Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation (Extended Version)

Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren,
Xiangyao Yu, Mahesh Balakrishnan[†], Philip A. Bernstein[‡]
University of Wisconsin-Madison, [†]Confluent, Inc., [‡]Microsoft Research
{zhihan,xzeng,kanwu,wuh-chwen,ziwei,yxy}@cs.wisc.edu
mbalakrishnan@confluent.io,philbe@microsoft.com

ABSTRACT

Two-phase commit (2PC) is widely used in distributed databases to ensure atomicity of distributed transactions. Conventional 2PC was originally designed for the shared-nothing architecture and has two limitations: *long latency* due to two eager log writes on the critical path, and *blocking* of progress when a coordinator fails.

Modern cloud-native databases are moving to a storage disaggregation architecture where storage is a shared highly-available service. Our key observation is that disaggregated storage enables protocol innovations that can address both the long-latency and blocking problems. We develop Cornus, an optimized 2PC protocol to achieve this goal. The only extra functionality Cornus requires is an atomic compare-and-swap capability in the storage layer, which many existing storage services already support. We present Cornus in detail with proofs and show how it addresses the two limitations. We also deploy it on real storage services including Azure Blob Storage and Redis. Empirical evaluations show that Cornus can achieve up to 1.9× latency reduction over conventional 2PC.

1 INTRODUCTION

Databases are migrating to the cloud because of desirable features such as elasticity, high availability, and cost competitiveness. Modern cloud-native databases feature a *storage-disaggregation* architecture where the storage is decoupled from computation as a standalone service as shown in Figure 1b. This architecture allows independent scaling and billing of computation and storage, which can improve resource utilization, reduce operational cost, and enable flexible cloud deployment with heterogeneous configurations. Many cloud-native database systems adopt such an architecture for both OLTP [22, 49, 62, 67] and OLAP [15–17, 24, 31, 60]. Nowadays, as storage services offer essential functions such as fault tolerance, scalability, and security at low-cost, systems start to layer their designs on the existing disaggregated storage services [23, 27].

This paper focuses on efficient deployment of the two-phase commit protocol on existing disaggregated storage services. Two-phase commit (2PC) is the most widely used atomic commit protocol, which ensures that distributed transactions commit in either all or none of the involved data partitions. 2PC was originally designed for the *shared-nothing* architecture and suffers from two major problems. The first is *long latency*: 2PC requires two round-trip network messages and associated logging operations. Previous work has demonstrated that the majority of a transaction's execution time can be attributed to 2PC [20, 21, 33, 42, 50, 52, 64]. The second problem is *blocking* [25, 26, 53]. Blocking occurs if a coordinator crashes before notifying participants of the final decision. These

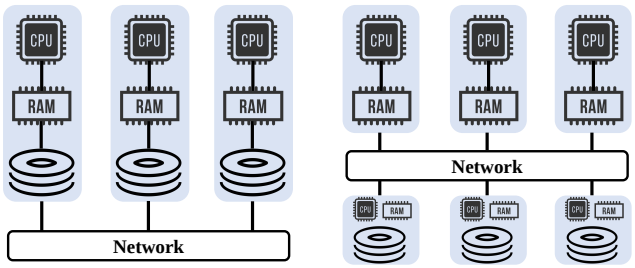


Figure 1: Shared-Nothing vs. Storage-Disaggregation.

two problems greatly limit the performance of 2PC, especially in a storage disaggregation architecture

Various techniques have been proposed to address these two problems with 2PC. Some proposed optimizations target the shared-nothing architecture and do not solve both problems simultaneously. These protocols either reduce latency by making strong assumptions about the workload and/or system that are not always practical for disaggregated storage [19–21, 26, 45, 46, 55, 56], or they mitigate the blocking problem by adding an extra phase and prolong latency [25, 41, 53]. Another line of research addresses both problems through customizing the storage. Examples include Paxos Commit [39], TAPIR [65], MDCC [44], and parallel commit in CockroachDB [57]. Existing solutions, however, are not applicable to general storage services because they require customized storage designs that perform conflict detection between transactions [6, 44, 57, 65] and/or need specific replication protocols [39, 44, 65]. Therefore, they cannot be readily applied to most existing storage services.

