K2+: A Multi-Region OLTP Database with Disaggregated Storage for High Availability and Strong Consistency

Haoze Song
The University of Hong Kong
hzsong@cs.hku.hk

Guoli Wei
USTC
weiguoli@mail.ustc.edu.cn

Xusheng Chen
Huawei
chenxusheng6@huawei.com

Yazhi Feng
Huawei
fengyazhi@huawei.com

Xieyun Fang
Huawei
fangxieyun@huawei.com

Yong Men
Huawei
menyong@huawei.com

Hao Feng
Huawei
feng.hao1@huawei.com

Yongkun Li
USTC
ykli@ustc.edu.cn

Heming Cui*
The University of Hong Kong
heming@cs.hku.hk

ABSTRACT

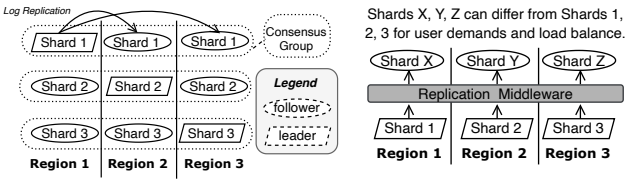
Cloud computing has fundamentally reshaped database system architectures by enabling the decoupling of compute and storage resources and supporting global, multi-region deployments. However, effectively integrating storage-disaggregated architectures with multi-region OLTP processing remains challenging, particularly when maintaining transactional consistency and performing cross-region replication. This paper presents K2+, a multi-region transactional key-value store. K2+ achieves fine-grained data partitioning across regions by combining region-local storage pools with independent, region-specific commit logs. To support replication within and across regions, K2+ incorporates regional partition management with dedicated cross-region replication middleware. This architecture allows different regions to split, merge, and place physical partitions independently while preserving high reliability, high availability, and strong consistency.

PVLDB Reference Format:
Haoze Song, Guoli Wei, Xusheng Chen, Yazhi Feng, Xieyun Fang, Yong Men, Hao Feng, Yongkun Li, and Heming Cui. K2+: A Multi-Region OLTP Database with Disaggregated Storage for High Availability and Strong Consistency. PVLDB, 19(12): 3860 - 3873, 2026.
doi:10.14778/3827998.3828001

1 INTRODUCTION

Over the past decade, cloud computing has enabled two notable trends in database design. First, *storage disaggregation* decouples compute from persistent storage, allowing elastic scaling of compute resources independent of storage pools [6, 12, 33, 39]. Second, *geo-distributed deployment* enables database systems to span multiple regions, improving fault tolerance, reducing client-perceived latency, and supporting disaster recovery [11, 34, 36, 43].

For example, strongly consistent, geo-distributed transactional databases have been extensively studied in the literature, and many have adopted storage-disaggregated designs. Spanner [11] achieves external consistency across regions using Multi-Paxos and relies on Colossus for storage disaggregation. CockroachDB [34] organizes data into per-range Raft groups and leverages cluster virtualization to share resources across tenants. FoundationDB [43] decouples transaction logs (TLogs) from storage nodes while enforcing a global commit order via commit proxies and log replication. Aurora DSQL [1] provides active-active synchronous replication across multiple regions, with a witness region for quorum. Within each region, compute nodes are decoupled from persistent storage.



(a) Replication via Consensus Group. (b) Replication via Middleware.

Figure 1: Comparison: existing solutions vs. ours.

Although these systems provide strong consistency and high availability, they share common limitations. Replication is tightly coupled with partition management, typically applied at the granularity of data shards or ranges via consensus protocols. When a replica resides in another region, partition operations such as splits, merges, or migrations may involve global coordination or synchronous updates to global metadata, limiting elasticity and responsiveness under skewed workloads. These constraints make region-local autonomy challenging, increase operational complexity, and reduce the system's flexibility in adapting to dynamic workloads.

To efficiently implement a multi-region OLTP database on storage-disaggregated infrastructure, we present K2+. K2+ builds on two existing foundations: (1) K2 [32], a strongly consistent multi-region transaction protocol that ensures serializability across geographically distributed nodes by coordinating transactions via TrueTime APIs; and (2) a region-local storage pool (DFV, §2.2), which provides elastic, highly reliable, and highly available storage within each region, enabling compute nodes to efficiently access and share storage without pre-allocating dedicated disks.

While these foundations provide strong consistency and region-local efficiency, they do not natively support storage disaggregation across regions. This paper proposes the system architecture that

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828001

*Corresponding author.

integrates these components into a multi-region, globally storage-disaggregated OLTP system with intra-region partition autonomy.

Specifically, to provide a unified and consistent view of data across multiple regions, K2+ introduces a *new replication middleware layer* that sits between compute nodes and the underlying storage layer. This layer cleanly separates local partition management from global data replication, enabling each region to maintain independent physical partition layouts for globally replicated data. As a result, regions can: (1) place partitions according to local resource availability and workload patterns without affecting other regions; (2) perform cross-region replication without imposing strict physical layout constraints, reducing coordination overhead.

By decoupling replication from local partitioning, K2+ achieves a balance between flexibility, efficiency, and correctness in fully disaggregated, multi-region OLTP deployments (§4).

However, several challenges remain in achieving high reliability, high availability, and strong consistency. For reliability, communication between compute nodes and the storage pool (DFV) occurs over an asynchronous network, leading to issues such as false-positive writes, where data appears to be persisted but no acknowledgment is returned. Handling persistence failures within a distributed transaction protocol is nontrivial. To address this, K2+ co-designs persistence, log replay, and failover mechanisms with the transaction protocol to efficiently handle false-positive persistence failures and node crashes. We systematically analyze these failure points in §2.3 and design intra-region and cross-region persistence strategies to achieve high performance (§5.1).

For high availability, it is crucial that K2+ promptly detects and recovers unhealthy or failed nodes. When a node becomes overloaded, K2+ redistributes data partitions to idle nodes to achieve load balancing. Thanks to the decoupling provided by K2+'s middleware replication layer, each region can independently split, merge, and place partitions without requiring cross-region coordination.

In the event of a node failure, recovery is performed by restoring the failed node's state onto new nodes using local storage. To prevent split-brain scenarios during network partitions and maintain strong consistency, K2+ leverages the primitives supported by its storage layer (DFV). Assuming regional failures are rare, K2+ prioritizes recovery within the same region. This region-focused strategy is efficient and aligns well with the locality of users' workloads (§5.2). During an actual regional failure, K2+ experiences longer recovery times, but reliability is still guaranteed (§5.3).

Overall, K2+ is a multi-region OLTP database that leverages a replication middleware to decouple replication from local partitioning. Building on this architecture, K2+ achieves high reliability and availability. K2+ has been tested and deployed on Huawei Cloud to serve mission-critical internal customer workloads (see the examples in §6.6). The key contributions of K2+ are fourfold:

- K2+ introduces a global/regional partition-management layer that keeps global metadata minimal while allowing each region to split, merge, and place partitions independently.
- K2+ introduces a replication middleware that decouples cross-region replication from request-serving data nodes.
- K2+ co-designs persistence mechanisms with its transaction protocol to handle persistence and node failures.
- K2+ reports production experience and measurements from multi-region deployments and real workloads.

2 GOALS AND OPPORTUNITIES

2.1 Design Goals

K2+'s design is guided by the following objectives:

- ***Storage-Disaggregation.*** K2+ embraces storage disaggregation, decoupling compute and storage for independent scaling, better resource efficiency, and improved failure isolation.
- ***High Reliability.*** Data loss is generally unacceptable. To prevent it, K2+ should replicate data across availability zones and support both synchronous and asynchronous cross-region replication. Applications can choose either mode per table.
- **High Availability.** Availability complements reliability and is critical for SLOs. It is the proportion of time a system, service, or component remains operational and accessible. Brief unavailability may occur for affected partitions or storage during failover. K2+ is designed to recover quickly from failures, restore service promptly, and keep unaffected partitions available.
- ***Strong Consistency.*** Mission-critical applications demand strong consistency to prevent anomalies. K2+ ensures strict serializability [18] for transactions, while asynchronous replicas also offer snapshot reads with bounded staleness.

2.2 DFV: The Storage Pool in Huawei Cloud

Data Functions Virtualization (DFV) is Huawei Cloud's unified data storage platform. It underpins every data-centric service running on the platform, from object, block, and file storage to databases and big data analytics. As the cornerstone of Huawei Cloud's core data services, DFV is engineered primarily for high reliability and availability, providing an elegant abstraction for storage disaggregation that aligns with our design goals. Our approach, therefore, fully leverages DFV's proven design pillars.

