# Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads

Juchang Lee
SAP Labs Korea
Seoul, Korea
juc.lee@sap.com

SeungHyun Moon
Pohang University of Science and Technology (POSTECH)
Pohang, Korea
shmoon@dblab.postech.ac.kr

Kyu Hwan Kim
SAP Labs Korea
Seoul, Korea
kyu.hwan.kim@sap.com

Deok Hoe Kim
SAP Labs Korea
Seoul, Korea
deok.hoe.kim@sap.com

Sang Kyun Cha
Seoul National University
Seoul, Korea
chask@snu.ac.kr

Wook-Shin Han*
Pohang University of Science and Technology (POSTECH)
Pohang, Korea
wshan@dblab.postech.ac.kr

## 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 of OLAP query performance. In order to tackle this challenging problem, we propose a novel database replication architecture called 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) in order to enable real-time reporting by minimizing the propagation delay between the primary and replicas. Through extensive experiments with a concrete implementation available in a commercial database system, we demonstrate that ATR achieves sub-second visibility delay even for update-intensive workloads, providing scalable OLAP performance without notable overhead to the primary.

*corresponding author


*Proceedings of the VLDB Endowment,* Vol. 10, No. 12


## 1. INTRODUCTION

Modern database systems need to support mixed workloads of online transaction processing (OLTP) and online analytical processing (OLAP) workloads [11, 17, 18]. 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 ETL (Extraction-Transformation-Loading) [20, 21]. However, this ETL-style, application-driven data replication between two different OLTP system 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 [11, 18] focuses on scaling up mixed workloads in a single hardware host, which eventually limits the maximum scalability of analytical query processing.

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.

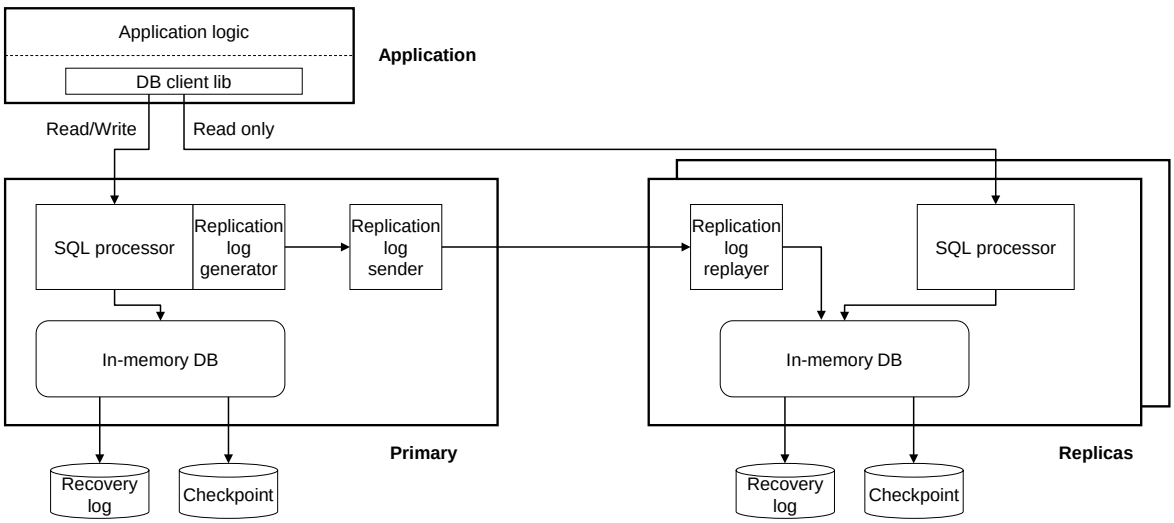


**Figure 1: Overall architecture.**

Data replication is a widely studied and popular mechanism for achieving higher availability and higher performance [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16]. 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 [3], 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 [9] 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 elastic scalability of the analytical query performance and shows less CPU consumption for replayed transactions. Thus, it can minimize the propagation (or snapshot) delay between the primary and replicas.

