay occurring at the eager replication, using RDMA [28] is another considerable practical option, but it goes beyond the scope of this paper.

8 Potential future optimizations and extensions

In this section, we describe ongoing or potential future extensions of ATR and the technical issues that need to be addressed for them. Note that, at the time of writing this paper, most of the extensions discussed in this section are not yet fully incorporated in SAP HANA productive versions

but we present them to highlight flexibility and potentials of ATR.

8.1 Sub-table replication

So far, we have discussed ATR with assuming that the entire records and columns of a designated table are replicated. In addition to it, we are considering two additional levels of sub-table replication: vertical and horizontal sub-table replication. In *vertical sub-table replication*, only a few selected columns are replicated to replicas. In this configuration, replicas still maintain RVID values which are assigned per record. Using this RVID column, the matching replica records can be uniquely identified. The column filter is added to the primary side to exclude out unnecessary data during ATR log generation. On the other hand, in *horizontal sub-table replication*, only when the new after-update record image meets a predefined predicate (or called record filter), the corresponding ATR log entry is generated. In this configuration, the record filter should be applied also when a replica is initialized or recovered. Note that the vertical sub-table replication and the horizontal sub-table replication can be used together for the same table.

8.2 Replication across formats

In Sects. 2 and 3.1, we already presented that ATR can be used for replication across the OLTP-favored row store and the OLAP-favored column store based on the ATR's logical representation of the changed data set. By further leveraging the ATR's format-independent expressiveness, we can consider additional forms of cross-format replication configurations as follows.

First, ATR enables the primary table and its replicas to have different table partitioning schemes. For example, the primary table can be a non-partitioned row store table, and its replica table can be a range-partitioned column store table. Then, the incoming OLTP transactions can be processed without the additional partition-pruning overhead, while the OLAP queries can be processed (or parallelized) on its partitioned replica table more efficiently. Moreover, it is also possible that the partitions of a replica table is distributed to multiple nodes for leveraging more CPU resource for OLAP queries on the replica table. It can be also considered another form of multi-replica configuration compared to the configuration of having multiple redundant replicas of a primary table.

Second and similarly, ATR enables the primary table and its replicas to have different set of secondary indexes or different database configuration such as different checkpoint interval or different merge interval [11,20] from its delta storage to main storage.

Third, replication to a volatile temporary table is another thinkable extension option of ATR. With this option, even at a database node that does not necessarily have its own persistent storage volume, we can maintain a replica of a table and then use for scalable query processing.

8.3 Write workload scalability and semi-multi-master replication

The replication mechanism discussed so far falls into *master-slave replication* where a single primary server handling all the incoming write transactions which are again replicated to all the read-only replicas. This has been so far a preferred configuration of ATR due to the following reasons compared to *multi-master replication* where each replica can server not only read workloads but also write workloads.

- In order to make all the replicas execute the write transactions in the same order even against conflicting transactions, the multi-master replication may need to involve a complex consensus protocol or the increased possibility of multi-node deadlocks [13].
- The "scale-up" approach rather than the "scale-out" approach is selected for a high volume of OLTP workloads. For example, a large-scale ERP customer is running SAP HANA on top of a single hardware host having 16 CPU sockets.
- The "scale-out" approach for the OLTP workloads is still an option, but, in this case, careful table placement is needed to avoid the two-phase commit overhead incurred by multi-node write transactions.

Even though ATR basically takes the master-slave replication architecture due to the above reasons, it also offers the option of placing the master copies of tables in different database nodes. Still, write transactions for a particular table are directed to a particular database node, but write transactions for another table can be processed in a different database node in order to distribute write workloads to multiple nodes overall. We call this architecture *semi-multi-master replication* to distinguish from the plain forms of multi-master or master-slave replication architecture.

In this semi-multi-master replication of ATR, there is a possibility of multi-node deadlock, but it is automatically detected by using the existing multi-node deadlock detector of SAP HANA [23]. In addition, to automatically suggest optimal table placement for a given workload, we are also developing a workload-driven replication suggestion tool, similarly to [7]. By exploiting the SAP HANA Capture and Replay feature [40], the tool analyzes the captured workload and finds out whether the overall system performance (in terms of query performance, two-phase commit overhead, load balancing, and memory consumption) can be improved

with adjusting the primary table location and/or adding replicas to particular tables.