As shown in Figure 2, the storage nodes of DFV span three availability zones (AZs). DFV offers a persistent log (Plog) abstraction, which is a contiguous, append-only storage segment. The primary interface to the persistence layer is through Plog clients. Within a single AZ, DFV's resilience is achieved through multiple replicas and background erasure coding. Cross-AZ redundancy is ensured by performing synchronous writes to replicas in all three AZs.

This synchronous cross-AZ write path reflects a deliberate durability–availability trade-off. DFV acknowledges writes only after persistence in all three AZs, providing stronger regional durability than quorum-based schemes. A storage-node failure may briefly block writes to the affected group during repair, while live replicas continue serving reads. As DFV is designed to recover promptly and such failures are rare in practice, it achieves a desirable design point, providing 99.999999999% (11 9s) durability and 99.999% (5 9s) availability in production.

Supporting K2+ in DFV presents two challenges. First, the interaction between K2+ and DFV through Plog clients may lead to false-positive failures due to asynchronous network communication. Second, DFV is limited to single-region deployments. Enabling cross-region replication requires middleware for efficient data replication. Solutions to these challenges are detailed in §3 and §5.

Portability beyond DFV. While our deployment uses DFV as the region-local storage pool, K2+ is portable and can operate over other regional storage services that provide intra-region durability and availability guarantees. Specifically, K2+ only requires that

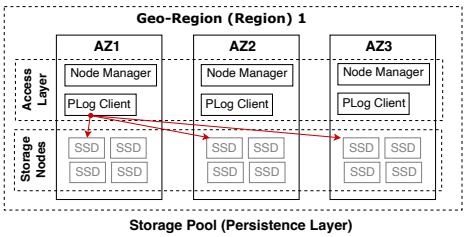


Figure 2: A storage pool spanning three availability zones.

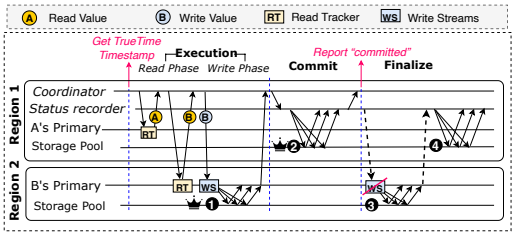


Figure 3: Message flow for coordinating a transaction.

each region offers a reliable storage pool with two capabilities: (1) an append-only durable log abstraction, and (2) a sealing primitive that prevents further writes to a log segment. These requirements are not specific to DFV and can be realized by a variety of regional storage systems. In our design, the sealing primitive is used to establish a single valid recovery boundary during data node failover, preventing split-brain scenarios (to be illustrated in §5.2). Some storage systems provide similar fencing primitives (e.g., Azure Append Blob [24] and Delos [4]). For log systems that do not natively support sealing, additional fencing mechanisms should be implemented to ensure safe failover in the presence of multiple writers.

To demonstrate that K2+ does not depend on DFV-specific features, our ablation study shows that it remains fully functional when using a simpler local-file backend (§6.3), while DFV primarily contributes improved latency and operational robustness.

2.3 TrueTime-based Transaction Processing

K2+ employs an adapted multi-version timestamp-ordering (MVTO) protocol and leverages TrueTime to coordinate transactions, ensuring high performance and strong consistency. The protocol is detailed in [32]. Below we outline the necessary protocol background. Figure 3 illustrates the message flow for coordinating a transaction. Like other MVTO protocols, K2+ associates each transaction with a unique timestamp before execution. With well-synchronized clocks, K2+ ensures that timestamps increase monotonically on a global scale. To guarantee serializability between concurrent transactions, K2+ implements read trackers (RT) and write streams (WS) to monitor transaction conflicts. RT records the largest timestamp of any transaction that has ever read the key. WS records all ongoing writes with the timestamp of the transaction that created them.

Consequently, each read/write is validated by comparing the transaction's timestamp against RT and WS. Writes that would violate the read order are aborted, and reads that would bypass uncommitted writes are blocked until those writes commit or abort.

After completing the conflict check, every successful read updates its corresponding RT, and every successful write creates a provisional entry in the WS. WS entries are persisted immediately to prevent write loss in the event of a node failure (❶ in Figure 3). If all operations succeed, the transaction proceeds to the commit phase. In the commit phase, K2+ records and persists the transaction's commit status (❷). To optimize the common-case performance, K2+ finalizes writes asynchronously (❸). It converts provisional WS entries into permanent versions after reporting commit results to clients. This remains safe because the final status of a write can always be reconstructed by combining the provisional WS with the transaction's final outcome. Once a transaction's writes are fully finalized, its in-memory record is discarded. Then, K2+ durably updates the transaction status from committed to finalized (❹).

In summary, this workflow has four persistence points with different correctness requirements. Persisting provisional write streams and the transaction commit status (❶–❷) is required for correctness: if a transaction commits, its provisional writes and final decision must be recoverable after failures. Finalizing writes and recording the finalized status (❸–❹) are required for subsequent read performance, but they are recoverable from the committed transaction status and provisional writes. This distinction is important for K2+'s persistence policy in §5.1: the system must retry or recover uncertain persistence at correctness-critical points, while bounded retries are sufficient for recoverable finalization points.

3 SYSTEM OVERVIEW

The architecture of K2+ is shown in Figure 4. The data access paths are shown in red, while the control paths (including data placement) are shown in blue. To achieve our goals for efficient multi-region data management with storage disaggregation (§2.1), K2+ introduces two new designs. First, it introduces a new partition management layer (including global and regional managers) that decouples global data management from regional data management by keeping global metadata minimal. Second, it introduces a decoupled replication middleware (§4) that performs cross-region replication on top of intra-region disaggregated storage (DFV).

Based on these two new designs, we further develop efficient persistence policies and fast recovery mechanisms to ensure high reliability and availability, leveraging the new routing and replication layers (§5). Before diving into details, this section presents an overview of K2+'s key components.

Driver. Starting from the top, user applications interact with K2+ via a driver. K2+ implements its query language based on the CQL protocol [20] with transaction support.

Frontend. Drivers connect to the frontend through sessions. In K2+, the frontend is responsible for query parsing, transaction coordination, and QoS flow control. To route users' read and write requests to the appropriate data nodes, the frontend communicates with regional managers to locate data.

TrueTime (TTC) Oracle. K2+ uses TrueTime primitives for strictly serializable transaction processing (§2.3). Frontends obtain valid timestamps for transactions from the TrueTime Oracle via the Timestamp API before execution. In each region, TTC runs on multiple servers across availability zones with availability > 99.9999%. If no TTC replica in a region can provide a timestamp within a bounded timeout, K2+ temporarily stops admitting new read-write transactions in that region for correctness.

Data Nodes. K2+ processes read and write requests through data nodes. Each data node holds a portion of the overall data and stores data using an LSM tree structure. The primary data model for storing regional data is based on key-value pairs. Similar to other

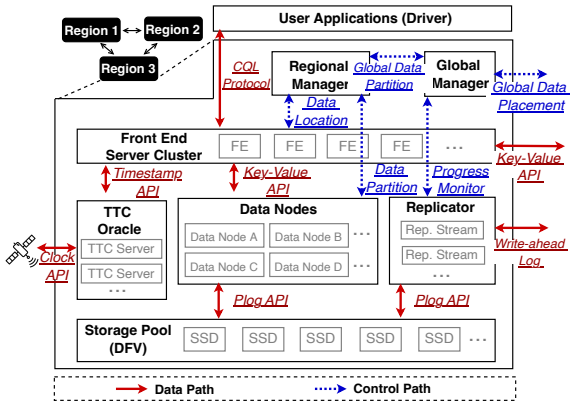


Figure 4: K2+ architecture overview.

NewSQL databases [3, 19], K2+ uses the table's primary key as the key and the corresponding row data as the value. In addition to customer data, K2+'s data nodes also manage the transaction recorder, read version tracker (RT), and write streams (WS), which are used for concurrency control (see §2.3). For fast lookups and updates, this metadata is indexed using B+ Trees.

Storage Pool. Data nodes persist data to the storage pool via the Plog client (see Figure 2). The storage pool (i.e., DFV, illustrated in §2.2) provides cross-availability zone redundancy within a single region, ensuring reliability and availability. For each data partition, two types of data are persisted. The first is the write-ahead log (WAL), which records the commit history of transactions. The second is the compacted data from each level of the LSM tree, which enables quick recovery of data nodes in the event of a node failure.

Replicator. K2+ introduces a new middleware layer above the storage pool to perform either synchronous or asynchronous cross-region replication. The key components of this middleware are replicators. To replicate data, the replicator reads the WAL of partitions from DFV and transmits them to receivers in other regions.

