Concurrency Control as a Service

Weixing Zhou
Northeastern University
China
zhouwx@stumail.neu.edu.cn

Yanfeng Zhang
Northeastern University
China
zhangyf@mail.neu.edu.cn

Xinji Zhou
Northeastern University
China
zhouxj@stumail.neu.edu.cn

Zhiyou Wang
Northeastern University
China
wangzy@stumail.neu.edu.cn

Zeshun Peng
Northeastern University
China
pengzs@stumail.neu.edu.cn

Yang Ren
Huawei Tec. Co., Ltd
China
renyang1@huawei.com

Sihao Li
Huawei Tec. Co., Ltd
China
sean.lisihao@huawei.com

Huanchen Zhang
Tsinghua University
China
huanchen@tsinghua.edu.cn

Guoliang Li
Tsinghua University
China
liguoliang@tsinghua.edu.cn

Ge Yu
Northeastern University
China
yuge@mail.neu.edu.cn

ABSTRACT

Existing disaggregated databases separate execution and storage layers, enabling independent and elastic scaling of resources. In most cases, this design makes transaction concurrency control (CC) a critical bottleneck, which demands significant computing resources for concurrent conflict management and struggles to scale due to the coordination overhead for concurrent conflict resolution. Coupling CC with execution or storage limits performance and elasticity, as CC's resource needs do not align with the free scaling of the transaction execution layer or the storage-bound data layer.

This paper proposes Concurrency Control as a Service (CCaaS), which decouples CC from databases, building an execution-CC-storage three-layer decoupled database, allowing independent scaling and upgrades for improved elasticity, resource utilization, and development agility. However, adding a new layer increases latency due to the shift in communication from hardware to network. To address this, we propose a Sharded Multi-Write OCC (SM-OCC) algorithm with an asynchronous log push-down mechanism to minimize network communications overhead and transaction latency. Additionally, we implement a multi-write architecture with a deterministic conflict resolution method to reduce coordination overhead in the CC layer, thereby improving scalability. CCaaS is designed to be connected by a variety of execution and storage engines. Existing disaggregated databases can be revolutionized with CCaaS to achieve high elasticity, scalability, and high performance. Results show that CCaaS achieves 1.02-3.11× higher throughput and 1.11-2.75× lower latency than SoTA disaggregated databases.

PVLDB Reference Format:
Weixing Zhou, Yanfeng Zhang, Xinji Zhou, Zhiyou Wang, Zeshun Peng, Yang Ren, Sihao Li, Huanchen Zhang, Guoliang Li, and Ge Yu. Concurrency Control as a Service. PVLDB, 18(9): 2761 - 2774, 2025.
doi:10.14778/3746405.3746406

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746406

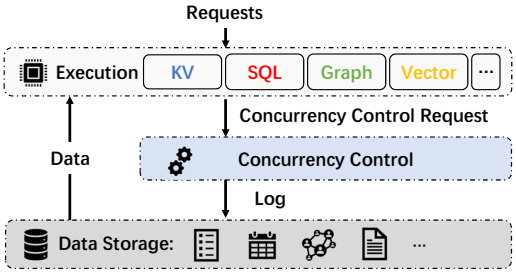


Figure 1: An execution-CC-storage three-layer decoupled database architecture.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/iDC-NEU/CCaaS.

1 INTRODUCTION

Database systems are evolving to a compute-storage disaggregated architecture [24, 33, 48, 52, 73, 80, 83, 87], such as Amazon Aurora [20], Socrates [23], PolarDB [28], and AlloyDB [1]. These databases typically decouple the system into an execution layer, which requires substantial computational resources, and a storage layer, which necessitates significant storage capacity. Compared to traditional databases where execution and storage are bundled together, these two-layer databases allow compute and storage resources to be scaled independently, thereby providing greater elasticity in the cloud environment, which are also called cloud-native databases. A set of works [43, 80, 83, 86] are proposed to improve these cloud-native databases from various aspects. As more and more enterprises move their applications to the cloud, these disaggregated databases are gaining wide popularity.

The spirit of cloud-native architecture is decoupling. A system should be decoupled into independent function modules, each with specific resource requirements. Cloud provides the elasticity of different decoupled resources (*e.g.*, computation, memory, and storage), allowing the growing or shrinking of resource capacity to adjust to changing demands. Each decoupled function module can be scaled independently to meet varying demands, fully utilizing the

decoupled resources. This approach enables the system to achieve elasticity. Furthermore, these function modules can be designed as independent services [2, 7, 22, 27, 29, 89], so that each function module can be reused by various applications and can be upgraded independently, thereby bringing more agility.

Concurrency Control (CC) is a key function module in databases to ensure that concurrent data access operations proposed by different users do not break data consistency. CC deals with concurrent conflicts and guarantees transaction ACID properties, and is evolving towards distributed with the evolution towards highly scalable cloud-native architectures. Adding nodes can enhance distributed transaction CC performance. However, this requires coordination among participates to resolve concurrent conflicts and ensure the ACID properties. System performance drops if the coordination overhead outweighs the benefits of the increased computing resources when adding too much nodes. CC has limited scalability, but it still needs high computational resources to resolve concurrent conflicts. The resource requirements of CC are neither consistent with SQL execution nor with data storage. Yet, most existing cloud-native databases simply couple CC either with the execution layer [20, 23, 28, 33, 52] or the storage layer [48, 71, 89], which limits the performance and elasticity of these systems (more details are discussed in Section 2).

Furthermore, the core of CC is resolving read-write conflicts on data items without caring about data types. Such a general applicability is often overlooked by existing databases, which are typically designed for specific engines and data models.

Considering the principle of cloud-native design (*i.e.,* decoupling functionality), it is desirable to decouple CC from the database system to maximize scalability and elasticity. By making CC an independent service, it can be connected to multiple engines with different data models (*e.g.,* relational, KV, Graph). This approach allows the CC service to be reused more easily and independently upgraded and evolved, promoting development agility.

This paper presents ***Concurrency Control as a Service*** **(`CCaaS`)**, a concept aimed at decoupling CC into a separate service and building an execution-CC-storage three-layer database (as shown in Figure 1). The system can dynamically adjust the resources of each layer based on the workloads (*e.g.,* computation, transaction processing, and data storage), thereby improving elasticity and resource utilization. Machines designed for specific scenarios (*e.g.,* compute-intensive, parallelism, storage-oriented) achieve better resource utilization by being deployed in different layers. In this architecture, execution engines independently execute transaction requests, read data from storage, and send resolving requests to `CCaaS` for concurrency control. Once conflicts are resolved, commit or abort notifications are sent back to the execution layer, and committed transaction logs are pushed to the storage layer for data updates.

Software-level disaggregation results in a significant performance reduction because processing over the network is slower than on a local machine [61]. Adding a new layer increases latency, as communication between execution and CC is switched from hardware to network. To tackle this, we propose a Sharded Multi-Write OCC (SM-OCC) algorithm with asynchronous log push-down mechanism. Employing an optimistic execution strategy can reduce the number of network communications between the execution and CC layer, as execution engines no longer need to send locking requests.

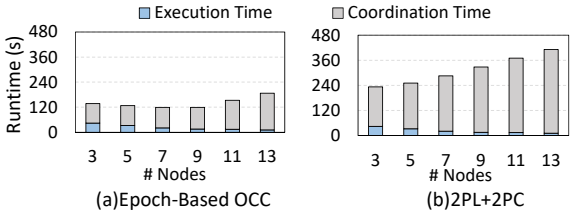


Figure 2: Breakdown of total processing time for distributed transactions with a changing number of nodes.

Asynchronous logging further decreases latency by allowing transactions to commit once logs are persisted in the CC layer, instead of waiting for data updates in the storage layer. Additionally, the CC layer encounters limited scalability when resolving transaction concurrent conflicts, which necessitates coordinated communication among multiple nodes. To address this, we aim to use a deterministic decision-making method to minimize coordination overhead between nodes to enhance the scalability.

As CC focuses on resolving read and write conflicts on data items, the influence of data models can be mitigated by logically abstracting data access. We design a set of interfaces, which only expose data operation types to the CC layer, so that `CCaaS` can be connected by multiple different engines with various data models.

In summary:

- We propose Concurrency Control as a Service (`CCaaS`), a novel execution-CC-storage three-layer decoupled database architecture. CC is decoupled from the database and works as a service, enhancing system scalability, elasticity, and agility. (Section 3).
- We propose a sharded multi-write optimistic concurrency control algorithm (SM-OCC) with asynchronous logging to enhance the scalability of CC and overall system performance (Section 4).
- We make several case studies on connecting existing execution/storage engines to `CCaaS` to demonstrate the benefits of the three-layer architecture (Section 6).

2 THE CASE FOR CCAAS

2.1 Resource Requirements of CC

To study the resource requirements of distributed CC, we evaluate the runtime of an epoch-based optimistic concurrency control algorithm (epoch-based OCC) and a distributed two phase locking + two phase commit (2PL+2PC) algorithm under a distributed environment. The atomicity and distributed consistency are validated in an epoch-based manner in epoch-based OCC and ensured by 2PC in 2PL+2PC. We generate 10 million distributed transactions using the YCSB-A benchmark, with each transaction containing 10 random operations (5 reads + 5 writes). We process these distributed transactions under different environment setups with different numbers of nodes (3 to 13 nodes), where the data is evenly and randomly distributed among nodes in each run. For more details, please refer to the description of Competitors and the distributed environment setup in Section 7. We record transaction execution time and inter-node coordination time (*i.e.,* waiting for messages from other nodes).

