Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation

Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, Daming Shao
Alibaba Group
China
{junbin.kangjb,le.cai,lifeifei,xingxuan.zxx,mingsong.cw,zijia,mingzong.sdm}@alibaba-inc.com

ABSTRACT

Shared-nothing, distributed databases scale transactional and analytical processing over a large data volume by spreading data across servers. However, static sharding of data across nodes makes such systems fail to timely adapt to changing workloads and struggle to obey the cloud pay-as-you-go model. Migrating shards between nodes online is a key technique to react to dynamic changes of workloads for cloud elasticity. Existing approaches introduce severely degraded performance and service interruption, resulting in SLA violation on the cloud; or they are tailor-made to deterministic databases. In this paper, we propose *Remus*, a new live migration approach for shared-nothing, distributed databases with snapshot isolation. *Remus* migrates shards between nodes with zero service interruption and minimal performance impact. This is achieved by an efficient unidirectional dual execution during migration. We implement *Remus* on a shared-nothing, distributed version of *PolarDB-PG* and evaluate it against state-of-the-art approaches using standard OLTP workloads TPC-C and YCSB, and hybrid workloads consisting of long-lived and short transactions. The results demonstrate *Remus* is the only effective approach to achieve the goal of zero transaction interruption, zero downtime and marginal performance impact, paving the way for applying the shared-nothing architecture to a cloud database which needs to provide elasticity while guaranteeing strict SLAs.

CCS CONCEPTS

• **Information systems → Relational parallel and distributed DBMSs**.

KEYWORDS

live migration, shared-nothing database, snapshot isolation

ACM Reference Format:
Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, Daming Shao. 2022. Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526047

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526047

1 INTRODUCTION

Cloud venders provide relational database-as-a-service for applications to process structured data in a cost-effective way. These applications often require cloud databases to process requests over ever-growing data volumes under extremely high concurrency. This drives an increased demand for large-scale transactional and analytical processing systems. The shared-nothing distributed databases can scale both storage and computing capacity beyond what a single node can offer through data sharding [4, 15, 16, 35, 40]. However, static sharding limits the ability of such systems to adapt to rapid workload variation. This may result in degraded performance and service level agreement (SLA) violations due to load imbalance [23, 29, 43, 49] and insufficient provisioning of cloud resources in the face of sudden load increases [3, 18]. Live migration that migrates data between nodes with little impact on the ongoing services is widely adopted by shared-nothing databases to adapt to dynamic workloads [8, 16, 23, 29, 34].

Most existing live migration techniques can be classified into two main categories: *push-migration* [8, 16, 18, 49] and *pull-migration* [23, 24]. *Push-migration* copies the snapshot of migrating data to the destination node and then pushes incremental updates iteratively. Some downtime is enforced to transfer the ownership of migrating data before running transactions on the destination [18, 49]. Transactions that access migrating data may be aborted on the source [8, 16], or restarted/resumed on the destination node [18, 49]. Aborting or restarting transactions can cause significant delays in case of long-running transactions such as large batch insertions. *Pull-migration* routes newly arrived transactions to the destination node during a migration. Missing data chunks are pulled on-demand from the source node. However, each data *pull* locks the corresponding data partitions and takes a long I/O time to complete during which many contending transactions may be blocked, resulting in considerable performance degradation [34, 49].

The limitations of existing approaches put obstacles in the way of applying the shared-nothing architecture to general database productions on the cloud, which may run short OLTP transactions, long analytical queries and batch transactions, and even a hybrid of these workloads while requiring strict SLAs. Any transaction aborts and performance impact induced by a migration may violate SLAs that must be guaranteed by the cloud vendors.

This paper introduces *Remus*, a new live migration technique designed for a shared-nothing database supporting snapshot isolation (SI), which incurs zero service interruption and minimal performance impact. We developed a shared-nothing, distributed version of PolarDB for PostgreSQL (aka *PolarDB-PG*), a commercial cloud-native database system on Alibaba Cloud [10] and applied the techniques described in this paper to *PolarDB-PG*.

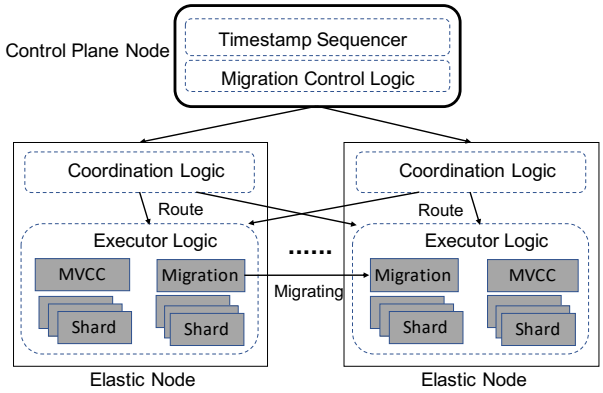


Figure 1: The architecture of *PolarDB-PG*.

Remus adopts the *push-migration* model but differs in that it eliminates any service interruption and incurs marginal performance impact during migration under different types of workloads. The key to *Remus* is a new technique called *ordered diversion* that leverages multi-version concurrency control (MVCC) and distributed SI to enable efficient unidirectional dual execution when transferring the ownership of migrating data, so as not to disrupt any running transactions.

During dual execution, newly arrived transactions are routed to the destination while allowing the existing transactions on the source to run to completion. We propose a concurrency control protocol, namely *MOCC*, that combines multi-versioning with a variant of optimistic concurrency control (OCC) [31] to maintain transaction consistency during dual execution. *MOCC* can piggyback on existing timestamp ordering protocols (e.g., centralized timestamp scheme (*GTS*) [40] and decentralized timestamp scheme (*DTS*)) to guarantee SI during a migration.

We implemented in *PolarDB-PG Remus* as well as Squall [23], the state-of-the-art *pull* approach, and other *push* approaches that adopt locking [8, 16] or remastering [1] to transfer data ownership, and evaluated them. Multiple types of workloads are tested in the experiments, including standard OLTP workload (TPC-C and YCSB) and hybrid workloads that consist of OLTP and long-lived transactions (i.e., analytical queries and large batch inserts). The experimental results demonstrate that *Remus* is the only approach that can achieve marginal performance variation, no downtime and zero migration-induced transaction aborts simultaneously under a broad spectrum of workloads, paving the way for applying the shared-nothing architecture to a cloud database which needs to provide elasticity while guaranteeing strict SLAs.

2 BACKGROUND AND MOTIVATION

2.1 The Target System

Remus is designed for shared-nothing, distributed databases and implemented in a commercial cloud native database *PolarDB-PG*, which is being open-sourced [11]. *PolarDB-PG* leverages MVCC and timestamp ordering across nodes to support SI [6, 7, 40]. The overall architecture is shown in Figure 1. The main components consist of a control plane node and multiple elastic nodes. The control plane is responsible for timestamp service of *GTS* and migration controller. Each elastic node is a PostgreSQL based instance mainly consisting of coordination, execution and timestamp ordering based concurrency control logic. Clients can submit requests to any one of the elastic nodes to execute transactions. When a node accepts a client connection, it starts a coordinator process and acts as a coordinator node performing distributed query planning and coordinating the query execution across nodes. Those nodes create worker processes to execute queries in a transactional way.

In *PolarDB-PG*, each user table is sharded across nodes by using consistent hashing [20]. Each shard is managed as a regular table on one node and the indices and constraints of the user table are created for its shards. A shard map is maintained on each node and is used to route queries to appropriate nodes.

2.2 Distributed Snapshot Isolation

SI has become a widely used transaction isolation level because it allows high concurrency between reads and writes, which is usually built on top of MVCC and is employed by many modern DBMS systems (e.g., PostgreSQL, MySQL and Oracle) [5, 9, 25, 37]. *PolarDB-PG* combines distributed MVCC with timestamp ordering to provide SI across nodes [6, 7, 22, 40]. It adopts the multi-version store of PostgreSQL and extends the tuple header to accommodate commit timestamp along with the unique id (*xid*) of the transaction that creates the version. PostgreSQL uses a commit log (*CLOG*) to record each transaction's status. When one transaction completes, it sets its status as committed/aborted in the *CLOG*, which can be looked up by other transactions during MVCC visibility validation. We extend the *CLOG* to record commit timestamps: when committing one transaction, its commit timestamp is stored in the *CLOG*.

To guarantee SI, *PolarDB-PG* must guarantee that one transaction T_R can see another transaction T_W's writes iff T_R's start timestamp is equal to or larger than T_W's commit timestamp. In an asynchronous network, it is challenging to maintain such timestamp order in a distributed database as transaction start/commit messages may arrive at different nodes in any order.