Our contributions are summarized as follows: 1) Through deep analysis in design requirements and decisions, we propose a novel database replication architecture for real-time analytical queries on operational data. 2) We propose a novel optimistic lock-free parallel log replay scheme which exploits characteristics of multi-version concurrency control (MVCC) and minimizes the propagation delay. 3) We propose a framework for adaptive query routing depending on its predefined max acceptable staleness range. 4) Through extensive experiments with a concrete implementation available in a commercial product, SAP HANA [17], we show that ATR provides sub-second visibility delay even for write-intensive workloads, achieving scalable, OLAP performance without notable overhead to the primary.

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 Section 4, we present a post-failure replica recovery mechanism and ATR's various implementation issues. Section 5 presents the results of performance evaluations, and Section 6 gives an overview of related work. Section 7 summarizes and concludes the paper.

# 2. ARCHITECTURE

## 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 queriable and transactionally-consistent state. The database client library transparently 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 capability of processing mixed OLTP

and OLAP workloads under the common database schema and under 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 decisions 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).

**Table 1: Summary of ATR design decisions.**

**(a) Common**

| | |
|---|---|
| D1.1 | Replicate across formats (across row store format and column store format) |
| D1.2 | Decouple and separate the replication log from the storage-level recovery log |
| D1.3 | Tightly couple the replication log generator and sender within the DBMS engine |

**(b) Primary server**

| | |
|---|---|
| D2.1 | Log the record-level SQL execution result to avoid non-deterministic behaviors and the potential conflict during parallel log replay |
| D2.2 | Ship generated replication log entries as soon as execution of its DML statement is completed |

**(c) Replicas**

| | |
|---|---|
| D3.1 | Enable to 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 | Support efficient post-failure replica recovery |

Now, we explain each decision in detail and elaborate its rationale. First, ATR replicates across different table formats (across row store format and column store format) (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 of 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 [12]). 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*) (D2.1). 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 the so-called RVID (record version ID), which will be explained in more detail in Section 3.3.

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

Fourth, although ATR supports both *lazy* (or called *asynchronous*) replication and *eager* (or called *synchronous*) replication [6], 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 embedded inside the database engine (D1.3) instead of relying on an external application-driven replicator like ETL tool [21] or middleware-based replication [3] 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 (D2.2) even before the transaction is completed, differently from [9]. As illustrated in Figure 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 database systems employing the Optimistic Concurrency Control [22], SAP HANA can show relatively lower abort ratios because it relies on pessimistic write locks for concurrency control among

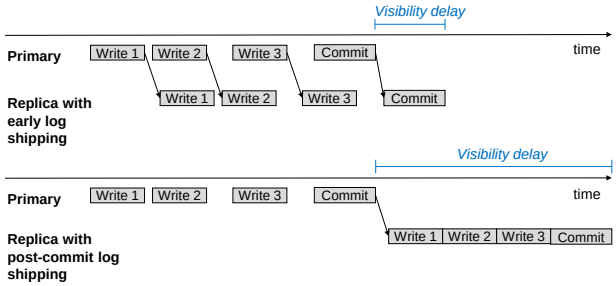


**Figure 2: Early log shipping vs. Post-commit log shipping.**

the write transactions. Notice that the read queries in SAP HANA do not require any lock based on the MVCC implementation [14].

- *Parallel log replay*: ATR performs parallel log replay in replicas to minimize visibility delay (D3.1). As SAP HANA is typically deployed to shared-memory multiprocessor architectures, 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 log replay scheme which is explained in Section 3.

Fifth, together with the above approaches for reducing the visibility delay, ATR also 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.

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 (D3.3), which will be explained in Section 4.1.