Figure 2 shows the total time for execution and coordination. No matter epoch-based OCC or 2PL+2PC, most time is spent on inter-node coordination. The total time for transaction execution is

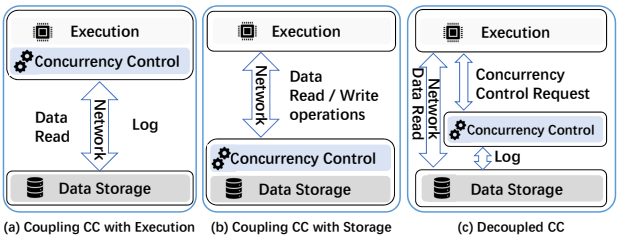


Figure 3: Comparison of different decoupled databases.

steadily decreasing as the number of nodes increases, since more nodes are involved in performing data operations. However, the total time for inter-node coordination is increasing as the number of nodes increases in 2PL+2PC. This is because more nodes could introduce more inter-node coordination overhead. Even though the total runtime is slightly decreasing as the number of nodes increases from 3 to 9 in epoch-based OCC, adding more nodes (greater than 9) can introduce significant coordination overhead and then outweigh the benefits of more compute resources. According to these results, we observe that CC often has limited scaling capabilities due to coordination overhead, and the resource requirement of CC is not consistent with that of transaction execution, where execution prefers relatively more compute nodes but CC prefers fewer nodes. This motivates us to decouple CC from database architecture and make it as an independent service.

2.2 Limitations of Existing Decoupled DBs

Most existing decoupled databases overlook the specific requirements of CC and simply couple CC with transaction execution or data storage, leading to performance and scalability limitations.

Databases like Aurora [20] and PolarDB [28] **couple CC with transaction execution**, as illustrated in Figure 3a. Such a design allows execution nodes to quickly process user requests and manage transaction conflicts. Adding more nodes increases computing resources, distributes the execution load, and prevents single-node bottleneck. However, it also raises coordination overhead by involving more nodes in conflict resolution, scaling too much nodes will hurts the system performance (limited scalability of CC). Moreover, coupling CC with execution limits system agility. Execution engines should be tailored to optimize execution plans for various data models or hardware types. For instance, graph databases like Nebula Graph [78] and Neo4j [12] support graph queries such as sub-graph matching and shortest path. Vector databases like Milvus [10] are built for similarity search, while GPU databases like GDB [45], MapD [58], and GPUDB [84] are optimized for parallel processing [68]. However, the core of CC algorithms [44, 55, 57, 72, 81, 82, 88] are similar, *i.e.,* handling concurrent read-write or write-write conflicts. Coupling CC with the execution layer would incur redevelopment costs for resolving transaction conflicts.

Some other disaggregated databases, such as Solar [89] and TiDB [48], **couple CC with data storage**, as depicted in Figure 3b. In these systems, all persistent states reside in the storage layer, rendering execution nodes stateless. The execution nodes are only responsible for computation, forwarding data operations to storage. This design enables dynamic adjustment of execution nodes to efficiently handle varying workloads, optimizing resource utilization. Conversely, storage nodes are stateful, and expanding them for more capacity involves tasks like data re-sharding and migration, making them less flexible to scale. Furthermore, storage nodes typically have large amounts of storage space but limited computing resources, and storing large data volumes necessitates numerous storage nodes. Coupling CC with storage makes CC hardly elastically scalable. Managing CC with limited computation resources provided by storage nodes could hurt performance.

Interfaces for the execution layer:

Begin(ExecutionInfo)	➢ Begin a transaction, generate TxnID and other metadata.
TxnCommit(*TxnID, RS, WS*)	➢ Validate the read and write sets and commit the transaction.
Lock(*TxnID, Key[], Value[] OpType[]*)	➢ Try to lock requested keys, abort the transaction if lock request fails.
Commit(*TxnID*)	➢ Commit the transaction, write logs and release locks in CCaaS.
Abort(*TxnID*)	➢ Abort the transaction, release locks in CCaaS.

Interfaces for the storage layer:

LogPush(LogAdaptorID)	➢ Register a Callback at CCaaS, CCaaS uses it to actively push logs to the storage layer.
LogPull(LSN[])	➢ Request the missed logs from CCaaS specified by LSNs for data recovery.

Figure 4: CCaaS Interfaces.

3 SYSTEM ARCHITECTURE

Motivated by the aforementioned analysis, we build a separate CC service for improving system scalability, elasticity, and agility.

3.1 CCaaS Interface

Execution Layer to CCaaS. Although data processing in execution engines varies significantly by data models, all have two basic operations: read and write. Based on this, we define unified interfaces that only expose the read and write operations to the CC layer and hide the impact of different data models, for different execution engines to interact with.

For OCC, after transaction execution, an execution engine generates a set of records, including read operations (*i.e.,* readset) and write operations (*i.e.,* writeset). During CC conflict resolution, the readset and writeset are used to detect read-write conflict and write-write conflict. Therefore, we define a unified interface, **`TxnCommit()`**, where execution engines standardize transaction execution results using a defined data structure (Unified Transaction Read Set and Write Set, as shown in Figure 5) and send them to `CCaaS` for conflict detection. On the other hand, for PCC, a set of interfaces (**`Lock()`**, **`Commit()`, and `Abort()`**, as shown in Figure 4) is provided for the execution layer to interact with `CCaaS`, ensuring correct transaction processing with locks.

Notably, decoupling CC from existing compute-storage disaggregated databases does not change their ability to support existing data operations. However, it encounters challenges depending on the type of CC algorithms used in `CCaaS`. OCC is an ideal choice for decoupling CC, as it requires only slight modifications to the transaction commit process, which sends the read and write sets of transactions to CCaaS for conflict resolution. Decoupling CC with pessimistic mode (PCC) poses greater challenges and incurs higher costs. Lock requests need to be sent to CCaaS for conflict detection before accessing the data, introducing multiple network round-trips in PCC mode. In cases where the data to be accessed is unknown beforehand, such as accessing data via secondary indexes, additional operations (e.g., an initial optimistic execution to identify

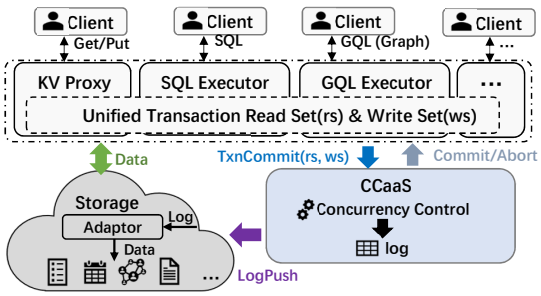


Figure 5: The overall architecture of CCaaS (OCC-based).

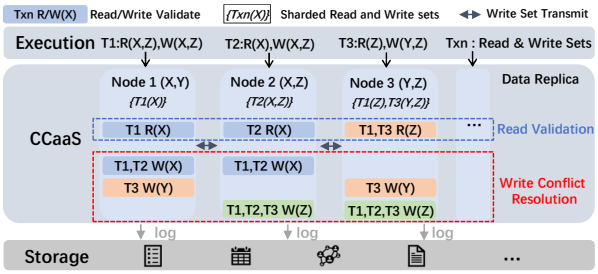


Figure 6: The architecture of CCaaS with SM-OCC.

the data) are required. In this paper, we mainly focus on supporting OCC as a first step toward achieving a decoupled CC service.

CCaaS to Storage Layer. Storage engines (*e.g.,* columnar [21], row-column hybrid [46], graph-native [35], and vector [75]) differ significantly in their data structures and update interfaces, pre-developing interfaces in CCaaS to update data for all storage engines is impractical. Thus, we choose to **transform the write sets of transactions into logs** and send the logs to the storage layer. Meanwhile, a **LogAdaptor** is required for a storage engine to provide a specification on how to transform the logs to the corresponding data with a specific schema (Section 4.4). We define a Callback **`LogPush()`** that is registered at CCaaS, so that CCaaS will use it to push logs to the storage layer actively, and the LogAdaptor receives these logs and converts them to data. Furthermore, the storage layer relies on the **`LogPull()`** interface to verify the completeness of the logs received according to the log sequence number (LSN) and requests missed logs from CCaaS.

3.2 System Workflow

By connecting execution and storage engines to CCaaS, we construct an execution-CC-storage three-layer decoupled database. The execution layer, which may consist of multiple engines with different data models (*e.g.,* KV, Relational, Graph), receives user requests, executes computation, reads from the storage, and sends unified CC requests to CCaaS for conflict resolution. It is noticeable that for KV engines that lack transaction support, such as LevelDB [9] and HBase [3], only get/put operations are supported on a single item. An additional transaction proxy (*e.g.,* the KV proxy as shown in Figure 5) is required to provide transaction semantics (*e.g.,* `BeginTransaction`, `Commit`, `Abort`) for users to interact with.