In this paper, we aim to maximize the flexibility brought by disaggregation without requiring customized APIs for the storage service. Therefore, a database can adopt existing highly optimized storage services and thereby avoid the expense of developing a new one, and can also allow the storage to adopt new mechanisms (e.g., new replication protocols) independently. We aim to answer the following research question: *What is the minimal requirement from the storage layer to enable 2PC optimizations addressing high latency and blocking?* Our answer is that the only requirement is the ability to provide *log-once* functionality, which ensures that for each transaction, only one update of its state in the log is allowed. We show that log-once semantics can be achieved with a simple *compare-and-swap-like API*, which is supported by almost every storage service today, including Redis [10], Microsoft Azure Storage [28], Amazon Dynamo [32], and Google BigTable [29].

(a) 2PC with no failure.

(b) 2PC with coordinator failure (cooperative termination protocol).

Figure 2: Illustration of Two-Phase Commit (2PC) — The lifecycle of a committing transaction (a) and a scenario of coordinator failure (b).

We introduce **Cornus**, an optimized 2PC protocol specifically designed for the storage disaggregation architecture in cloud-native databases. Cornus makes two major changes to conventional 2PC. First, it eliminates decision logging by the coordinator, which significantly reduces the latency of 2PC. A transaction is committed as soon as all participants have written `VOTE-YES` in their log in response to prepare requests in phase one. This optimization is feasible because highly-available disaggregated storage ensures that after log is written, it will not be lost. Second, Cornus uses a *LogOnce()* API based on compare-and-swap functionality to address the blocking problem. The API allows multiple participants to read and update the same log file and ensures that only the first log append for a transaction T can update T's state. This feature allows any participant that is uncertain about the transaction's outcome to force an abort by writing a vote to an unresponsive participant's log.

We study the deployment of Cornus over commercial cloud storage services (e.g., Azure Table, Azure Blob, Redis, Amazon S3, etc.) and discuss its implications for performance and privacy. We implement Cornus in an open-source distributed DBMS, Sundial [64], and deploy it on Azure Blob [28] and Redis [10].

In summary, this paper makes the following contributions:

- We revisit the design of 2PC in the context of a storage-dsiaggregation architecture.
- We develop Cornus, an optimized 2PC protocol for the storage-disaggregation architecture. It reduces transaction latency during normal processing and alleviates the blocking problem.
- We prove the correctness of Cornus by showing it satisfies all the required properties of an atomic commit protocol.
- We deploy Cornus on practical storage services, Redis [10] and Azure Blob [28], and show up to a 1.9× speedup in latency.

The rest of the paper is organized as follows: Section 2 provides background about two-phase commit and the architectural changes brought by storage disaggregation. Section 3 describes Cornus, including pseudocode, proofs, optimizations for read-only transactions, and analyses of failure cases. Section 4 discusses the deployment of Cornus on practical storage services and Section 5 evaluates it. Section 6 discusses related work and Section 7 is the conclusion.

2 BACKGROUND AND MOTIVATION

2.1 Two Phase Commit (2PC)

In a partitioned database, each partition has a corresponding process called a *resource manager* that handles reads and writes to the partition. 2PC is an atomic commit protocol that ensures a transaction involving multiple partitions either commits everywhere or aborts everywhere. 2PC contains a *prepare phase* and a *commit phase*, shown in Figure 2. The process initiating 2PC is called the *coordinator*. Resource managers that took part in the distributed transaction are called *participants*.

Figure 2a shows the logging events and network messages when all participating resource managers agree to commit the transaction and no failure occurs. If the coordinator fails before sending the decision to all participants, as shown in Figure 2b, some participants may not know the decision. A participant that voted yes and then times out waiting for the coordinator's decision will initiate a *termination protocol*. In a *naive termination protocol*, the participant needs to wait until the coordinator recovers. If the coordinator includes a list of participants in the prepare request, the uncertain participants can run a *cooperative termination protocol*. In this case, an uncertain participant contacts other participants to learn the decision. It repeats this process until at least one participant knows the decision as shown in Figure 2b. In 2PC, the coordinator's decision log record serves as the ground truth of the commit/abort decision — the final outcome of the transaction relies on the success of logging this record. The conventional 2PC has the following two limitations.

Limitation 1: Latency of Two Phases

In the standard 2PC protocol, the transaction caller experiences an average latency of *one network round-trip* and *two logging operations*, as shown in Figure 2a. Such a delay directly affects the transaction response time that an end-user will experience.

