Asymmetric-Partition Replication for Highly Scalable Distributed Transaction Processing in Practice

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

Hyejeong Lee
SAP Labs Korea
hyejeong.lee@sap.com

Seongyun Ko
POSTECH
syko@dblab.postech.ac.kr

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

Mihnea Andrei
SAP Labs France
mihnea.andrei@sap.com

Friedrich Keller
SAP SE
friedrich.keller@sap.com

Wook-Shin Han*
POSTECH
wshan@dblab.postech.ac.kr

ABSTRACT

Database replication is widely known and used for high availability or load balancing in many practical database systems. In this paper, we show how a replication engine can be used for three important practical cases that have not previously been studied very well. The three practical use cases include: 1) scaling out OLTP/OLAP-mixed workloads with partitioned replicas, 2) efficiently maintaining a distributed secondary index for a partitioned table, and 3) efficiently implementing an online re-partitioning operation. All three use cases are crucial for enabling a high-performance shared-nothing distributed database system. To support the three use cases more efficiently, we propose the concept of *asymmetric-partition replication*, so that replicas of a table can be independently partitioned regardless of whether or how its primary copy is partitioned. In addition, we propose the *optimistic synchronous commit* protocol which avoids the expensive two-phase commit without sacrificing transactional consistency. The proposed asymmetric-partition replication and its optimized commit protocol are incorporated in the production versions of the SAP HANA in-memory database system. Through extensive experiments, we demonstrate the significant benefits that the proposed replication engine brings to the three use cases.

PVLDB Reference Format:
Juchang Lee, Hyejeong Lee, Seongyun Ko, Kyu Hwan Kim, Mihnea Andrei, Friedrich Keller, Wook-Shin Han. Asymmetric-Partition Replication for Highly Scalable Distributed Transaction Processing in Practice. *PVLDB*, 13(12): 3112-3124, 2020.
DOI: https://doi.org/10.14778/3415478.3415538

*Corresponding author

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415538

1. INTRODUCTION

Database replication is one of the most widely studied and used techniques in the database area. It has two major use cases: 1) achieving a higher degree of system availability and 2) achieving better scalability by distributing the incoming workloads to the multiple replicas. With geographically distributed replicas, replication can also help reduce the query latency when it accesses a local replica.

Beyond those well-known use cases, this paper shows how an advanced replication engine can serve three other practical use cases that have not been studied very well. In addition, in order to lay a solid foundation for such extended use cases of replication, we propose the novel notion of *asymmetric-partition replication* where replicas of a table can be independently partitioned regardless of whether or how the primary copy of the table is partitioned. Note that, in this paper, the term *table partition* denotes a part (or a subset) of such a partitioned table.

Furthermore, to address a challenge encountered while applying the asymmetric-partition replication to a practical system, we propose a novel optimization called *optimistic synchronous commit* that avoids the expensive two-phase commit protocol while preserving the strict transactional consistency. This optimization is possible by fully exploiting the fact that the replica table is a derivable and reconstructable data structure based on the corresponding primary table data.

The three practical use cases are presented as follows.

1.1 Independently Partitioned Replica

First, when handling both OLTP and OLAP workloads in the same database system, asymmetric-partition replication enables the system to maintain an updateable primary copy of a table in a single, larger physical node, while its replica copies are independently partitioned and distributed across multiple smaller physical nodes, as shown in Figure 1. With this architecture, we can avoid the cross-partition two-phase commit for OLTP workloads, while serving partition-friendly OLAP workloads in a more scalable way with the multiple smaller replica nodes.

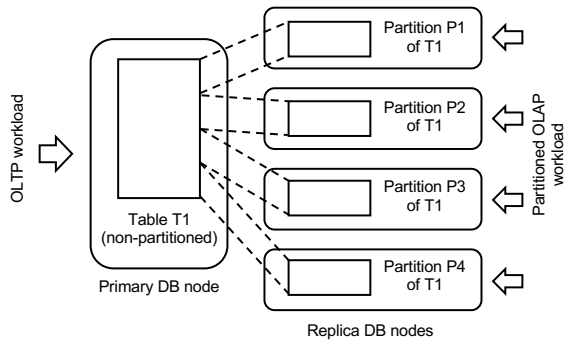

Figure 1: Scalable Processing of OLTP/OLAP-mixed workloads (Use Case 1).

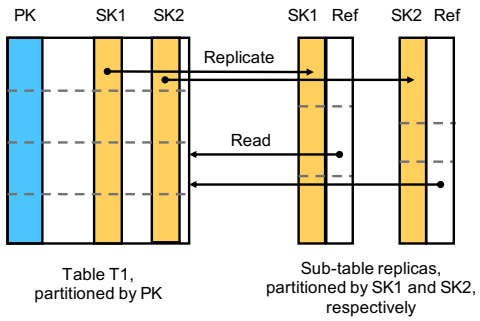

Figure 2: Partitioned Secondary Index with Asymmetric-Partitioned Replication (Use Case 2).

A recent work with an academic prototype [17] suggested a similar replication architecture, called STAR. In the proposed replication architecture, partitions of a primary table are co-located in the same node while partitions of a replica table can be scattered across multiple nodes. However, it assumes that the primary and replica copies of a table are still partitioned in the same layout. Compared to that, our proposed asymmetric-partition replication allows the replica tables to be differently partitioned and differently distributed from the primary table. As a result, in our proposed architecture, the replica table can be re-partitioned and re-distributed without affecting the primary table's partitioning scheme.

1.2 Distributed Secondary Index

The second use case of asymmetric-partition replication is to efficiently maintain a secondary index for a partitioned and distributed table. When a table is partitioned by its primary key or its subset, it is relatively easy to identify a target partition for an incoming point query that includes the primary key predicate. Its partition pruning can be performed with the input parameter value of the predicate at either the client library or the server that initially received the query. However, when a point query needs to be evaluated with a secondary key, then its target partition is not so obvious, since the table is partitioned by the primary key. That can force the query to access all the database nodes to identify the corresponding table partition.

Asymmetric-partition replication provides a convenient and efficient solution to the distributed secondary index management problem. A secondary index for a partitioned table can be modeled as a sub-table replica of its primary table. The sub-table replica can be considered a vertical sub-table replica, since it only needs to contain a few necessary columns from the primary table: secondary key columns and a reference (source table partition ID and row ID) pointing to the corresponding primary table record. Subsequently, the sub-table replica can be partitioned by the secondary key itself, not necessarily with the same partitioning key as its primary table. This architecture is illustrated in Figure 2 with an example of the two secondary indexes (for two secondary keys, SK1 and SK2).

One may implement the described distributed secondary index from scratch without relying on a replication engine, but it requires significant time and developer resources. Moreover, the proposed optimistic synchronous commit protocol enables elimination of the expensive two-phase commit cost when a change made at a primary table needs to be synchronized with its remote sub-table replica.

1.3 Online Table Re-partitioning

The third use case of asymmetric-partition replication involves re-partitioning an existing table online. To support dynamic workloads, the system needs to be able to adjust the partitioning scheme of an existing table dynamically. One solution uses a table-level exclusive lock, but this would block other concurrent DML transactions during the re-partitioning operation. Thus, the re-partitioning should be processed very efficiently.

Here again, asymmetric-partition replication provides a convenient and efficient solution. The online repartitioning operation can be logically mapped to a sequence of operations: 1) creating an online replica by copying the table snapshot, 2) replicating delta changes that are incoming to the original primary table during the snapshot copy operation, 3) transferring the primary ownership of the table to the new replica, and 4) dropping the original primary table in the background. In this procedure, asymmetric-partition replication allows the new replica to have a partition scheme different from that of its original table. Similar to the second use case, this type of online DDL can be implemented from scratch, but we can save development time and resources by leveraging asymmetric-partition replication.