When a transaction request arrives, the execution layer calls the `Begin()` interface to initialize a transaction. In OCC, the execution layer optimistically executes the transaction. For example, SQL engine parses SQL statements (*e.g.,* SELECT) and retrieves metadata (*e.g.,* table schema, data distribution) from storage to generate a physical execution plan. Based on the plan, the engines read data from the storage layer without locking (stale reads may occur) and cache reads and writes locally. When transaction execution is finished, the execution layer sends the read and write sets to CCaaS for conflict detection. In PCC, CCaaS provides a distributed lock manager for lock authorization and conflict detection. Unlike OCC, PCC requires several interactions between the execution engines and CCaaS. When the execution layer invokes the `Commit()` interface in PCC or after resolving conflicts in OCC, CCaaS returns Commit/Abort to the execution layer and pushes the updates in the format of logs to the storage layer through the `LogPush` interface.

Since OCC requires much fewer interactions between the execution layer and the CC layer, OCC is more suitable as the CC protocol in CCaaS. To enhance CC scalability and performance, we propose a **Sharded Multi-Write OCC (SM-OCC)** algorithm as the default CC mechanism in CCaaS in Section 4.

4 CCAAS DESIGN

4.1 CCaaS Overview

Requirements. Since the CC layer operates as an independent service, CCaaS should satisfy a set of specific requirements:

First, CCaaS must be highly available. Classic master-follower architecture provides high availability but comes with several significant drawbacks. 1) All the conflicts must be resolved in the master node; a single-node bottleneck occurs when concurrent CC requests increases, causing heavy resource contention. 2) The master-follower architecture experiences temporary unavailable when the master node goes down unexpectedly. The system necessitates a pause for complete log replaying in the followers. To address this challenge, we adopt a **Multi-Master** architecture [4, 34, 79], where each master has the capacity to resolve transaction conflicts. CC requests are distributed across multiple masters, preventing single-point performance bottlenecks. If a master fails, other masters continue providing CC services without interruption.

Second, while the use of a multi-master architecture can fully utilize each replica and provide high availability, it incurs a write amplification problem. The same set of writes needs to be processed in every master to maintain replica consistency. To mitigate this, **Sharding** strategies [32, 33, 65, 66, 70] can be used to enhance the scalability, where each node handles only a subset of shards, reducing the write amplification impact.

Third, the transaction execution should be **Optimistic**. Prior studies have proposed various concurrency control algorithms, including pessimistic [44, 57, 82], optimistic [56, 74, 81, 88], and deterministic [47, 55, 63, 64, 72] methods, each with its own pros and cons. Using a PCC algorithm in the CC layer involves sending numerous lock requests from the execution layer, leading to increased network traffic and transaction latency. Deterministic algorithms provide strong consistency guarantees among multiple replicas through *deterministic execution*, which reduces coordination overhead. However, they are not universally applicable, as they struggle with interactive transactions and often require pre-processing. To leverage the strengths of both optimistic and deterministic approaches, we propose a combined method. Transactions

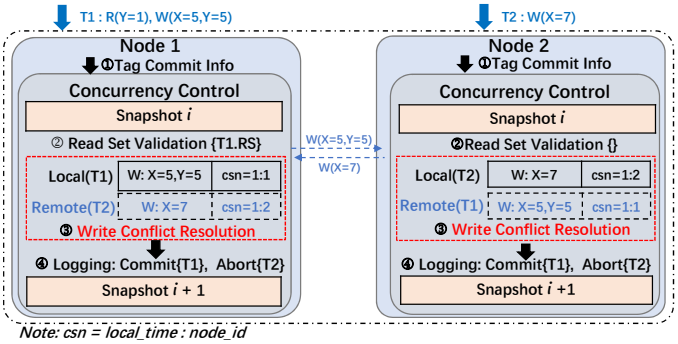


Figure 7: Conflict resolution within a data shard.

are ***optimistically executed*** in the execution layer. Once a transaction is submitted to CCaaS, the multi-master nodes in the CC layer ***deterministically compare*** the read and write sets according to predefined rules. It is worth noting that deterministic comparisons occur only during the validation phase. Users can still interact with the database during transaction execution. Support for multi-round and interactive transactions remains unaffected.

Fourth, **Epoch-Based Committing** suits the sharded architecture better. Typically, sharded systems use protocols like two-phase commit (2PC) for transaction atomicity, which requires multiple round-trip acknowledgments, leading to significant network overhead. The Epoch-Based Commit protocol [56] groups transactions into epochs, using the entire epoch as the coordination unit to minimize communication overhead. The original transaction-granularity synchronization is transformed into epoch-granularity, effectively reducing coordination overhead.

Given these, we propose a **Sharded Multi-Write OCC (SM-OCC)** algorithm. Notably, the CC algorithm (*e.g.,* SM-OCC) is changeable. Lock-based algorithms like 2PL can also be used in CCaaS to reduce the abort rate caused by using OCC. If the storage layer supports multi-version data, MVCC can also be integrated into CCaaS [41, 52]. Algorithms [30, 31, 38, 49, 76, 77] that leverage RDMA to reduce network overhead can also be applied to CCaaS to enhance performance. Users can choose an appropriate CC algorithm that suits their workloads but must consider its impact on system availability, scalability, and performance.

CCaaS Architecture with SM-OCC. In CCaaS, each node maintains several shards of committed transaction metadata. For example, as shown in Figure 6, node 1 manages Shards X and Y, node 2 manages Shards X and Z. Identical replicas are deployed across nodes, using the Raft [59] consensus protocol to synchronize updates and ensure consistency. Each node acts as a master, allowing nodes to manage transaction conflicts independently (*e.g.,* both node 1 and node 2 can resolve conflicts on Shard X). They receive read and write sets (referred to as transactions) and partition them based on a sharding strategy (*e.g.,* range-based or hash-based), routing subtransactions to corresponding nodes for conflict resolution. Since each node has the capacity for conflict resolution, subtransactions can be routed to any one of the nodes (masters). For instance, node 1 shards transaction $T1$ into $T1(X)$ and $T1(Z)$, sending $T1(Z)$ to Node3. Alternatively, $T1(Z)$ can also be send to node 2 since it holds a replica of Z.

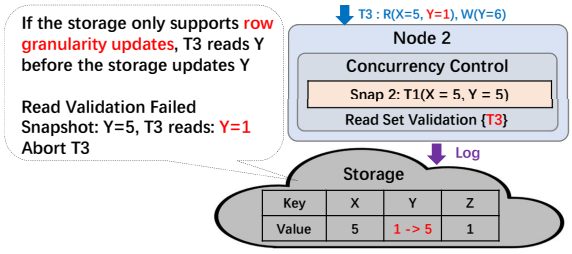


Figure 8: Read validation when connecting to the storage engines only supports row granularity updating.

CCaaS provides a certain degree of scalability and elasticity. When scaling out for more computational resources, CCaaS assigns shard replicas to the new node. After synchronizing metadata with peers, the new node begins resolving transaction conflicts. When scaling in, nodes transfer the replicas to other nodes for replacement. If the access rate of a shard increases, resulting in higher resource contention on some nodes, CCaaS can share the overhead of read-set validation by increasing the number of replicas, or re-partition the shard to distribute the overhead of write-set resolution. CCaaS only maintains the meta-information of committed transactions, and when re-sharding is performed, only the meta-information managed by the CC nodes is redistributed. CCaaS does not experience significant network bandwidth usage.

4.2 Sharded Multi-Write OCC

Multi-Write OCC. For convenience, we first introduce conflict resolution within a shard and then present the difference with sharding. The main workflow are shown in Figure 7. First of all, CCaaS divides physical time into epochs (*e.g.,* 10 ms per epoch) and assigns incremental unique numbers to these epochs. Each node collects read and write sets (transactions) and packs them at epoch granularity based on the reception time. Upon receiving a transaction, CCaaS node tags it with commit-info, which includes the commit epoch number (CEN, indicating the epoch to which the transaction belongs) and the commit sequence number (CSN, identifying the transaction), for subsequent processing.

Nodes synchronize transactions with each other at epoch granularity, and operate in an epoch manner: after transactions of the i epoch have been executed, snapshot i is generated and nodes start the conflict resolution for transactions of epoch $i+1$ (*e.g.,* $T1, T2$ in node 1 and node 2).

There are two types of conflicts between transactions: read-write and write-write conflicts. The core procedure of SM-OCC is divided into **Read Set Validation** and **Write Set Resolution** phases.