8.4 An alternative to handling log serialization error: log forwarding with out-of-order log replay

When a log serialization error is encountered, the corresponding DML replayer waits until the log serialization error is resolved, as described with Algorithm 1. Alternatively, it is also possible to forward the waiting-state log entry to a separate waiter queue in order to keep processing the next log entries in the normal DML queue. In this case, if a DML log entry of a transaction is forwarded to a waiter queue and then the next DML log entry of the same transaction is replayed in the original normal DML queue, then the DML operations of the same transaction can be replayed in a different order than their original execution order. However, with the proposed parallel log replay mechanism, this does not lead to any issue because the RVID-based record-level serialization holds even among the DML operations of the same transaction. If two DML operations of the same transaction touch the same database record and the earlier one is under the waiting status, then the next DML log entry will be also forwarded to the waiter queue. Remark that, in this scheme, the commit processing should be performed after it is ensured that all the DML log entries distributed to the waiter queue are replayed. It can be simply implemented by maintaining a reference counter per replayed transaction.

8.5 Replication log buffer management for better scalability at the primary

The proposed mechanism introduces a few new critical sections which are implemented by a lock-free structure using an atomic CAS instruction. The replication log buffer (Sect. 3.2) and the per-table RVID generator (Sect. 3.1) are such cases at the primary. Although a CAS operation might still be problematic on multi-socket hardware, it has been a practical and viable design choice considering that there are already several pre-existing critical sections such as recovery log buffer and commit timestamp generator [26]. In addition, even the per-table RVID generator pre-existed in SAP HANA for more efficient query processing.

In spite of this, in the future, if the multi-socket synchronization overhead of CAS operation becomes a more visible issue (as the number of hardware sockets increases), it is another option to create multiple DML log buffers (one per hardware socket, for example) also at the primary side, similarly to the DML log buffers maintained for parallel replay at the replica. Together with these parallel DML log buffers at the primary, a single commit log buffer might still be necessary to enforce the strict ordering of transaction commit log entries, but, by combining with the pre-existing group commit scheme, the access frequency to the shared commit log buffer can be further reduced.

8.6 Log size reduction

In order to reduce the size of the log that needs to be shipped, two potential optimizations can be considered. First, the discussed sub-table replication (Sect. 8.1) not only reduces the memory footprint required for the replica tables but also reduces the size of the replication log that needs to be shipped. Second, applying a loss-less data compression technique is another thinkable option to reduce the physical network usage between the primary and the replicas. To reduce the CPU consumption involved for compression and decompression, exploiting hardware-accelerated data compression technique can be applied as programmable NIC and FPGA are now widely deployed in datacenter-scale [27,38].

8.7 Transaction-consistent online replica creation

In order to deal with dynamic variation of incoming workloads, it is a desirable property to add or remove replicas without service downtime at run time. Particularly, with increasing demands of cloud computing and multi-tenant database systems, such *elasticity* [12,14,15,34,43] is becoming an essential requirement for modern database systems.

To enable adding replicas online and in a transaction-consistent manner, the following protocol is possible with ATR. For initializing the target replica table, the primary creates a table snapshot (or checkpoint) image and copies it to the replica system. And, for the new changes occurred during the snapshot creation and copy operation, a replication log generator is activated right before creating the snapshot. At the replica side, if the After-Update RVID of a replication log entry is smaller than or equal to the current RVID value of the target replica record, then the replication log entry is abandoned because its contained change is already available at the replica side. After the table copy operation completes, the metadata manager is notified of the existence of new replica. Then, on the next query execution to the corresponding table, the newly added replica becomes a possible candidate that is considered during query compilation.