# 3. LOG GENERATION AND REPLAY

After describing the structure of the replication log entries (Section 3.1), this section presents how they are generated by the primary server (Section 3.2) and then replayed by the replica server in parallel (Section 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 precommit 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 replication log entries to the parallel log replayers, which will be explained in more detail in Section 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 for quickly locating 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.

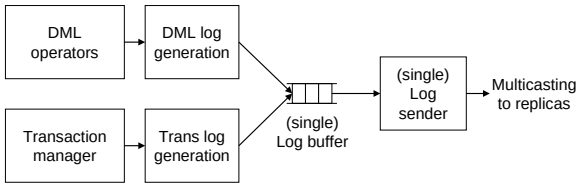


**Figure 3: Log generator and sender.**

## 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 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 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.

## 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 Figure 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 precommit 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 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

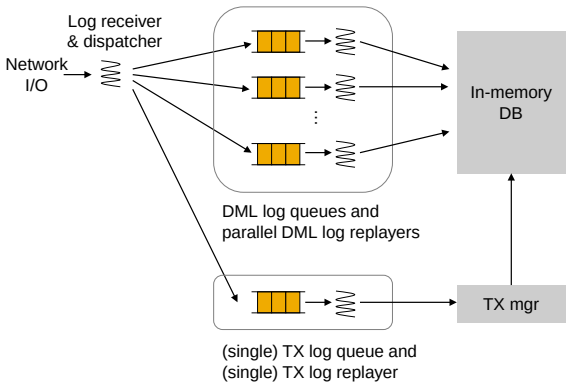


**Figure 4: Parallel log replay.**

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 to 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 [9], 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 [9], 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 rereading the target database record (lines 9 to 15 and 17 to 23 in Algorithm 1).

In order to correctly detect the log serialization error, ATR exploits the characteristics of the MVCC implementation of 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, it means that the preceding DML operation for the 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 Figure 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 Figure 5. Under this scenario, after replaying Log1 and Log2, Log5 can be encountered by

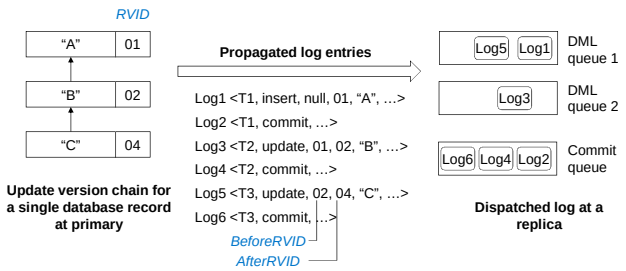


**Figure 5: Parallel log replay example.**

a DML replayer before Log3 is replayed. However, while trying to replay Log5, the DML replayer recognizes that there is no record version whose RVID is equals to Log5'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 Implementation Issues

### 3.4.1 DML Replay

The detailed DML log replay algorithm is presented in Algorithm 1. In this algorithm, note that the DML replay operation skips the integrity constraint check because it was already done at the primary. It also skips the record locking because there are no other concurrent write transactions in the replica except the other DML log replayer and the transaction serialization is ensured by checking RVID visibility among the DML log replayers, as discussed in Section 3.3. Due to the skipped integrity check during parallel log replay, it is possible that 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.

### 3.4.2 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, *precommit*, *commit*, and *postcommit*, and then delegate the precommit work to the parallel DML log replayers by using the precommit log entry and delegates the postcommit work to asynchronous background threads. The precommit 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 4. Right after this operation, 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.

Note that, during the DML log replay, the recovery redo and undo log entries are generated for the recovery of the

**Algorithm 1** Replay a DML log entry ($\tau$, $\beta$, and $\alpha$ denote TableID, Before-update RVID, and After-update RVID, respectively.)