Read Set Validation. Each node first independently validates the read sets of locally received transactions based on snapshot i. CCaaS adopts Snapshot Isolation (SI) by default to validate the reads. When CCaaS connects to storage engines with only row-level updates, an additional read-error scenario may occur: a transaction may read data partially modified by another transaction, beyond the conflicts seen with traditional transaction-level updates. Figure 8 shows this issue. This read error occurs when transaction $T1$ updates Y, which has already been read by $T3$. $T1$ updates X and Y to 5 and commits. $T3$ reads X as 5 and Y as 1. Despite no conflicts in the same epoch, $T3$ must be aborted to maintain atomicity (*i.e.,* it should read all

Algorithm 1: Write Set Conflict Resolution

Input: a transaction $txn.\{CSN, WS\}$.
Output: Commit or Abort

```
1  Function WriteSetResolution(Transaction txn):
2      result = Commit;
3      foreach r in txn.WS do
4          row = GlobalWriteVersionMap.Find(r.key);
5          if row is not Null and r.type is Insert then
6              result = Abort; //row already exists.
7          else if row is Null and r.type is not Insert then
8              result = Abort; //deleted in previous epoch.
9          else
10             result = Compare(r);
11     if result is Abort then
12         EpochAbortSet.Insert(txn.CSN);
13     else
14         foreach row in txn.WS do
15             GlobalWriteVersionMap.Set(row);
16     return result;
17 Function Compare(Record r):
18     row = EpochWriteVersionMap.Find(r.key);
19     if row is Null then
20         //row has not been updated in current epoch.
21         row.CSN = txn.CSN;
22         EpochWriteVersionMap.Set(row);
23     else if (row.CSN > txn.CSN) then
24         //mark the transaction with CSN row.CSN as abort.
25         EpochAbortSet.Insert(row.CSN);
26         row.CSN = txn.CSN;
27         EpochWriteVersionMap.Set(row);
28     else
29         return Abort;
30     return Commit;
```

or none of $T1$'s updates). To ensure atomicity, transactions like $T3$ that are aborted by checking snapshots.

Write Set Conflict Resolution. For transactions passing the read validation phase, their write sets are sent to remote nodes (other masters) to ensure data consistency between replicas. Once all local write sets of epoch $i+1$ are sent and all peers' write sets are received (epoch synchronization), the write resolution phase begins.

Algorithm 1 is designed as **deterministic** to resolve write conflicts, allowing each node to resolve conflicts independently without coordination, thus reducing communication overhead. Each node uses a *GlobalWriteVersionMap* (snapshot) to record information of committed transactions. For each epoch, each node uses an *EpochWriteVersionMap* to track transaction write intents within the current epoch and maintains an *EpochAbortSet* to store the CSNs of transactions that should be aborted.

During the resolution phase, records (*e.g.,* rows) in the write set are traversed to detect conflicts. For each write operation, it is essential to first check its validity: operations that attempt to update or delete a non-existing row, or insert a row that already exists, are not allowed as they conflict with committed transactions. To verify this, the *GlobalWriteVersionMap* is first searched to determine the row's current state, ensuring the write operation can proceed correctly at the storage layer (lines 5-8).

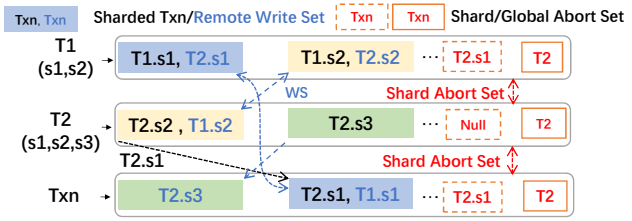


Figure 9: Transaction processing in sharded multi-write OCC.

If no conflicts with committed transactions are found, the `Compare` function (lines 17-30) is used to check for write conflicts within the current epoch. In the function, the *EpochWriteVersionMap* is used to detect if another transaction is attempting to update the same row : 1) $row == NULL$ means that no other transaction has tried to update the row, so the current transaction can proceed. The updating intent with the transaction's CSN is inserted into the map(lines 16-19). 2) Otherwise, a write-write conflict occurs, and a partial order '$\prec$' between CSNs is used to determine which transaction wins (lines 20-26). The CSN of a transaction is composed of local time + node id. To avoid single-point bottlenecks caused by using a central sequencer, the CSN is assigned using local timestamp with node id.

Definition 1. $T1$ '$\prec$' $T2$ if '$T1.local\_time < T2.local\_time$' or '$T1.local\_time = T2.local\_time$ & $T1.node\_id < T2.node\_id$'.

Notably, no two transactions with the same CSN exist. The case of $row.CSN == txn.CSN$ will never occur by using the local clock with node ID to assign CSN. Moreover, the comparison rules are changeable. The rules mentioned above lead to the fact that transactions arriving on nodes with smaller local times (clock skew) have a higher probability of winning, which may lead to inequities. For example, the comparison rules can be changed to commit transactions based on polling, making the comparison a bit fairer.

After completing the conflict resolution of all transactions, each node updates the *GlobalWriteVersionMap*, and writes logs. Then, a new snapshot based on the current epoch is generated.

To reduce data replication overhead, each node can performs write-set conflict resolution on local received transactions (local) first and only send the write-sets of the winning transactions to peer nodes. In this way, part of the write conflict resolution load can be distributed among nodes (masters), improving `CCaaS` scalability. Since the comparison rule is deterministic, transactions that lose in local conflict detection must be aborted even if their write-sets are sent to remote nodes. Performing local resolution and only sending wining write-sets does not affect the correctness of `CCaaS`.

Sharding. Figure 9 shows the sharded architecture with three nodes, forming a three-shard, two-replica setup. In this setup, each `CCaaS` node manages a portion of the read and write conflicts (*e.g.,* node 1 and node 3 are both responsible for shard 1, blue shard). Transactions are sharded into subtransactions and rerouted to the corresponding nodes. For instance, node 1 shards $T1$ into $T1.s1$ and $T1.s2$. Since node 1 hosts replicas of shard 1 and shard 2, it validates both sub-transactions locally. In contrast, node 2 sends $T2.s1$ to node 3, as node 2 does not maintain a replica of shard 1.

Once conflict resolution is completed, each node generates an **EpochAbortSet** (Shard). Since each node only manages part of the sub-transactions, the AbortSet on each node may do not account

for all transactions that need to be aborted. Therefore, an additional round of EpochAbortSet replication is needed to construct a **globally consistent abort set** for the epoch to ensure transaction atomicity. Then, each node aborts the relevant transactions based on the globally abort set and logs the committed write sets.

4.3 Isolation

CCaaS uses an epoch-based mechanism and defaults to Snapshot Isolation (SI). For storage engines with multi-version read support, read set validation can be skipped to speed up conflict resolution. At the start of a transaction, a start timestamp is recorded for validation. Upon commit, the transaction's read and write sets, along with the timestamp, are sent to CCaaS. CCaaS uses this timestamp to determine the appropriate snapshot for validation. Unlike traditional methods of preventing phantom reads, such as using index locks, CCaaS detects phantom reads based on the transaction's execution result. If the read version does not match the snapshot, the transaction is aborted.

CCaaS also supports Read Committed (RC) and Repeatable Read (RR) levels when the storage engine supports transaction-level updates. However, achieving Serializable (SER) isolation would require global read-write dependency tracking across nodes and epochs, introducing high network and computation overhead, which limits scalability. Considering these performance concerns, CCaaS currently does not support this isolation level. A possible approach is to build a read-write dependency graph and break dependency cycles [40].

4.4 Log Pushing

As discussed in Section 3, LogAdaptor converts logs into data structures compatible with each storage engine. To minimize engine modifications, the adaptor is preferably implemented in CCaaS. For example, for HBase [3], we implement the adaptor in CCaaS, which converts logs into structures (rowArray, rowOffset, and rowLength), and then invoke HBase's interface Put to update data. For engines without direct data modification support, modifications to the database are required. For instance, openGauss [13], which only provides SQL interfaces for updates, the log adaptor is implemented within openGauss, invoking internal update mechanisms to apply changes.

During logging, CCaaS first writes logs to local disks for persistence before pushing them to the storage layer and returning resolution results to the execution layer. When using SM-OCC, an asynchronous log-pushing mechanism can be used to reduce committing latency, where each node returns the resolution results to the execution layer before finishing pushing the log. This does not affect correctness, as CCaaS maintains up-to-date meta-information for conflict resolution. However, in write-intensive workloads, it may increase transaction abort rates due to stale data reads.

4.5 Fault Recovery

Building a execution-CC-storage three layered database requires careful handling of failures at each layer.

Failure in the execution layer. Execution nodes optimistically read data and temporarily store updates in memory. If a node fails before committing the transaction to CCaaS, the updates are lost, but the other layers remains unaffected. The user can resubmit the transaction to another active node. If a node fails while waiting for the CC result, the user can connect to another node to check if the data has been updated and confirm if the transaction was committed. Since the transaction is already sent to CCaaS, it will be correctly processed without effectiveness.

Failure in the CCaaS layer. CCaaS uses the Raft protocol [60] to ensure fault tolerance. In SM-OCC, each shard master maintains a Raft instance (Shard Raft) for to make a consensus on the status of live masters, which can prevent permanent blocking (waiting for the write sets from a failed master). Additionally, each node maintains a Raft instance (Txn Raft) to replicate its locally received transactions to other nodes for backup. Once Txn Raft replication completes, nodes proceed with transaction sharding and conflict resolution.

When a node fails, CCaaS takes different actions depending on whether the node completed the write set exchange for the current epoch. For simplicity, consider a non-sharded example: node 1 and node 2 receive transactions $T1$ and $T2$ respectively, which have a write-write conflict. If node 1 fails before $T1$'s write set is replicated, node 2 produces a GlobalAbortSet without $T1$. During log push, the newly elected Txn Raft leader checks the GlobalAbortSet and pushes $T1$'s log, leading to inconsistencies as both $T1$ and $T2$ are pushed to the storage layer. To prevent this, CCaaS re-executes the CC for the current epoch if a failed node hasn't completed write set replication. The newly elected Txn Raft leader takes over conflict resolution for the failed node and transmits the write sets of the transactions received by the faulty node to other nodes.