1.4 Contribution and Paper Organization

The key contribution of this paper is as follows.

- We propose the novel notion of *asymmetric-partition replication*, so that replicas of a table can be independently partitioned regardless of whether or how the primary copy of the table is partitioned.
- We show how asymmetric-partition replication enables a scalable distributed database system - economically and efficiently. More specifically, we describe 1) how OLTP and OLAP workloads can efficiently be processed with the proposed replication architecture, 2) how the secondary key access query for a partitioned table can be accelerated in distributed systems, and 3) how the online repartitioning DDL operation can be implemented while minimizing its interference with other concurrent DML transactions.
- To overcome the challenge in the write performance overhead encountered while applying the replication

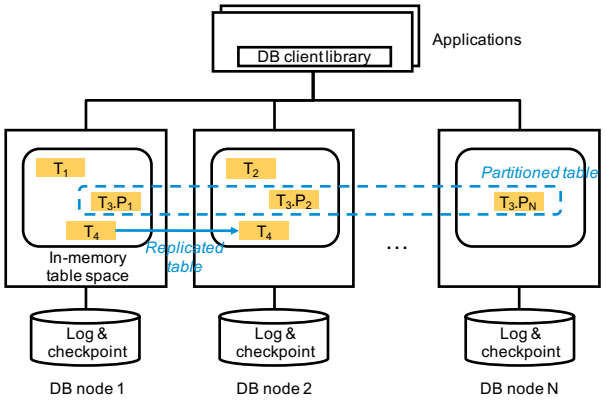

Figure 3: A simplified view of HANA distributed in-memory database architecture.

engine to those practical use cases, we propose *optimistic synchronous commit protocol* that avoids the expensive two-phase commit protocol while preserving the strict transactional consistency. The basic idea of optimistic synchronous commit protocol has also been briefly discussed in [13], but we detail implementation issues in this paper (Section 4.1).

- The three use cases are studied with extensive experiments using a concrete implementation in SAP HANA.

The rest of this paper is organized as follows. In Section 2, we provide technical background for the presented work in this paper. In Section 3, we present the three practical use cases in more detail. Section 4 describes the optimizations implemented to more efficiently support the three use cases. Section 5 provides the experimental evaluation results. Section 6 gives an overview of related works. Section 7 discusses future works and Section 8 concludes the paper.

2. BACKGROUND

In this section, we briefly summarize the key characteristics of SAP HANA distributed system architecture and HANA Table Replication both of which lay background for the work to be presented in this paper.

2.1 SAP HANA Distributed Database System Architecture

Figure 3 gives a simplified view of SAP HANA distributed in-memory database, which generally follows the shared-nothing partitioned database architecture. For the purpose of increasing the total computation resource and/or the total in-memory database space, it exploits multiple independent database nodes which are inter-connected through a commodity network.

The multiple database nodes belong to the same database schema whose tables can be distributed across the database nodes. In addition, a single table can be horizontally partitioned into multiple partitions, each of which contains a disjoint set of records of the table. The partitions of a table can be distributed across multiple database nodes. A *partition key* needs to be designated to partition the table along with a *partitioning function*, such as *hash*, *range*, or their combination. Typically, the primary key, or its subset, is chosen as the partition key of the table.

Regardless of how tables are partitioned and distributed, the multiple database nodes belong to the same transaction domain, ensuring strict ACID properties even for cross-node transactions. For this, two-phase commit and distributed snapshot isolation [14, 9] are incorporated. While queries can be routed to any of the database nodes by the client library directly, the client library finds an optimal target database node for a given query by looking up a part of its compiled query plan, which is transparently cached and refreshed at the client library. For a partitioned table, its partitioning function is also cached at the client library, so that the target table partition can be pruned at run time by evaluating a part of query predicate. This *client-side statement routing* is described in more detail in [14]. When a query needs to involve multiple nodes, a server-side query execution engine coordinates the distributed query process by exchanging intermediate query results.

2.2 HANA Table Replication

HANA Table Replication [15, 13] has been designed and implemented primarily for the purpose of enabling real-time reporting over operational data by minimizing the propagation delay between the primary and its replicas. Its key features are highlighted as follows.

First, it supports replicating a table or only a sub-table (a subset of columns of a table, for example) without necessarily replicating the entire database or the entire table contents. The replica of a table or sub-table can be located in a database node, while the primary copy of the table is located in another node.

Second, it supports cross-format replication with a storage-neutral logical log format. To maximize the performance of the mixed OLTP/OLAP workloads, it enables creation of a column-oriented replica layout for a row-oriented primary table. For this, the replication log is defined as a logical format decoupled from its underlying storage layout.

Third, it uses record-wise value logging with row ID (called RVID in [15, 13]) to maintain mapping between a primary record and its replica record. To avoid the non-deterministic behavior of a function logging, it captures record-level DML execution results. Together with the cross-format logical logging, it enables the replica table to have a different table structure (or partitioning scheme) from its primary table.

Fourth, in order to minimize the propagation delay between the source and the replica, HANA Table Replication employs a few novel optimizations such as lock-free parallel log replay with record-wise ordering and an early log shipping mechanism, which are explained in detail in [15, 13].

Finally, HANA Table Replication supports transaction-consistent online replica creation. By using the characteristics of multi-version concurrency control, HANA Table Replication allows other concurrent DML transactions to continue even while a new replica is being added online. Section 4.2 provides more detail about the online replica creation protocol.

3. ASYMMETRIC-PARTITION REPLICATION AND ITS PRACTICAL USE CASES

In this section, we describe how the presented HANA Table Replication is evolved and extended to asymmetric-partition replication and then present its three practical use cases in detail.

Table 1: Summary of Key Properties of Asymmetric-Partition Replication.

Properties	Usefulness in the three scenarios
Replication object granularity with table or sub-table	Enables applying the asymmetric-partition replication only to the necessary tables and columns (Use Cases 1 to 3)
Cross-format logical logging and record-wise value logging	Enables replica table structure to be decoupled from its primary table structure because the primary-to-replica relationship is maintained at record level (Use Cases 1 to 3)
Lock-free parallel log replay	Reduces the write transaction overhead by fast update propagation (Use Cases 1 and 2) or reduces the DDL elapsed time (Use Case 3)
Two commit options with asynchronous commit and optimistic synchronous commit	Enables avoiding the expensive two-phase commit (Use Cases 1 to 3)
Online replica addition with MVCC	Provides a basis for lock-free repartitioning operation (Use Case 3)

3.1 Asymmetric-Partition Replication

Asymmetric-partition replication enables replicas of a table to be independently partitioned regardless of whether or how its primary table is partitioned. It is an extended form of HANA Table Replication which inherits and expands upon the beneficial properties of HANA Table Replication described in Section 2.2. Table 1 provides a summary of its key properties and how they contribute to efficient and convenient implementation for the three practical use cases.

To extend HANA Table Replication to asymmetric-partition replication, the mapping between a primary-table record and its replica record is managed in a different way compared to HANA Table Replication. When a replica partition is initially populated, its initial snapshot is generated from the primary table by performing the partition pruning function, instead of physically copying the entire table image. When a DML operation occurs at the primary table, the corresponding DML replication log entry is generated and then its target replica partition is dynamically determined by performing the partition pruning function for the changed record. When a column value corresponding to the replica's partitioning key is updated at the primary table, a single DML operation at the primary can generate two DML log entries depending on the before-update value and the after-update value: a deletion log entry for a replica partition and an insertion log entry for another replica partition.