As the two-phase commit (2PC) protocol is used in distributed transactions for atomicity, the 2PC prepare-wait mechanism [22, 40] is adopted to maintain timestamp order across nodes consistently. Specifically, each distributed transaction tags its transaction status as prepared (a reserved special timestamp) in the *CLOG* in the prepare phase. In the commit phase, a commit timestamp is assigned to the transaction. Its prepared status in the *CLOG* is replaced with the commit timestamp. For a single-node transaction, it also marks its status as prepared in the *CLOG* first before assigning its commit timestamp. To read a tuple, a transaction (T_R) traverses its version chain until finding the latest version that is committed with the commit timestamp before T_R's start timestamp. For each traversed version, the status and commit timestamp of the transaction (T_W) that creates the version is consulted from the *CLOG* to perform visibility checking. If the result is a prepared status, T_R would wait for T_W to complete.

The correctness of the prepare-wait mechanism [22, 40] is derived as follows. T_R performs any MVCC reads after it acquires a start timestamp. T_W is assigned commit timestamp after its prepare phase completes. Existing timestamp protocols obey either a globally increasing property [40] or Lamport's causality increasing

property [30, 32]. Hence, if T_R's start timestamp >= T_W's commit timestamp and they have access dependencies, T_W must complete its prepare phase when T_R starts to perform reads. Then T_R would see T_W's commit timestamp or prepared status. In the latter case, T_R would wait for T_W to complete and see its writes.

Centralized Coordination: *PolarDB-PG* supports centralized coordination for timestamp generation by using a centralized sequencer called *GTS* [6, 40]. The *GTS*, implemented in the control plane node, generates monotonically increasing timestamps across nodes. Each transaction interacts with *GTS* to request a start timestamp as a snapshot to perform reads and a commit timestamp when committing it. The globally monotonic timestamps guarantee linearizability.

Decentralized Coordination: To avoid the centralized bottleneck, we also use a decentralized timestamp scheme, namely *DTS*. Each server maintains a Hybrid Logical Clock (a variant of [30]), which is a mixture of a logical time (*LT*) [32] with a synchronized physical time (*PT*) [1]. The key property of *DTS* is to leverage *LT* to track causal ordering among transactions for ensuring SI while using *PT* to generate fresh snapshots across nodes.

The choice between *GTS* and *DTS*: *GTS* guarantees linearizability across sessions. *DTS* can guarantee linearizability across sessions that connect to the same node. For sessions starting on different nodes, *DTS* allows stale snapshot reads due to physical clock skew among nodes while ensuring SI like in many other systems [1, 19, 22, 48]. Users can use *DTS* for better performance if their application tolerates such stale reads within clock skew across sessions.

2.3 Analysis of Existing Migration Approaches

2.3.1 Migration Costs. Data migration usually introduces service downtime, performance degradation, numerous aborted transactions and transaction restarts [17, 34]. We call them migration costs. Among them, transaction aborts or restarts result in wasted resource consumption and increased execution time, especially in cases of long-running transactions which are common. In scenarios such as IoT [26] and hybrid serving/analytical processing [27], the edge computer nodes or other service systems collect real-time streaming data generated by edge devices or by online user activities. The data is then ingested into a database system using long-running batch transactions. Such data generation and ingestion happen continuously as these scenarios require real-time computation: performing analytical queries as well as point lookups over the latest data, in order to generate business intelligence reports or to retrain machine learning models [26, 27].

Another type of long-running transactions are analytical queries. Furthermore, a fusion of analytical queries and large batch inserts, such as *INSERT...SELECT*, is often used to insert aggregated results from one or multiple tables into another table in real-time analytics [16, 41]. Restarting a transaction inside a database is hard to implement, and sometimes impossible for interactive transactions, since partial results may already be sent to applications [45].

2.3.2 Pull Migration. Squall [23] is a state-of-the-art *pull* based migration which combines asynchronous background pulling with on-demand pulling to migrate data. The former is used to migrate the data in the background while the latter pulls missing data accessed by ongoing transactions during a migration. Squall divides continuous key ranges into chunks, and migrates a data chunk at one time for both reactive and background pulls. A migration-status tracking table is created on both the source and destination to track each chunk's on-the-fly location. Once a data chunk is migrated, transactions that access it on the source node would be aborted and retried on the destination node. Squall locks the source and destination data partitions during a *pull* to prevent any concurrent access to the migrating chunk, so as to maintain consistency. This partition locking mechanism is provided by H-store [39], the OLTP database system where Squall is implemented. Although Squall optimizes the *pull* migration compared to Zephyr [24], it still suffers severe performance degradation during a migration [34, 49], as also demonstrated in our evaluation. This is because the lengthy reactive and background pulls can block many concurrent transactions accessing the migrating data, leading to significant throughput drop. Meanwhile, Squall does not fit long running transactions as they may hold partition locks for a long time, blocking other concurrent access and migration *pulls*.

2.3.3 Push Migration. Compared to pull migration, push migration adopts an iterative-state-copying (*ISC*) method [17, 18, 49] to support live migration. At the start of a migration, a snapshot of shards to be migrated is created on the source node and is then copied to the destination node. Incremental updates are tracked during snapshot copying since transactions are still routed to the source node. Those tracked updates are sent to the destination iteratively during the catch-up phase. When the number of un-synchronized updates drops below a threshold, *ISC* enters into the ownership transfer phase. At this phase, active transactions are blocked and cannot access the migrating data. The remaining final updates are copied to the destination before transferring the ownership of migrating data. Recent work [1, 8, 16–18, 49] follows this *ISC* method. The main difference among them lies in how to achieve atomic ownership transfer, which can be classified into *lock-and-abort*, *suspend-and-resume*, and *wait-and-remaster* techniques.

Lock-and-abort [8, 16]: During the ownership transfer phase, this approach locks the migrating shards to prevent any writes, replays the final updates and then modifies the shard map table on all coordinator nodes to route incoming transactions to the destination. Any transactions that hold the locks in a conflict mode are terminated in advance when trying to lock the shards. When the transfer completes, the blocked transactions are aborted. This approach can result in many write transaction aborted and may severely impact the performance of long-running transactions.

Suspend-and-resume [17, 18, 49]: During the ownership transfer phase, Albatross [17, 18] suspends all active transactions accessing the migrating data on the source. The approach then copies the final updates as well as the state of active transactions to the destination, and resumes them on the destination. Such approach, designed for simple OLTP workloads, is not suitable for queries that consist of complex operators such as joins and aggregations. Those queries may build a large amount of intermediate results, such as hash tables for hash join. Migrating those intermediate results during the ownership transfer phase may increase service downtime,

[1] We can use NTP (Network Time Protocol) or PTP (Precision Time Protocol) to synchronize physical time between servers.

	Lock [8, 16]	Suspend [18, 49]	Remaster [1]	Squall [23]	MgCrab [34]	Remus
Service Downtime	Yes	Yes	Yes	No	No	No
Transaction Abort	Yes	No	No	Yes	No	No
OLTP Throughput Drop	Low	Low (Not Always)	Low (Not Always)	High	Low	Low
Batch Throughput Drop	High	Low	Low	Median	Not Sure	Low
Support of Interactive transaction	Yes	Yes	Yes	Yes	No	Yes
Concurrency Control	MVCC	OCC	MVCC	Partition Lock	Determinism	MVCC

Table 1: Comparison of existing state-of-the-art migration approaches with Remus. "Batch Throughput Drop" means the throughput drop of batch write transactions. "Not Always" means the OTLP throughput drop may be high in case of long lived transactions. Lock, Suspend and Remaster are short for *lock-and-abort*, *suspend-and-resume* and *wait-and-remaster*.

leading to a significant throughput drop. Restarting transactions on the destination may remedy the above problem. However, it is not feasible for interactive transactions and introduces significant cost for long-running transactions.

Wait-and-remaster: DynaMast [1] proposes a light-weight remastering protocol to transfer data ownership among multiple fully replicated nodes. This remastering protocol can be leveraged to migrate data while introducing no transaction aborts in the migration scheme. During the ownership transfer phase, *wait-and-remaster* suspends routing newly-arrived transactions, waits for ongoing transactions of writing the migrating shards to complete (**wait**) and then updates the shard map table to route incoming transactions to the destination (**remastering**). This approach may lead to lengthy downtime in the case of long-running transactions that are writing the migrating data. Furthermore, as the transaction write set is often unknown prior to execution, it needs to wait for any on-the-fly transactions to complete, even though they would not access the migrating data.