In a sharded setup, incomplete EpochShardAbortSets from different masters can result in an incorrect GlobalAbortSet, necessitating epoch re-execution. When a node fails, new Raft leaders are elected. The new Shard Raft leader takes response for the failed node. If the write set transfer was completed before the failure, other shard masters can process the complete write sets for the current shard and produce a correct conflict resolution result (*i.e.,* EpochShardAbortSet). If not, CCaaS will re-execute the current epoch with the updated Raft leaders.

CCaaS 's multi-master architecture ensures that individual node failures do not affect availability. Execution nodes connected to failed nodes can reroute to healthy ones, ensuring uninterrupted CC service. When the failed node recovers, Raft leadership is restored, and the system returns to normal. In cases where multiple nodes in a Raft instance fail, conflict resolution cannot proceed without achieving Raft consensus. CCaaS responds by re-sharding the data, rebuilding Raft instances, and re-executing the epochs. Even if a majority of CCaaS nodes fail due to network partitions or power outages, the database system can still provide read-only service by passing CCaaS. In such cases, no Raft leader is elected, log push-down is terminated. Upon recovery, nodes first push their remaining local logs. As all states stored in main memory are lost, the nodes then catch up with the latest state by fetching Raft logs before resuming service.

Failure in the storage layer. As described in Section 3, the storage layer may include distributed systems like HBase [3] or standalone databases like openGauss [13]. Distributed systems already provide

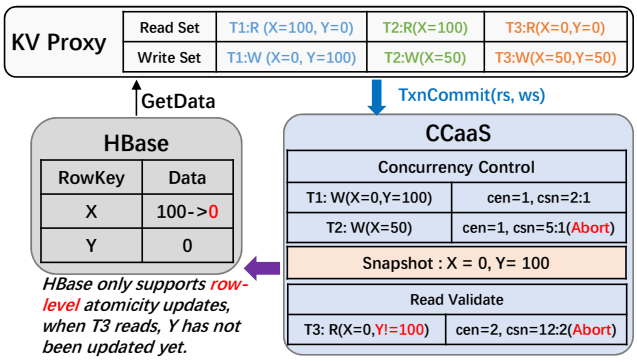


Figure 10: An example of empowering HBase with TP capability.

fault tolerance, so no additional mechanisms are needed. For standalone databases, multiple instances must be deployed, each with a full replica (see Section 6.2). The failure of a majority of storage nodes does not affect the storage layer's ability to serve data. When a node recovers, it pulls and replays logs from CCaaS or peer nodes.

5 DISCUSSION

Opportunities of Decoupling. Most database developers tend to decouple database systems based on the disaggregation of storage and computing provided by cloud providers, often prioritizing hardware disaggregation while overlooking resource contention among functional modules. Considering the principles of decoupling, the performance implications of module coupling need to be considered. For example, CC requires efficient handling of concurrent transactions, while logging favors sequential I/O, and record storage demands efficient random access. In serverless architectures, different query operators exhibit varying computational demands, necessitating tailored resource allocation (*e.g.,* parallel scanning vs. high-frequency processing).

Another key aspect is functional generality. Decoupled function modules can be designed as generalized services, *e.g.,* CCaaS not only resolves conflicts across execution nodes but also supports heterogeneous execution engines, facilitating a heterogeneous execution layer (Section 6.3). Additionally, CCaaS ensures broad compatibility through log adaptors, allowing integration with various storage engines (e.g., columnar, row-based, disk, or memory) based on system requirements.

Trade-offs. Decoupling CC offers several benefits, and we demonstrate several case studies in Section 6 for illustration. However, the impact on system performance and maintenance complexity should also be considered. Decoupling CC introduces additional network communication overhead, and CCaaS is more suitable for OCC. Decoupling CC in the case of using PCC algorithms tends to incur higher overhead. Furthermore, decoupling may lead to increased overhead due to the need to manage the three distinct layers. Each layer will have its own maintenance requirements, which could increase maintenance complexity. Additionally, in highly partitioned workloads, coordination overhead across database nodes is minimal. The scalability of the system can not be limited by coupling CC with other layers. In such situations, decoupling CC for higher scalability is not necessary.

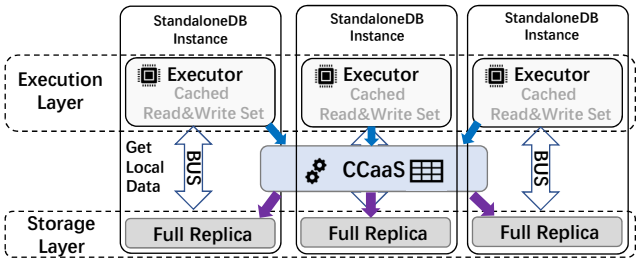


Figure 11: An illustration of building a multi-master DB.

Implications to Existing Cloud-Native Databases. Existing cloud-native databases adopt the *Log-as-the-Database* principle to reduce network I/O. CCaaS and the SM-OCC algorithm can seamlessly integrate with the storage-disaggregated architectures [61], introducing several key improvements: CCaaS enables traditional master-follower architectures to evolve into multi-master architectures, improving elasticity and availability. Execution nodes can retain their local CC mechanisms while CCaaS enforces global consistency, allowing for a flexible concurrency control design that adapts to different workload characteristics. Furthermore, SM-OCC employs row-level conflict resolution, improving parallelism compared to page-level approaches [34, 79, 90]. With CCaaS, existing databases can gain more flexibility in system architecture and performance optimization.

6 CASE STUDIES

In this section, we demonstrate several advantages of CCaaS by presenting some study cases.

6.1 Empowering NoSQL DBs with TP Capability

Most NoSQL databases [3, 9, 12, 78] do not provide transaction semantics for better scaling. We offer a solution to add transaction support without modifying the original logic. By connecting existing NoSQL databases to CCaaS, these databases can be empowered with TP capability and ACID properties. For example, Figure 10 shows how we connect HBase, a distributed key-value store, to CCaaS to enable TP. First, we implement a KV Proxy to provide transaction semantics, linking it with both CCaaS and HBase. The proxy provides `Start, Get, Put, RollBack` and `Commit` interfaces for transaction operations and executes user read/write requests, and caches the data locally. When a transaction is committed, the proxy sends the cached read and write sets to CCaaS. As described in Section 4.4, a log adaptor in CCaaS uses HBase's putRow interface to update the data. We use HBase [3] as an example, and CCaaS has also been connected to NoSQL DBs like LevelDB [9] and NebulaGraph [78].

6.2 Making Standalone TP Distributed

The performance of standalone databases like openGauss [13], PostgreSQL [15], and MySQL [11] is inherently constrained by the scalability limits of single-node hardware. These systems struggle under high TP workloads and face availability challenges due to their single-node deployment model. By connecting multiple standalone instances to CCaaS, a multi-master distributed TP system can be easily established (Figure 11). Each instance maintains a full replica of data, independently processes users' requests, and sends

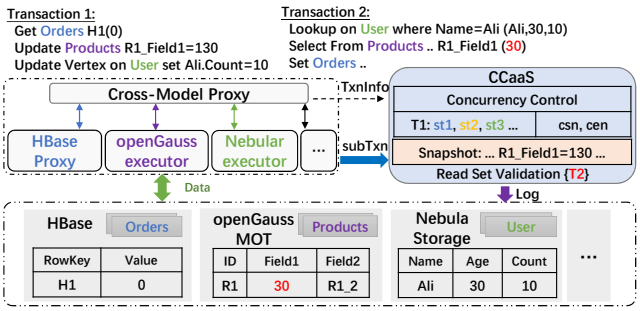


Figure 12: Cross-model transaction processing with CCaaS.

CC requests to CCaaS to resolve transaction conflicts within and between instances. Compared to the master-follower architecture, this design can maximize resource utilization across all instances, distribute the workload, and eliminate single-node bottlenecks. The system also achieves high availability, as other instances can continue serving user requests when some fail.

6.3 Supporting Cross-Model Transactions

Modern business involves multiple data models (*e.g.,* Relational, KV, Graph, Vector) stored in various databases, often adopting different storage engines for diverse data needs. Businesses may need a transaction with ACID properties to modify data in different databases. We assume that a cross-model database is a database that can store, index, and query data across multiple data models. However, ensuring ACID transactions across these databases is challenging due to heterogeneous query languages and the complexity of maintaining consistency across multiple data stores [36, 41, 50, 85].

CCaaS provides a trivial solution by decoupling concurrency control from data models, enabling cross-model transaction support. As shown in Figure 12, users send requests to a Cross-Model Proxy, which forwards them to relevant execution engines, splitting a cross-model transaction into multiple single-model sub-transactions. Once all single-model transactions are executed, the Proxy sends commit commands to each execution engine to submit single-model transactions to CCaaS. Meanwhile, it sends transaction information to CCaaS, indicating which single-model transactions belong to the same cross-model transaction. Upon receiving all sub-transactions and metadata, CCaaS merges them and applies the same conflict resolution algorithm, ensuring consistency across heterogeneous storage engines.