Note that the asymmetric-partition replication implementation has been incorporated and officially available in production versions of SAP HANA since April 2018. The following is one example of a possible SQL that creates an asymmetric-partition replica with hash partitioning. In addition to hash partitioning, range, round robin, or their cascaded partitioning are also possible [4]. The choice of the partitioning scheme for the replica is totally independent from how its corresponding primary table is partitioned.

```
CREATE TABLE <replica table name>
    LIKE <primary table name>
    [ASYNCHRONOUS|SYNCHRONOUS] REPLICA PARTITION BY
        HASH (<partition key for the replica>)
        PARTITIONS <the number of the partitions>
        AT <locations of the replica partitions>;
```

3.2 Use Case 1: Partitioned Replication for scaling out mixed OLTP/OLAP Workloads

The capability to handle both OLTP and OLAP workloads in the same database system is an attractive feature for modern database systems. Compared to traditional ETL-based OLAP systems, it can reduce visibility delay between the source OLTP and target OLAP systems by eliminating the need of an intermediate application-driven ETL processing layer. In addition, it eliminates the application-side burden of maintaining consistency and compatibility between the OLTP and OLAP database schemas.

For such a database system that handles OLTP and OLAP workloads, asymmetric-partition replication offers an interesting optimization opportunity. While the updateable primary copy of a table is still maintained in a single larger physical node, its replicas can be independently partitioned and distributed across multiple smaller physical nodes as already shown in Figure 1.

A similar replication architecture for mixed OLTP/OLAP workloads has already been proposed by the authors' earlier works [15, 13]. In the previous replication architecture, multiple replicas of a table can be created to scale out OLAP performance. In contrast, the proposed architecture in this paper can leverage multiple smaller replica nodes by horizontally partitioning a single logical replica table into multiple smaller sub-tables. While the replication architecture in [15, 13] requires each of the replica tables to maintain a full copy of the entire table contents, the partitioned replication architecture proposed in this paper enables a reduction of memory consumption at each replica and also a decrease in the required network resource consumption between the primary node and its replica nodes. Therefore, for a partitionable OLAP workload, the proposed asymmetric-partition replication becomes a practical option.

Moreover, when only a subset of a table (e.g., the most recent set of data in the table) is needed by the OLAP queries, the asymmetric-partition replication enables the creation of replicas for the needed table partitions alone. Therefore, the proposed asymmetric-partition replication provides more flexibility in choosing the right configuration for the replica table. Note that the cross-format replication presented in [15, 13] (replicating from a row-store table to a column-store table) can be combined with the proposed asymmetric-partitioned replication. For example, the primary table can be formatted to a row-store non-partitioned layout while its replica table is formatted to a column-store partitioned layout.

Note that the proposed asymmetric-partition replication architecture does not exclude having multiple copies for a replica partition for higher level.

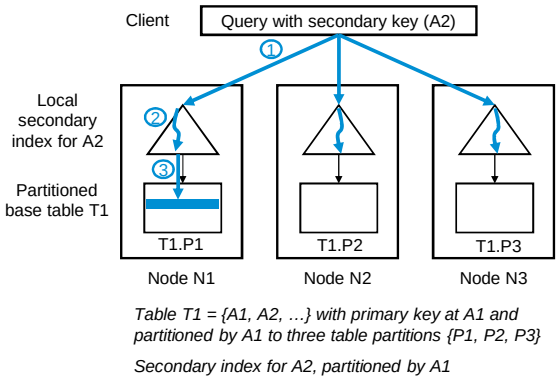

Figure 4: Local secondary index for a partitioned table by its primary key.

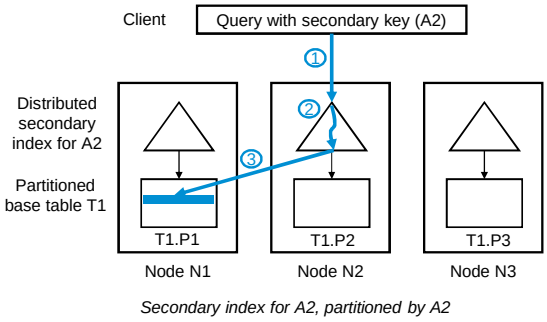

Figure 5: Independently partitioned secondary index for a partitioned table by its primary key.

3.3 Use Case 2: Scalable Distributed Secondary Index for Partitioned Tables

For queries having a partition key predicate, the target database node that owns the matching records can be easily identified by performing the partitioning function. On the other hand, queries having a non-partition key predicate require a full table scan at all nodes where the table is distributed. To more efficiently handle such secondary key access queries, maintaining a distributed secondary index is necessary.

A straightforward way of maintaining such a distributed secondary index is to co-partition a table and its secondary indexes together using the same partitioning key and then to maintain a separate *local index* for each table partition. Figure 4 shows an example of the local secondary index. A database table T1 is partitioned into three partitions by taking the primary key (A1) as its partition key. Each of the three table partitions has its own local secondary index that spans the records belonging to the co-located table partition.

With this approach, the full table scan can be avoided but all the database nodes still need to be accessed for the secondary key access query because the partition key and the secondary key do not match. As a result, as the number of nodes where the table is distributed increases, the cost of a secondary key access query grows proportionally both in network resource and CPU consumption. While the response time can be optimized by performing multiple local index accesses in parallel, the number of accessed nodes remains unchanged, and the resource cost remains high.

To overcome such a limitation, it might be desirable to partition the secondary index with its own secondary key independently from its base table. Figure 5 shows an example of an independently partitioned distributed secondary index. By early pruning the target secondary key index partition (at the step 1 in the figure), the number of accessed node for a secondary key access query can be reduced to one (when the secondary index lookup result points to a base record in the co-located table partition) or two (when the result points to a base record in a remote table partition), regardless of the number of table partitions and their distribution. However, this approach may need to pay the price on the write transaction side because an update at a table partition could involve updating a secondary index partition located in a remote node.

Asymmetric-partition replication makes implementing such a distributed secondary index more efficiently and with less development cost. The secondary index is created as a form of replica table which is independently partitioned from its primary table. Then, on a DML operation at a base table partition, the DML results are automatically propagated and applied to the corresponding secondary index partition (i.e. a replica partition) in a transaction-consistent way. Regarding how to generate and where to forward the DML replication log entries, it follows the same DML replication protocol of the asymmetric-partition replication which was explained in Section 3.1.

Such asymmetric-partition replication brings the following important performance benefits to the distributed secondary index implementation.

- *Optimistic transaction commit*: At the time of committing a transaction that updated both a local table partition and a remote secondary index partition, the optimistic synchronous commit ensures the cross-node atomicity without relying on the expensive two-phase commit protocol. This commit protocol will be presented in more detail in Section 4.
- *Fine-grained sub-table replication*: The fine-grained sub-table replication capability enables replication of only the necessary columns for the secondary index and thus saves in memory footprint and network bandwidth. At the secondary index partitions, only the secondary key columns and a reference (base table partition ID and row ID) to the original source record are maintained.
- Client-side statement routing: Combined with the client-side statement routing described in Section 2.1, the input value of the secondary key parameter of a query can be evaluated with the corresponding partition function cached at the client library. Then, the client library can early route the query to an optimal database node location where the corresponding secondary index partition is located.

One may implement such a distributed secondary index by using SQL triggers and a separate table for maintaining the secondary key values and the references. However, this is subject to the performance overhead incurred by (1) the general-purpose SQL trigger and (2) the two-phase commit for cross-node write transactions. Another may use more native and specialized implementation from scratch but it requires non-trivial time and developer resources.