Global Manager (GM). The global manager is a synchronous cross-region subsystem responsible for schema management and global data placement. The workload and metadata volume managed by the global manager are minimal: K2+ only records regional keys partitioned by table definitions obtained at initialization and leaves fine-grained key management to the regional manager (RM) in each region. Thus, GM is off the transaction-critical path and is unlikely to become a centralized bottleneck. In steady state, all read/write requests are routed using the cached global range view at the RM, and GM is contacted only for (i) cache misses (e.g., after a cold start) and (ii) infrequent control-plane events such as schema changes and global range-view updates (e.g., split/merge coordination metadata and region failover orchestration). Consequently, GM's request rate is typically orders of magnitude lower than the data-plane QPS.

Cache correctness is enforced via a ZooKeeper-stored commit epoch: GM publishes a new global range view by persisting View $[e+1]$ and then atomically advancing global_epoch to $e+1$. RMs tag cached views with e_{local} and periodically validate it by checking global_epoch with a short promised TTL (200 ms in our configuration); GM will not update the global view within the lease period. On an epoch mismatch, they refresh the latest view before routing. If the epoch cannot be validated beyond the TTL (e.g., due to network errors), requests are conservatively rejected.

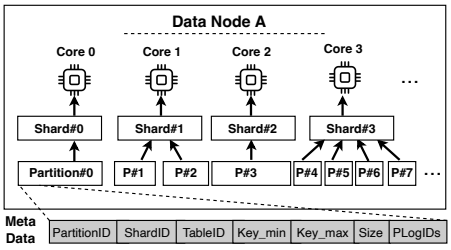


Figure 5: Shard and partition layout in a data node.

Regional Manager. The regional manager further divides the assigned global data into smaller portions and maps them to data nodes within the same region. It maintains heartbeat connections with all nodes (including frontends, data nodes, and the TTC Oracle) to detect failures within the region. Regional managers are replicated across AZs using quorum-based replication, and recovery is performed by view change among the active copies.

4 MULTI-REGION DATA MANAGEMENT

In this section, we describe our design for a unified, consistent data view across compute nodes, leveraging fine-grained partitioning (§4.1) and routing (§4.2) over globally replicated data (§4.3).

4.1 Data Shards and Partitions

Data Shards. K2+ is built on Seastar [31], a high-performance, event-driven C++ framework with coroutine-based async programming. Seastar partitions resources by CPU core (each core has dedicated memory) to avoid cross-core contention; K2+ follows this model by binding each core to a data shard.

Partition Strategy. Binding data to cores complicates resharding under dynamic workloads: moving shards can be costly because cores have different memory spaces, yet resharding is necessary for availability under overload. K2+ addresses this by adding a routing layer between shards and physical partitions (Figure 5). Physical partitions (sets of rows from a table) are the minimal management unit, and each shard can contain multiple physical partitions.

From a storage perspective, each physical partition corresponds to an independent LSM tree. Therefore, it is important to maintain a manageable number of physical partitions to avoid storage fragmentation and excessive complexity in partition routing. To address this, K2+ defines three types of physical partitions: large, medium, and small, designed to consume 100%, 50%, and 25% of CPU resources, respectively. As a result, each shard can have a maximum of four physical partitions, avoiding storage fragmentation.

4.2 Partition Routing

Next, we show how K2+ routes reads and writes to partitions. This partition routing complements the replication middleware (§4.3), enabling transparent global data management.

We use the table defined in Figure 6 as an example. The table has three columns; $(C1, C2)$ is defined as the primary key, and $C1$ serves as the Global Key. The Global Key is designed as a subset of the primary key and is not required to be unique. It serves as a coarse-grained locality hint for global data placement, allowing application-level locality semantics to guide the placement of global ranges. For instance, a subsidiary project management department

```
CREATE TABLE demo (
        C1 STRING NOTNULL,
        C2 INT NOTNULL,
        C3 INT DEFAULT NULL,
        PRIMARY KEY (C1, C2),
        GLOBAL KEY (C1),
        GLOBAL RANGE MAP (
            "" ::p[R1]::r[R2, R3],
            "100"::p[R2]::r[R1, R3],
            "200"::p[R3]::r[R1, R2] ),
    );
```

Figure 6: A statement creates a demo table.

operating in New York can use the location attribute (e.g., 302) as the Global Key, so that the department's primary data copy can be placed in the $R3$ region to improve locality. In this sense, the Global Key bridges logical semantics and coarse-grained placement.

Alternatively, if records in a table do not exhibit strong locality semantics, users may omit the Global Key and the global range mapping in the DDL. In that case, K2+ treats the table as residing in the region where it is created, and the system automatically manages all lower-level intra-region partitioning internally.

In this example, rows with $C1$ in range $(-inf, 100)$ are stored in Region R1, with replicas in R2 and R3. Rows with $C1$ in $[100, 200)$ are stored in Region R2, with replicas in R1 and R3. Rows with $(200, inf)$ are stored in Region R3, with replicas in R1 and R2.

Global Routing. Figure 7 shows the procedure for routing read/write requests. After the SQL request is parsed by the frontend, the read or write request is directed to the region that holds the relevant data. For write requests, the region containing the primary copy is selected. For read requests, the nearest region that contains either the primary or a replica is chosen. The global routing is determined by the Global Key and maintained as a routing table in the global manager. The routing table is cached at frontends and regional managers. As schema changes are infrequent, most read/write requests can be routed using cache without involving the global manager.

Regional Routing. After a read or write request is directed to the appropriate region, the request is subsequently forwarded by the regional range view. The regional routing is based on the remaining part of the primary key. In this example, $C2$ will be used for regional routing. Finally, the request is forwarded to the corresponding CPU core, along with its Root Plog ID that indicates the entry point of the persistent storage layer. Each Root Plog is associated with index Plogs, which record the offsets of the WAL Plogs and LSM levels.

4.3 Replication Middleware

K2+ achieves cross-region replication by employing a middleware layer built on top of the intra-region shared storage pool. Figure 8 shows an overview of the middleware.

To achieve global decoupling between the storage layer and data nodes, K2+'s key idea is to ensure that physical partition information is visible only within its respective region. In other words, partition metadata managed by regional managers is not replicated across regions. This design not only significantly reduces the amount of metadata ZooKeeper needs to manage (only the global manager's data view is required) but also decouples partitioning policies across different regions. For each table range, partitions are segmented and managed independently within each region. This independence allows data to be grouped into partitions that reflect regional biases in workload access patterns. Load-balancing operations, such as splitting or merging partitions, can be carried

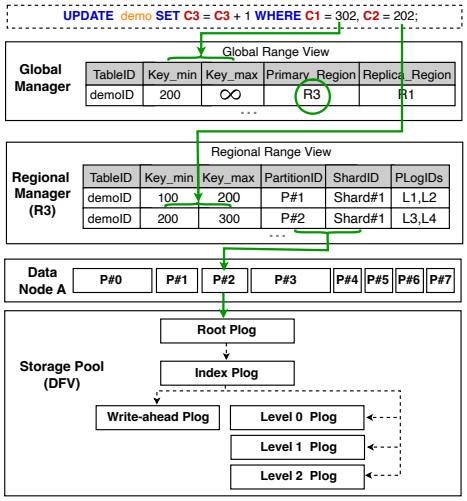


Figure 7: An example of read/write request routing.

out locally. As a result, the correspondence of partitions between primary and secondary regions forms an N-to-M relationship rather than a strict one-to-one correspondence. In our example (see Figure 8), Region R1 contains two data nodes, each managing two data partitions (four in total), while Region R2 has a single data node with only one data partition.

Replicator Manager. K2+ introduces an independent Replicator Manager to manage and record the replication progress of each partition for efficiency and availability. For each partition in the primary region, the Replicator Manager maintains a list of Plog backups stored in the replica region (i.e., the three Rep. Streams shown in Figure 8). As partitions in the primary region may split or merge during replication, the Replicator Manager guarantees that all predecessor partitions have been replicated before starting replication for any new partition lineage. To achieve this, when new partitions are split or merged at the primary nodes, K2+ notifies both the regional manager and replicator manager beforehand.

Replication Mode. K2+ supports both synchronous and asynchronous cross-region replication (§2.1), with the mode configurable independently for each global key range or table. In asynchronous mode, primary data nodes return responses immediately after persisting data to DFV. In synchronous mode, primary data nodes hold requests (i.e., corresponding to the four persistence points described in §2.3) until they receive acknowledgment that cross-region replication is complete (the blue path in Figure 8). A key feature of K2+ is that, instead of waiting for data node Z to send acknowledgments ($ACKs$) after responding to the client, replication is acknowledged immediately after the WAL is persisted in the replica's region. This design reduces the cross-region replication latency. To further minimize the number of $ACKs$, K2+ performs packet aggregation by combining acknowledgments for different partitions within the same primary data node. Each ACK message includes the Plog ID and offset up to which the WAL has been replicated. Thus, requests from the same partition are notified only with the maximum offset.