In the current implementation of ATR in SAP HANA, the snapshot creation is performed by a normal transaction which follows the *snapshot isolation* consistency level. It means that the created snapshot contains only the committed database images. As a result, if there was an active transaction at the time of creating the table snapshot and the transaction had already performed update operations without ATR log generation, then those previous update operations are not reflected in any of the created table snapshots or the ATR log entries. To avoid this problem, when the snapshot creation

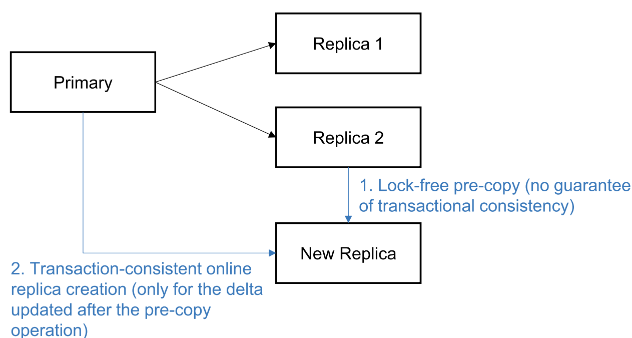

Fig. 17 Replica creation optimization under 1-to-n replication configuration

transaction gets its transaction snapshot timestamp (or read timestamp), a short-term table lock is acquired to disallow the running of any concurrent write transaction on the same table. Remark that the table lock is released as soon as the snapshot timestamp is acquired, even before the snapshot image of the table is created. Even though the described protocol is already implemented in SAP HANA productive version, we are in the course of eliminating another coarse-grained lock unnecessarily acquired by a pre-existing legacy code which physically copies the table snapshot, at the time of writing this paper.

8.8 Non-disruptive replica creation under 1-to-n replication

When there is already an active replica in the system, the table snapshot can be created not necessarily from the primary but from one of the existing replicas. Even though the existing replica can provide an outdated state of the table snapshot compared to the primary's, the gap can be filled by running the RVID-based recovery procedure of Algorithm 5 between the primary and the new replica as the next step, which is also illustrated in Fig. 17. With this optimization, we can save CPU and network resources of the primary system during the online replica addition and also we can minimize the interference to the active write transactions running at the primary.

9 Related work

9.1 Database replication

Database replication is a widely studied and popular concept for achieving higher availability and higher performance. There are a number of different replication techniques depending on their purposes or application domains.

Cross-datacenter system replication is an option for increasing high availability against datacenter outages [6,33,39]. For such a high availability purpose, even SAP HANA provides another different replication option, called HANA System Replication[39], which basically focuses on replicating entire database contents across data center. On the contrary, HANA ATR focuses on load balancing and scalable read performance by replicating a selected list of tables within a single data center although we do not exclude the possibility of extending ATR for the purpose of high availability or geo-replication.

When we need to allow the replication system to span heterogeneous database systems while decoupling the replication engine from the underlying DBMS servers or to transform the extracted source data as in ETL processing, *middleware-based database replication* has been another practical technique [2,4,10,35,41,42]. However, differently from those techniques, HANA ATR embeds the replication engines inside the DBMS kernel aiming at the real-time replication between HANA systems without making any additional hop during the replication.

Depending on where the incoming write workloads can be processed, there are two replication options: master-slave replication and multi-master replication. In the *multi-master replication* [2,3,10,17], each replica can serve both read and write workloads. However, in order to make all the replicas execute the write transactions in the same order even against conflicting transactions, the multi-master replication may need to involve a complex consensus protocol or the increased possibility of multi-node deadlocks [13]. Like [9,16,35], ATR takes the *master-slave replication* architecture, simplifying the transaction commit protocol and avoiding the danger of multi-master deadlocks. However, in contrast to [9,16,35], ATR employs the transparent and automatic routing protocol as explained in Sect. 2 so that the application developer need not be concerned about the location of the primary copy of a particular table. Additionally, based on its table-wise replication feature, ATR offers the option of the *semi-multi-master replication* as discussed in Sect. 8.3.