Limitation 2: Blocking Problem

In 2PC, a participant learns the decision of a transaction either directly from the coordinator or indirectly from other participants. In an unfortunate corner case shown in Figure 2b where the coordinator fails before sending any notification, no participant can make or learn the decision. While the decision is uncertain, any new transaction that conflicts with the current uncertain transaction cannot commit, since it depends on the outcome of the uncertain transaction. This is the well known blocking problem which causes data to be inaccessible

due to the failure of another resource manager not holding the data, limiting the performance and data availability of 2PC.

Many optimizations have been developed to improve the two limitations above in a shared-nothing architecture, including eliminating the prepare phase, at the cost of making extra system assumptions [19, 21, 46, 55, 56], or eliminating blocking, at the cost of a third phase [25, 41, 53]. In Section 6, we describe these protocols in more detail and explain their relationship with Cornus.

2.2 2PC in Storage-Disaggregation Architecture

Modern cloud-native DBMSs use a storage disaggregation architecture where storage and computation are separate services connected by the data center network (Figure 1b), in contrast to shared-nothing systems with direct-attached storage (Figure 1a). Many cloud-native OLTP databases adopt this architecture including Amazon Aurora [62], Apple FoundationDB [66], CockroachDB [6], Google Spanner [49], and Microsoft SQL Server [22]. The disaggregation architecture allows the computation and storage layers to scale and be developed independently. It offers better elasticity, flexibility, and resource utilization, and separates the concerns to achieve easier management and lower operational cost.

Naively deploying 2PC over a storage-disaggregation architecture increases the logging latency since the storage service is at the other side of the network. Previous research has tried to leverage the architectural features of storage disaggregation to optimize classic 2PC for better performance. Paxos Commit [39] was the first to develop a theoretical framework. In particular, the protocol contains four key ideas: (1) A transaction's fate is no longer decided by the decision logging of the coordinator, but by the votes logged by all participants including the coordinator. (2) Paxos acceptors can pre-prepare to save the first phase in Paxos. (3) Participants can directly propose the prepared message to replicas (i.e., acceptors), thereby skipping the Paxos leader to save one message delay. (4) The coordinator is also the Paxos leader of all Paxos instances, so the coordinator can learn the decision from acceptors directly saving two more message delays.

Paxos Commit and its followup protocols [6, 37, 44, 48, 63, 65] co-design 2PC with the underlying consensus protocol in the storage service. While they are non-blocking and achieve low latency, they require significant customization of the storage layer which cannot be readily deployed in existing storage services. In this paper, we aim to solve the long latency and blocking problems *without* customizing the storage service. We aim to develop a protocol that largely retains the performance advantage of Paxos Commit-like optimizations while being deployable in existing storage services.

3 CORNUS

Cornus is a non-blocking, low-latency 2PC variant that has minimal requirements on the disaggregated storage service. The only new storage-layer function needed by Cornus is the logging procedure *LogOnce()*, which can be implemented using compare-and-swap — a capability supported in many cloud-native storage services. This section presents the Cornus protocol in detail.

After presenting the big picture in Section 3.1, we describe the APIs and protocols in Sections 3.2 and 3.3 respectively. Section 3.4 describes how Cornus handles failures and recovery. Section 3.5

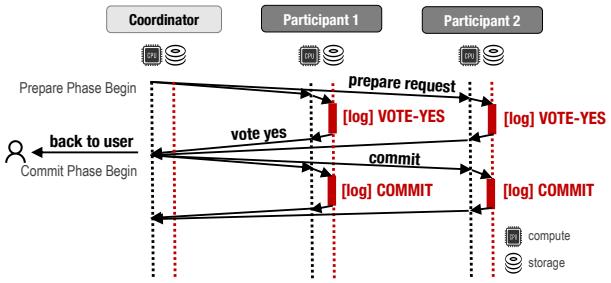


Figure 3: Illustration of Cornus.

proves the correctness of Cornus. Section 3.6 discusses read-only transactions and Section 3.7 discusses further optimizations.

3.1 Design Overview

Disaggregated storage enables the optimizations in Cornus as it has the following features that a conventional shared-nothing architecture does not provide:

Feature #1: A disaggregated storage service has *built-in data replication to support high availability*.

Feature #2: A disaggregated storage service can be *accessed by all the participants*.

Feature #3: A disaggregated storage service can support *limited computation tasks beyond basic reads and writes*.

Below we explain why these new features of storage disaggregation can enable optimizations that reduce commit latency and avoid blocking at the same time.