Deduplication. After receiving the WAL from the original region, replicator writes it into local DFV. To prevent duplicating log entries, the receiver tracks the highest successfully applied offset for each WAL stream. If replication resumes from an earlier offset due to network retries or failover, any records below this offset are skipped.

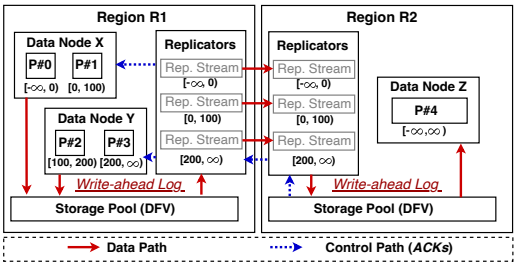


Figure 8: Replication middleware.

Data Replay. After replication, data nodes in the replicated region asynchronously replay the WAL to update the partition's memtable and LSM tree. Since a data node may fail during replay, K2+ should ensure that user-visible data is presented incrementally during failover, while preserving the same order as in the primary.

K2+ ensures correctness through two mechanisms. First, the visibility of replayed data is controlled by the epoch ID generated by K2+'s transaction protocol, which guarantees monotonicity across the system. An epoch becomes visible to user requests only after all operations within that epoch have been replayed. At the implementation level, each WAL record is tagged with an epoch ID when created. During replay, records from previously replayed epochs are filtered out, preserving monotonic visibility across epochs. Second, within the same epoch, replay is idempotent. Once a transaction or write reaches a committed or finalized state, it cannot revert to an aborted or in-progress state (as illustrated in Figure 9 and Figure 10). This ensures that reapplying the same WAL record only repeats an already determined transition without changing the transaction outcome. As a result, K2+ is able to resume replay at the data node from any point within any epoch during failover.

Replicator Failover. On replicator failure, a new replicator will take over the responsibility of continuing replication. The new replicator will resume progress based on the Plog ID and finished offset recorded by the replicator manager (for either replicators in the primaries' region or the replicas' region). In the event of a data node failover, a standard data-node failover procedure is followed (see §5.2). The corresponding replicator will be paused until a new primary data node recovers in the same region. The failure of replica data nodes does not affect the progress of replication.

4.4 Securing Strong Consistency

With synchronous replication, K2+ guarantees strictly serializable data access across all data nodes. This is achieved via strongly consistent request routing and the ACK-based replication middleware. Under asynchronous replication, K2+ offers snapshot reads with bounded staleness. To validate these consistency guarantees in our replication middleware, we formally analyze K2+'s replication protocol in [32] and further use TLA+ model checking to verify key safety invariants (e.g., no dual decision / no lost committed writes) in [7]. The results corroborate the design of K2+.

5 HIGH RELIABILITY AND AVAILABILITY

5.1 Taming Asynchronous Persistence

K2+ co-designs persistence with its transaction protocol (§2.3) to handle asynchronous network interactions with the storage layer (§2.2). In this environment, concurrent persistence requests from

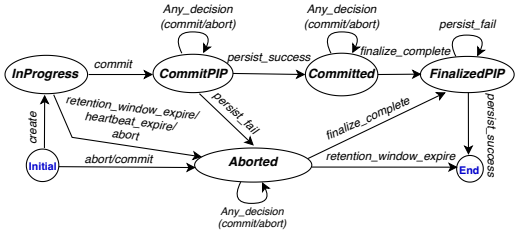


Figure 9: State machine of transaction status.

different coroutines may be delayed or fail due to conflicts or network packet loss. For instance, consider a transaction (txn_1) that is about to commit. To do so, a coroutine (co_1) on the recorder's data node will issue a commit request and wait for persistence to succeed. However, this request may be blocked and delayed in the network. Meanwhile, another coroutine (co_2) may attempt to abort txn_1, for example, because the heartbeat between the frontend and the data node has expired. As a result, the abort request may be successfully written into the write-ahead log, and K2+ will initiate a rollback to clean up the temporary writes on other nodes. Later, the delayed request from co_1 may resume and persist the status of txn_1 as committed in the append-only log. This causes an anomaly: although txn_1 has been aborted, the latest status is "committed."

To address this, K2+ introduces a new state for transactions called Persist-in-Progress (PIP), which serializes competing persistence attempts without relying on OS-level locks. PIP means the system has issued a persistence request whose outcome is not yet known; before resolving the uncertainty, conflicting state transitions must be constrained. The PIP state is applied to all four persistence points in a transaction's lifetime (§2.3), resulting in two state machines: one for transaction records and one for provisional writes (see Figure 9 and Figure 10).

The correctness guarantee is that only a single commit or abort decision takes effect for each transaction. Delayed or retried persistence requests cannot overwrite a finalized decision.

5.1.1 Transaction Status State Machine. The key correctness invariant is that if a transaction ultimately commits, the commit decision must have been successfully persisted.

- **InProgress:** The transaction is active. Transitions to CommitPIP upon a commit request, or to Aborted otherwise.
- **CommitPIP:** Commit decision has been issued and persistence is in progress. To ensure correctness (i.e., for the failed example), K2+ must retry the persistence request until a valid response (either success or failure) is received, rather than an unknown outcome. Transition to Committed occurs upon persist_success; transition to Aborted occurs upon persist_fail.
- **Committed:** Transaction has been durably committed. Transition to FinalizedPIP only when asynchronous finalization of transaction writes completes.
- **FinalizedPIP:** Finalization of transaction writes is complete but the persistence change of transaction status is in progress. After successful completion, the transaction reaches End.
- **Aborted:** Terminal state indicating the transaction was aborted.

5.1.2 Write Operation State Machine. The key correctness invariant is that if a transaction ultimately commits, the temporary write must be persisted to prevent data loss in the event of a node failure.

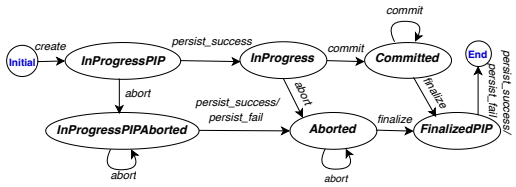


Figure 10: State machine of write operation in a transaction.

- **InProgressPIP:** Write has been created but has not been persisted. If the transaction aborts, it transitions to InProgressPIPAborted. On successful persistence, it transitions to InProgress.
- **InProgress:** Write has been created, but whether it will be committed is unknown. State transitions occur based on the decision.
- **InProgressPIPAborted:** Write was aborted during persistence. Automatically transitions to Aborted.
- **Aborted:** Terminal state for aborted writes.
- **Committed:** Write is committed.
- **FinalizedPIP:** Write can be finalized. Persistence is in progress. K2+ only performs a limited number of retries for this request. If the outcome of persistence remains unclear after retries, K2+ can safely proceed, leaving the work to be resolved at the stage of resolving subsequent write-read conflicts.

5.2 Node Failover

Next, we illustrate the node's intra-regional failover mechanism, designed to ensure high reliability and availability.

Data Node Failover. K2+ always attempts to recover a data node within the same region as the failed node. This failover strategy is based on two key observations. First, regional failures are rare. Intra-region storage pool (DFV) offers both high reliability and high availability, enabling quick data recovery within the region. Second, the placement of primary nodes and replicas is determined by customer workloads to optimize data locality. Recovering data nodes within the same region helps maintain data locality.

Implementation-wise, when recovering a data node, K2+ migrates all data partitions from the failed node to a new data node with available capacity. To facilitate quick recovery and load balancing, K2+ always maintains several active idle data nodes (a.k.a. hot backups). An example of migrating a partition is shown in Figure 11. Before migration begins, the regional managers update the partition placement information to reflect these changes (❶). After that, a new data partition is constructed on an underutilized data node. K2+ does not recover in-progress transactions; instead, it treats them as aborted. This design choice is due to the fact that read version trackers (which function similarly to read locks) are not persisted. Attempting to recover in-progress transactions could degrade the consistency guarantees of K2+ (§2.1).

Preventing Split-Brain Scenarios. A major concern during failover is the potential for split-brain, where failed nodes enter a zombie state that could compromise correctness. This may occur due to network partitions, delayed persistence requests, or overloaded nodes not being terminated promptly. To prevent split-brain for writes, K2+ leverages I/O fencing API provided by DFV, a feature also available in other systems (e.g., [24, 35]).

In particular, before a new data node takes over a partition, it issues Seal operations to underlying Plogs (Root Plogs, Index Plogs, and WAL Plog in sequence) to prevent further writes from the old node. This is essential because the old and new nodes share the same LSM-tree state via the storage pool for fast recovery (as shown in Figure 11). Sealing the old WAL ensures no further append becomes valid, and the old writer cannot modify the LSM trees. As a result, the seal defines a stable recovery boundary.