Since different storage engines cannot directly communicate, updates in a cross-model transaction cannot occur atomically across engines. This may lead to inconsistent reads in new transactions (e.g., $T2$ in Figure 12), violating transaction atomicity. CCaaS detects such anomalies by validating $T2$'s read set against snapshots and aborts $T2$ if inconsistency is found. After resolving concurrency conflicts, CCaaS returns the results to the Proxy and execution nodes, then pushes logs to the respective storage engines. Since cross-model transactions span multiple engines, each log entry is labeled with an identifier specifying its target engine, ensuring correct log propagation and consistency.

7 EVALUATION

This section evaluates the performance of CCaaS with SM-OCC.

Implementation. We implement CCaaS with ~10,000 lines of C++ code. We develop a KV Proxy as the KV execution engine based on code [14] to support transaction semantics for LevelDB [9] and HBase [3]. Since LevelDB is a standalone KV store, we implement data access interfaces by using brpc [5] to enable remote data access. OpenGauss [13] is a standalone database and only exposes its query interfaces for users. We modify its source code (less than 1,500 LoC), decouple the execution engines (openGauss-execution) and the storage engines (openGuass-MOT [25]), and build an openGauss-execution - CCaaS- openGauss-MOT database. Similar to openGauss, we also modified NebulaGraph [78] source code, enabling its execution engine (Nebula Graph) and storage engine (Nebula Storage) to connect to CCaaS.

Physical Environment. Our experimental cluster is deployed on Aliyun, and each layer consists of 3 nodes by default. Each node (ecs.c6.4xlarge instance) is equipped with 16 vCPUs and 32 GB RAM and runs Ubuntu 22.04 LTS. All nodes are connected with a local area network of 5 Gbps. The default epoch length of CCaaS is 5ms. We use openGauss-execution - CCaaS- openGauss-MOT database as our default choice. Since openGauss-MOT [25] is an in-memory store, we deployed it on a 16vCPU and 64GB RAM node to store a large amount of data in memory.

Competitors. We implement an epoch-based OCC protocol and a 2PL+2PC protocol under the same codebase with CCaaS for comparison. The data is evenly divided according to the number of nodes. During the 2PC, the node, which receives the transaction requests will participate in the transaction committing as the coordinator.

To compare the performance in the cloud as well as across data models, we select TiDB [48], FoundationDB [87] and Epoxy [50] as competitors. TiDB is a compute-storage disaggregated NewSQL database with the storage layer handling concurrency control (CC). It uses optimistic execution and Percolator [62] for transaction conflict resolution. FoundationDB (FDB) is a distributed key-value store that decouples CC from logging and storage, building a transaction-log-storage three-layer data store. FDB uses lock-free concurrency management with a deterministic transaction order and implements Serializable Snapshot Isolation (SSI) by combining OCC with MVCC. Epoxy is a middleware that enables connectivity to existing databases and ensures ACID transactions across heterogeneous storage systems by building a transaction metadata management layer. While Epoxy and CCaaS have different objectives in the architecture, both enhance distributed transaction processing. This shared approach makes Epoxy a relevant competitor for evaluating CCaaS's performance in multi-engine and disaggregated databases.

Deployment. We follow the official documents [6, 16] to deploy TiDB and FoundationDB. In TiDB, where the CC and storage layers are coupled, we deploy the execution layer with 3 machines, each with 16 vCPUs, and the storage layer with 3 machines, each with 32 vCPUs (16 for CC and 16 for storage). For FDB, We deploy each layer with 3 machines, each with 16 vCPUs. We deploy Epoxy with 3 machines, each with 16 vCPUs. Since the execution and data storage of the underlying databases (openGauss-MOT, LevelDB, Nebula) are coupled together, we deployed them on 32 vCPUs machines. We deployed stand-alone database instances on three nodes by default, forming a multi-master architecture. Distributed databases (*e.g.,* HBase) are also built on a three-node configuration. By default, we

Table 1: Summary of Workloads

Name		Data Size	Operation Type
YCSB [19]	**YCSB-A**	1 M rows	10op/txn (50% read - 50% write)
	YCSB-B		10op/txn (95% read - 5% write)
TPC-C [17]		100 warehouse	50% NewOrder - 50% Payment
LDBC-SNB [8]		SF10	10op/txn (50% fetch - 50% insert)
Cross-Model		1 M rows (KV) 1 M rows (SQL) SF10 (Graph)	10op/txn (5 KV, 4 SQL, 1 Graph) (KV, SQL: 80% read - 20% write) (Graph: 50% fetch - 50% insert)

configure 3 client servers for each database, with each client-server hosting 32 clients that connect to the database and send transaction requests. When the number of execution nodes is expanded, the number of client servers scales correspondingly.

Workloads. Table 1 presents the workloads used in the evaluation. To make YCSB a transactional workload, we wrap 10 operations into transactions (txn) with a Zipfian access distribution. For the TPC-C workload, we run a 50% NewOrder - 50% Payment mix to focuse on conflict resolution. In LDBC-SNB, we wrap 10 random fetch/insert operations into a transaction. Additionally, to evaluate the cross-model transactions, we synthetically generate a multi-model transactional workload.

7.1 Overall Performance

We compared CCaaS with TiDB [48], FoundationDB [87] and Epoxy [50]. Figure 13 reports the throughput and latency results. Since FoundationDB does not support TPC-C, we only compare TiDB and Epoxy for the TPC-C workload. For the YCSB-B workload, CCaaS achieves 1.11-2.62 times higher throughput and 1.02-2.62 times lower latency than others. For the TPC-C workload, CCaaS achieves 1.19-2.05 times higher throughput and 1.21-2.15 times lower latency. Additionally, we evaluate TiDB, FDB, and CCaaS under the YCSB-B workload by varying the number of nodes per layer (from 3 to 7) and scaling the execution nodes while keeping other layers constant. The results in Figure 14 demonstrate that CCaaS achieves 1.31–3.11 times higher throughput.

The performance of CCaaS benefits from SM-OCC (Section 4.2), allowing each node to resolve transaction conflicts independently. Specifically, our epoch-based mechanism only requires replicating write sets among the CCaaS nodes, which reduces network round-trips (RTTs) caused by coordination. Additionally, the asynchronous log push-down method reduces transaction latency caused by updating data storage. The results can immediately be returned after finishing conflict resolution. In CCaaS, committing a transaction only requires 3 RTTs of network communication.

FoundationDB decouples logging from CC. Transaction commit results cannot be returned until the corresponding logs are replicated, requiring 4 RTTs and resulting in higher latency. TiDB uses a variation of Percolator [62] for CC and Raft [60] to replicate logs. Committing a transaction requires multiple RTTs (2PC + Raft, over 4.5 RTTs), increasing latency. Additionally, its pessimistic locking mechanism limits transaction concurrency. Epoxy uses multiple standalone databases for data storage. The execution time of a single statement in Epoxy is low. However, it needs to use Epoxy Coordinator and Epoxy Shim as proxies to forward user requests, maintain additional meta-information to ensure MVCC across multiple storage engines, and use an optimized two-phase

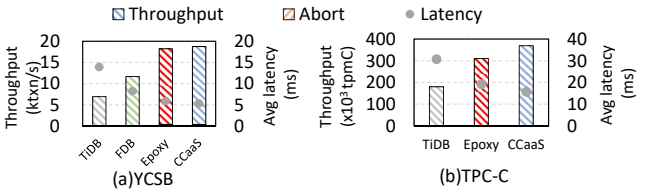


Figure 13: Experiment results with competitors.

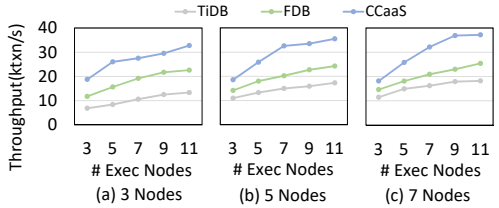


Figure 14: Comparison with existing disaggregated databases under YCSB-B workload.

commit protocol to ensure that transactions are committed atomically, introducing a certain amount of latency into the system. Therefore, its performance is approximate to that of CCaaS using SM-OCC for concurrency control.

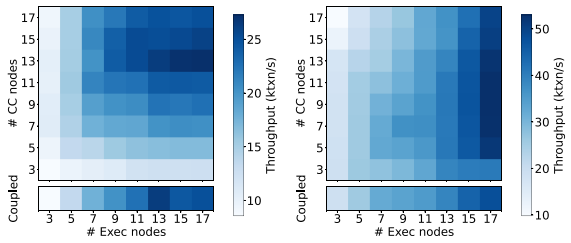
7.2 Scalability

In this experiment, we evaluate the scaling performance when CC is coupled with the execution engine and when scaling execution nodes and CC nodes independently. The results in Figure 15 show that the CC-execution decoupled system shows better throughput performance than the CC-execution coupled system, with various CC-execution node number combinations under both YCSB-A and YCSB-B workloads. Especially under the YCSB-B workload, the optimal performance is obtained with 17 execution nodes and 7 CC nodes. This verifies the necessity of independent scaling of CC, which also requires much less cost. In addition, CCaaS shows better performance under the read-intensive YCSB-B workload than under the write-intensive YCSB-A workload. This is because large write sets incur more network overhead in the CC-decoupled system, reducing system performance.