Latency reduction. In conventional 2PC, the final outcome of a transaction is determined by the coordinator's decision log. In Cornus, we follow the first idea in Paxos Commit (cf. Section 2.2) and change the criterion to be the *collective votes in all participants logs*; namely, a transaction commits if and only if all participants have logged `VOTE-YES`. Figure 3 shows the procedure of a committing transaction in Cornus. Compared to conventional 2PC as shown in Figure 2a, the coordinator no longer needs to log the decision, which eliminates this source of latency.

Features #1 and #2 are critical in achieving this latency reduction. They guarantee that once a vote is written to storage, it will not be lost and can be read by all resource managers. In Section 3.3, we show how Cornus ensures correctness by leveraging these features.

Non-blocking. Conventional 2PC can block when a coordinator fails and its decision log cannot be accessed. Cornus avoids blocking because the collective decision logs are written to the storage service (not the coordinator's local disk) and thus are highly available (Feature #1). If a participant is uncertain about the transaction's outcome, it can read the votes of all participants to learn that outcome. The uncertain participant can even write to a log on behalf of an unresponsive participant to enforce the final decision (Feature #2). Here we assume that each data partition has a single log written by all transactions touching this partition. A single-partition transaction writes to the log in the corresponding data partition; a distributed transaction writes to logs of all corresponding partitions that it accesses. These behaviors are identical to 2PC.

If multiple participants try to resolve the outcome of an unresponsive participant by writing the same log, they can create a data race. To avoid this, we leverage Feature #3 and introduce a new API in the storage service called *LogOnce()*, which guarantees that a transaction's state in the log is write-once — after the first update of the transaction's state, future updates to the state have no effect. *LogOnce()* is the only extra storage-layer API required by Cornus and is powerful enough to ensure correctness and avoid blocking. We will explain the *LogOnce()* API in more detail in Section 3.2 and its implementation using commercial storage services in Section 4. We note that Cornus may still block due to a failure of the underlying storage system. However, in any storage-disaggregation database, the entire system would fail if the underlying storage becomes unavailable, since the database is not able to access the raw table data anymore. Moreover, modern cloud storage services are typically highly available. For example, Azure Blob Storage has "11 nines" durability with locally redundant storage and "12 nines" with zone-redundant storage [5].

3.2 Cornus APIs

We first describe our RPC notation for remote procedure calls and then introduce the logging APIs used in Cornus.

Remote Procedure Calls (RPC)

We model communication between participants as RPCs. An RPC can be either *synchronous* or *asynchronous*. A synchronous call blocks until the callee returns. An asynchronous call allows the caller to continue executing until it explicitly waits for the response.

We represent an RPC using the following notation:

$$RPC^{n}_{sync/async}::FuncName()$$

where the subscript can be *sync* or *async* for synchronous and asynchronous RPCs respectively. The superscript *n* denotes the destination. *FuncName()* is the function to be called on the remote site and can take arbitrary arguments. In this paper, we consider the following two RPC functions:

Log(txn, type)

The *Log(txn, type)* function simply appends a log record of a certain *type* to the end of transaction *txn*'s log. It is used in both the conventional 2PC protocol and Cornus.

LogOnce(txn, type)

In Cornus, we introduce *LogOnce(txn, type)* to guarantee that a transaction's state can be written at most once. It atomically checks if a log record already exists for *txn*, and if not assigns the value in *type* to the record, namely `VOTE-YES`, `COMMIT`, or `ABORT`. It returns the state of *txn* after performing the atomic operation, which is either *type* or the existing state, depending on whether the operation updated the state. This function is used in Cornus but not in conventional 2PC.

3.3 Cornus Protocol

The pseudocode for Cornus is shown in Algorithm 1. We use a gray background to highlight the key changes in Cornus compared to standard 2PC with cooperative termination protocol introduced in Section 2.1, where a coordinator will send out each prepare request along with a list of coordinator's and participants' addresses. In the following, we explain the pseudocode for the coordinator, the participant, and the termination protocol.