9.2 Lazy replication

Compared to [17] which relies on *eager (or synchronous) replication*, ATR basically follows *lazy (or asynchronous) replication* to reduce the overhead at the primary-side transaction execution as in [3,5,9,10,16,35]. However, differently from those other lazy replication techniques, ATR is optimized to reduce the visibility delay between the primary and its replicas by employing a couple of optimizations such as early log shipping and parallel log replay. Note that [2,35] discuss techniques to achieve stronger consistency under lazy replication by letting the replica-side read queries wait until certain conditions are met. It is similar to the eager replication option based on the *reader-pays-cost* model, discussed in

Sect. 7. Regarding the early log shipping, [32] also proposed a similar idea and showed how immediate update propagation, which does not wait for the commit of the write transaction, improves data freshness.

As also described in [44], lazy (or asynchronous) replication can be seen as a form of eventual consistency. [44] discusses about variations of eventual consistency model and [1] discusses about providing expected bounds on data staleness under eventual consistency model.

9.3 Parallel replication

For parallel replay under lazy replication, [16] relies on a run-time inter-transaction dependency tracker, which may become a contention point as shown in Sect. 6.2. Compared to such a *pessimistic parallel log replay* approach, ATR employs an *optimistic lock-free parallel log replay* algorithm by leveraging the record version ID of MVCC implementation. In [16], transactions belonging to the same *barrier* group can be committed out of order, but their changes become visible to the replica queries after all the transactions in the barrier group are replayed and committed. As a result, the barrier length can affect the log replay throughput and the visibility delay; for example, if the length of a barrier increases, the log replay throughput can increase, but the visibility delay may increase. In ATR, all the commit log replay operations are serialized by the single queue and single replayer, and the committed transaction results become immediately visible to the replica queries. In addition to the optimistic lock-free parallel log replay algorithm, with careful separation of the serialized portion of commit operations from the other parallelized DML, pre-commit, and post-commit operations, ATR achieves both high-throughput parallel log replay and shorter visibility delay.

9.4 Cross-format replication

The idea itself of scaling out mixed OLTP/OLAP workloads with replication is not a new one. For example, [31] extends Hyper [18] to achieve scalable analytics performance with a master-slave replication. Particularly, [31] takes a different approach than ATR in that [31] multicasts the redo log generated at the primary node, while ATR decouples the replication log from the redo log.

Most recently, [28] proposed BatchDB where OLTP and OLAP replicas can have different storage layouts to efficiently handle hybrid OLTP and OLAP workloads. In spite of having similar goals, ATR is clearly distinguished from BatchDB in its internal mechanisms. One of the key ideas of BatchDB is to enqueue OLAP queries at replicas and then execute at a time in batches to implicitly share the resource among the executed queries. Although BatchDB also uses RowID for fast application of updates to replicated records, there is no deeper discussion on parallel replication based on RVID or log-less replica recovery. In addition, while BatchDB focuses only on lazy replication, ATR is extended also to high-performance eager parallel replication enabled by ATR's optimizations for minimizing the propagation delay between the primary and its replicas. Remark that ATR implementation has been already available in productive versions of SAP HANA since its SPS 10 (released in July 2015) [22,25]. While both of [28,31] also discuss about replicating from a row store to a column store, we have extended the possible replication configurations into more dimensions as in Sect. 8.2.

Conventional logical logging mechanisms can also meet the need of cross-format replication naturally. However, as described in Sect. 2.2, SAP HANA recovery log format is tightly coupled with the physical format of the target table type. Additionally, changing the recovery log format of already deployed productive systems was not an easy practical option. Under this given background, we have decided to decouple and separate the replication log from the storage-level recovery log and then, it led to several subsequent unconventional-but-practical design choices like early log shipping, RVID-based parallel log replay and RVID-based log-less replica recovery.

Compared to such conventional logical logging mechanisms, there might be concern about maintaining two different code paths. However, especially based on the proposed log-less replica recovery mechanism, the storage-level recovery log of the primary system becomes the single point of truth for not only primary recovery but also replica recovery, which excludes the possibility of logical conflict or divergence between the primary and the replica during recovery.

9.5 Pub/sub-style logical replication

Databus [8] is a source-agnostic change data capture system, which provides ways of capturing data change events from a source system in a transaction-consistent order. For this, it relies on Trigger or a parser of binary-format recovery log (in case that the format is interpretable). With this inherent decoupling between the source system and the change capture system, Databus can also be used to offer the cross-format logical replication. However, our proposed replication mechanism is differentiated from [8] in the following aspects.