7.3 Elasticity

To evaluate the elasticity improvements facilitated by CCaaS, we test the system under dynamic workloads using a mix of TPC-H and YCSB-B. TPC-H represents computationally heavy analytical processing (AP) workloads handled by the execution layer, while YCSB-B models transactional processing (TP) workloads requiring conflict resolution in the CC layer. We dynamically adjust the number of AP and TP clients over time to simulate the dynamic AP-TP mixed workloads, as shown at the bottom of Figure 16. In response to these workload changes, we dynamically adjust the number of nodes in the execution and CC layers and observe changes in TP throughput, as shown in the upper part of Figure 16.

As shown in Figure 16, when the AP workload increases at 12s, TP throughput drops due to CPU contention in the execution layer. To mitigate this, we scale out the execution layer by adding 2 nodes at 18s, quickly restoring TP throughput. At 45s, increasing TP clients makes the CC layer a bottleneck due to its limited capacity for handling concurrent transactions. Scaling out the CC layer at



(a) YCSB-A (b) YCSB-B

Figure 15: Throughput when scaling the number of execution nodes and CC nodes in CCaaS.

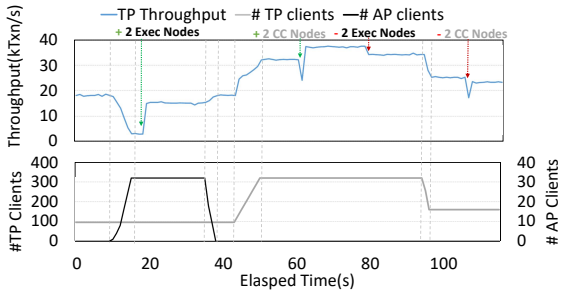


Figure 16: Elasticity performance under changing workloads.

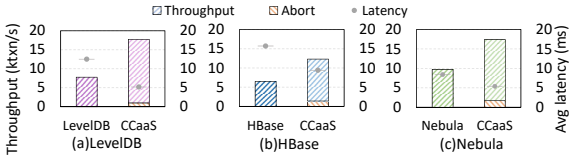


Figure 17: Performance of NoSQL DBs empowered with ACID TP capability.

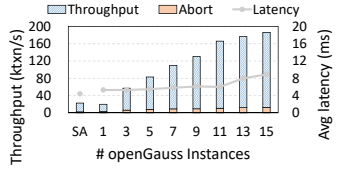


Figure 18: Performance of serving standalone TP engines.

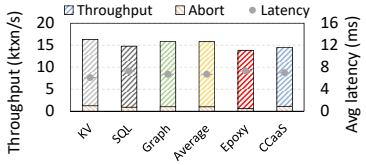


Figure 19: Performance of cross-model (CM) transactions.

60s improves throughput but triggers re-sharding in CCaaS, causing a temporary 27% drop due to the redistribution of committed transaction metadata. After 40s, the AP workload ends, reducing computational demand. At 80s, we remove 2 execution nodes to save costs, causing a slight TP throughput drop. At 100s, with fewer TP clients, the CC layer is over-provisioned, so we remove 2 CC nodes, with minimal performance impact. The results demonstrate that the decoupled CC layer enables flexible resource allocation and efficient response to workload changes. By allowing independent scaling of the execution and CC layers, CCaaS enhances elasticity while reducing costs.

Notably, CCaaS supports re-sharding without interrupting the CC service. When re-sharding begins, CCaaS first requires that conflict resolution follows the new sharding policy in a pre-determined number of epochs. During re-sharding, CC nodes transfer metadata (snapshot and subsequent metadata updates) to designated nodes while continuing conflict resolution under the original sharding strategy. Once re-sharding is completed, CCaaS switches to the new sharding policy and resumes normal processing.

7.4 Case Studies

Supporting TP for NoSQL DBs. To demonstrate the TP performance provided by CCaaS, we use the YCSB-B workload to evaluate the throughput and the average latency of original NoSQL databases and that of the CCaaS-enhanced ones. As shown in Figure 17, by connecting to CCaaS, these NoSQL databases gain transaction processing capability and show higher operation throughput and lower latency due to the log asynchronous push-down method.

Building a Multi-Master Database. Figure 18 shows the TP's horizontal scalability by connecting multiple openGauss instances to CCaaS under the YCSB-B workload. When only one openGauss instance is connected to CCaaS, the performance is lower, and latency is higher compared to the standalone instance due to the additional network I/O introduced by CCaaS. Decoupling allows modules to scale independently to meet resource demands under varying workloads. However, when the workload is within the capacity of a single instance, the benefits of decoupling are not realized, and the network overhead becomes a burden. In contrast, as system load increases, adding more execution nodes allows the workload to be distributed across multiple nodes, avoiding single-point bottlenecks and improving throughput.

Cross-Model Transactions. We use the cross-model workload to test CCaaS and Epoxy. For comparison, we use the same data to generate a single model transaction (10 op/txn) to test each single-model storage engine. The weighted average throughput and latency of these single-model transactions are also reported. The results are shown in Figure 19. As shown, the KV storage engine is the fastest since it does not incur overhead from statement parsing, execution plan generation, *etc.* Due to the processing of Graphs in the workload is simpler than that of SQL, SQL has the lowest performance with the highest latency. Because CCaaS needs to collect all single-model sub-transactions, the performance of multi-model transactions is determined by the slowest engine. In supporting for cross-model transaction processing, Epoxy demonstrates performance that approximates that of CCaaS.

7.5 Varying Workloads

We evaluate the performance of CCaaS under zipfian and uniform access distributions using YCSB workloads. As shown in Figure 20, a high write rate with Zipfian distribution will cause more transactions to be aborted. This is because the SM-OCC only commits one transaction when multiple transactions update the same record. Under the Zipfian distribution, transactions are easier to access the same records, causing a higher abort rate. Comparing system performance under YCSB-A and B workloads, CCaaS has better performance under YCSB-B workloads. This is because under YCSB-A load, transactions have more write operations, causing higher network overhead in CCaaS and reducing system performance.

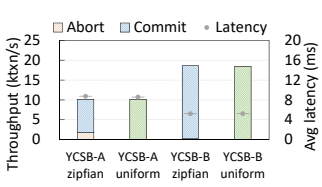


Figure 20: Performance under different contention.

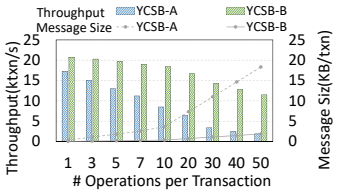


Figure 21: Performance when varying number of operations.

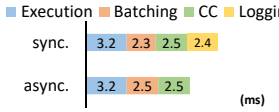


Figure 22: Runtime breakdown (sync./async. logging).

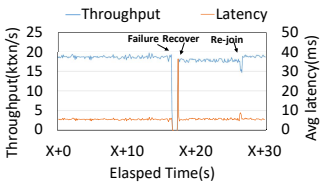


Figure 23: Performance variation when system recovery.

Additionally, we adjust the operation count of transactions to evaluate system performance with varying read-write set sizes. Figure 21 shows that the system works well with low operation numbers. Adding more operations lead to longer execution times and larger message sizes, reducing throughput. CCaaS performs better under the read-intensive workload (YCSB-B) because it validates reads by checking metadata in the read set (*e.g.*, CSN), which increases slowly. In the write-intensive workload (YCSB-A), additional write operations significantly increase the message size, and increasing write operations increases transmission and replication overhead in CCaaS. The write sets replication consume a lot of computation resources, causing performance degradation.

7.6 Sync. Logging vs. Async. Logging

As shown in Figure 22, we perform a breakdown analysis for a single transaction under the YCSB-A workload. Since YCSB-A is a write-intensive workload, the log push-down takes 2.4 ms on average per transaction. In comparison, the log push down in the read-intensive workload YCSB-B is less than 1ms (not shown in Figure 22 due to space constraints). When using asynchronous logging, the overall latency is reduced. However, since more transactions are processed per epoch, the batching and CC phases take longer.

7.7 Fault Recovery

To evaluate performance under failure, we manually shut down a CCaaS node and see how CCaaS acts. Figure 23 shows the changes in throughput and latency. There is a temporary performance drop following a node failure at 16 seconds, as active nodes wait for write sets from the failed node. CCaaS quickly responds to this failure due to Raft-based membership management (with a 500 ms timeout setup). After changing the leader of the Raft instances (Section 4.5), CCaaS resumes providing service. With only two nodes in CCaaS, overall throughput slightly decreases and latency increases due to the increased load on each node. After the crashed node recovers (at 26 seconds), the Raft-based membership management notices and adds the recovery node back to the cluster, and then the throughput and latency return to normal.

8 RELATED WORK

Decouple Transaction Management Component. Earlier works [33, 39, 51, 53, 54, 67] introduce transaction components (TM) to handle conflicts via virtual resources. Deuteronomy [51] uses TM to provide transaction processing (TP) capabilities for KV stores. In [39], TM is used to detect transaction dependencies and release locks early during 2PC. Omid [26, 41, 67] and Tell [52] use TM to implement multi-version concurrency control (MVCC), improving transaction throughput. DIBS [42] implements the TM with predicate locking to guarantee transaction isolation. These systems generally focus on enabling TP for existing data stores, using centralized TM to resolve conflicts. FoundationDB [87] decouple the TM based on the roles (coordinator and participant) in the 2PC for better scalability. CCaaS follows the principle of cloud-native design to decouple the CC layer, tries to use a multi-master architecture to improve the scalability of CC, and supports processing with multiple models by abstracting the CC from the data models.