Algorithm 1: API of Resource Managers in Cornus — Assuming a committing transaction. Differences between Cornus and 2PC are highlighted in gray.

```
1  Function Coordinator::StartCornus(txn)
2      for p in txn.participants do
3          send VOTE-REQ to p asynchronously
4      wait for all responses from participants
5          on receiving ABORT decision ← ABORT
6          on receiving all responses decision ← COMMIT
7          on timeout decision ← TerminationProtocol(txn)
8      reply decision to the txn caller
9      for p in txn.participants do
10         send decision to p asynchronously
11 Function Participant::StartCornus(txn)
12     wait for VOTE-REQ from coordinator
13         on timeout RPC_sync^local log::Log(ABORT) return
14     if participant votes yes for txn then
15         resp ← RPC_sync^local log::LogOnce(VOTE-YES)
16         if resp is ABORT then
               # Another participant has logged ABORT for it
17             reply ABORT to coordinator
18         else
19             reply VOTE-YES to coordinator
20             wait for decision from coordinator
21                 on timeout decision ← TerminationProtocol(txn)
22             RPC_sync^local log::Log(decision)
23     else
24         RPC_async^local log::Log(ABORT)
25         reply ABORT to coordinator
26 Function TerminationProtocol(txn)
27     for every paticipant p other than self do
28         RPC_async^p.log::LogOnce(ABORT)
29     wait for responses
30         on receiving ABORT decision ← ABORT
31         on receiving COMMIT decision ← COMMIT
32         on receiving all responses decision ← COMMIT
33         on timeout retry from the beginning
34     return decision
```

Coordinator::StartCornus(txn)

After a transaction *txn* finishes the execution phase, the coordinator calls *StartCornus(txn)* to start the atomic commit protocol. The coordinator sends out vote requests along with a list of all participants involved in the transaction to all participants (lines 2–3). Then the coordinator waits for responses from all participants (line 4). If it receives an `ABORT`, the transaction reaches an abort decision (line 5). If it receives `VOTE-YES` from all participants (i.e., none of them is `ABORT`), the transaction reaches a commit decision (line 6). If it times out, it invokes the termination protocol to finalize a decision (line 7). The latter is unlike 2PC, which unilaterally aborts the transaction without running the termination protocol.

Once the decision is reached, it can be returned to the transaction caller immediately without logging the decision. This is a key difference between Cornus and 2PC; the latter would reply to the caller only after the decision is written to stable storage. This optimization reduces the caller-observed latency by the duration of one logging operation. Finally, the coordinator broadcasts the decision to all participants asynchronously (lines 9–10).

Participant::StartCornus(txn)

The participant logic is very similar for Cornus and 2PC. A participant waits for a `VOTE-REQ` message from the coordinator (line 12). If it times out, it can unilaterally abort the transaction (line 13).

After receiving a `VOTE-REQ`, a participant votes `VOTE-YES` or `VOTE-NO` based on its local state of the transaction. For a `VOTE-NO`, it logs an `ABORT` record and replies to the coordinator (lines 24–25). This logging operation can be asynchronous following the *presumed abort* optimization in conventional 2PC [50].

For a `VOTE-YES`, the participant logs the record using *LogOnce()* (line 15). There are two possible outcomes. If the function returns `ABORT`, then another participant already aborted the transaction on behalf of this participant through the termination protocol. In this case, the participant aborts the transaction and returns `ABORT` to the coordinator (lines 16–17). Otherwise the function returns `VOTE-YES`, in which case the participant returns `VOTE-YES` to the coordinator (lines 19) and starts to wait for the coordinator's decision message (line 20). If it times out, it executes the termination protocol (lines 21). Otherwise, the decision received from the coordinator is written to the local log (line 22).

TerminationProtocol(txn)

In both 2PC and Cornus, a participant executes the termination protocol when it times out while waiting for a message and cannot unilaterally abort the transaction. In 2PC, the participant running the cooperative termination protocol contacts all the other participants for the outcome of the transaction. If any participant returns the outcome, the uncertainty is resolved. Otherwise, it will block until the failure is recovered.

Cornus avoids this problem. Instead of contacting the resource managers, the participant running the termination protocol tries to log an `ABORT` record in each participant's log using the function *LogOnce()* (lines 27–28). If the remote storage has not received any log record for the transaction yet, the `ABORT` record will be logged and returned (line 29–30). If the log already received a decision log record (i.e., `COMMIT` or `ABORT`) for this transaction, the function will return that decision (line 30–31). Another case is that the *LogOnce()* returns a `VOTE-YES` record. If the current participant receives this response from all the logs, it commits the transaction (line 32). If the current participant experiences a timeout again, it retries the termination protocol (line 33).

The only case where Cornus blocks is when it cannot reach the storage service, which we assume is highly unlikely since the storage service is designed to be highly available.