- Contrasted to ATR's in-database replication (Sect. 2.2), [8] uses an external process (called replay process) to capture changes from the primary database (by Trigger of by parsing the recovery log). In this sense, as pointed out in Sect. 2.2, the approach in [8] can involve an additional network round trip for replication, compared to the proposed ATR mechanism.

- Contrasted to push-based replication in ATR, [8] pulls the changes from the primary database first by the replay process and then again by the replication consumers (called subscripotion clients).
- Differently from ATR that relies on RVID for parallel log replay with record-wise partial ordering, [8] relies on a total ordering based on global commit timestamp to enforce the transactional ordering during replication. In addition, [8] does not discuss about any parallelism during replay. Moreover, because the commit timestamp is determined at the time of the transaction commit, it cannot take the advantage of the early log shipping proposed in this paper.

In a summary, [8] has a different design criteria from ATR in that it is designed primarily for focusing on reducing the propagation delay between the primary and its replicas.

Kafka [19] is another well-known pub/sub messaging-based replication system. From the authors' perspectives, Kafka is orthogonal to what we propose with ATR. Compared to Kafka which provides an intermediate store with producer and subscriber APIs, ATR is more about (1) how to generate change logs from a source database system for replication and about (2) how to replay the generated replication logs for a target database system. In that sense, it is not impossible to combine ATR with a messaging system like Kafka. The generated ATR log entries can be stored in a remote messaging system and then, replicas can be registered to the messaging system as subscribers. The replicas, instead of receiving the ATR log entries directly from the primary system, can receive the ATR log entries by using the messaging system's consumer API. After receiving them, the proposed ATR parallel replay scheme can be applied at the subscriber side for better efficiency.

9.6 ETL

Although we have already compared with conventional ETL-based replication approach in Sect. 1, it is worth mentioning that ETL is not only for replication but also for data transformation which helps accelerate reporting queries. However, contrasted to this conventional ETL-based OLAP system management, SAP has been pursuing a different vision and principle of having a common physical database schema across OLTP and OLAP systems without relying on intermediate data transformation layers between them [30,36]. This new architecture paradigm is beneficial not only for reducing the data propagation delay between OLTP and OLAP systems, but also for eliminating the application-side burden of maintaining the transformation rules between the physical database schema and the corresponding reporting queries used by BI tools. For more systematic and consistent mapping between the physical database schema and BI tool, SAP HANA has also offered to create a layered architecture of database views on top of the common database schema, instead of ETL-based application-managing data transformation, as described in more detail in [29,30]. Note that it is already possible to additionally create database views on top of the ATR replicas.

10 Conclusion

In this paper, we presented an efficient and scalable replication architecture called ATR in SAP HANA. We empirically showed that ATR enables real-time replication with sub-second visibility delay even for update-intensive workloads, showing scalable OLAP performance without notable overhead to the primary.

We first proposed the novel replication architecture for scaling out mixed workloads of OLTP and OLAP along with important design choices made. We then proposed an efficient, scalable log generation and parallel replay scheme. Here, the log buffer at log generation is implemented by a lock-free structure using an atomic CAS instruction, while a parallel log replayer exploits a novel, optimistic lock-free scheme by exploiting characteristics of MVCC. Specifically, we proposes two novel concepts of (1) the parallel log replay with RVID-based record-wise partial ordering and (2) the so-called optimistic interleaving technique for higher parallelism under high-conflict workloads. In order to support full-fledged replication in a commercial in-memory database system, we next proposed the RVID-based log-less post-failure replica recovery mechanism and presented various implementation issues including how to handle various transactional consistency issues under the proposed replication architecture.

Through extensive experiments with a concrete implementation available in a commercial main-memory DBMS product, we showed that ATR achieves sub-second visibility delay even for update-intensive workloads, providing scalable, OLAP performance without notable overhead to the primary. In addition, with extending to eager replication, we demonstrated how the ATR's parallel log replay and its log-less replica recovery mechanisms improve run-time transaction performance under eager replication.