**Require:** A DML log entry $L$.
Find the transaction object $T$ for $L.TransactionID$.
**if** $T$ is empty **then**
  Create a transaction object for $L.TransactionID$.
**end if**
**if** $L.OperationType = Insert$ **then**
  Insert $L.Data$ into the table $L.\tau$.
  Set the inserted record's $RVID$ as $L.\alpha$.
**else if** $L.OperationType = Delete$ **then**
  **while** true **do**
    Find the record version $R$ whose $RVID$ equals to $L.\beta$ in the table $L.\tau$.
    **if** $R$ is not empty **then**
      Delete $R$. **return**
    **end if**
  **end while**
**else if** $L.OperationType = Update$ **then**
  **while** true **do**
    Find the record version $R$ whose $RVID$ equals to $L.\beta$ in the table $L.\tau$.
    **if** $R$ is not empty **then**
      Update $R$ with $L.Data$ and $L.\alpha$. **return**
    **end if**
  **end while**
**end if**

**Algorithm 2** Replay a precommit log entry

**Require:** A precommit log entry $L$.
Find the transaction object $T$ for $L.TransactionID$.
Mark $T$ś state as *precommitted*.

**Algorithm 3** Replay an abort log entry

**Require:** An abort log entry $L$.
Find the transaction object $T$ for $L.TransactionID$.
Abort $T$.

**Algorithm 4** Replay a commit log entry

**Require:** A commit log entry $L$.
Find the transaction object $T$ for $L.TransactionID$.
Wait until $T$ś state becomes *precommitted*.
Increment the transaction commit timestamp of the replica server by marking the $T$ś generated record versions with a new commit timestamp value.

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. The detailed recovery algorithm will be explained in Section 4.

#### 3.4.3 Query Processing at Replicas

Queries running at the replicas just follows the existing visibility rule of MVCC in SAP HANA. When a query starts, it takes its snapshot timestamp value from the replica commit timestamp which is incremented by the commit log replayer as in Algorithm 4. 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. The query visibility decision protocol is also described in [14].

# 4. RECOVERY AND OTHER IMPLEMENTATION ISSUES

## 4.1 Post-Failure 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.

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** Re-synchronize 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

For example, in the state of Figure 6, $P = \{r_1, r_3, r_5, r_9\}$ 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.

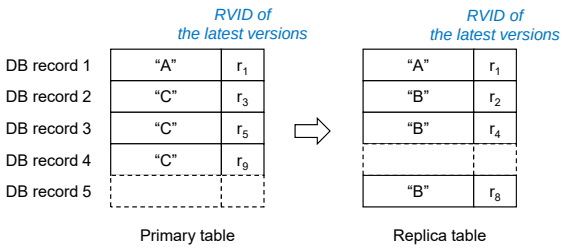


**Figure 6: Post-failure replica recovery.**

Although comparing the entire RVID columns of two tables looks expensive at a glance, we took this approach especially considering the characteristics of SAP HANA in-memory column store. Because the RVID column values of the entire table are stored on a contiguous memory in a compressed form [17] in SAP HANA column store, scanning the entire RVID column values of a column store table can be done rapidly.

## 4.2 Other Implementation Issues

This section presents a few other implementation issues or corner cases that we have additionally addressed during the course of implementing the ATR architecture.

#### 4.2.1 Run-time Error Handling

In addition to the type of failures that have been discussed in Section 4.1, a run-time error such as an out-of-memory exception can also happen in the middle of log replay. Our simple and practical solution is to switch off the replication for the particular problematic table after retrying during a predefined time period. If a particular replica is switched off, then the next queries to the problematic replica table are automatically re-routed to the other replicas or its primary table without disruption of the overall service.

#### 4.2.2 DDL Transaction Replication

Following the SAP HANA distributed system architecture [13], 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 entries for the table are replayed.

#### 4.2.3 Routing Read-own-write Transactions

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. Regarding this problem, two practical solutions are considered. First, it can be solved by maintaining additional watermarks incremented on every DML and by letting the replica-routed query check whether the sufficient number of DML logs are replayed. Or, it can simply be avoided by maintaining the changed table list for the active transaction and then directly routing such detected read-own-write queries to the primary. In the current production version of ATR, the second approach is available.