3.4 Failure and Recovery

This section discusses the behavior of Cornus when failures occur. For simplicity, we assume each site has one process and we discuss cases where only one site fails at a time. Table 1 and Table 2 discuss the system behavior of a coordinator and participant, respectively — when each one fails and when it recovers. The tables also describe the behavior of the failed node after it is recovered.

Coordinator Failure

Here we list the system behaviors if the coordinator fails at different points of the Cornus protocol.

Case 1: The coordinator fails before the protocol starts. In this case, a participant times out waiting for `VOTE-REQ` (line 13 in Algorithm 1). Therefore, all participants can unilaterally abort the transaction locally.

Case 2: The coordinator fails after sending some but not all vote requests. A participant that did not receive the request has the same behavior as Case 1; namely, it unilaterally aborts the transaction. A participant that received the request logs the vote, sends a response to the coordinator, and times out while waiting for the final decision because the coordinator failed (line 21 in Algorithm 1). Then the participant runs the termination protocol to check the votes of all participants. It either learns the abort decision or appends an `ABORT` to their logs, thereby aborting the transaction.

Case 3: The coordinator fails after sending all the vote requests but before sending any decision. In this case, all participants will time out waiting for the decision from the coordinator. They all run the termination protocol to learn the final outcome of the transaction and act accordingly.

Figure 4a illustrates such a case. The figure can be compared with Figure 2b showing how Cornus avoids blocking. After a participant's timeout, instead of contacting the coordinator, which has failed, it contacts all the logs in the shared storage using the *LogOnce()* function. Since all have `VOTE-YES` in their logs, each participant learns the decision of `COMMIT` and avoids blocking.

Case 4: The coordinator fails after sending out the decision to some but not all participants. For participants that have already received the decision, their local protocol terminates. The other participants time out waiting for the decision and run the termination protocol to learn the final decision.

Case 5: The coordinator fails after sending out the decision to all participants. In this case, all participants have completed the local protocol and thus have no effect.

Regardless of the cases, the coordinator has no actions during the recovery since it keeps no state and participants can terminate the transaction on their own.

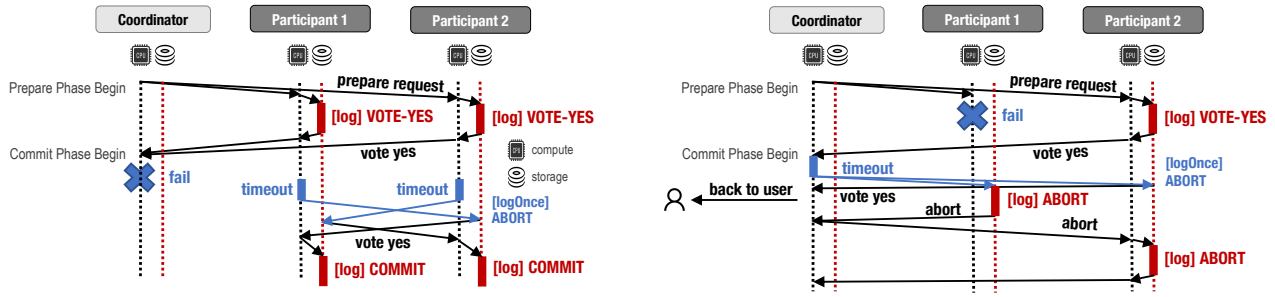
Participant Failure

We list the effects of failure of a participant at different points of the Cornus protocol.

Case 1: The participant fails before receiving the vote request from the coordinator. In this case, the coordinator times out waiting for all responses from participants and then runs the termination protocol (line 7 in Algorithm 2).

The coordinator logs an `ABORT` record for the failed participant, thereby aborting the transaction. The coordinator then broadcasts the decision to the remaining participants. If another participant also times out and initiates the termination protocol, the end effect is the same. After the failed participant recovers, it runs the termination protocol to learn the final decision.

Case 2: The participant fails after it receives the vote request but before logging its vote. The behavior of the system is the same



(a) Coordinator fails before sending decision

(b) Participant fails before logging vote

Figure 4: Cornus under Failures — The behavior of Cornus under two failures scenarios.

Time of Coordinator Failure	Effect of Failure	During Recovery
Before Cornus starts	Participants (if any) time out and unilaterally abort the transaction.	No action needed
After sending some vote requests	Participants that did not receive the request time out and abort unilaterally. Participants that receive the request time out waiting for the decision and execute the termination protocol, which aborts the transaction.	No action needed
After sending all vote requests but before sending any decision	All participants time out while waiting for the decision and execute the termination protocol to learn the outcome.	No action needed
After sending some decisions	Participants that did not receive the decision run termination protocol to learn the decision.	No action needed
After sending all decisions	No Effect	No action needed

Table 1: Effects of Coordinator Failures.