Overall, we believe that we proposed a modern, practical, and production-grade replication architecture and also our comprehensive study for replication across formats lays a foundation for future research in scale-out in-memory database systems.

Acknowledgements The authors would like to acknowledge Hyejeong Lee, Deok Koo Kim, Kyungyul Park, Christian Bensberg, Martin Heidel, Joern Schmidt, Michael Muehle, Mihnea Andrei, Alexander Boehm and many other colleagues in HANA development team who supported and helped ATR development. Also, the authors would like to deeply

thank anonymous VLDB Journal reviewers who provided invaluable comments and suggested ideas to improve the contents.

References

1. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Quantifying eventual consistency with PBS. VLDB J. **23**(2), 279–302 (2014)
2. Bornea, M.A., Hodson, O., Elnikety, S., Fekete, A.: One-copy serializability with snapshot isolation under the hood. In: Proceedings of the 27th IEEE ICDE Conference, pp. 625–636 (2011)
3. Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., Silberschatz, A.: Update propagation protocols for replicated databases. In: Proceedings of the ACM SIGMOD Conference, pp. 97–108 (1999)
4. Cecchet, E., Candea, G., Ailamaki, A.: Middleware-based database replication: the gaps between theory and practice. In: Proceedings of the ACM SIGMOD Conference, pp. 739–752 (2008)
5. Chairunnanda, P., Daudjee, K., Özsu, M.T.: Confluxdb: multi-master replication for partitioned snapshot isolation databases. PVLDB **7**(11), 947–958 (2014)
6. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Googles globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8 (2013)
7. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proc. VLDB Endow. **3**(1–2), 48–57 (2010)
8. Das, S., Botev, C., Surlaker, K., Ghosh, B., Varadarajan, B., Nagaraj, S., Zhang, D., Gao, L., Westerman, J., Ganti, P., et al.: All aboard the databus!: Linkedin's scalable consistent change data capture platform. In: Proceedings of the Third ACM Symposium on Cloud Computing, p. 18. ACM (2012)
9. Daudjee, K., Salem, K.: Lazy database replication with snapshot isolation. In: Proceedings of the VLDB Conference, pp. 715–726 (2006)
10. Elnikety, S., Dropsho, S.G., Pedone, F.: Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In: Proceedings of the EuroSys Conference, pp. 117–130 (2006)
11. Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., Dees, J.: The SAP HANA database—an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)
12. Galante, G., de Bona, L.C.E.: A survey on cloud computing elasticity. In: 2012 IEEE Fifth International Conference on Utility and Cloud Computing (UCC), pp. 263–270. IEEE (2012)
13. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. ACM SIGMOD Rec. **25**(2), 173–182 (1996)
14. Heinze, T., Jerzak, Z., Hackenbroich, G., Fetzer, C.: Latency-aware elastic scaling for distributed data stream processing systems. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, pp. 13–22. ACM (2014)
15. Herbst, N.R., Kounev, S., Reussner, R.H.: Elasticity in cloud computing: what it is, and what it is not. In: ICAC, pp. 23–27 (2013)
16. Hong, C., Zhou, D., Yang, M., Kuo, C., Zhang, L., Zhou, L.: KuaFu: closing the parallelism gap in database replication. In: Proceedings of the 29th IEEE ICDE Conference, pp. 1186–1195 (2013)
17. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: Proceedings of the 26th VLDB Conference, pp. 134–143 (2000)
18. Kemper, A., Neumann, T.: Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of IEEE ICDE Conference, pp. 195–206 (2011)
19. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)
20. Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., Zeier, A.: Fast updates on read-optimized databases using multi-core CPUs. PVLDB **5**(1), 61–72 (2011)
21. Lee, J., Kim, K., Cha, S.K.: Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In: Proceedings of the 17th IEEE ICDE Conference, pp. 173–182 (2001)