2.3.4 Dual Execution Migration. In order to eliminate downtime and interruption, MgCrab [34] proposes a dual-execution based migration for shared-nothing deterministic databases [46]. In a deterministic database like Calvin [46], concurrent transactions are executed in a globally predetermined order on each node via sequencers. MgCrab leverages such determinism to route each incoming transaction during migration to both source and destination node while maintaining consistency. MgCrab guarantees that each pair of dual-execution transactions would run on a consistent view of both nodes because of deterministic execution. The source node transaction reactively pushes missing data chunks to the destination to assist its paired transaction in accessing not-yet-migrated data on the destination. MgCrab [34] is tailor-made for deterministic databases, which need to determine a global execution order for a batch of transactions prior to execution. As a result, MgCrab cannot support interactive transactions. Moreover, MgCrab relies on lock-based deterministic concurrency control, which prevents its application to the widely used MVCC database systems.

2.3.5 Summary. In Table 1, we compare existing state-of-the-art migration techniques with Remus in six dimensions. Some dimensions such as downtime and transaction aborts have been discussed in the context of OLTP transactions in prior work [34]. We focus on the migration impact on complex hybrid workloads consisting of both short OLTP and long lived transactions. In term of downtime and transaction aborts, only MgCrab and Remus can achieve both no downtime and zero transaction abort. For the throughput of short OLTP transactions, most approaches except for Squall introduce insignificant performance degradation. However, only Remus and *lock-and-abort* can avoid the impact on OLTP throughput in the presence of long-live transactions. *lock-and-abort* can cause large throughput drop for batch write transactions as it may abort them during migration. MgCrab is implemented on a deterministic database and cannot support interactive transactions. By comparison, Remus is based on MVCC and can support more general workloads with high performance, including interactive transactions, batch and analytical transactions and stored procedures.

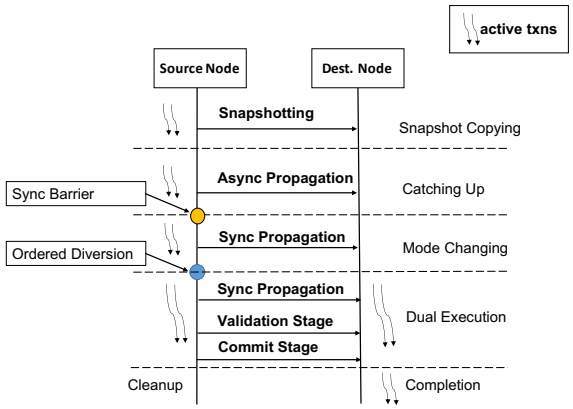


Figure 2: The migration phases in *Remus*. The yellow solid dot denotes the event of sync barrier that makes the source enter into sync propagation mode. The blue solid dot denotes the event of ordered diversion after which newly arriving transactions are redirected to the destination.

3 DESIGN

3.1 Overview

The primary design goal of *Remus* is to support live migration of shards between nodes with zero service interruption and marginal performance impact. The approach should be applicable to general-purpose databases under a wide variety of workloads, e.g., short OLTP transactions, long batch and analytical transactions and hybrid workloads. *Remus* adopts the *push-migration* method to migrate shards between nodes. Differing from previous work on *push-migration*, *Remus* supports efficient dual execution when transferring the ownership of migrating data to the destination node, so as to avoid any interruption and to minimize performance impact.

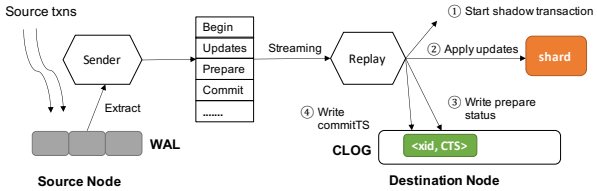


Figure 3: The update propagation in *Remus*.

As shown in Figure 2, the migration process of a shard in *Remus* includes four phases: snapshot copying, asynchronous update propagating, sync mode changing and dual executing. In the first phase, *Remus* leverages MVCC to create a snapshot of a shard and copies the snapshot to the destination node without interfering normal transaction processing. During async propagation phase, changes committed after the snapshot are propagated continuously to the destination (*async execution phase*). When the destination catches up with the source node, *Remus* changes the propagation mode from asynchronous to synchronous mode by using a *sync barrier*, thus entering into the *sync execution phase*. Then the unidirectional dual execution starts by performing *ordered diversion*: newly arrived transactions accessing the migrating data are routed to the destination node while allowing the execution of existing transactions on the source node to run to completion without any interruption and suspension. A concurrency control protocol, namely *MOCC*, is adopted to maintain transaction consistency and SI between the source and destination nodes during dual execution.

We call a transaction that runs on the source node and has its writes propagated to the destination node as a source transaction. A transaction that is forwarded to the destination node to access the migrating data is termed a destination transaction.

3.2 Snapshot Copying

Remus uses multi-versioning to create a snapshot for a shard to be migrated and then installs the snapshot on the destination. Specifically, *Remus* scans the target shard to retrieve valid tuple versions that are committed before the snapshot timestamp. The retrieved tuples are then inserted into an empty shard on the destination node with a reserved minimal commit timestamp. This allows those data to be visible to any transactions on the destination starting after the snapshot. The snapshot scan and installation are executed in a streaming way, such that no extra storage is required for the migrating shard.

3.3 Asynchronous Update Propagation

In this stage, *Remus* asynchronously propagates to the destination node the changes of the migrating shard that are committed after the snapshot timestamp. This process does not interfere the source's transaction processing so as not to impact its performance.

In order to minimize the performance impact on user queries, *Remus* uses Write-ahead Log (WAL) [36] of PostgreSQL to track the changes that need to be propagated. Since each transaction's updates are logged in the WAL, incremental changes over a snapshot can be tracked by traversing WAL records.

As shown in Figure 3, *Remus* starts a propagation (send) process on the source node to propagate the changes of committed transactions from the WAL to the destination node. The propagation process reads streaming records from the WAL continuously and builds a update cache queue for each encountered transaction to cache its modifications extracted from the WAL. Note that *Remus* only extracts changes related to the migrating data. When encountering a commit record of a transaction, the propagation process sends all its change records in its update cache queue to the destination if its commit timestamp is greater than the snapshot timestamp. If a source transaction is found to be aborted in the WAL or its commit timestamp is smaller than or equal to the snapshot timestamp *Remus* simply drops its modifications and releases its update cache queue. For transactions with a large write set *Remus* also allows their change records being spilled to disk. When starting to propagate one spilled transaction, *Remus* would reload and send its spilled change records in batches.

On the destination node, *Remus* creates a replay process that applies the propagated changes of each source transaction in sequence. Specifically, for each source transaction the replay process starts a shadow transaction with the same start timestamp to re-execute its changes and uses the same commit timestamp to commit. As all propagated changes are applied in the same order as they commit on the source node, the data of the migrating shard on the destination is consistent to that on the source.

In *PolarDB-PG*, each shard table has a primary unique key. Each propagated change record includes the primary key value of the modified tuple. For each propagated record that modifies an existing tuple, the replay process first locates the tuple by using a primary index scan and then applies the modification.

Remus uses the above framework for update propagation in both synchronous and asynchronous modes as shown in Figure 2. In asynchronous propagation mode, any source transaction can complete immediately after its updates are flushed to the WAL, which are propagated to the destination node asynchronously. In synchronous propagation mode, a source transaction needs to wait for its updates to be applied on the destination and uses *MOCC* to commit (§3.5.2).

3.4 Propagation Mode Changing

When the number of changes that have not been applied on the destination drops below a threshold, the live migration performs ownership transfer and enters into dual execution phase. During unidirectional dual execution, a destination transaction should be able to access the changes of a source transaction that commits before the destination transaction starts. This requires all updates of migrating shards to be available on the destination. *Remus* achieves it by ensuring that: first, source transactions running through dual execution have their updates propagated to and applied on the destination before they commit; second, all updates committed before dual execution are available on the destination. A mode changing phase is introduced to achieve this goal.