Time of Participant Failure	Effect of Failure	During Recovery
Before receiving the vote request	The coordinator will timeout, running the termination protocol to abort the transaction	Abort the transaction
After receiving the vote request, before logging vote	Same as above	Same as above
After logging vote, before replying to coordinator	The coordinator will run the termination protocol to see the vote and learn the final outcome.	Abort the transaction if local vote is abort; otherwise run termination protocol to learn the outcome.
After replying vote to the coordinator	No effect	If decision log exists, follow the decision. Otherwise, same as above.

Table 2: Effect of a Participant Failures.

as in Case 1 because, from the other participants' perspectives, the behavior of the failed participant is identical.

Figure 4b shows an example of this case. The coordinator receives a `VOTE-YES` from participant 2 and times out while waiting for a response from participant 1. At this point, the coordinator runs the termination protocol to try to log `ABORT` on behalf of the participants via *LogOnce()* . After the coordinator logs `ABORT` for participant 1 and learns that participant 2 logged `VOTE-YES`, it logs its abort decision and sends it to participant 2. The example assumes only the coordinator has a timeout. Participant 2 might also experience a timeout, which will lead to the same final outcome.

Case 3: The participant fails after it logs the vote, but before replying to the coordinator. In this case, the coordinator times out waiting for votes and runs the termination protocol. Then it can see all the participants' votes from the storage and learns the outcome. The remaining participants learn the decision either from the coordinator or by running the termination protocol themselves. After the failed participant recovers, it aborts the transaction if it found its local vote is an abort; otherwise, it runs the termination protocol to learn the outcome.

Case 4: The participant fails after sending out the vote. This failure does not affect the others — the coordinator and remaining participants execute the rest of the protocol normally. After the failed participant recovers, it either finds that it already logged a decision or else runs the termination protocol to learn the outcome.

3.5 Proof of Correctness

This section formally proves the correctness of Cornus. Our proof follows the structure used in [26] where an atomic commit protocol is proved as five separate properties (AC1–5, see below). We start by defining a *global decision* of a distributed transaction.

Definition 1 [Global Decision]: A transaction reaches a `COMMIT` global decision if all participants have logged `VOTE-YES`; it reaches an `ABORT` global decision if any participant has logged `ABORT`. Otherwise the decision is undetermined.

We first introduce and prove the following lemma.

Lemma 1 [Irreversible Global Decision]: Once a global decision is reached for a transaction, the decision will not change.
Proof: There are two cases to consider. In case 1, an abort global decision has been reached meaning one log contains an ABORT record. According to the semantics of *LogOnce()* , no VOTE−YES can be appended to that log anymore, meaning that the global decision cannot switch to commit. In case 2, a commit global decision is reached and all participants have VOTE−YES in their logs. The only way to append an ABORT record is through the termination protocol. But the semantics of *LogOnce()* implies it will not append ABORT because VOTE−YES already exists. □

We now prove the five properties in order.

Theorem 1 [AC1]: The decision of each participant is identical to the global decision.
Proof: A participant reaches a decision through either learning the decision from the coordinator or directly seeing or enforcing the vote in each log through the termination protocol. In both cases, a local commit decision is reached only when all logs contain VOTE−YES and an abort local decision is reached only when at least one log contains ABORT (which may be written through the termination protocol), so that the local decision is identical to the global decision.

Theorem 2 [AC2]: A participant cannot reverse its decision after it has reached one.
Proof: Due to Lemma 1, once a global decision is reached, it cannot reverse. Due to Theorem 1, each participant will reach the same decision as the global decision, finishing the proof. □

The following two properties must hold for correctness of 2PC [26].
AC3: The *commit* decision can only be reached if all participants voted Yes.
AC4: If there are no failures and all participants voted Yes, then the decision will be to *commit*.

For the proof of Cornus, we combine the two properties into the following theorem.

Theorem 3 [AC3&4]: The decision of a transaction is a *commit* if and only if all participants vote Yes and write VOTE−YES to their corresponding logs.
Proof: By Definition 1, a transaction's *global decision* is a *commit* if and only if all participants write VOTE−YES to their logs. By Theorem 1, the decision of each participant is identical to the global decision, finishing the proof. □

Finally, 2PC requires the following property.
AC5: Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

For Cornus, we prove the following theorem which achieves a stronger property.