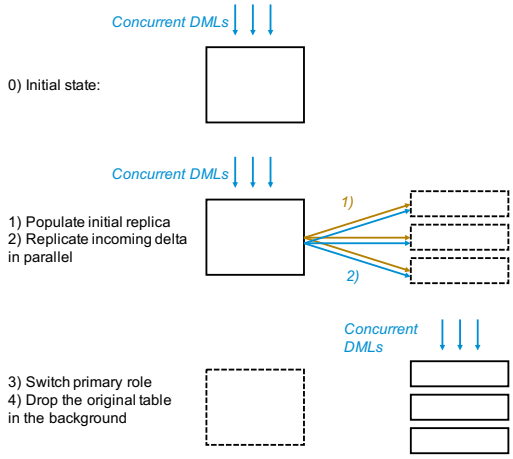

Figure 6: Online table repartitioning with asymmetric-partition replication.

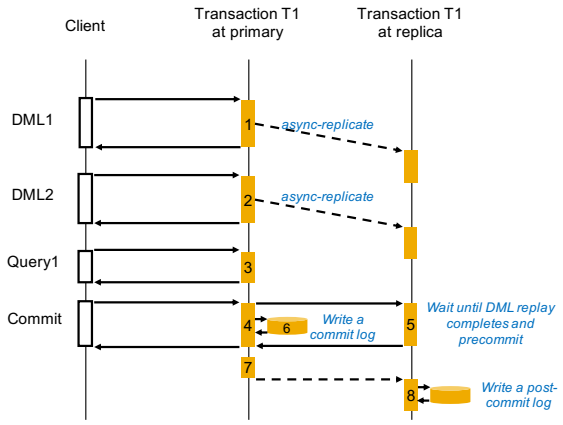

Figure 7: Replication protocol using optimistic synchronous commit (OSC-R).

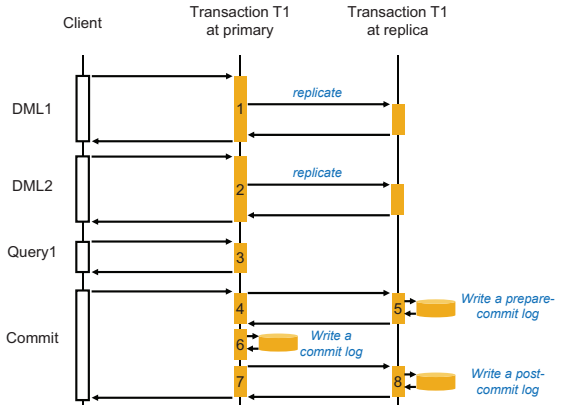

Figure 8: Replication protocol using synchronous DML trigger and two-phase commit (2PC-R).

3.4 Use Case 3: Online Table Re-partitioning

As discussed in Section 1, the overall performance of a shared-nothing distributed database system can be significantly affected by how well the database is partitioned for a given workload. Considering the dynamic nature of many enterprise application workloads and also the ever growing data volume, it is often required to continuously re-partition a given table. Under such a situation, it is desirable to support an efficient online re-partitioning operation that can minimize service downtime or interference to other concurrent normal transactions during the re-partitioning operation. Relying on the table-level exclusive lock during the re-partitioning operation can not meet such a demand.

Asymmetric-partition replication offers a beneficial option for implementing such an online re-partitioning operation exploiting the MVCC-based online replica addition protocol, instead of relying on a long-duration table lock. The online repartitioning operation can be logically mapped to a sequence of operations, as illustrated in Figure 6. First, an online replica is initially populated by copying the table snapshot (step 1). Second, delta changes that are incoming during the step 1 are asynchronously replicated by replication log entries (step 2). Third, once the step 1 is completed, a synchronization point between the primary and its replica is made and then the primary ownership of the table is transferred to the new replica (step 3). After that, the original primary table is dropped asynchronously in the background (step 4). In this procedure, asymmetric-partition replication allows the new replica to have a partition scheme different from that of its original table.

Like the second use case, the described online re-partitioning operation can be implemented by a more native and specialized implementation from scratch but it requires non-trivial time and developer resource. Note that the other types of online DDL operations that require copying the table data (for example, conversion from a row-oriented table format to a column-oriented table format) can be benefited from such an online replica creation protocol in general.

4. OPTIMIZATIONS

In this section, we present two key optimizations that make asymmetric-partition replication more efficient for the three use cases.

4.1 Optimistic Synchronous Commit Protocol

Optimistic synchronous commit protocol, called OSC-R in this paper, takes a hybrid approach by combining asynchronous DML replication with synchronous transaction commit. A DML log entry is generated at the DML time, but its propagation to the replica is asynchronously initiated without affecting the DML response time. After waiting until all its previous DML log entries are successfully applied to the replicas, the transaction proceeds to the commit processing. To highlight the performance gain from OSC-R, we compare OSC-R with an alternative replication protocol that is based on synchronous DML and two-phase commit, called 2PC-R. Figure 7 and Figure 8 show OSC-R and 2PC-R protocols, respectively.

At OSC-R, replication does not affect DML response time as in steps 1 and 2 of Figure 7, while each DML involves a synchronous network round trip to the replica in 2PC-R (steps 1 an 2 in Figure 8).

At the transaction commit phase, in contrast to 2PC-R that incurs two network round trips (Steps 4 and 7) and three log I/O operations (Steps 5, 6, and 8), OSC-R can commit the transaction immediately after one network round trip (Step 4) and one log I/O operation (Step 6), both of which are interleaved with each other. Such an op-

timistic optimization is possible by leveraging the fact that a replica is basically a data structure derived from its primary table. When a failure occurs in the middle of the OSC-R transaction's commit phase, there is still a chance that the replica can be recovered to the latest committed state by re-synchronizing with the primary. By the same reason, the recovery redo log and the post-commit log entries generated at the replica can be persisted asynchronously (Step 8). The replica recovery protocol for HANA Table Replication is described in more detail in Section 4.1 of [13].

The transaction commit at OSC-R can proceed after waiting until its previous DML replay operations complete (Step 5 in Figure 7). However, for multi-statement transactions, which are quite common in many enterprise applications, there is high chance that the asynchronous replication of an earlier DML statement is overlapped with the next statement execution at the primary transaction and thus the delay at Step 5 is minimized.

Differently from the commit protocol under the lazy replication presented in [15, 13], OSC-R must ensure visibility atomicity of the changes made by the primary transaction and its corresponding replayer transactions. Under the lazy replication, if a database client accesses a replica node after it receives the commit acknowledgement of its primary transaction, then its written changes may not be yet visible at the replica node. To ensure visibility atomicity across the primary and its replicas at OSC-R, however, when a concurrent query tries to access a replica record version in *precommitted* state after Step 5 and before Step 8 (Figure 7), the access to the record version is postponed until the state of the record version is finally determined to be either *post-committed* or *aborted* by Step 7 at the primary node and then informed to the replica node by Step 8.

Figure 7 omitted the transaction abort scenario, but it can be explained as follows. If the primary transaction aborts after sending the precommit request to the replica (for example, by a failure while writing a commit log to the disk at Step 6), then the corresponding replica record versions are marked as *aborted* by Step 8. And thus, those replica record versions are ignored by the pending concurrent query. The overall protocol for reading the replica record versions under OSC-R is provided by Algorithm 1.

Algorithm 1 Read a replica record version under OSC-R

Require: A query, Q.
Require: A replica record version, V, found by evaluating the search condition of Q.
1: **if** V.State = *precommitted* **then**
2: Wait until V.State gets updated.
3: **end if**
4: **if** V.State = *committed* AND V.Timestamp <= Q.Timestamp **then**
5: **return** V.Value.
6: **else**
7: **return** NULL.
8: **end if**

One potential drawback of OSC-R compared to 2PC-R is that the uncommitted changes made by a transaction might not yet be visible if the same transaction tries to read its own uncommitted changes at a replica - by the consequence of asynchronous DML replication. This *read-own-write* can be still achieved under OSC-R by assigning monotonically increasing sequence numbers to the executed statements in a transaction and then letting the replica-routed read statement wait until the previous statement is applied to the replica node. Or, more simply, such a read-own-write query can be routed to the primary node directly by the client library.