For the first guarantee, a *sync* barrier is adopted to change the asynchronous propagation mode to a synchronous mode. To create a *sync* barrier, *Remus* sets a flag in a shared memory area of the source node. This flag is checked by source transactions before they

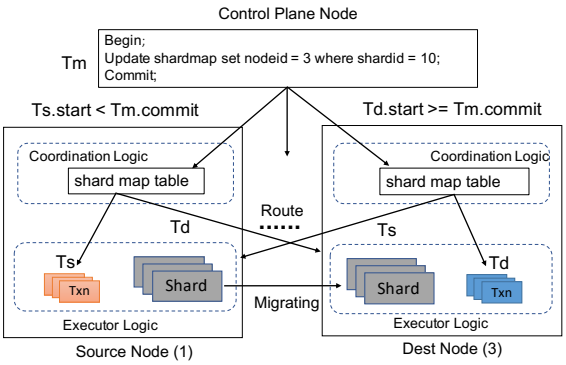


Figure 4: The illustration of *ordered diversion* in case of migrating shard 10 from node 1 to node 3.

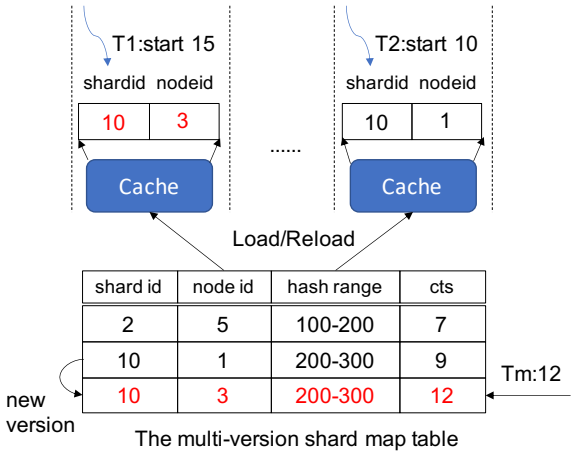


Figure 5: The illustration of shard map table and its cache. cts: the commit timestamp of a tuple version.

commit. When the flag is set, the commit progress will wait until its updates have been propagated to and applied on the destination. We call these transactions as synchronized source transactions. Synchronized source transactions use *MOCC* (§3.5.2) to commit such that when entering dual execution any source transactions would have their updates propagated to the destination and write-write conflicts (*WW*-conflicts) resolved on the destination before they can commit.

For the second guarantee, *Remus* records all transactions on the source already entering their commit progress before the *sync* barrier is set. The set of these transactions is called TS_{unsync}. After all the transactions in TS_{unsync} complete, the current flush (tail) position in the WAL is recorded as LSN_{unsync}, which can cover all the updates from TS_{unsync}. When all the migrating-shard related changes till LSN_{unsync} have been propagated to and applied on the destination, the dual execution mode can be started, since all committed updates from TS_{unsync} are available on the destination and newly arrived source transactions use *MOCC* to commit. During the mode changing phase, incoming transactions are routed to the source node and run without any interruption and suspension.

3.5 Dual Execution

As shown in Figure 2, after the propagation mode changing phase completes, *Remus* performs shard ownership transfer to enter into dual execution. During the dual execution, existing transactions continue to run on the source node, called source transactions, while newly arriving transactions accessing the migrating shard, called destination transactions, are directed to the destination node. In order to ensure SI between source transactions and destination transactions, the dual execution phase should address two key problems: (1) how to ensure that both source and destination transactions run on a consistent view of the migrating data, and (2) how to resolve *WW*-conflicts and maintain timestamp order between source transactions and destination transactions. The former is solved by using a technique called *ordered diversion* while the latter is achieved through a concurrency control protocol termed *MOCC*.

3.5.1 Ordered Diversion. In order to maintain a consistent snapshot of migrating data between concurrent source and destination transactions, we propose an *ordered diversion* technique that enables a unidirectional propagation based dual execution with low overhead. The key to the *ordered diversion* is to leverage distributed multi-versioning metadata and SI to steer incoming transactions to the appropriate node during dual execution.

As shown in Figure 5, *Remus* organizes the shard map on each node as a regular multi-versioning table which we call shard map table. During shard ownership handover, a distributed transaction, called T_m, is employed to update the table on each node. The shard map table records the shard ID, consistent hash range, and node ID for each shard. To route a query of one transaction T_1, the coordinator process parses the query predicates to determine the shards that the query needs to access and then reads the shard map table using T_1's start timestamp as a snapshot to determine the nodes where the target shards lie. As shown in Figure 4, T_m actually updates the node ID of the migrating shard and uses 2PC to commit. As a result, a transaction starting after T_m commits is steered to the destination node to access the migrating shard. The commit timestamp of T_m becomes a barrier to divide transactions into two groups: those starting before the barrier are routed to the source (T_s) and others are directed to the destination (T_d).

For example, as shown in Figure 5, as T2's start timestamp (10) is smaller than T_m's commit timestamp (12), it would be still routed to the source node while T1 is directed to the destination as T1's start (15) is larger than T_m's commit timestamp. However, within an asynchronous network, T_m's commit message may arrive on the node handing T1 after T1 starts which leads to T1 not seeing T_m's updates. *Remus* relies on existing timestamp protocols as discussed in §2.2 to ensure timestamp order consistency among transactions across nodes. Specifically, the prepare-wait mechanism is adopted to ensure timestamp order.

Theorem 3.1. *Using T_m as an ordering barrier, Remus enables unidirectional propagation from the source node to the destination node, as the changes made to the migrating data on the destination node are invisible to the transactions running on the source node.*

Under dual execution, we assume any two transactions T_s and T_d, that access the migrating shard, on the source and destination node, respectively.

Proof. Although T_s and T_d may be coordinated by different nodes, *Remus* leverages timestamp ordering to guarantee that T_m's effect becomes visible across nodes consistently (§2.2). A transaction is directed to destination if and only if its start timestamp is greater than or equal to T_m's commit timestamp (T_m.commitTS), no matter which node the transaction is coordinated and routed. As T_d is directed to the destination node, it must see T_m's changes made on the shard map, i.e., T_d.startTS >= T_m.commitTS. Similarly, as T_s runs on the source node, T_m's updates are invisible to T_s, indicating T_s.startTS < T_m.commitTS. As a transaction's commit timestamp is always larger than its start timestamp, we can derive that

$$T_d.commitTS > T_d.startTS > T_s.startTS \quad (1)$$

Consequently, T_d's writes are invisible to T_s. □

Consistency of shard map cache *PolarDB-PG* builds a fast private *ordered* cache from the shard map table for each coordinating process to speed up query routing as shown in Figure 5. The private cache is ordered by the consistent shard hash ranges and is organized as an ordered array, so as to enable fast binary searching to locate the shard for point-lookup and efficient shard pruning for range-scan.

The adoption of private ordered cache would sabotage the transaction semantic of T_m that the *ordered diversion* relies on to enable unidirectional dual execution. After T_m commits, the cache in each coordinator process would be invalidated and reloaded. However, there is a vulnerable time window between the event of T_m's commit and that of invalidation. As a result, stale shard map values in the cache may still be used to route one transaction T_1 with start timestamp larger than T_m 's commit timestamp after T_m commits. To amend this, *Remus* adopts a strategy that marks each node as *cache-read-through* state with migrating shard IDs temporarily before the execution of T_m and clears the state after T_m commits. When one coordinator process starts to route a transaction (e.g., T1) during *cache-read-through*, it first uses T1's start timestamp to read the tuples corresponding to the migrating shard IDs from the shard map table. Then its cache is updated if there are new visible tuple versions. If the *cache-read-through* state ends up with one process's cache entries having not yet been updated, the process will refresh its cache entries to the new version from the shard map table after completing the current transaction. This is safe as subsequent transactions would be assigned start timestamps larger than T_m.commitTS.

As shown in Figure 5, since *T1.* startTS is larger than T_m.commitTS, the process coordinating T1 refreshes its cache with T_m's updates and then routes T1 to the destination node. In contrast, the cache of the process routing T2 is stale as T_m's updates are invisible to T2 and T2 is still directed to the source.

3.5.2 MOCC. We propose *MOCC*, a concurrency control protocol combining multi-versioning with a variant of OCC [31], to ensure SI between source and destination transactions. For a source transaction T_s and a destination transaction T_d, T_d's updates are invisible to T_s under SI according to Theorem 3.1. There are no write-read (*WR*) dependencies from T_d to T_s. *MOCC* does not need to validate the read set of each source transaction. Hence, only *WW*-conflicts between T_d and T_s need to be processed. *MOCC* allows concurrent read and write accesses to the migrating shard on both nodes during dual execution while *WW*-conflicts are resolved on the destination.

In *MOCC*, the destination node starts a shadow transaction for each source transaction to execute its propagated changes. The shadow transaction is assigned the same start timestamp as its source transaction and runs as a normal transaction conducting reads and writes to versioned tuples and following SI. Its updates can be read by destination transactions following MVCC visibility validation. Constraint checking and tuple locking are also maintained for each change on the destination node.