22. Lee, J., Kim, K.H., Na, H.J., Park, C.G., Lee, H.: Rowid-based data synchronization for asynchronous table replication. US Patent App. 14/657,938 (2015)
23. Lee, J., Kwon, Y.S., Färber, F., Muehle, M., Lee, C., Bensberg, C., Lee, J.Y., Lee, A.H., Lehner, W.: SAP HANA distributed in-memory database system: transaction, session, and metadata management. In: Proceedings of the 29th IEEE ICDE Conference, pp. 1165–1173 (2013)
24. Lee, J., Moon, S., Kim, K.H., Kim, D.H., Cha, S.K., Han, W.S.: Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. PVLDB **10**(12), 1598–1609 (2017)
25. Lee, J., Park, C.G., Na, H.J., Kim, K.H.: Transactional and parallel log replay for asynchronous table replication. US Patent App. 14/657,948 (2015)
26. Lee, J., Shin, H., Park, C.G., Ko, S., Noh, J., Chuh, Y., Stephan, W., Han, W.S.: Hybrid garbage collection for multi-version concurrency control in SAP HANA. In: Proceedings of the ACM SIGMOD Conference, pp. 1307–1318 (2016)
27. Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., Zhang, L.: KV-direct: high-performance in-memory key-value store with programmable NIC. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 137–152. ACM (2017)
28. Makreshanski, D., Giceva, J., Barthels, C., Alonso, G.: BatchDB: efficient isolated execution of hybrid OLTP + OLAP workloads for interactive applications. In: Proceedings of the ACM SIGMOD Conference, pp. 37–50 (2017)
29. May, N., Böhm, A., Block, M., Lehner, W.: Managed query processing within the SAP HANA database platform. Datenbank-Spektrum **15**(2), 141–152 (2015)
30. May, N., Bohm, A., Lehner, W.: SAP HANA—the evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads. Datenbanksysteme für Business, Technologie und Web (BTW 2017) (2017)
31. Mühlbauer, T., Rödiger, W., Reiser, A., Kemper, A., Neumann, T., et al.: Scyper: a hybrid OLTP& OLAP distributed main memory database system for scalable real-time analytics. In: BTW, pp. 499–502 (2013)
32. Pacitti, E., Simon, E.: Update propagation strategies to improve freshness in lazy master replicated databases. VLDB J. **8**(3–4), 305–318 (2000)
33. Patterson, S., Elmore, A.J., Nawab, F., Agrawal, D., El Abbadi, A.: Serializability, not serial: concurrency control and availability in multi-datacenter datastores. PVLDB **5**(11), 1459–1470 (2012)
34. Perez-Sorrosal, F., Patiño-Martinez, M., Jimenez-Peris, R., Kemme, B.: Elastic SI-Cache: consistent and scalable caching in multi-tier architectures. VLDB J. **20**(6), 841–865 (2011)
35. Plattner, C., Alonso, G.: Ganymed: Scalable replication for transactional web applications. In: Proceedings of the ACM USENIX Middleware Conference, pp. 155–174 (2004)
36. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: Proceedings of the ACM SIGMOD Conference, pp. 1–2. ACM (2009)
37. Psaroudakis, I., Wolf, F., May, N., Neumann, T., Böhm, A., Ailamaki, A., Sattler, K.U.: Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In: Technology Confer-

ence on Performance Evaluation and Benchmarking, pp. 97–112. Springer (2014)
38. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp. 13–24. IEEE (2014)
39. SAP: high availability for SAP HANA. https://archive.sap.com/documents/docs/DOC-65585
40. SAP: SAP HANA capture and replay tool. https://blogs.sap.com/2016/06/14/introducing-the-new-sap-hana-capture-and-replay-tool-available-with-sap-hana-sps12/
41. SAP: SAP LT (SLT) replication server. http://www.sap.com/community/topic/lt-replication-server.html
42. Simitsis, A., Vassiliadis, P., Sellis, T.: Optimizing ETL processes in data warehouses. In: Proceedings of the 21st IEEE ICDE Conference, pp. 564–575 (2005)
43. Sousa, F.R., Machado, J.C.: Towards elastic multi-tenant database replication with quality of service. In: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, pp. 168–175. IEEE Computer Society (2012)
44. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
45. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Elsevier, Amsterdam (2001)
46. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc. VLDB Endow. **2**(1), 385–394 (2009)