4.2 MVCC-based Online Replica Addition

Figure 9 shows the procedure of adding a new replica online in a transaction-consistent way. It starts with activating replication logging for all the DMLs occurring at the target primary table (Step 1). Then, while a consistent snapshot image of the primary table is being retrieved and applied to the replica (Step 3), the replication logger captures and replicates all the newly committed changes (Step 3').

To allow concurrent DML operations during Step 3, it is important to perform this phase without relying on any exclusive lock on the primary table because this operation can take a long time depending on the size of the table. For this purpose, we exploit the characteristics of MVCC. The replica initializer is implemented as a snapshot isolation transaction [8] that accesses a consistent set of database record versions as of its acquired snapshot timestamp. In Figure 9, the transaction timestamp acquired at Step 2 becomes the snapshot timestamp of the replica initializer transaction. And then, based on the assigned snapshot timestamp and the MVCC protocol that is also described in [16], a consistent set of record versions is retrieved at Step 3.

Note that, as a potential side effect of using the snapshot isolation transaction for the replica initializer, the MVCC version space can keep growing during Step 3 in order to maintain a consistent set of record versions as of the acquired snapshot timestamp. However, by using the hybrid garbage collection technique [16] implemented in SAP HANA, the space overhead at the version space is effectively minimized.

The operation of copying the initial table snapshot (Step 3) and the DML replication for the delta changes (Step 3') are performed in parallel, as illustrated in Figure 9. This parallel processing does not cause any consistency issue because both Step 3 and Step 3' follow the record-wise ordering scheme based on RVID (record version ID) [13]. RVID is an unique identifier of a record version in a table. A new RVID value is assigned when a new record version is created. Using the RVID management, before applying a DML log entry to the replica, the log replayer checks whether the RVID value of the target replica record equals to the Before-Update RVID of the propagated DML log entry. If not, the DML log replay operation is postponed since it means that an earlier change is not yet applied to the replica. This RVID-based record-wise ordering is described in more detail in Section 3.3 of [13]. At Step 3', the delta changes are *asynchronously* propagated to the replica and thus the other concurrent DML transactions do not need to wait for its change propagation to the replica.

Finally, at Step 4, once Step 3 is completed, the replica starts accepting new queries after waiting until the previously generated delta changes are being applied to the replica - to make the replica state up-to-date. When creating a lazy-mode replica, the replica can already start accepting new queries as soon as Step 3 is completed.

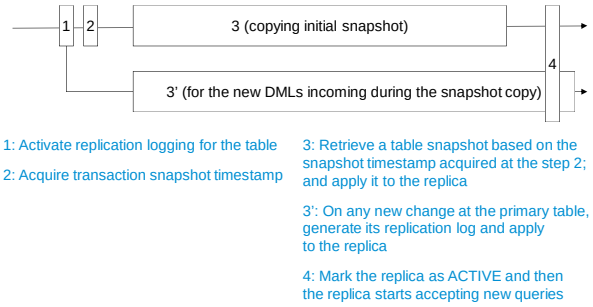

Figure 9: MVCC-based online replica creation.

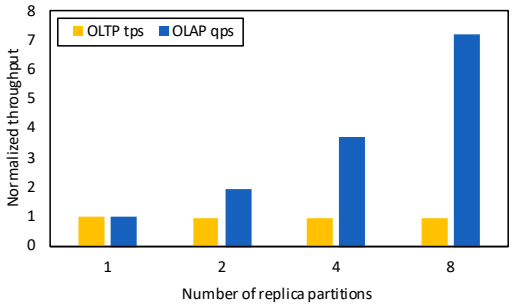

Figure 10: TPC-CH performance with asymmetric-partition replication.

5. EXPERIMENTAL EVALUATION

To evaluate the performance of the proposed architectures, we conduct a number of experiments with a development version of SAP HANA. The tested SAP HANA system is configured as a distributed system which consists of up to nine database nodes. Each of them has 60 physical CPU cores (120 logical cores with hyper-threading enabled), 1TB of main memory and four 10Gbit NICs. They are interconnected by a commodity network within the same local data center.

5.1 Partitioned Replication for OLTP/OLAP-mixed Workloads

To show the benefit of asymmetric-partition replication for mixed OLTP/OLAP workloads, we tested its performance with a TPC-CH benchmark, which was initially used in [20]. The benchmark generates both TPC-C and TPC-H workloads simultaneously over the same data set consisting of nine TPC-C tables and three other tables used exclusively by TPC-H (`NATION`, `REGION`, and `SUPPLIER`). To those tables, the initial data were populated with 100 warehouses as in [20].

The test system consists of one primary node and up to eight replica nodes. At the primary node, all 12 tables are created and placed without any partitioning. Each of those 12 primary tables creates its replica table. Among them, eight tables which have `Warehouse_ID` column create their replicas as a partitioned form by the `Warehouse_ID` column. For the other four tables (the three TPC-H tables and `ITEM` table), a full replica table is created for each replica node. For example, for the `ORDERLINE` table, we created its replica with range-partitioned asymmetric replicas using the below DDL. The number of partitions is set as the the number of the used replica nodes in each experiment.

```
CREATE TABLE REP.ORDERLINE LIKE
   SRC.ORDERLINE ASYNCHRONOUS REPLICA
   PARTITION BY RANGE (Warehouse_ID)
       (PARTITION 1 <= VALUES < 26,
       PARTITION 26 <= VALUES < 51,
       PARTITION 51 <= VALUES < 76,
       PARTITION 76 <= VALUES < 101)
   AT 'host1:port1', 'host2:port2',
   'host3:port3', 'host4:port4';
```

TPC-H read-only queries are distributed to the replica nodes simultaneously, while TPC-C read/write transactions are directly routed to the primary node. To evenly distribute the TPC-H queries over the four replica nodes, a query predicate (`Warehouse_ID = ?`) is added to each of them. The TPC-C workload is generated by 30 concurrent clients in total and the TPC-H workload is generated by 90 concurrent clients per replica node. Each of the TPC-H clients repeats execution of the 22 TPC-H queries serially during the test.

Figure 10 shows the experimental result. The TPC-H throughput values are normalized by 1-replica TPC-H throughput. The TPC-C throughput values are normalized by 1-replica TPC-C throughput. As the number of the replica nodes (and thus the number of the corresponding replica partitions) increases, the total aggregated processing throughput of TPC-H (OLAP qps) increases almost scalably. Meanwhile, the overhead imposed to the write transaction is maintained at a minimum, as the TPC-C processing throughput shows in the result (OLTP tps). Note that TPC-C throughput does not increase with the increasing number of the replica nodes in this experiment, because the number of the primary nodes remains unchanged and the TPC-C transactions are directed to the primary nodes only.

The plain forms of symmetric replication show nearly the same results as Figure 10. However, they impose other overheads. For example, in a form of symmetric replication, both the primary and its replica tables can be created as non-partitioned tables and each replica node can maintain a full copy of the primary table. Then, such a symmetric replication configuration requires additional memory consumption at each replica node and additional network resource consumption between the primary and its replica nodes. As an alternative under symmetric replication, the primary table can be partitioned by the same partitioning scheme as its replica table and then the partitions of the primary table can be co-located within the same single primary node. However, this alternative also adds a disadvantage in that the primary table must be re-partitioned whenever the partitioning scheme of the replica table changes. In the proposed asymmetric-partition replication, the replica table can be re-partitioned without necessarily affecting the partitioning scheme of the corresponding primary table.