# 5. EXPERIMENTS

In this section, with the following experiment goals, we provide our performance evaluation result 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 algorithm which relies on a run-time inter-transaction dependency tracker (Section 5.2).
- Based on its optimistic parallel log replay, ATR shows sub-second visibility delay in the given update-intensive benchmark (Section 5.3).
- Regardless of transaction conflict ratio, ATR log replayer constantly shows higher throughput than the primary (Section 5.4).
- The overhead of ATR at the primary is not significant in terms of primary-side write transaction throughput and CPU consumption (Section 5.5).
- 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 (Section 5.5).
- Finally, with the increasing number of replicas, ATR shows scalable OLAP performance with no notable overhead to the OLTP side (Section 5.6).

## 5.1 Experimental Setup

ATR has been successfully incorporated in the SAP HANA production version since its SPS 10 (released in June 2015). 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 [9].

To generate a OLTP and OLAP mixed workload, we used the same TPC-CH benchmark program as the one used in [18]. The benchmark program runs both TPC-C and TPC-H workloads simultaneously over the same data set, after initially populating 100 warehouses as in [18]. 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 [18], the absolute numbers for the TPC-CH benchmark are not disclosed but normalized by undisclosed constants, except for the micro-benchmark results conducted in Section 5.4 and Section 5.5.

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 40Gbit/sec. 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 Section 5.6, we scale up to four replica servers with one primary server and one client machine while the other experiments focus on single-replica configuration.

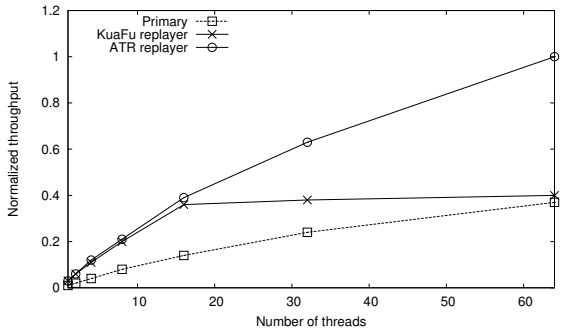


**Figure 7: TPC-C throughput over the number of threads (normalized by the 64-thread ATR replayer throughput).**

## 5.2 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 7 shows the experimental results. The 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 the KuaFu-style parallel replay algorithm based on our best understanding of their paper [9]. 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 Figure 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*[9] 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 Figure 7. 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 [9] 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.

## 5.3 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 seconds. 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 Section 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 are dynamically configured to the same number with the number of TPC-C clients.

Figure 8 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 seconds) due to the performance mismatch between the primary log generation and the replica log replay, as also indicated by Figure 7.

## 5.4 Impact of Inter-transaction Conflict

As explained in Section 3.3, the ATR replayer can waste CPU cycles for its retry operation especially when multiple replayer threads try to update the same record. 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 40 clients concurrently run update transactions on top of the table while varying the initial table size from 1 to 1 million records. Each update transaction commits after repeating the following update statement 10 times. The 10 primary keys used for each transaction are picked up randomly from the key range of the initially populated data

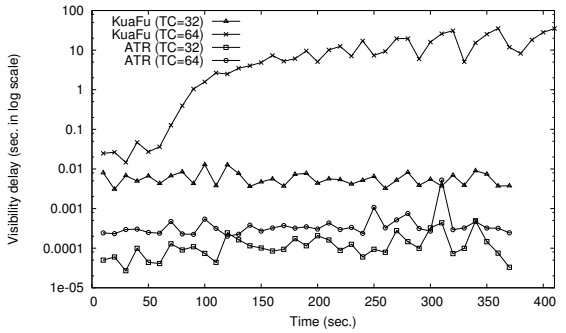


**Figure 8: Visibility delay at the replica in log scale while running TPC-C benchmark at the primary (TC denotes the number of TPC-C clients).**