In *MOCC*, each source transaction is committed by using two stages: validation stage and commit stage. *MOCC* utilizes the framework offered by the 2PC protocol to manage source transactions and their shadow transactions, e.g., when a shadow transaction rolls back, its source transaction does so.

Validation stage: During this stage, the source transaction writes a validation record, a special 2PC prepare log record to the WAL, and waits for the validation outcome from the destination. Upon a special prepare log record of a source transaction, the propagation process sends its changes in its update cache queue to the destination node.

For each propagated write of T_s, its shadow transaction T_{dual} re-executes its change on the destination. Specifically, T_{dual} inserts a new tuple for an insertion record from T_s. For any record of update, delete or explicit row-level lock, a valid tuple version according to T_s's start timestamp is first retrieved. Then the retrieved tuple version is checked whether it is has been marked as dead (i.e., deleted) or there are newer versions (updated). If so, both T_{dual} and T_s are aborted as the tuple is being or has been modified by other destination transactions, indicating a *WW*-conflict. Otherwise, T_{dual} re-executes the operation according to the propagated record. After all changes are validated and re-executed successfully, both T_{dual} and T_s can be committed.

Commit stage: If any *WW*-conflict occurs during the validation stage, *Remus* aborts both the source and its shadow transaction. Otherwise, the replay process uses 2PC to first prepare the shadow transaction and sends an ack of *validation-ok* back to the source node. Then the source transaction determines a commit timestamp and writes a commit record to the WAL, which would be propagated to the destination node asynchronously. Upon receiving the commit record, the destination node commits the corresponding prepared shadow transaction with the same commit timestamp.

When the source transaction is a distributed transaction and uses 2PC to commit, *Remus* combines its validate stage with the 2PC prepare phase. If the source transaction finally decides to abort due to failures on other nodes, the source node writes a rollback prepared record in the WAL, which would be propagated to the destination to roll back the shadow transaction. Otherwise, a commit prepared record for the source transaction would be written to the WAL on the source node, which would be propagated to the destination to commit the prepared shadow transaction with the same commit timestamp. The destination node starts a separate apply process to handle commit/rollback (prepared) records.

Distributed SI: *MOCC* can piggyback on existing timestamp ordering protocols (e.g., *GTS* and *DTS*) to provide SI during migration. Recall that *PolarDB-PG* adopts prepare-wait mechanism [22, 40] to maintain timestamp order between transactions across

nodes as discussed in §2.2. During dual execution, the validation stage propagates the prepare event of each source transaction to the destination which uses 2PC to prepare its shadow transaction. The commit timestamp of one source transaction is assigned after both the source and its shadow transaction complete the prepare phase. And the shadow transaction would commit with the same timestamp with its source transaction. The prepared status of a shadow transaction would block concurrent MVCC reads over its writes on the destination to maintain timestamp order between source and destination transactions.

3.6 Performance Impact

The main cost induced by *Remus* is the increased latency of source transactions as they need to wait for their shadow transactions to complete during synchronous propagation. The latency depends on two factors: (a) the speed of replaying propagated changes, i.e., assuming $speed_{replay}$; (b) the update speed to the migrating data, i.e., $speed_{update}$. If $speed_{replay}$ exceeds $speed_{update}$, the destination node can catch up with the source during the asynchronous propagation phase, making the performance impact minimal during the dual execution as demonstrated in our experiments (§4). Otherwise, the destination node would fail to catch up with the source and the mode changing phase can result in significantly increased latency during this phase.

One way to control performance impact is to integrate parallel replay mechanism [19] to increase $speed_{replay}$. On the other hand, shards can be managed in a flexible way as Akkio [2] and Spanner [15] to control $speed_{update}$. *Remus* implements a transaction-level parallel apply approach based on SI by tracking timestamp order.

3.7 Crash Recovery

As described in §3.5.1, *Remus* adopts a distributed transaction T_m to update the shard map table on each node. If any failure occurs during a migration, *Remus* decides to continue or roll back the unfinished migration according to whether T_m commits. If any failure occurs during T_m, the migration controller first recovers T_m by using 2PC recovery. If T_m is not committed, no transactions are directed to the destination node. The source node contains all updates to the migrating data. *Remus* terminates the ongoing migration and cleans up the partially migrated data on the destination node. The uncompleted migration can be initiated again. If T_m is found committed, *Remus* continues the migration as the destination node contains some latest updates of the migrating data. For both cases, *Remus* should first clean up residual shadow and source transactions.

Our recovery mechanism is based on the key property of *MOCC*: each source transaction is committed only after its shadow transaction has been prepared successfully. After a crash, *PolarDB-PG* follows normal recovery process of a distributed database system that cleans up residual distributed transactions. A 2PC transaction would be committed only if it enters into the second phase before a crash. This process can recover any prepared source transactions. Then the recovery of each prepared shadow transaction takes the same action as its source transaction. If its source transaction is committed, *Remus* queries its commit timestamp from the source node and commits the shadow transaction using the same timestamp. Otherwise, the shadow transaction would be rolled back. Note that any source transaction waiting for its validation stage result would be terminated first in the case of a crash occurred on the destination node. After recovering transactions in dual execution, *Remus* cleans up either the migrated data on the destination node or on the source node according to whether *Tm* commits.

Our approach is orthogonal to the fault-tolerant model (such as consensus-based and primary-backup replication) adopted by the database. Each node can have several synchronized replicas. In the event of failures occurred on the source and/or destination nodes, its one replica will take over as the new primary (leader) node. Then the above recovery process can be performed to clean up residual source and shadow transactions and complete the unfinished migration.

3.8 Collocated Migration

In order to improve the performance of analytic queries, distributed databases usually support collocated sharding between tables such that joins on the sharding keys between collocated tables can avoid expensive data shuffling [16, 38, 44]. Migrating one shard can sabotage such collocation, thus harming the performance of analytic queries. One way to amend such issue is to migrate collocated shards one by one. However, collocated joins still suffer from data shuffling during the period of migration.

To address this problem, *Remus* supports collocated migration,i.e., migrating collocated shards together. Specifically, *Remus* copies the snapshots of collocated shards in parallel to the destination node and then propagates their updates continuously during the *async* and *sync* execution phases. The update propagation scheme and dual execution remain the same with that of migrating one shard. *Remus* also supports migrates more than one shards that may be non-collocated at one time in a similar way.

4 EXPERIMENTAL EVALUATION

4.1 Setup

Our experiments were conducted on Alibaba Cloud using a cluster of six i2.16xlarge ECS (server) with CentOS 8.0 Linux. Each server contains 64 vCPU, 512 GB DRAM and NVMe SSDs and connects to other servers through a 10 Gbps network. We deploy one elastic node of *PolarDB-PG* on each server. Synchronous WAL logging and periodic checkpoints are enabled for all experiments. We run the benchmark workload clients on a separate server with the same hardware configuration. The migration controller node is deployed on a standalone server. As *DTS* shows much better performance than *GTS*, all the experiments are conducted on *PolarDB-PG* with *DTS* supporting SI. In all the experiments, 18 threads are started to apply changes in parallel on the destination node. Such parallelism is large enough to make $speed_{replay}$ exceed $speed_{update}$ (§3.6) in our experiments.

4.2 Approaches in Comparison

We compare *Remus* with some state-of-the-art approaches discussed in §2: *lock-and-abort*, *wait-and-remaster* and *Squall*. For fair comparison, we implemented them in *PolarDB-PG*. Both *lock-and-abort* and *wait-and-remaster* adopt the same snapshot copying, update propagation, and parallel apply protocols as *Remus*. For *wait-and-remaster*,

in order to support general transactions, our implementation waits for all ongoing transactions to complete during the ownership transfer phase. As Squall leverages partition-locks in H-store [39] to maintain consistency during pulling, an equivalent shard locking mechanism is implemented on top of MVCC to support *pull* migration. We call this approach as *PolarDB-Squall* or Squall for short. The pull chunk size is set to 8 MB, as suggested in the Squall paper [23]. We split ranges into 8 MB approximately. To speed up multi-shard migration, Squall starts multiple asynchronous workers, each of which pulls one migrating shard in the background. We do not implement MgCrab [34] since it is specifically designed for deterministic databases, such as Calvin[46].