5.2 Distributed Secondary Index

To show the benefit of the proposed replication-based distributed secondary index implementation, we extracted a real-application table and its involved queries from a real enterprise financial application workload [5]. The table called `DFKKOP` represents items in contract account documents. Th-

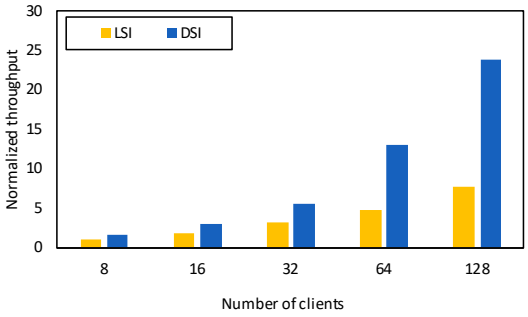

Figure 11: Multi-client scalability of secondary key search throughput (4 database nodes).

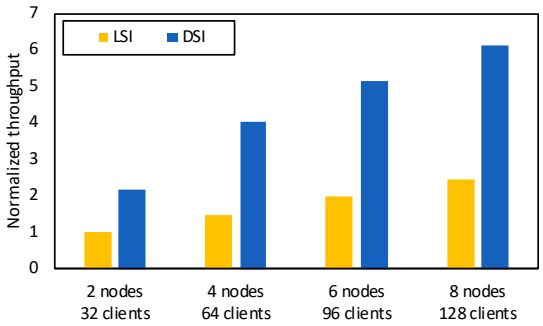

Figure 12: Multi-node scalability of secondary key search throughput (16 clients per node).

ere are in total 170 columns and its primary key consists of five columns among them. The table is hash-partitioned on a column called `OPBEL` (representing a contract document ID) which is one of the five primary key columns. The initially populated table size is about 56GB and it was evenly distributed across the database nodes by the hash-partitioning function. The table has a secondary index on a column called `VKONT` (representing a contract account ID), which does not belong to the primary key column set.

The experiment consists of two parts. One is to measure the improvement at the secondary key search performance and the other is to measure the overhead incurred at the write transaction. In both measurements, we compared the performance of the proposed replication-based distributed secondary index implementation (DSI) with that of a local secondary index implementation (LSI). At LSI, the secondary index is co-partitioned with the base table by the same partitioning key, `OPBEL`. At DSI, the secondary index is partitioned by the `VKONT` column.

In DSI, the following DDL will create partitioned and distributed sub-table replicas that will act as secondary indexes for the base table (`DFKKOP`).

```
CREATE TABLE DFKKOP_REPLICA LIKE
   DFKKOP SYNCHRONOUS REPLICA (VKONT)
   PARTITION BY HASH (VKONT)
   PARTITIONS 4 AT 'host1:port1', 'host2:port2',
   'host3:port3', 'host4:port4';
```

Note that the secondary index can be also created as a range-partitioned replica to support range queries. As the DDL indicates, the replica is created as a "synchronous" replica to ensure the atomic transactional visibility with its source table. Also, only the necessary secondary key column (`VKONT`) is stored in the replica together with two internal columns (`$reference_partid$` and `$reference_rowid$`) that are used for referring to the corresponding base record.

5.2.1 Search Performance

First, we measure the secondary key search performance with the following secondary key access query chosen from the captured real-application workload. The query returns a record for a given contract account ID with the additional constant predicate for a certain category and status. We omit the details of the constant predicate because it is specific to the application semantics and did not affect the experimental result significantly.

```
SELECT * FROM DFKKOP WHERE VKONT = ? AND ...;
```

Note that, since the necessary integration at the query processor for the distributed secondary index was not yet available at the time of writing this paper, we revise the test query into two consecutive queries for the DSI experiment as follows. The first query finds one or more references to the base records by looking up the replica table (`DFKKOP_REPLICA`). The secondary query retrieves the target base record by giving the `$reference_rowid$` value as its second parameter after finding the target table partition by giving the retrieved `$reference_partid$` value as its first parameter (`PARTITION(?)`). `$rowid$` (64-bit numeric integer) denotes a unique identifier of a record within a table.

```
SELECT $reference_partid$, $reference_rowid$
   FROM DFKKOP_REPLICA WHERE VKONT = ?;
SELECT * FROM DFKKOP PARTITION(?)
   WHERE $rowid$ = ? AND ...;
```

Figure 11 shows the multi-client scalability of the secondary key search performance by varying the number of clients from 8 to 128. The number of database nodes is fixed to 4. The query throughput is normalized by that of the 8-client LSI case.

In the result, DSI shows a scalable throughput with the increasing number of clients and outperforms LSI significantly by factors of 1.5 to 3.1. While the secondary key access query at DSI involves accessing at most two database nodes regardless of the total number of the database nodes, the query at LSI accessed all the database nodes, including the node where no matching record exists. Considering the fact that DSI involves two separate queries, it is expected that the gain would increase further by the right integration with SQL processor in the future.

Figure 12 shows a multi-node scalability with the number of clients fixed to 16 per node. The query throughput is normalized by that of the 2-node LSI case. Similarly to the previous experiment, DSI scales better than LSI as the number of the database nodes increases and outperforms LSI significantly by factors of 2.2 to 2.5. It is worth noting that LSI exhibited 1.4 to 3.8 times higher CPU usage than DSI in the experiment.

5.2.2 Write Performance

In the second part of the experiment, we measure the performance overhead added to the write transaction. For this purpose, we use the following write transaction that involves 5 insert statements to the `DFKKOP` table.

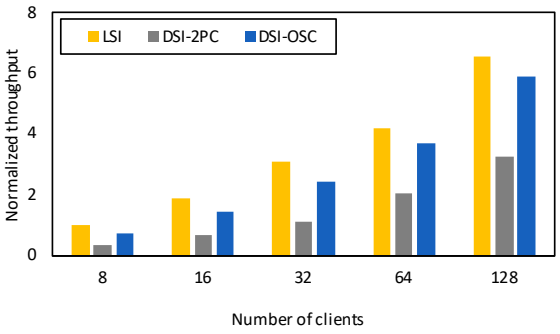

Figure 13: Multi-client scalability of write transaction throughput (4 database nodes).

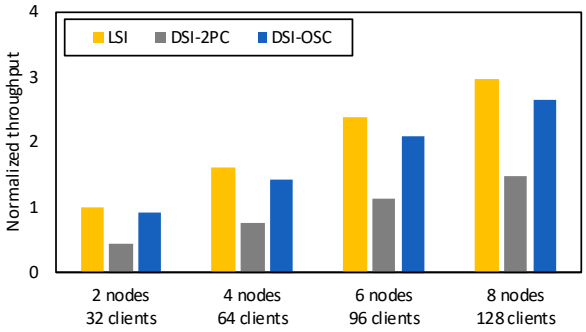

Figure 14: Multi-node scalability of write transaction throughput (16 clients per node).

```
INSERT INTO DFKKOP VALUES ...;
// insert 5 different records
SELECT COUNT(*) FROM DFKKOP WHERE OPBEL = ?;
COMMIT;
```

To highlight the performance benefit of the DSI implementation that uses the optimistic synchronous commit protocol, we compared three approaches: (1) LSI, (2) DSI with synchronous trigger and two-phase commit (DSI-2PC), and (3) DSI with asymmetric-partition replication and the optimistic synchronous commit protocol (DSI-OSC).

Figure 13 and Figure 14 respectively show multi-client scalability and multi-node scalability of the above presented write transaction. The throughput in Figure 13 is normalized by that of the 8-client LSI case. The throughput in Figure 14 is normalized by that of the 2-node LSI case. In Figure 13, DSI-OSC scales comparably to LSI as the increasing number of the clients and eventually DSI-OSC outperforms DSI-2PC significantly. In Figure 14, DSI-OSC again scales similarly to LSI as the increasing number of the database nodes and eventually DSI-OSC outperforms DSI-2PC significantly.