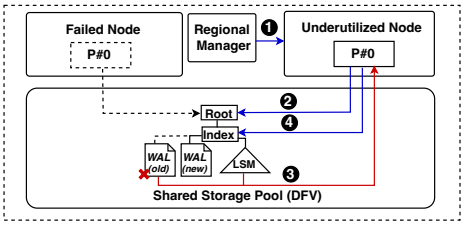


Figure 11: This diagram illustrates how data partition #0 can be migrated from a failed node to an underutilized node.

Once WAL Plogs of the old node are sealed, the new node switches the Root Plog and creates a new Index Plog, which generates new write-ahead entries and indexes the existing LSM tree (❷ in Figure 11). The old WAL and latest SSTables are then restored (❸), and the new node flushes reconstructed SSTables and begins serving requests with a new WAL (❹).

This workflow ensures that each Plog has a single writer, preventing split-brain in a lightweight manner. It relies on the Seal operation, under the assumption that the intra-region storage layer is fully managed and accessible only through its exposed APIs. For portability over log systems lacking native Seal support, a general-purpose fencing layer can be implemented above an opaque storage API; however, that approach is potentially more complex and less efficient, as it requires careful tracking of ownership and in-flight requests to maintain a consistent recovery boundary.

Unlike writes, reads can be served directly from memory without involving the persistence layer. To prevent split-brain scenarios for reads, K2+ employs a lease check mechanism on read requests to prevent reading stale data from failed data nodes. Since this lease check is performed after the read operation, it can effectively prevent stale data from being returned to the user. Note that a similar check cannot be applied to writes, as data nodes are only able to verify the lease before issuing a persistence request (instead of after the request). Thus, even if the request is issued before the lease expires, it could still be delayed by the network.

Frontend Failover. Like data nodes, K2+ does not recover in-progress transactions at failed frontends but treats them as aborted. New transactions are forwarded to other frontends.

Manager Failover. Regional and replicator managers run multi-AZ; on AZ failure, a replica in another AZ becomes primary. Global manager's (GM's) availability affects control-plane agility rather than data-plane availability. GM is replicated using ZooKeeper for leader election and strongly consistent metadata storage. During transient GM unavailability, the data plane continues serving requests using epoch-versioned routing metadata cached at regional manager; only control-plane operations (e.g., schema changes, new placement decisions, and failover orchestration that requires committing a new global view) are paused until GM recovers.

5.3 Regional Failover

K2+ classifies regional failures into three types:

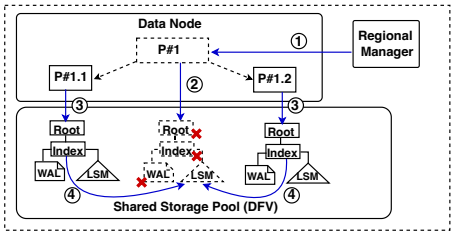


Figure 12: This diagram illustrates how data partition #1 can be divided into two separate partitions, #1.1 and #1.2.

- If all DFV instances in two availability zones (AZs) within a region fail, the associated DFV pool in that region becomes read-only. A regional failover is then planned.
- If the manager clusters (including regional manager and replicator manager) in two AZs within a region fail, this results in a manager failure, causing all data nodes or replicators in that region to potentially lose their heartbeats. A regional failover is required to restore management capability.
- If a region suffers a large-scale failure (e.g., ≥80% of data nodes crash due to power outage) and the system is unable to recover automatically, a forced regional failover should be performed to make data service continuously available.

When selecting the region to accommodate recovered partitions, the nearest region that contains the global range cover for that partition will be chosen. This approach aligns with K2+'s strategy to maintain the locality of the user's workload.

Recovering a partition in the new region is straightforward if synchronous replication mode (§4.3) was enabled before the failover occurred. K2+ reconstructs the data partition in the new region using the same procedure as for recovering from an intra-region data node failure (i.e., following the partition migration semantics; see Figure 11). The key difference is that K2+ updates not only the regional data partition location in the regional manager, but also the global range view in the global manager prior to that.

When asynchronous replication mode is used for less-critical data, some new writes may persist in the primary region without being replicated across regions. In such cases, K2+ may unavoidably lose a portion of data but guarantees an RPO $\leq$ 10*s*. The core challenge in recovery is how to restore the data to a snapshot that preserves the atomicity of transactional writes and the global order of transactions. K2+ leverages its visibility control algorithm [32]. In particular, the algorithm segments the transaction history into epochs. During recovery, K2+ can restore all data partitions affected by the failed partitions to a specific epoch. The recovery epoch is determined through a negotiation process. The latest epoch that is fully replicated on all affected partitions is chosen.

5.4 Mastering Timeouts

Beyond failure handling, K2+ also employs timeout mechanisms to reclaim both logical and physical resources, further ensuring high availability. The timeout mechanism in K2+ serves two purposes. First, K2+ relies on a timeout mechanism to abort unresponsive transactions. Since K2+ supports interactive transactions, if the client driver does not send requests to the frontend for an extended period, the frontend will abort the transaction and refuse to process any subsequent requests related to it. To facilitate this, K2+ allows users to configure a maximum lifetime for each interactive transaction. When this duration is exceeded, the frontend sends an abort request to the transaction recorder and closes the session between the frontend and the client.

Second, K2+ relies on a timeout mechanism to clean up aborted transactions. As discussed in §5.2, when a data node or frontend fails over, any in-progress transactions are treated as aborted and cannot be recovered since no metadata is persisted. Therefore, a timeout mechanism is necessary to ensure that unfinished transactions are properly cleaned up. In the event of a frontend failover, the transaction recorders will trigger a force_abort after the timeout period (shown as retention_window_expire in Figure 9).

In the event of a data node failover, any in-progress transaction recorders located on that node are lost. To clean temporary writes, other participant nodes will actively trigger a finalize procedure after a timeout for temporary writes. Specifically, the participant nodes will query the newly recovered data node for the final decision. If the response is Abort or Not_Found, the temporary writes can be safely discarded. Essentially, there are two subcases for a Not_Found response: (1) the in-progress transaction records have been lost, or (2) the transaction write has already been finalized and the recorder has been removed. Both subcases indicate that the temporary writes can be safely removed. Otherwise, if the response is Committed, the writes should be finalized accordingly.

5.5 Reaching High Availability

Availability is measured as the fraction of time the service is operational. Same as [17], downtime is any five-minute window with a server error rate > 5% given at least 60 requests/min. To achieve high availability, K2+ combines fast failover (§5.2, §5.3) with proactive health detection and load balancing [23].

K2+ monitors components from two perspectives. For each VM, it tracks compute usage (CPU, memory, connections, and abnormal process states) via health agents and heartbeat reports to the regional manager. For inter-server requests, it tracks latency and failure rates; if more than 5% of requests time out or error for two statistical periods (e.g., two minutes), the regional manager raises an alert and isolates affected components before recovering them.

When a data node nears overload (e.g., ≥90%), K2+ splits partitions and migrates them to underutilized nodes (Figure 12). The regional manager selects a split point and updates routing (①); the old partition pauses, flushes cleaned segments to DFV, and blocks new WAL writes (②); new storage objects are created (③); and existing SSTables are re-indexed by the new partitions' index Plogs (④). Migration then follows Figure 11, but the destination node skips WAL population/flush because data was already flushed in ②; only indexes/metadata are copied. In our evaluation, splitting and migration complete in a few seconds (§6.5).

Besides, if a manager node is deemed unhealthy, K2+ reassigns leadership roles to nodes in a different AZ. If a frontend becomes unhealthy, K2+ adds it to the load balancer's blacklist, preventing any new requests from being routed to it.

6 EVALUATION

All experiments are conducted on our cloud-based cluster, consisting of 24 machines, with the exception of the scalability experiments.

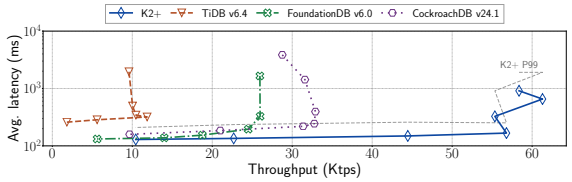


Figure 13: TPC-C performance on multi-region deployment (Cross-region Replication Mode: Synchronous).

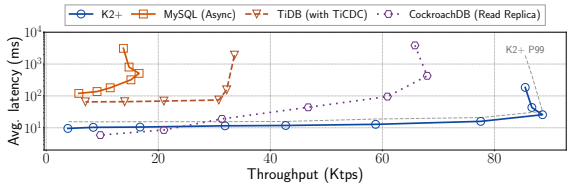


Figure 14: TPC-C performance on multi-region deployment (Cross-region Replication Mode: Asynchronous).