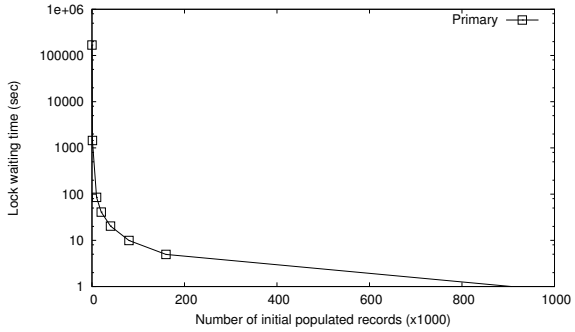


**Figure 9: Accumulated lock waiting time at the primary over the initiale table size.**

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.

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

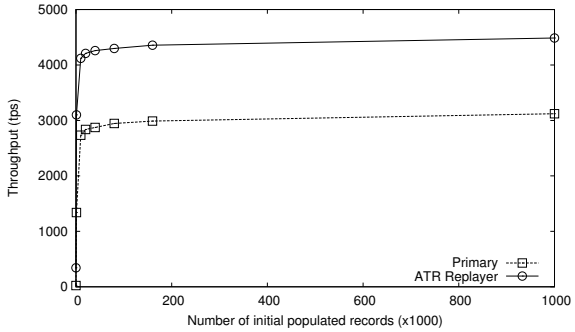


**Figure 10: Micro-benchmark throughput over the conflict ratio.**

**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 |

To confirm whether our designed micro-benchmark emulates the conflict ratio well, we first measured the accumulated lock waiting time throughout the benchmark for all the clients, as shown in Figure 9. As expected, depending on the initial table size, the conflict ratio varied significantly. The larger lock waiting time means a higher transaction conflict ratio because transactions have to wait until they acquire a lock when there exist transaction conflicts. For example, when the number of the initially populated records is set to one, all transactions are serially executed due to conflicting transactions with each other.

As the next step, we have measured the throughput of the ATR replayer and the primary, varying the initial table size. As in Section 5.2, we measured the log replay throughput after pre-generating and pre-loading the replication log entries. Figure 10 shows the experimental results. ATR constantly shows higher throughput than the primary regardless of the conflict ratio. Note that, in Figure 10, compared to Figure 7, the gap between the ATR replayer and the primary is smaller since Figure 7 was measured with the TPC-C benchmark consisting of read/write workloads while Figure 10 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.

## 5.5 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 Section 5.4 while populating 1 million records initially. However, differently from Section 5.4 where the replayers run with the pre-generated replication log, we measured the actual performance with the log online-replicated from the primary.

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 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 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 not so significant.

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). Again, according to our CPU profiling analysis, the CPU consumption at the replica was saved since the replayer skips the integrity constraint check and the record locking/unlocking during the log replay as mentioned in Section 3.4. Furthermore, the skipped primary key search at the replica is another important contributor to such CPU cost savings. Note that the target record is found by a simple hash operation with the 8-byte *Before-Update RVID* value at the replica 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. 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 Section 5.6.

## 5.6 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 while the TPC-H queries are evenly routed across all the available HANA database servers including the primary server.

Figure 11 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

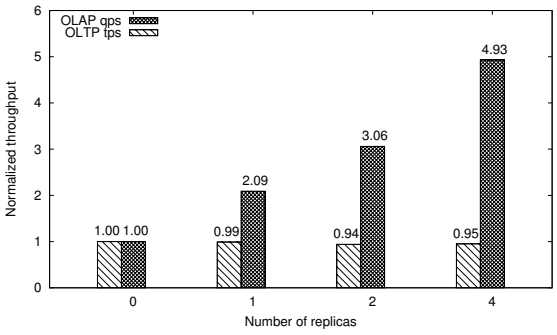


**Figure 11: TPC-CH throughput for varying the number of the replicas and TPC-H clients (normalized by the 0-replica throughput numbers respectively; N-replica configuration means that there are in total N+1 database servers including the primary).**

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 superlinear 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, as discussed in Section 5.5. As a result, each replica has a larger OLAP capacity than the primary in terms of available CPU resources.