For more detailed analysis of the write transaction performance, we measured the average response time of the DML and commit operations, as shown in Figure 15 and Figure 16, respectively. The number of database nodes varies from two to eight with the number of clients fixed to 16 per node.

Figure 15 shows that the DML response time of DSI-2PC takes on average 1.9 times longer than that of DSI-OSC. It

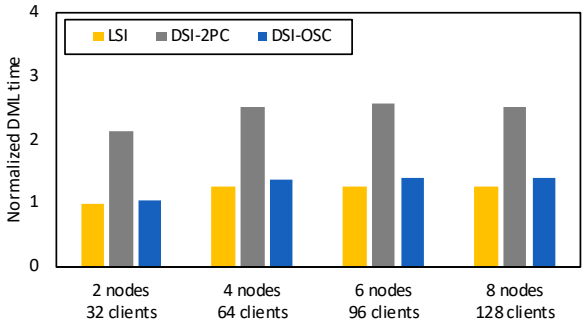

Figure 15: Average response time of DML operation.

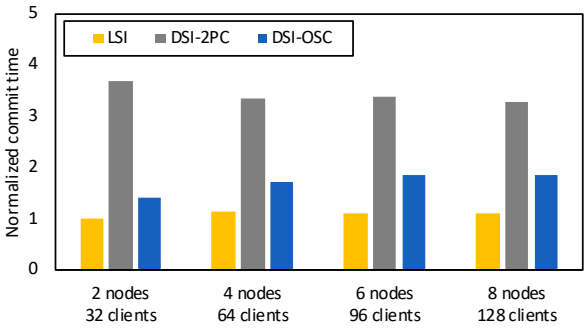

Figure 16: Average response time of commit operation.

is because the DSI-2PC is implemented based on the synchronous SQL trigger and thus, a DML operation can involve cross-node communication if the target secondary key index entry is located at a remote node.

Figure 16 shows that LSI shows the response time is shortened as expected because it accesses only a single database node and performs a local commit. Compared to that, while DSI-2PC shows significantly longer response time than the LSI, DSI-OSC adds relatively smaller overhead to the transaction commit operation. It is because the optimistic synchronous commit enables DSI-OSC to avoid the expensive two-phase commit. Compared to LSI, DSI-OSC still shows slightly higher response time because the primary transaction can commit after confirming that all its DML operations are successfully applied to the in-memory replicas, in order to ensure atomic transaction visibility as described in Section 4.

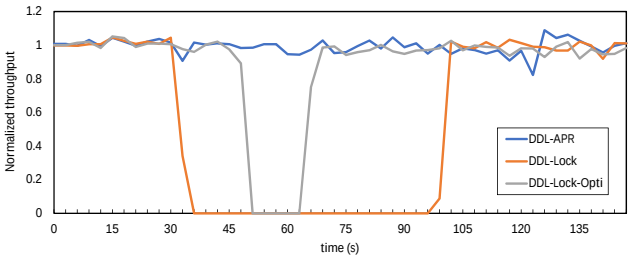

Figure 17: Impact of online repartitioning to TPC-C throughput.

5.3 Online Table Repartitioning

To show the benefit of the proposed online table repartitioning implementation, we compare three alternative approaches: (1) online table repartitioning based on asymmetric-partition replication (DDL-APR); (2) table repartitioning with long-duration table lock (DDL-Lock); (3) table repartitioning with long-duration table lock but a specialized optimization (DDL-Lock-Opti). At DDL-Lock, the exclusive table lock is acquired during the entire table repartitioning operation. While both DDL-APR and DDL-Lock are performing MVCC-based record-wise snapshot retrieval, DDL-Lock-Opti copies the physical full table image at once by exploiting the fact that there is no other concurrent DML operation during the table copy operation by the long-duration table lock acquisition.

To see how those three DDL implementations affect the other concurrent DML transactions, we perform the online repartitioning DDL while continuously running TPC-C workload to the same system. The initial TPC-C database is populated with 100 warehouses and the TPC-C workload is generated by 30 concurrent clients, consistently with the experiment of Section 5.1. The online repartitioning DDL is performed against `CUSTOMER` table which is updated by TPC-C `Payment` and `Delivery` transactions. `CUSTOMER` table is initially formatted as a non-partitioned table and then it is partitioned into four range partitions online.

Figure 17 shows the experiment result. The vertical axis represents the throughput of the TPC-C transactions, normalized by that measured while DDL operation is not yet involved. In all of the three approaches, the repartitioning DDL is performed in 30 seconds after the high-load phase of TPC-C workload starts.

In DDL-Lock and DDL-Lock-Opti, during the DDL execution, the TPC-C performance dropped to 0 because all the TPC-C transactions eventually get blocked by the long-term exclusive table lock acquired by the online re-partitioning operation. Compared to that, DDL-APR shows almost non-blocking behavior except small performance drop at two time points: one when starting the DDL operation (at the time point 30) and another when completing the DDL operation (at the time point 120). At the two points, the current implementation at SAP HANA is relying on short-term instant-duration table lock when acquiring the transaction snapshot timestamp and when transferring the primary ownership to the new replica. As already explained in Section 4.2, the first short-term table lock is removable in the future by additional implementation.

Remark that the overall elapsed time of DDL-APR increased compared to DDL-Lock or DDL-Lock-Opti. One reason is that, in DDL-APR, the TPC-C clients keep generating new DML changes and thus the total amount of data that needs to be applied to the new table increases compared to DDL-Lock or DDL-Lock-Opti. In DDL-Lock-Opti, the overall blocking period is shortened significantly compared to DDL-Lock by applying the table-level physical image copy operation. However, this optimization is applicable only when the exclusive lock is acquired during the table copy operation. Considering that such a repartitioning DDL operation in the practical systems is typically performed in the background, it is more important to reduce the interference to the normal DML transactions rather than reducing the elapsed time of the DDL operation itself.

6. RELATED WORK

6.1 Asymmetric-Partitioned Replication

The conventional logical replication exploits replication of logical statements from the primary to its replica systems, and thus it enables a replica table to be independently structured from its primary table. Such logical replication has been already widely used by academic or industrial systems. However, to the best of our knowledge, none of those previous works exactly matches with the concept of the asymmetric-partition replication proposed in this paper.

For example, MySQL allows that a table on the master can have more or fewer columns than the slave's copy of the table. However, in MySQL, "replication between tables having different partitioning is generally not supported" [2]. It is also known that the logical replication in MySQL allows the historical cold data of a table is replicated but not deleted at the replica side any longer. It can be seen as a special form of asymmetric-partition replication because the primary table has less data or less partitions than that of its replica table which holds both of recent (hot) and historical (cold) data. However, such a configuration is enabled simply by turning off the replication log generation for a particular database session that executes the delete operations for the historical data. It does not match with the generic form of the asymmetric-partition replication proposed by our paper.

[18] proposed another type of logical replication, called BatchDB. OLTP and OLAP replicas can have different storage layouts (row oriented format for OLTP, column oriented format for OLAP) to efficiently handle hybrid OLTP and OLAP workloads. In addition, it focuses only on lazy (asynchronous) replication. Compared to [18], our paper explores how logical replication can evolve into the concept of asymmetric-partition replication and how synchronous replication can be further optimized without sacrificing transactional consistency.

6.2 Distributed Secondary Index

The technique of implementing distributed secondary index by using a separate system table is not new. Our contribution is not about using a separate table for the implementation of distributed secondary index, but about using an advanced form of replication engine for economical and efficient implementation of distributed secondary index for a practical commercial DBMS. In addition, we addressed the challenge of the node-to-node write synchronization overhead by proposing the optimistic synchronous commit protocol without sacrificing any transactional consistency, as shown in Section 4.1.