4.3 Workloads

The TPC-C [47] workload: We create a TPC-C database of 480 warehouses which are partitioned into shards across nodes. Each shard contains a table's data belonging to one warehouse. Shards of different tables are collocated on the same node according to their warehouse ID. Transactions accessing only one warehouse are executed in one single node so as to avoid distributed transactions. The same number of clients as warehouses are started to perform TPC-C transactions. The default TPC-C configuration is used and the workload mainly consists of 45% *New-order* and 43% *Payment* transactions, among them around 10% of which are distributed transactions. Each client selects a warehouse as its home warehouse and randomly chooses a different remote warehouse for a distributed transaction. TPC-C thinking time between transactions is eliminated to produce high throughput as in OLTP-Bench [21].

The YCSB [14] workload: In the experiments, we create a YCSB database that consists of 100 million tuples. Each tuple has a 8 Byte primary key and is of around 1 KB size, resulting in a total of 100 GB data. We generate a YCSB workload consisting of 50% reads and 50% updates with a uniform or a skewed access pattern. The YCSB transactions are executed in a multi-statement interactive mode where each read/update statement is wrapped with explicit transaction *BEGIN* and *COMMIT* statements. As a result, the write-set of each YCSB transaction is unknown prior to execution. *wait-and-remaster* needs to wait for all the ongoing YCSB transactions to complete during the ownership transfer phase. For the YCSB database, we create 360 shards across 6 nodes by default. Each node owns 60 shards to simulate 60 partitions per node in H-Store, enabling a fine-grained parallelism for Squall.

Hybrid workload A: This hybrid benchmark runs a uniform YCSB workload with 400 clients while starting a batch ingestion client which issues 10 batch insert transactions to one coordinator node in a tight loop, simulating a mixed workload of point queries/updates and real-time data ingestion [26, 27]. Each batch insert transaction appends one million tuples into the YCSB table by using PostgreSQL's *COPY* command. The coordinator node executing a *COPY* command extracts stream data from a given file into tuples, routes them to the corresponding shards and finally uses 2PC to commit all ingestion. The batch client is collocated with the coordinator node and uses scripts to generate data files in advance. Data tuples are generated with monotonically increasing primary (sharding) keys starting from the maximum key value in the YCSB table plus one and are of 1 KB tuple size. For migration-induced aborts, we add repeatable retry logic for the batch insert client.

Hybrid workload B: The hybrid benchmark runs a uniform YCSB workload with 400 clients while executing an analytical query over the YCSB table, simulating hybrid transactional and analytical processing (HTAP) workload. The analytical query is defined as follows: *Begin; with checkresult as (select count(*)=1 as x from accounts group by aid) select count(*) from checkresult where x!='t'; Commit.* This query checks whether there are duplicated primary keys (*aid*) in the YCSB table *accounts* across nodes. We use it to verify database consistency during a migration.

4.4 Cluster Consolidation

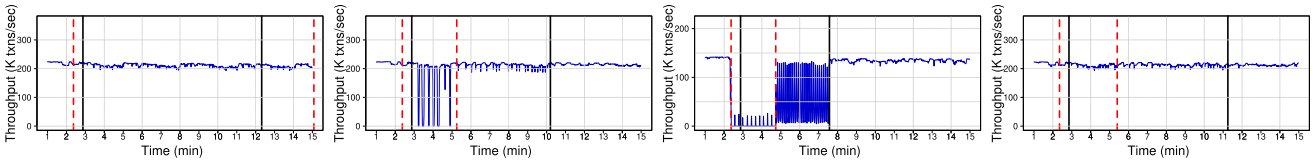
We compare *Remus* to the baselines under a scenario of cluster consolidation. The cluster consolidation removes one node from a six-node cluster. All of the shards on the source node (60 shards) are migrated to other nodes evenly.

4.4.1 Hybrid Workload A. The cluster consolidation is conducted after a 30-second run of the batch insertion workload. Two shards are migrated together each time, resulting in 30 consecutive migrations. As shown in Table 2, 97% of the batch insert transactions are aborted by *lock-and-abort* during the period of consolidation, resulting in significant throughput drop during consolidation for the batch insert workload, around 1/33 of that before consolidation. The batch ingestion takes 3X more time to complete in *lock-and-abort* than in *Remus*. The abort ratio of *lock-and-abort* is related to the average elapse time of each batch transaction and the interval time between consecutive migrations. If the former is longer, *lock-and-abort* can result in most batch transactions to fail.

During consolidation, Squall aborts 13% of batch transactions when they try to insert tuples into migrated ranges on the source node. Note that the ingestion throughput of Squall is higher than those of other approaches. Batch insert transactions run faster on Squall since they hold the locks of all inserting shards, at the cost of blocking YCSB transactions and having much lower YCSB throughput. *Remus* and *wait-and-remaster* do not cause any transaction aborts. They maintain a steady insertion throughput during cluster consolidation.

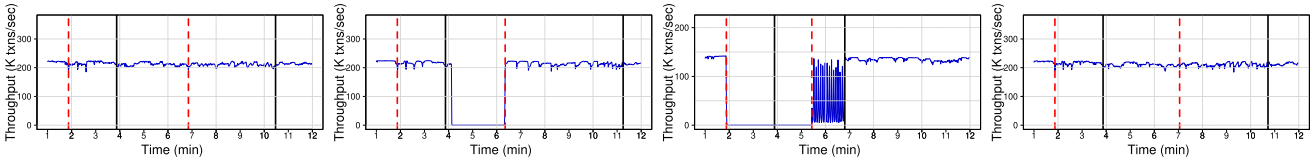
As shown in Figure 6, YCSB throughput varies slightly during consolidation for both *Remus* and *lock-and-abort*. This is because *Remus* adopts dual execution to transfer data ownership without any wait and downtime. *lock-and-abort* instead aborts long-running batch insert transactions and YCSB transactions to achieve the same goal. As a comparison, *wait-and-remaster* causes several sharp drops, even reaching zero for a short time. The reason is that *wait-and-remaster* waits for long-running batch transactions to complete during the ownership transfer phase. This blocks newly arrived YCSB transactions until the ownership is transferred. Such lengthy downtime occurs for each migration. After the batch insertion completes, *wait-and-remaster* renders marginal throughput drops as it only needs to wait for ongoing short YCSB transactions to complete.

The YCSB throughput of Squall reaches zero for most of the time during batch insertion. This is because each batch insert transaction acquires all the locks of shards to insert, resulting in significant



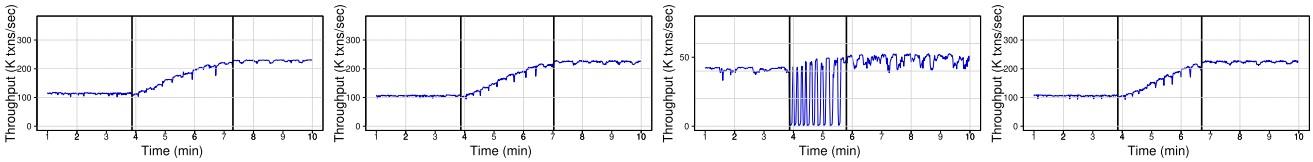
(a) Lock-and-abort (b) Wait-and-remaster (c) Squall (d) Remus

Figure 6: The YCSB throughput under hybrid workload A during consolidation. The vertical solid lines delimitate the entire migration start and end. The red dashed lines delimitate the start and the end of the batch insert workload.



(a) Lock-and-abort (b) Wait-and-remaster (c) Squall (d) Remus

Figure 7: The YCSB throughput under hybrid workload B during consolidation. The vertical solid lines delimitate the entire migration start and end. The red dashed lines delimitate the start and the end of the analytical transaction.



(a) Lock-and-abort (b) Wait-and-remaster (c) Squall (d) Remus

Figure 8: The YCSB throughput during load balancing.

	Lock-and-abort	Wait-and-remaster	Squall	Remus
Abort Ratio During Consolidation	97%	0%	13%	0%
Avg. Throughput During/Before Consolidation	1.8/59	59/59	67/80	55/59

Table 2: The batch insert throughput (K tuples/s) under hybrid workload A (Ingested tuple size: 1KB).

blocking time for YCSB transactions and Squall's *pulls*. After the batch inserts complete, Squall still renders large throughput fluctuation as each migration *pull* takes tens of milliseconds to pull one data chunk of around 8 MB size from the source node and to store it on the destination. This *pull* latency leads to many transactions to be blocked and delayed. The overall YCSB throughput of Squall in Figure 6c is much smaller than the other approaches as Squall adopts shard-lock for concurrency control which achieves much lower concurrency than MVCC.