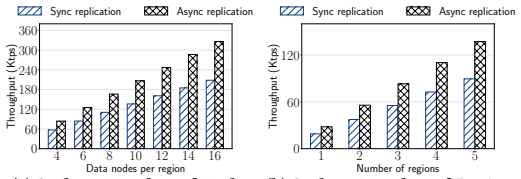
Each machine has 8 CPU cores (16 vCPUs) and 64 GB of memory. All servers run Ubuntu 20.04 with the Linux 5.4 kernel. By default, experiments use a three-region deployment (except for intra-region failover). That is, we allocate 8 machines per region. The measured average inter-region latency is 44.2*ms*.

Baselines. We emphasize that comparing industrial OLTP systems is inherently difficult and cannot be perfectly normalized. Different systems are implemented in different architectures, programming languages, runtime ecosystems (e.g., C/C++, Java, Go), and adopt distinct design patterns. These choices lead to non-trivial differences in performance. As a result, an exact "apples-to-apples" comparison is often unattainable in practice.

To mitigate this, we follow two principles throughout the evaluation. First, we ensure *resource parity*. In each region for K2+, we allocate one machine for TrueTime assignment and the regional/global managers, one machine for replicators, two machines for frontends, and four machines for data nodes. Clients are collocated with frontends to generate benchmarks. For baseline systems, we follow their recommended component configurations while keeping the number of CPU cores dedicated to transaction processing equal to that of K2+. In our default setup, this corresponds to 64 transaction-processing cores per region (4 data nodes × 16 vCPUs each). Across all experiments, all systems consistently achieve over 80% utilization of their allocated transaction-processing cores. Second, in addition to cross-system comparisons, we present *within-system breakdowns* for K2+ that isolate the impact of storage disaggregation and the replication middleware.

Workloads. We run two standard benchmarks and two real-world applications. We use TPC-C [37] to compare K2+ with state-of-the-art open-source OLTP databases. TPC-C is deployed in a partitioned configuration: the dataset is initialized with 1,000 warehouses and 10 districts per warehouse, and the data are evenly distributed across CPU-core shards. We use YCSB [13] to vary workload properties and to reach peak throughput in our failover study. Characteristics of the real-world applications are described in §6.6.

Overall, our evaluation aims to quantify (i) end-to-end performance of K2+ under multi-region deployments (§6.1, §6.2, and §6.6), (ii) the overhead introduced by storage disaggregation and the replication middleware (§6.3), and (iii) K2+'s recovery behavior under representative failure/sub-healthy scenarios (§6.4 & §6.5).



(a) Scale to Number of Nodes (b) Scale to Number of Regions

Figure 15: TPC-C Scalability of K2+.

6.1 End-to-End Performance

To evaluate the end-to-end performance of K2+, we conduct an experiment that progressively drives the system to saturation. We evaluate K2+ under both asynchronous and synchronous cross-region replication modes. The results are shown in Figure 13 and Figure 14. We use an close-loop setup and sweep the total number of client threads from 6 to 3072. For each configuration point, we repeat the experiment for 10 rounds and report the mean throughput (transactions per second) and average latency for all evaluated systems. For K2+, we additionally report P99 tail latency. Overall, K2+ reaches up to 87Ktps in asynchronous mode and 62Ktps with synchronous cross-region replication, while the baselines achieve lower peak throughput under the same experimental settings.

For synchronous deployments, we compare K2+ with baselines that use the synchronous replication mechanisms supported in their codebases. TiDB [19] and CockroachDB [34] implement the Raft protocol, while FoundationDB [43] uses master-slave replication.

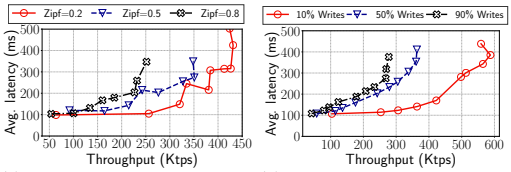
Among the baselines, TiDB's performance saturates earlier when 768 clients are deployed across three regions. TiDB's transaction protocol is not specifically designed for geo-distributed deployments: it relies on a centralized Placement Driver (PD) for data location and timestamp assignment, creating regional bottlenecks. CockroachDB and FoundationDB are representative multi-region databases. They outperform TiDB.

CockroachDB uses hybrid logical clocks (HLC) yet still relies on per-range Raft groups for cross-region replication. As a result, coordination overhead scales with the number of ranges involved in each transaction. In contrast, K2+ separates replication from range leadership and can replicate data from different ranges within a single replication view, allowing the transaction coordinator to track replication progress with lower overhead.

FoundationDB decouples the transaction log (TLog) from the transaction layer by introducing log servers. In FoundationDB, TLogs are synchronously replicated across regions to provide durable, ordered commit records, while the underlying storage layer pulls and applies these logs asynchronously. However, FoundationDB still relies on global commit ordering, achieved by commit proxies in coordination with resolvers and TLogs. These commit proxies can become a bottleneck under stress tests. In contrast, because K2+ uses TrueTime for ordering, it goes further by fully decoupling the transaction layer from the replication layer.

For asynchronous deployments, we mainly compare K2+ with MySQL asynchronous primary–replica replication [27], TiDB with TiCDC [28], and CockroachDB with read-only replicas [10].

Compared to synchronous replication, asynchronous replication removes wide-area acknowledgements from the write path. However, the systems differ in how much cross-region communication remains on the critical path. MySQL's primary-replica replication (similar to the systems in Figure 1a) centralizes all writes at a single



(a) Impact of Zipf ratio on K2+ (b) Impact of R/W ratio on K2+

Figure 16: Sensitivity analysis with YCSB.

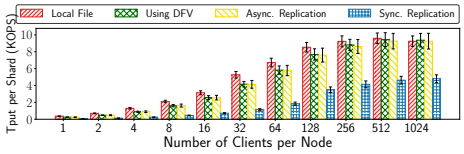


Figure 17: Performance impact of storage and replication overhead.

primary. As the workload scales, the primary becomes a bottleneck, and clients in remote regions still pay cross-region latency to reach the primary even when replication itself is asynchronous. Similarly, TiDB relies on centralized components (e.g., PD) for metadata and timestamp services; while TiCDC can replicate changes asynchronously, transaction execution still depends on those services and thus remains sensitive to cross-region latency.

Scalability. The default configuration spans three regions with four data nodes per region. In Figure 15a, we vary the number of data nodes per region from 4 to 16. Throughput scales with the number of data nodes. Asynchronous replication achieves better because it avoids waiting for cross-region acknowledgments on the transaction critical path. We also evaluate deployments with 1 to 5 regions (4 data nodes/region). The results are shown in Figure 15b.

Control-plane overhead. We measure Global Manager (GM) load to validate that our routing metadata service is not a bottleneck.

As described in §3, GM is off the transaction critical path: it serves only (i) routing-metadata lookups on FE/RM cache misses, and (ii) control-plane updates. We quantify GM load under peak TPC-C runs (in the scalability experiments) and trigger global-key repartitioning for load balancing. As K2+ does not rely on global-key repartitioning for load balance in normal operation, load balance is primarily achieved within each region via resource elasticity enabled by disaggregated storage. Under this stress workload, >99.8% of data-plane requests are routed using FE/RM cached global range views, and GM is contacted only on cache misses (primarily during migration/repartition transitions). In our experiments, GM's CPU utilization was <43%, indicating that GM is not the bottleneck.

6.2 Performance Sensitivity.

YCSB. To evaluate the impact of data skewness and the read/write ratio on K2+ 's performance, we use YCSB to vary workload parameters. Figure 16 presents the results. As expected, higher conflict rates and a higher proportion of writes naturally reduce performance, since more operations must be serialized, limiting overall parallelism. Nevertheless, K2+ 's performance remains robust across varying client counts, thanks to resource elasticity (enabled by resource disaggregation) and effective load balancing across nodes.

6.3 Disaggregation and Decoupling

The key benefits of storage disaggregation and component decoupling are improved resource utilization (§6.5), faster fault recovery (§6.4), and more flexible replication placement policies (§4.3),

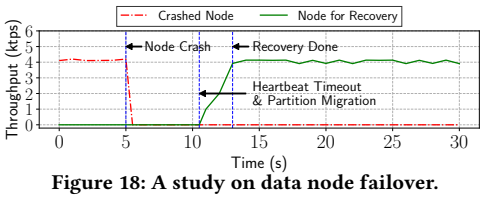


Figure 18: A study on data node failover.

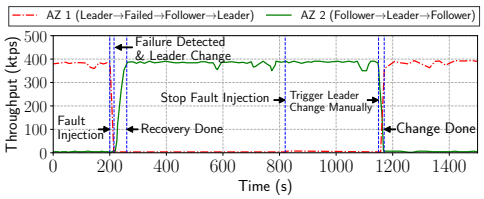


Figure 19: A study on availability zone failover.

thereby meeting our design goals (§2.1). These advantages, however, may come with additional performance overhead. In this experiment, we quantify that overhead in terms of both throughput and latency, and use a within-K2+ ablation to attribute the overhead to (i) the storage substrate (LocalFile → DFV) and (ii) cross-region replication (DFV → Async → Sync).