For example, [12] discusses two secondary index approaches in the context of distributed NoSQL key-value store. One is to store an inverted list in a separate system table and another to co-locate data and its local secondary index. The first approach is equivalent to what we presented as distributed secondary index and the second to local secondary index in Section 3.3. [12] shares the same conclusion with our study: the first table-based approach incurs higher communication overhead between nodes on write operations while the second co-location approach incurs the development cost because it has to be implemented from scratch. However, while [12] does not further explore how the write overhead is addressed, our work proposes a novel optimistic synchronous replication protocol as in Section 4.1.

Teradata [1] supports a special form of a table-based distributed secondary index implementation. It co-locates secondary index for the attributes with non-unique values, while using a table-based approach for the attributes that have unique values - via automatic hashing of the attribute values. However, by the nature of the hash-based partitioning used for the distributed secondary index, operations such as range searches need to be performed in all the nodes. Compared to that, our work allows the secondary index table to be partitioned by any general partitioning scheme including hash and range schemes. In addition, from [1], we could not find explicit description on how it addresses the node-to-node write synchronization overhead.

Both DynamoDB [3] and Megastore [7] support distributed secondary index. However, both rely on "asynchronous" change propagation to the remote secondary index with sacrificing the transactional consistency. Under such an asynchronous propagation, the query on the secondary index could return result that are not up to date. It imposes significant burden to application programmer since they have to anticipate and handle such a sporadic stale query result.

Google F1 [23] provides a synchronous secondary index that uses the two-phase commit for transactional consistency. While its write transaction performance will be bound by the two-phase commit overhead, our proposed optimistic synchronous commit protocol reduces the commit overhead by exploiting the fact that the secondary index is a data structure derivable and recoverable from the base table data.

ScyllaDB [6] exploits the materialized view to support a secondary index. For each secondary index, it creates a materialized view that has the indexed column as the partitioning key. Our work does not exclude the possibility of using materialized view instead of replication for maintaining the distributed secondary index. However, our proposed optimization such as optimistic synchronous commit will still hold for the implementation based on materialized view.

6.3 Online DDL

Many cloud DBMSs exploit replication for the purpose of online upgrades or online database migration [19]. Compared to those approaches, we propose exploiting table replication engine for online DDL operations that occur for a particular table.

For online DDL or online schema change operations, various techniques are already known. For example, Google F1 [21] proposes a new protocol for online and asynchronous schema evolution. In order to guarantee the database consistency, it transforms the schema change operations into multiple steps of consistency-preserving schema changes. PRISM/PRISM++ [11, 10] presents a special language for schema change operations, tools to evaluate the effect of changes, and a query rewriting method. Controvol [22] develops a framework for controlled schema changes for NoSQL applications.

In contrast to those prior works, we showed that the online DDL can be implemented economically and efficiently be extending a replication engine and utilizing the characteristics of MVCC.

7. DISCUSSION AND FUTURE WORK

We expect that our work can be further extended by contributions from academia in the following aspects. First, it is desirable in practice to automate the decision of which tables need to be replicated for a given workload; how many replicas are needed; and how the replica table needs to be partitioned. Since replication itself needs to pay a cost for additional resource consumption, it is not always trivial to make a decision that maximizes the additional value over the paid cost. Second, the presented optimistic synchronous commit protocol can be further generalized to apply to other scenarios that require transactional consistency between the base data and its derived remote data, beyond the scope of the replication. For example, tables that require transactional consistency with each other by triggers or cascaded-update foreign-key constraints will present another practical use case. Third, the proposed asymmetric-partition replication and the optimistic synchronous commit protocol can be extended to support multi-master replication, for example by adding a conflict detection and resolution mechanism among the changes made at different master replica nodes.

8. CONCLUSION

In this paper, we presented a novel concept of asymmetric-partition replication engine as a foundation to serve three important practical use cases in distributed database systems. In the asymmetric-partition replication, replicas of a table can be independently partitioned regardless of whether or how its primary copy is partitioned. The three practical use cases include (1) scaling out OLTP/OLAP-mixed workloads with partitioned replicas, (2) efficiently maintaining a distributed secondary index for a partitioned table, and (3) efficient implementation of the online re-partitioning operation. In addition, to address the challenge in node-to-node write synchronization overhead in asymmetric-partition replication, we proposed optimistic synchronous commit by fully exploiting the fact that a replica is a derivable and reconstructable data structure from the corresponding primary table data. The proposed asymmetric-partition replication and its optimizations are incorporated in SAP HANA in-memory database system. Through extensive experiments, we demonstrated the significant benefits that the proposed replication engine brings to those three practical use cases.

Overall, we believe that our work revealed that the database replication can serve even more practical use cases, beyond the traditional use cases, for modern distributed database systems.

9. ACKNOWLEDGMENTS

The authors would like to thank the entire SAP HANA development team for providing the solid foundation for the work presented in this paper. We would especially like to express our appreciation to Eunsang Kim, Hyoung Jun Na, Chang Gyoo Park, Kyungyul Park, Deok Koo Kim, Jungsu Lee, Joo Yeon Lee and Carsten Mueller. In addition, the authors would like to sincerely thank anonymous VLDB reviewers who provided invaluable comments and suggestions. Seongyun Ko and Wook-Shin Han are partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1A2B3007116).

10. REFERENCES

[1] Introduction to teradata. https://www.teradatapoint.com/teradata/secondary-index-in-teradata.htm.

[2] Mysql: Replication and partitioning. https://dev.mysql.com/doc/refman/8.0/en/replication-features-partitioning.html, 2012.

[3] Using global secondary indexes in dynamodb. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html, 2012.

[4] Sap hana sql and system views reference 2.0 sps 03. https://help.sap.com/viewer/product/SAP_HANA_PLATFORM/2.0.03, 2018.

[5] Sap s/4hana. https://www.sap.com/products/s4hana-erp/features.html, 2019.

[6] Scylladb. https://www.scylladb.com/, 2019.

[7] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[9] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB Journal-The International Journal on Very Large Data Bases*, 23(6):987–1011, 2014.

[10] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *The VLDB Journal-The International Journal on Very Large Data Bases*, 22(1):73–98, 2013.

[11] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761772, Aug. 2008.

[12] J. V. Dsilva, R. Ruiz-Carrillo, C. Yu, M. Y. Ahmad, and B. Kemme. Secondary indexing techniques for key-value stores: Two rings to rule them all. In *International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*, 2017.

[13] J. Lee, W.-S. Han, H. J. Na, C. G. Park, K. H. Kim, D. H. Kim, J. Y. Lee, S. K. Cha, and S. Moon. Parallel replication across formats for scaling out mixed oltp/olap workloads in main-memory databases. *The VLDB Journal-The International Journal on Very Large Data Bases*, 27(3):421–444, 2018.

[14] 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 *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1165–1173. IEEE, 2013.

[15] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, and W.-S. Han. Parallel replication across formats in sap hana for scaling out mixed oltp/olap workloads. *Proc. VLDB Endow.*, 10(12):15981609, Aug. 2017.

[16] 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 *Proceedings of the 2016 International Conference on Management of Data*, pages 1307–1318. ACM, 2016.

[17] Y. Lu, X. Yu, and S. Madden. Star: Scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):13161329, July 2019.

[18] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. Batchdb: Efficient isolated execution of hybrid oltp+ olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50. ACM, 2017.

[19] T. Mishima and Y. Fujiwara. Madeus: database live migration middleware under heavy workloads for cloud environment. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 315–329, 2015.

[20] 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.

[21] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in f1. *Proc. VLDB Endow.*, 6(11):10451056, Aug. 2013.

[22] S. Scherzinger, T. Cerqueus, and E. C. de Almeida. Controvol: A framework for controlled schema evolution in nosql application development. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1464–1467. IEEE, 2015.

[23] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):10681079, Aug. 2013.