Overall, under hybrid workload of batch insert transactions and short transactions, *Remus* is the only approach incurring zero transaction interruption, zero downtime, and slight throughput drop during cluster consolidation.

4.4.2 Hybrid Workload B. Figure 7 shows the YCSB performance under hybrid workload B. Four shards are migrated together each time, resulting in 15 consecutive migrations. For Squall, the YCSB throughput drops to zero when the analytical transaction runs. The latter locks all the shards, thus blocking all the YCSB clients as well as migration *pulls*. After the analytical transaction completes, the YCSB throughput starts to increase but fluctuates largely. This is mainly caused by the conflict between YCSB transactions and Squall migration *pulls*. The consolidation in Squall completes much faster than the other approaches as the *pull* migration does not transfer any extra data during migration while the catch-up phase in *push* migration propagates a number of incremental updates.

The throughput on *wait-and-remaster* drops to zero after the consolidation starts, which lasts until the analytical transaction completes. As both the analytical and YCSB transactions are executed as multi-statement interactive transactions, *wait-and-remaster* needs to wait for them to complete before transferring ownership and blocks incoming transactions, resulting in downtime.

This experiment indicates that both Squall and *wait-and-remaster* are not suitable for HTAP workloads. In contrast, both *Remus* and *lock-and-abort* shows marginal performance impact during cluster consolidation under hybrid workload B.

Workload	*Remus*	*lock-and-abort*	Txn Latency
Hybrid A	1.9	27	2.1
Hybrid B	1.7	33	2.1
Load balancing	6.6	51	2.8
Scale-out	4.1	94	4-15

Table 3: Average latency increase (ms) caused by *Remus* and *lock-and-abort*. Txn Latency: average latency (ms) of TPC-C/YCSB write transactions.

4.5 Load Balancing

In this experiment, the YCSB workload is skewed and generates 50 hotspot shards on one of six nodes. The load balancing process migrates 40 of those 50 hotspot shards to the other five nodes evenly, during which four shards are migrated together each time.

As shown in Figure 8, for *Remus*, *lock-and-abort* and *wait-and-remaster*, the YCSB throughput increases gradually and varies slightly during load balancing. However, for *lock-and-abort*, our experiment recorded several thousand transaction aborts caused by migration and a few hundred ones from *WW*-conflicts. In contrast, both *Remus* and *wait-and-remaster* incurs zero transaction interruption from migration. The throughput of Squall drops considerably and has significant fluctuation due to YCSB transactions blocked by *pulls*. The reason resulting in the throughput varying considerably after load balancing on Squall is mainly due to the severe shard-lock contention on hotspot shards.

4.6 Scaling Out

In this scenario, we use a 480-warehouse TPC-C workload and start with a five-node cluster. Initially, one overloaded node contains 160 warehouses, twice of that of the other nodes (80 warehouses). The scaling-out experiment migrates 80 warehouses of the overloaded node to one newly added node. We migrate 3 warehouses (a total of 24 shards given 8 TPC-C distributed tables) together from the source node to the destination node at one time, resulting in 27 consecutive migrations. Squall is not shown in this evaluation because our implementation does not support multi-key range partitioning.

Figure 10 shows the TPC-C throughput increases and reaches a higher number after scale-out for all approaches. The figure also shows many throughput fluctuations, which are caused by 27 migration operations and data ownership transfers. Among these approaches, *Remus* achieves much smaller throughput variation than *lock-and-abort* and *wait-and-remaster* as it adopts dual execution to transfer ownership smoothly. The throughput fluctuation of these two approaches becomes much larger than that under the YCSB workload. For *wait-and-remaster*, since TPC-C transactions are longer than YCSB transactions, the approach takes longer time to wait for all on-the-fly transactions to complete during the ownership transfer phase, resulting in significant throughput fluctuation. For *lock-and-abort*, this is because the ownership transfer phase takes much longer time to complete under such heavy workload. The performance impact of *lock-and-abort* is sensitive to the duration of the ownership transfer phase during which migrating shards are locked and accessing transactions are blocked.

4.7 Latency Increase

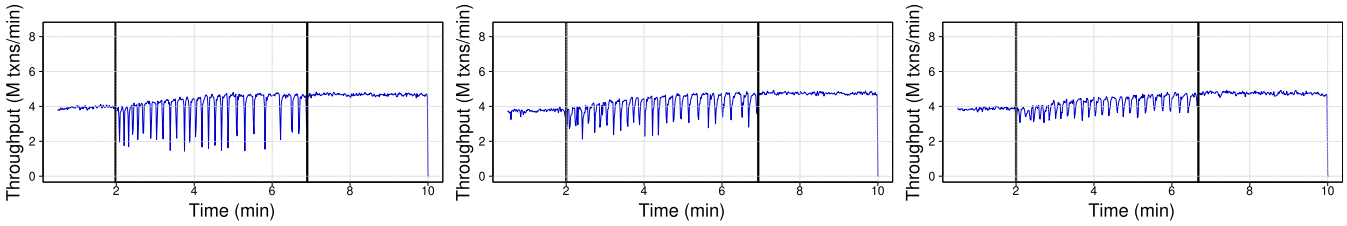
Remus may increase the latency of synchronized source transactions, as they need to wait for their updates to be applied. Meanwhile, *lock-and-abort* incurs some downtime during the ownership transfer phase, which locks the migrating shards to prevent any writes, replays all the remaining final updates and then modifies the shard map table on each coordinator node using 2PC to route transactions to the destination. This introduces latency to the source transactions which are blocked during the ownership transfer phase and then are aborted and retried after this phase completes.

Table 3 shows the average latency increase of *Remus* and *lock-and-abort* under the four workloads in our experiments. *Remus* only incurs several milliseconds (1.7 to 6.6 ms) latency increase on average. In contrast, *lock-and-abort*'s latency increase is more significant, reaching tens of milliseconds (27 to 94 ms). This is mainly because for *Remus* each synchronized transaction can commit immediately after its own updates have been replayed. Instead, *lock-and-abort*'s latency increase includes the time to replay all the remaining final updates during the ownership transfer phase. In addition, this latency increase also includes the time to update the shard map table across all the nodes using a 2PC transaction.

Table 3 also shows the average latency of TPC-C/YCSB transactions in the four experiments. For TPC-C (only in the scale-out test), the table column shows a range value, which presents the latency numbers of the single-node/distributed *new-order* and *payment* transactions. The average latency increase during the sync phase in *Remus* is within the same order of magnitude as the TPC-C/YCSB transaction latency, and the sync execution phase usually lasts for quite a short time (tens of milliseconds), leading to insignificant performance impact for latency-sensitive applications.

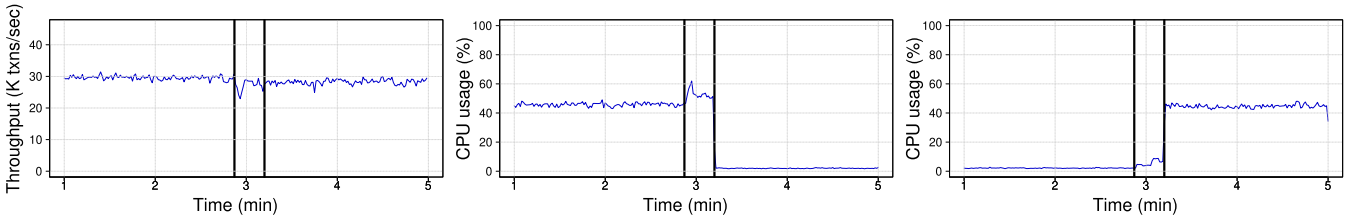
4.8 High Contention Workload

We evaluate the CPU usage and the performance impact of *Remus* using a high-contention YCSB workload on a hot shard to be migrated. The workload starts 200 clients to read and update 100 tuples randomly in this shard, which leads to almost one million *WW*-conflicts during the five-minute test. Figure 10 shows the throughput and the CPU usage of the source node and the destination node. The throughput suffers from a sharp drop (around 26%) caused by the snapshot copying. Such long-running transaction prevents the stale tuple versions before its snapshot timestamp from being timely reclaimed on the source node. When the heavy update workload happens to a small number of tuples, the length of their version chains can increase quickly, slowing down tuple access [28, 33]. Note that the performance impact of long-running transactions is marginal in §4.4, because the YCSB update load is dispersed over 100-million tuples and no lengthy version chains are built up. Correspondingly, during the snapshot copying, the CPU usage of the source node increases by up to 15%. After the snapshot copying completes, the throughput is restored, and the CPU usage increases around 6% on the source node, mainly for propagating update records continuously. On the destination, *Remus* introduces around 8% CPU usage during migration, which is consumed by transaction-level parallel replay. This experiment shows few *WW*-conflicts (8) between shadow and destination transactions during dual execution. The main reason is the dual execution lasts for quite



(a) Lock-and-abort (b) Wait-and-remaster (c) Remus

Figure 9: The TPC-C throughput during scaling out.