The throughput is studied in Figure 17. Compared to using local files for data persistence, using DFV slightly reduces throughput while improving data reliability and availability by replicating data across three availability zones. Thanks to Seastar's asynchronous scheduling framework, the throughput overhead remains minimal (< 2%). Latency-wise, DFV takes $300\mu s$ for appending new data while local files take less than $100\mu s$.

We then study the incremental overhead of cross-region replication on top of DFV. The throughput overhead of asynchronous replication is under 3%, since K2+ globally decouples the replication layer from data nodes: replicators only read the write-ahead logs produced by data nodes in DFV and do not block the data path. In addition, asynchronous replication adds negligible latency overhead. Differently, synchronous replication can reduce throughput by up to 42%, because it requires additional acknowledgment messages between replicators and data nodes before a transaction can complete. However, we view this overhead as an unavoidable cost of tolerating regional failures (i.e., requiring a cross-region durability point). The latency overhead of synchronous replication primarily depends on cross-region network latency and is about 45ms in our multi-region deployment.

Overall, our results suggest that storage disaggregation and component decoupling incur acceptable performance overhead in K2+.

6.4 Evaluation on Failure Recovery

Failure injection and recovery metrics. We use controlled failure injection under a fixed open-loop YCSB workload to quantify recovery behavior. At time t_0, we inject failures using scripts: we crash a target process via `kill -9` and emulate network isolation using `iptables` rules that drop traffic on the affected node/region links. We define *recovery time* as the elapsed time from t_0 until system throughput reaches at least 95% of the steady-state throughput measured immediately before the crash. We also report the transient throughput drop during the failure window.

Node failure. As shown in Figure 18, we inject a data-node crash at 5*s* by terminating the process using scripts. In our experience,

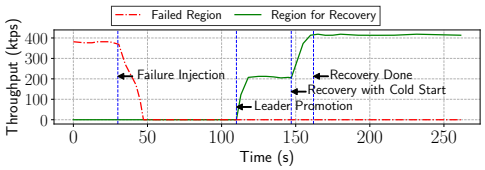


Figure 20: A study on regional failover.

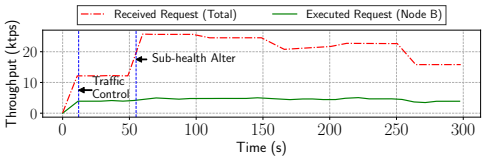


Figure 21: A study on scaling out and migration.

the dominant failover cost is failure detection: the regional manager monitors data nodes via heartbeats, and K2+ employs a conservative 5*s* threshold to avoid false-positive failure reports. After detection (at 10*s*), the regional manager updates in-region data-placement metadata and migrates affected partitions to nodes with available capacity. Recovery to the pre-failure throughput takes about 3*s*.

AZ failure. Next, we study failover of an availability zone (AZ). As shown in Figure 19, we inject a failure at 200*s* by manually enforcing a network partition between AZs. The failure is detected by a regional-manager replica in another AZ at 215*s*, which then triggers a leader promotion to initiate recovery. By 260*s*, all data nodes have recovered and migrated to new AZs. At 820*s*, we restore connectivity between the failed AZ and the other AZs. The regional manager in the previously failed AZ then rejoins the consensus group as a follower. At 1150*s*, we manually change the leader role back to the previously failed AZ. This step is not required for failover; we include it to study K2+'s throughput during normal leader changes. During the change, the throughput of the original leader temporarily decreases while the new leader ramps up. The aggregate throughput across AZs remains stable.

Region failure. In Figure 20, we evaluate K2+ under a regional failure. As discussed in §1 and §4.3, K2+ enables flexible replica management: it keeps a data partition active in another region only when that partition serves read-only transactions, while leaving reliability entirely to DFV and the replicators. Consequently, when recovering a data partition in a new region, there are two recovery modes. For partitions currently served by data nodes, we promote the corresponding replica to primary by updating metadata in the global and regional managers. For partitions that are inactive, we reconstruct in-memory components (e.g., memtables and indexes) and register the data partition with the managers; we refer to this as *cold-start recovery*. In our experiments, replicas are promoted to primaries immediately after the global manager confirms a regional failure (at 110 s in Figure 20), whereas cold-start recovery requires an additional ∼ 27 s. Note that the additional ∼ 27 s is incurred only for partitions that are inactive and thus require reconstructing in-memory state. This is a resource-efficiency trade-off, based on the assumption that regional failure is rare. SLO-critical partitions can be pinned as always-active in a secondary region to avoid cold-start.

Global Manager (GM) Failure. GM is coordinated by ZooKeeper; we rely on ZooKeeper's leader election to handle GM process failures and its strong consistency to serialize global view change. To stress this mechanism, we kill the active GM leader during peak workload; ZooKeeper triggers re-election and a new GM leader takes over automatically. Throughout the failover window, FEs/RMs continue routing requests using versioned cached global range views (re-election will not update view versions).

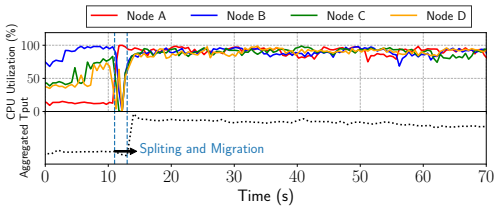


Figure 22: A study on load balance between data nodes.

6.5 Evaluation on Elasticity

To evaluate the resource elasticity enabled by storage disaggregation and component decoupling, we stress-test K2+ under overload. Figure 21 shows K2+ 's macro-level behavior when the workload exceeds Node B's processing capacity. K2+ uses pre-failure detection to identify unhealthy nodes and applies traffic control to preserve the goodput of overloaded nodes (§5.5). If the hotspot does not subside, the regional manager triggers load balancing across data nodes or adds a new node. Figure 22 reports the CPU utilization of Nodes A-D and the aggregate transaction throughput.

Overall, the data partitions on Node B are split and migrated to the new Node A and underutilized Nodes C and D, improving aggregate throughput by 5.7×. Splitting and migration complete within 3 s, enabled by K2+'s efficient storage management (see Figure 11 and Figure 12) and component decoupling.

This experiment highlights two key design benefits of K2+. First, different regions can maintain independent physical partition layouts for the same global range, so replica regions do not need to mirror splits or migrations. As a result, no global coordination or cross-region communication is involved. Second, each CPU core can host multiple physical partitions, enabling finer-grained intra-region load balancing: only selected partitions of a hot shard are migrated to underutilized nodes or cores. This allows workloads to be balanced across CPU cores. As shown in Figure 22, the CPU utilization of all data nodes reaches over 80% after balancing.

6.6 Real-world Use Cases

In our final experiment, we evaluate K2+ 's performance using two representative real-world applications that align with our design goals. They were chosen because they stress different parts of the system: one employs synchronous cross-region replication for mission-critical workloads, while the other leverages asynchronous replication and dynamic load balancing for write-heavy workloads.

App#1 is a user management system for cloud-resource management, which authenticates users and enforces access-control policies. It is mission-critical and requires strong consistency. The service is deployed on K2+ across three data centers (i.e., Wulan, Dongguan, and Shenzhen) on 32 machines with 2,048 CPU cores. Figure 23 shows the speedup over MongoDB. Cross-region read-only replicas accelerate authentication queries that access remote partitions. Figure 24 reports data freshness via visibility delay; with a 1s snapshot interval, K2+ achieves an average delay of 521ms.

App#2: A graph data processing service that supports large-scale graph construction, traversal, and analytics over highly connected

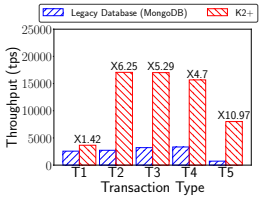


Figure 23: App#1 Workloads.

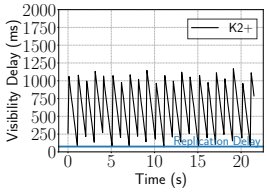


Figure 24: Cross-region VD.

cloud data. The data access patterns on graphs are highly skewed, requiring dynamic load balancing and migration. It was deployed on 512 CPU cores across two data centers. Asynchronous replication improves performance for non-critical data. In a single-region deployment, K2+ achieves a 2.76× speedup over FoundationDB: 1.94× from efficient transaction management, 1.42× from load balancing, with minimal replication overhead (Table 1).