Cross-Engine Transaction. Conventionally, cross-data store transactions are implemented through a distributed transaction protocol such as X/Open XA [69] or WS-TX [18]. Such protocols use two-phase commit to ensure atomicity. Cherry Garcia [37] and Omid [41] provide ACID transactions across multiple key-value stores but only support key-value operations. Skeena [85] proposes a holistic approach to cross-engine transactions and uses an atomic commit protocol to efficiently ensure correctness and isolation. Epoxy [50] provides ACID transactions across heterogeneous data stores by using an additional MVCC control panel, but it needs a primary transactional DBMS as a transaction coordinator for transaction processing. In comparison, CCaaS makes the CC module an independent service, allowing it to be connected by various data stores concurrently. By maintaining transaction meta-information and resolving conflicts at epoch-granularity, CCaaS allows the system to connect to multiple storage engines simultaneously.

9 CONCLUSION

This paper proposes Concurrency Control as a Service (CCaaS), an execution-CC-storage three-layer database architecture. We demonstrate that databases can be revolutionized with CCaaS: NoSQL databases can gain ACID TP ability, standalone TP databases can support distributed transactions with horizontal scalability, and cross-model transactions can be realized. Our evaluation results show that CCaaS outperforms existing disaggregated databases, including TiDB and FoundationDB, and exhibits a certain degree of scalability of transaction processing. CC, as an independent service, has potential that has not been fully developed. Our future work will focus on optimizations like the support of more isolation levels and consistency choices, and even cross-device CC service.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (62461146205), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and Huawei. Yanfeng Zhang and Zeshun Peng are the corresponding authors.

REFERENCES

[1] . 2024. AlloyDB for PostgreSQL. https://cloud.google.com/alloydb?hl=en.
[2] . 2024. Amazon S3. https://aws.amazon.com/s3/.
[3] . 2024. Apache HBase. https://hbase.apache.org/.
[4] . 2024. AWS Aurora Multi-Master. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Amazon_Aurora_Multi-Master_Scaling_out_database_write_performance_DAT404-R1.pdf.
[5] . 2024. bRPC: An industrial-grade RPC framework. https://brpc.apache.org/.
[6] . 2024. FoundationDB Official Documents. https://apple.github.io/foundationdb/configuration.html.
[7] . 2024. Google Cloud Storage. https://cloud.google.com/storage?hl=en.
[8] . 2024. LDBC-SNB. https://ldbcouncil.org/benchmarks/snb/.
[9] . 2024. LevelDB. https://github.com/google/leveldb.
[10] . 2024. Milvus. https://milvus.io/.
[11] . 2024. MySQL. https://dev.mysql.com/.
[12] . 2024. Neo4j. https://neo4j.com/.
[13] . 2024. openGauss. ĭlhttps://opengauss.org/.
[14] . 2024. PingCAP Go-YCSB. https://github.com/pingcap/go-ycsb.
[15] . 2024. PostgreSQL. https://www.postgresql.org/.
[16] . 2024. TiDB Official Documents. https://docs.pingcap.com/tidb/stable/hardware-and-software-requirements.
[17] . 2024. TPC-C. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.
[18] . 2024. Web Services Atomic Transaction. https://docs.oasis-open.org/ws-tx/wsat/2006/06.
[19] . 2024. YCSB. https://github.com/brianfrankcooper/YCSB/.
[20] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 666. https://doi.org/10.1145/872757.872855
[21] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712
[22] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, SOCA, 44–51.
[23] 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* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047
[24] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-Invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045
[25] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, et al. 2020. Industrial-strength OLTP using main memory and many cores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3099–3111.
[26] 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 17)*. 167–180.
[27] 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.
[28] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2477–2489. https://doi.org/10.1145/3448016.3457560
[29] Tomas Cerny, Michael J Donahoo, and Michal Trnka. 2018. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review* 17, 4 (2018), 29–45.
[30] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast In-Memory Transaction Processing Using RDMA and HTM. *ACM Trans. Comput. Syst.* 35, 1, Article 3 (July 2017), 37 pages. https://doi.org/10.1145/3092701
[31] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/2901318.2901349
[32] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 48–57. https://doi.org/10.14778/1920841.1920853
[33] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)* 38, 1 (2013), 1–45.
[34] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: Bringing Multi-Master to the Cloud. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3488–3500. https://doi.org/10.14778/3611540.3611542
[35] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. Tigergraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248* (2019).
[36] Akon Dey, Alan Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 125–136.
[37] Akon Dey, Alan Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 125–136.
[38] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. 2023. Fast One-Sided RDMA-Based State Machine Replication for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* 20, 2, Article 31 (April 2023), 25 pages. https://doi.org/10.1145/3587096
[39] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report MSR-TR-2016-1001. https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/
[40] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (jun 2005), 492–528. https://doi.org/10.1145/1071610.1071615
[41] Daniel Gomez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. 2014. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 676–687.
[42] Kevin P Gaffney, Robert Claus, and Jignesh M Patel. 2021. Database isolation by scheduling. *Proceedings of the VLDB Endowment* 14, 9 (2021).
[43] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. 2022. Cornus: atomic commit for a cloud DBMS with storage disaggregation. *Proc. VLDB Endow.* 16, 2 (oct 2022), 379–392. https://doi.org/10.14778/3565816.3565837
[44] Theo Härder. 1984. Observations on optimistic concurrency control schemes. *Information Systems* 9, 2 (1984), 111–120.
[45] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. 2009. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–39.
[46] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*. 1199–1208.
[47] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proc. VLDB Endow.* 17, 3 (nov 2023), 469–482. https://doi.org/10.14778/3632093.3632109
[48] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[49] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 185–201.
[50] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions Across Diverse Data Stores. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2742–2754.
[51] Justin Levandoski, David Lomet, and Kevin Keliang Zhao. 2011. Deuteronomy: Transaction support for cloud data. In *Conference on innovative data systems research (CIDR)*.
[52] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery,

New York, NY, USA, 663–676. https://doi.org/10.1145/2723372.2751519
[53] David Lomet, Alan Fekete, Gerhard Weikum, and Mike Zwilling. 2009. Unbundling transaction services in the cloud. *arXiv preprint arXiv:0909.1768* (2009).
[54] David Lomet and Mohamed F. Mokbel. 2009. Locking key ranges with unbundled transaction services. *Proceedings of the VLDB Endowment* 2, 1 (aug 2009), 265–276. https://doi.org/10.14778/1687627.1687658
[55] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (jul 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808
[56] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based commit and replication in distributed OLTP databases. *Proceedings of the VLDB Endowment* 14, 5 (jan 2021), 743–756. https://doi.org/10.14778/3446095.3446098
[57] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: scaling transactions through asymmetric replication. *Proceedings of the VLDB Endowment* 12, 11 (jul 2019), 1316–1329. https://doi.org/10.14778/3342263.3342270
[58] Todd Mostak. 2013. An overview of MapD (massively parallel database). *White paper. Massachusetts Institute of Technology* (2013).
[59] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319.
[60] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.
[61] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proc. ACM Manag. Data* 2, 3, Article 180 (May 2024), 26 pages. https://doi.org/10.1145/3654983
[62] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 251–264.
[63] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication.. In *EDBT*. 73–84.
[64] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647
[65] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1035–1046. https://doi.org/10.14778/2732977.2732979
[66] 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 (nov 2016), 445–456. https://doi.org/10.14778/3025111.3025125
[67] Ohad Shacham, Yonatan Gottesman, Aran Bergman, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2018. Taking omid to the clouds: Fast, scalable transactions for real-time cloud analytics. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1795–1808.
[68] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595
[69] CAE Specification. 1991. *Distributed Transaction Processing: the XA Specification*. X/Open.
[70] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 245–256. https://doi.org/10.14778/2735508.2735514
[71] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134
[72] 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* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838
[73] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.
[74] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713
[75] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. https://doi.org/10.1145/3448016.3457550
[76] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: hybrid is better. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 233–251.
[77] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419
[78] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An open source distributed graph database. *arXiv preprint arXiv:2206.07278* (2022).
[79] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/-PODS '24)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3626246.3653377
[80] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2101–2113. https://doi.org/10.14778/3476249.3476265
[81] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935
[82] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: harmonizing concurrency control and caching in a distributed oltp database management system. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.
[83] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1805.
[84] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.
[85] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and Consistent Cross-Engine Transactions. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 34–48. https://doi.org/10.1145/3514221.3526171
[86] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun 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. *Proc. VLDB Endow.* 14, 10 (jun 2021), 1900–1912.
[87] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2653–2666. https://doi.org/10.1145/3448016.3457559
[88] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, et al. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
[89] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. 2019. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage* 15, 2, Article 11, 26 pages. https://doi.org/10.1145/3318158
[90] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 685–699. https://doi.org/10.1145/3514221.3526187