(a) The YCSB throughput (b) Source CPU usage (c) Destination CPU usage

Figure 10: The throughput and CPU usage under the high contention YCSB workload with *Remus* migration.

a short time, which starts after T_m commits and ends quickly after the existing source transactions complete.

5 DISCUSSION

Our approach can be applied to many other PostgreSQL based distributed DBMS products [4, 8, 13, 16]. The main features needed by *Remus* are 2PC, timestamp based MVCC, SI and table sharding. Some of these features are implemented in all those products, such as 2PC and SI. The missing features can be supported with manageable efforts. CockroachDB [45] adopts the *push migration* model based on its timestamped MVCC [12]. CockroachDB can adopt our dual execution model to transfer the ownership of migrating data without aborting transactions.

6 FURTHER RELATED WORK

Albatross: Albatross [18] proposes a *suspend-and-resume* migration design for databases with a computation and storage independent architecture. Such approach may incur a lengthy downtime when migrating a large transaction state as discussed in §2.3. Our approach designed for a shared-nothing architecture can be extended for shared-storage databases to eliminate any downtime. For example, we can use copy-on-write (CoW) to build a snapshot over shared storage, and then transfer the ownership of migrating data through our update propagation scheme and dual execution model. Such extension will be our future work.

Zephyr: Zephyr [24] proposes a pull-based live migration approach that supports synchronized dual execution during migration. Newly arrived transactions are routed to the destination node and pull missing data pages from the source node on demand. Once a page is migrated, transactions that access it on the source node would be aborted. Zephyr also incurs significant performance drops like Squall [23] due to interleaved pull blocking.

ProRea: ProRea [42] adopts *pull-migration* and uses bi-directional synchronization during dual execution to ensure SI. Compared with our unidirectional synchronization, the bi-directional synchronization may lead to frequent bouncing of migrating pages between the source and destination, deteriorating database performance.

Industrial Solutions: CockroachDB [45] supports data migration by using snapshot copying and catching up phases [12]. During the ownership transfer phase, CockroachDB hands over the lease to the new node and aborts access to the old leaseholder. Greenplum [13] and Amazon Redshift [4] adopt a *stop-and-copy* strategy to support data redistribution (migration), during which the whole system stops processing write transactions (read-only), resulting in a much longer downtime than the *lock-and-abort* approach adopted by Microsoft Citus [16] and Huawei LibrA [8]. None of these industrial approaches can achieve the goal of zero downtime, no transaction aborts and minimal performance impact which our approach targets.

Migration for non-transactional store: Prior work such as [29] also proposes live migration approaches for key-value stores. However, these approaches do not need to consider fully transactional support required by traditional relational databases, which also simplifies their design.

Update propagation scheme: Prior work [19] proposes a lazy update propagation scheme for database replication and maintains consistency between the primary node and its replica nodes under SI. The asynchronous propagation phase of *Remus* is similar to the lazy replication [19] as well as PostgreSQL's logical replication.

7 CONCLUSION

This paper presents *Remus*, a new live migration technique for general, shared-nothing distributed databases, which migrates data between nodes with no service interruption and minimal performance impact. We implement *Remus* in a PostgreSQL based distributed database *PolarDB-PG*. Extensive experimental results demonstrate the ability of *Remus* in adapting to changing workloads while incurring marginal performance impact, zero downtime and zero transaction aborts under a broad spectrum of workloads.

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive Dynamic Mastering for Replicated Systems. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1381–1392.

[2] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 445–460.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58.

[4] Amazon AWS. 2021. Overview of managing clusters in Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/mgmt/managing-cluster-operations.html.

[5] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*. 1–10.

[6] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. 2014. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *VLDB J.* 23, 6 (2014), 987–1011.

[7] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. 2017. Omid, Reloaded: Scalable and Highly-Available Transaction Processing. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*. 167–180.

[8] Le Cai, Jianjun Chen, Jun Chen, Yu Chen, Kuorong Chiang, Marko A. Dimitrijevic, Yonghua Ding, Yu Dong, Ahmad Ghazal, Jacques Hebert, Kamini Jagtiani, Suzhen Lin, Ye Liu, Demai Ni, Chunfeng Pei, Jason Sun, Li Zhang, Mingyi Zhang, and Cheng Zhu. 2018. FusionInsight LibrA: Huawei's Enterprise Cloud Data Analytics Platform. *Proc. VLDB Endow.* 11, 12 (2018), 1822–1834.

[9] Andrea Cerone and Alexey Gotsman. 2016. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*. 55–64.

[10] Alibaba Cloud. 2021. PolarDB. https://www.alibabacloud.com/product/polardb.

[11] Alibaba Cloud. 2021. PolarDB for PostgreSQL distributed version. https://github.com/ApsaraDB/PolarDB-for-PostgreSQL/tree/distributed.

[12] Cockroachlabs. 2021. CockroachDB Replication Layer (v21.2). https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html.

[13] Jeffrey Cohen, John Eshleman, Brian Hagenbuch, Joy Kent, Christopher Pedrotti, Gavin Sherry, and Florian Waas. 2011. Online Expansion of Largescale Data Warehouses. *Proc. VLDB Endow.* 4, 12 (2011), 1249–1259.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. 143–154.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 251–264.

[16] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2490–2502.

[17] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.* 38, 1 (2013), 5:1–5:45.

[18] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (2011), 494–505.

[19] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 715–726.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 205–220.

[21] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[22] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. 173–184.

[23] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 299–313.

[24] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). 301–312.

[25] Alan D. Fekete, Shirley Goldrei, and Jorge Perez Asenjo. 2009. Quantifying Isolation Anomalies. *Proc. VLDB Endow.* 2, 1 (2009), 467–478.

[26] Christian Garcia-Arellano, Adam J. Storm, David Kalmuk, Hamdi Roumani, Ronald Barber, Yuanyuan Tian, Richard Sidle, Fatma Özcan, Matt Spilchen, Josh Tiefenbach, Daniel C. Zilio, Lan Pham, Kostas Rakopoulos, Alexander Cheung, Darren Pepper, Imran Sayyid, Gidon Gershinsky, Gal Lushi, and Hamid Pirahesh. 2020. Db2 Event Store: A Purpose-Built IoT Database Engine. *Proc. VLDB Endow.* 13, 12 (2020), 3299–3312.

[27] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, and Kai Zeng. 2020. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. VLDB Endow.* 13, 12 (2020), 3272–3284.

[28] Jong-Bin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived Transactions Made Less Harmful. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. 495–510.

[29] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2017. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 390–405.

[30] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8878)*. 17–32.

[31] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226.

[32] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[33] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 1307–1318.

[34] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron J. Elmore, and Shan-Hung Wu. 2019. MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems. *Proc. VLDB Endow.* 12, 5 (2019), 597–610.

[35] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2530–2542.

[36] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[37] Ragnar Normann and Lene T. Østby. 2010. A theoretical study of 'Snapshot Isolation'. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings (ACM International Conference Proceeding Series)*. 44–49.

[38] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. 2020. Fast and Effective Distribution-Key Recommendation for Amazon Redshift. *Proc. VLDB Endow.* 13, 11 (2020), 2411–2423.

[39] Andrew Pavlo, Evan P. C. Jones, and Stanley B. Zdonik. 2011. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proc. VLDB Endow.* 5, 2 (2011), 85–96.

[40] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. 251–264.

[41] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr

El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shivakumar Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proc. VLDB Endow.* 11, 12 (2018), 1835–1848.

[42] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. 2013. ProRea: live database migration for multi-tenant RDBMS with snapshot isolation. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. 53–64.

[43] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (2016), 445–456.

[44] Muthian Sivathanu, Midhul Vuppalapati, Bhargav S. Gulavani, Kaushik Rajan, Jyoti Leeka, Jayashree Mohan, and Piyus Kedia. 2019. INSTalytics: Cluster Filesystem Co-design for Big-data Analytics. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. 235–248.

[45] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. 1493–1509.

[46] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 1–12.

[47] TPC. 2010. TPC-C Benchmark (Revision 5.11). http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[48] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. 2021. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. 357–372.

[49] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 335–347.