7 RELATED WORK AND LESSONS LEARNED

7.1 Serverless Transactional Database

Storage disaggregation. Over the past decade, storage disaggregation has become a foundational paradigm for cloud-native transactional databases [9]. It enables elastic compute scaling and fast recovery by separating compute from a shared, highly available storage substrate. Representative systems include Amazon Aurora [38], Microsoft Socrates [2], Google AlloyDB [16], PolarDB/PolarFS [5], Taurus [12], and Neon [25]. These systems predominantly target *intra-region* deployments, where low-latency access to the shared storage and a single-region failure model simplify transaction ordering and replication. In contrast, K2+ extends the serverless/disaggregated paradigm across *multiple geo-regions* by introducing a decoupled middleware layer for cross-region replication and management, while retaining region-local shared storage pools. This design makes globally decoupled compute and storage practical under WAN and enables region-local partitioning and elasticity without forcing storage/compute coupling at remote replicas.

Memory disaggregation. Memory disaggregation further decouples volatile state from CPUs to improve elasticity and accelerate recovery. Systems such as LegoBase [42], PilotDB [30], *PolarCXLMem* [40], and GaussDB [22] explore pooled memory via RDMA or CXL to hold buffers and metadata remotely. However, these efforts largely assume a single-region fabric and do not address the end-to-end challenges of multi-region transaction processing. Integrating memory disaggregation with geo-distributed transactional semantics remains open and is beyond the scope of this paper.

7.2 Multi-region Transactional Database

Extensive work has studied strongly consistent transactional databases in geo-distributed settings [8, 14, 15, 26, 29, 36, 41]. Some of them have been commercialized. For example, Spanner [11] uses Multi-Paxos [21] for cross-region replication and relies on TrueTime for external consistency. CockroachDB [34] uses per-range Raft groups. FoundationDB [43] unbundles transaction logs (TLogs) and storage, but still enforces a global commit order through commit proxies, resolvers, and log replication. Aurora DSQL [1] is another new multi-region relational database providing active-active synchronous replication across two read/write Regions and a third witness Region that participates in the transaction quorum. While these designs ensure strong consistency, their replication mechanisms remain coupled with data ranges or shards. In contrast, K2+ implements cross-region replication via decoupled WAL-shipping over region-local storage pools, enabling N-to-M correspondence between primary and replica partitions, and allowing region-local partition management without requiring global coordination during splits, merges, or migration.

Serverless multi-region systems. More recently, CockroachDB Serverless [33] enhanced flexibility via cluster virtualization, enabling resource sharing across tenants. It separates the SQL and data layers, analogous to the separation of frontends and data nodes in our system. Relative to this, K2+'s contribution is to push disaggregation between storage and data nodes further, so that cross-region replication no longer forces compute/storage coupling and does not require per-range consensus for each replica.

	FDB	K2+ (DFV w/o LB)	K2+ (DFV)	K2+ (DFV+async. rep)
Tput (ktps)	150.19	291.55	415.05	408.35

Table 1: Performance factor analysis for App#2.

7.3 Lessons Learned

L#1: Enable region-local partition autonomy. In K2+, layered routing policies and the replication middleware let each region manage partitions independently—split/merge and load balancing become purely local operations. Cross-region replication only needs to respect WAL order and split/merge lineage, so partition elasticity no longer depends on synchronous global-metadata updates. This significantly improves operational responsiveness under skewed load and frequent resharding.

L#2: Keep only critical partitions warm. Regional failures are rare and high availability across regions is not free: keeping all partitions warm everywhere is prohibitively expensive, while on-demand activation introduces cold-start recovery time. We adopt an "active-on-demand" strategy for efficiency, while explicitly pinning critical partitions as always active in backup regions. This makes the trade-off effective and protects latency-sensitive workloads.

L#3: High reliability and availability require full-stack co-design. In production, multi-region HA and reliability only hold if you engineer and operate them end-to-end: transaction ordering, durability/replication, and failover orchestration must share explicit invariants and failure semantics, backed by fault-injection testing and runbooks that make rare failures predictable.

8 CONCLUSION

This paper presents K2+, a multi-region database for high reliability, availability, and strong consistency. Built on robust intra-region storage pools and a new middleware for efficient cross-region replication, K2+ globally decouples storage from compute to enable flexible partition management. The results show that K2+ achieves high performance and rapidly recovers from faults.

ACKNOWLEDGMENTS

The work is supported in part by National Key R&D Program of China (2022ZD0160201), HK RGC RIF (R7030-22), HK RGC GRF (ref No.: 17208526, 17208223, and 17204424), a Huawei flagship research grant in 2023, SupernetAI, and the HKU-CAS Joint Laboratory for Intelligent System Software.

REFERENCES

[1] Amazon Web Services, Inc. 2026. What is Amazon Aurora DSQL? https://docs.aws.amazon.com/aurora-dsql/latest/userguide/what-is-aurora-dsql.html. Accessed: 2026-07-09.

[2] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[3] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 331–343.

[4] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 617–632.

[5] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[6] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[7] Claudia Cauli, Timo Lang, Shuo Chen, Sebti Mouelhi, Xin Jin, Subhajit Bandopadhyay, Xusheng Chen, Yazhi Feng, Haoze Song, Linhua Tang, Zhenli Sheng, and Ananth Shrinivas Srinath. 2026. Lessons Learned from Incorporating Formal Methods in Huawei Cloud Reliability. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 2223–2238. doi:10.1145/3767295.3803626

[8] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 210–227.

[9] Zongzhi Chen, Xinjun Yang, Mo Sha, Feifei Li, Kang Wang, Zheyu Miao, Jie Xu, Jianfeng Wang, and Sheng Wang. 2025. CloudJump II: Optimizing Cloud Databases for Shared Storage. In *Companion of the 2025 International Conference on Management of Data*. 336–349.

[10] Cockroach Labs. 2022. CockroachDB Docs: Follower Reads (Read-only replicas). Online documentation. https://www.cockroachlabs.com/docs/stable/follower-reads.html Accessed: 2026-02-13.

[11] 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. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[12] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1463–1478.

[13] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 223–230.

[14] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[15] Jinkun Geng, Shuai Mu, Anirudh Sivaraman, and Balaji Prabhakar. 2025. Tiga: Accelerating geo-distributed transactions with synchronized clocks. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 555–571.

[16] Google Cloud. 2026. AlloyDB for PostgreSQL. https://cloud.google.com/products/alloydb Accessed: 2026-06-23.

[17] Google Cloud. 2026. Cloud Spanner Service Level Agreement (SLA). https://cloud.google.com/spanner/sla Accessed: 2026-06-23.

[18] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[19] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[20] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[21] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121)* (2001), 51–58.

[22] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.

[23] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. 2023. Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 49–64.

[24] Microsoft. 2026. Append Block — Azure Blob Storage REST API. https://learn.microsoft.com/en-us/rest/api/storageservices/append-block Accessed: 2026-07-09.

[25] Neon, Inc. 2026. Neon: Serverless Postgres. https://github.com/neondatabase/neon. Accessed: 2026-07-09.

[26] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[27] Oracle. 2026. MySQL 8.4 Reference Manual: Replication. Online documentation. https://dev.mysql.com/doc/refman/8.4/en/replication.html Accessed: 2026-02-13.

[28] PingCAP. 2026. TiCDC Overview. Online documentation. https://docs.pingcap.com/tidb/stable/ticdc-overview/ Accessed: 2026-07-09.

[29] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1747–1761.

[30] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 498–512.

[31] ScyllaDB. 2026. Seastar: High Performance Server-side Application Framework. https://github.com/scylladb/seastar. Accessed: 2026-07-09.

[32] Haoze Song, Yongqi Wang, Xusheng Chen, Hao Feng, Yazhi Feng, Xieyun Fang, Heming Cui, and Linghe Kong. 2025. K2: On Optimizing Distributed Transactions in a Multi-Region Data Store with TrueTime Clocks. *Proceedings of the VLDB Endowment* 18, 6 (2025), 1756–1769.

[33] Jeff Swenson, Andy Kimball, Raphael 'kena' Poss, Rebecca Taft, Jay Lim, Adam Storm, Sumeer Bhola, Paul Bulkley-Logston, Pj Tatlow, Rachael Harding, Rafi Shamim, Aditya Maru, and Irfan Sharif. 2025. CockroachDB Serverless: Sub-second Scaling from Zero with Multi-region Cluster Virtualization. In *Companion of the 2025 International Conference on Management of Data*. 648–661.

[34] 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 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[35] The Apache Software Foundation. 2024. Apache HBase. https://hbase.apache.org/. Version 2.6.0. Accessed: 2026-07-09.

[36] 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 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[37] Transaction Processing Performance Council. 2014. TPC-C. http://www.tpc.org/tpcc/. Accessed: 2026-07-09.

[38] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[39] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, Chengjun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[40] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data*. 689–702.

[41] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–37.

[42] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912.

[43] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data*. 2653–2666.