# 6. RELATED WORK

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 [5, 15, 19]. For such a high availability purpose, even SAP HANA provides another different replication option, called HANA System Replication[19], 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 [1, 3, 7, 16, 21, 20]. 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* [1, 2, 7, 10], 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 [8]. Like [6, 9, 16], ATR takes the *master-slave replication* architecture, simplifying the transaction commit protocol and avoiding the danger of multi-master deadlocks. However, in contrast to [6, 9, 16], ATR employs the transparent and automatic routing protocol as explained in Section 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 placing the master copy 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 for distributing the 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. [4] can also be classified as a semi-multi-master replication like ATR. 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 [13]. A detailed description on the semi-multi-master configuration goes beyond the scope of this paper.

Compared to [10] which relies on *eager (or synchronous) replication*, ATR follows *lazy (or asynchronous) replication* to reduce the overhead at the primary-side transaction execution as in [2, 4, 6, 7, 9, 16]. 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.

For parallel replay under lazy replication, [9] relies on a run-time inter-transaction dependency tracker, which may become a contention point as shown in Section 5.2. 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 [9], 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.

# 7. 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. Particularly, the novel concepts of SessionID-based log dispatch method and RVID-based dynamic detection of serialization error were proposed. In order to support full-fledged replication in a commercial in-memory database system, we next proposed the RVID-based post-failure replica recovery mechanism and presented various implementation issues.

Through extensive experiments with a concrete implementation available in a commercial 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. Overall, we believe that our comprehensive study for replication across formats lays a foundation for future research in scale-out in-memory database systems.

# 8. ADDITIONAL AUTHORS

Chang Gyoo Park (SAP Labs Korea, `chang.gyoo.park@sap.com`), Hyoung Jun Na (SAP Labs Korea, `hyoung.jun.na@sap.com`) and Joo Yeon Lee (SAP Labs Korea, `joo.lee@sap.com`)

# 9. REFERENCES

[1] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *IEEE ICDE conference*, pages 625–636, 2011.

[2] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD conference*, pages 97–108, 1999.

[3] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *ACM SIGMOD conference*, pages 739–752, 2008.

[4] P. Chairunnanda, K. Daudjee, and M. T. Özsu. Confluxdb: Multi-master replication for partitioned snapshot isolation databases. *PVLDB*, 7(11):947–958, 2014.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8, 2013.

[6] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB conference*, pages 715–726, 2006.

[7] S. Elnikety, S. G. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys conference*, pages 117–130, 2006.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.

[9] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou. Kuafu: Closing the parallelism gap in database replication. In *IEEE ICDE conference*, pages 1186–1195, 2013.

[10] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement database replication. In *VLDB conference*, pages 134–143, 2000.

[11] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *IEEE ICDE conference*, pages 195–206, 2011.

[12] J. Lee, K. Kim, and S. K. Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *IEEE ICDE conference*, pages 173–182, 2001.

[13] J. Lee, Y. S. Kwon, F. Färber, M. Muehle, C. Lee, C. Bensberg, J. Y. Lee, A. H. Lee, and W. Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *IEEE ICDE conference*, pages 1165–1173, 2013.

[14] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *ACM SIGMOD conference*, pages 1307–1318, 2016.

[15] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[16] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM USENIX middleware conference*, pages 155–174, 2004.

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

[18] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K.-U. Sattler. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 97–112. Springer, 2014.

[19] SAP. High availability for SAP HANA. `https://archive.sap.com/documents/docs/DOC-65585`.

[20] SAP. SAP LT (SLT) replication server. `http://www.sap.com/community/topic/lt-replication-server.html`.

[21] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing ETL processes in data warehouses. In *IEEE ICDE conference*, pages 564–575, 2005.

[22] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.