Theorem 4 [AC5]: If the storage layer is fault tolerant, then with any failures in the compute layer, the remaining participants will reach a decision in bounded time without requiring the failed sites to be recovered.
Proof: Since the storage layer is fault tolerant, once a global decision is reached, an active participant can always learn the global decision through the termination protocol. The only case where a decision cannot be reached is when one participant fails to log its vote. In this case, a coordinator or participant that experiences a timeout will run the termination protocol and directly write an ABORT into the pending participant's log, which enforces a global decision. □

3.6 Read-Only Transactions

Conventional 2PC can be optimized for read-only participants [50]. Specifically, if a transaction issues only read requests to one participant, this participant does not need to log anything during the prepare phase and can simply release locks. For simplicity, we will call such participant a read-only participant. In Cornus, this optimization has a small subtlety that we will explain in the following two cases.

The simple case is that the entire transaction is read-only and the coordinator knows this fact before starting 2PC. In this case, similar to 2PC, all participants can skip logging in the prepare phase in Cornus. In practice, we believe it is possible to learn read-only transactions before starting 2PC in many scenarios.

The second case is that the transaction is read-only but does not know the fact before starting 2PC, or only a subset of partitions are read-only. In this case, all partitions including the read-only ones must log VOTE−YES in Cornus (in contrast to 2PC which skips logging for read-only partitions). Such a log is necessary because otherwise other participants will interpret the absence of a VOTE−YES in a read-only participant's log as an abort (e.g., consider a read-only transaction that times out waiting for a VOTE−REQ and aborts). Although writing such log for read-only participants is additional overhead, it can happen in parallel with log writes of read-write participants. Therefore, it does not increase the number of log writes on the critical path, though it might affect tail latency if a read-only participant is slow. By contrast, in 2PC, the coordinator has an extra log write on the critical path, to log the commit decision. Therefore, Cornus still has lower latency compared to conventional 2PC.

3.7 Further Optimization Opportunities

With Cornus, we leverage functionality that is *already supported in existing storage systems* to optimize 2PC, namely, *LogOnce()* through compare-and-swap. In this subsection, we discuss other optimization opportunities (e.g., those discussed in the Paxos Commit framework) if further functionality is supported in the storage.

Optimization #1. Upon receiving a request, an existing storage service would send the response only to the requesting participant. By letting the storage service respond to multiple participants, we can further reduce the latency of the protocol. Specifically, after a participant's vote is logged in the storage layer, the response can be sent to both the requesting participant and the coordinator. As a result, the coordinator can learn the votes directly from storage instead of requiring another message hop from the participant. This saves one message delay in the critical path, without introducing any extra messages.

Optimization #2. The optimization above can be extended by having the storage broadcast its vote to all participants of a transaction. Then each participant can learn the final decision directly from distributed votes, rather than waiting for the coordinator to send a decision message after it receives all votes. This further reduces

the overall latency but does not impact the user-observed latency. Unlike optimization #1 above, this optimization incurs extra network messages due to broadcasting.

These two optimizations can be implemented as extensions to existing storage services, without changing the underlying replication or consensus protocol. If the internals of the storage layer are exposed to the compute layer, further optimizations can be enabled, as in Paxos Commit as shown in Section 5.6. However, this exposure is contrary to our goal of having 2PC optimizations work with any storage service as long as they support the needed APIs.

4 DEPLOYMENT

In this section, we discuss the deployment of Cornus in practical cloud storage systems. In particular, this requires implementing *Log()* and *LogOnce()* introduced in Section 3.2, using existing cloud storage services. This entails two requirements.

One requirement is to guarantee that after the decision is made it will not be altered, even if multiple participants concurrently execute the termination protocol. We can leverage existing compare-and-swap-like APIs in cloud storage services to support this feature. The other requirement is access control. In 2PC, logging in the prepare phase persists both transaction states and the actual data. In Cornus, a participant may need to access other participants' transaction states. This requires separate access control for transactions' states and data so that a participant cannot read others' log of actual data while accessing the transaction states.

In this paper, we studied a wide range of modern in-cloud storage services on how these services support CAS and data privacy, and we deploy Cornus on two of them — Redis and Azure Blob Storage (a.k.a. Windows Azure Storage for Blob) for quantitative evaluations in next section.

4.1 Deployment on Redis

Redis [10] is an in-memory data structure store that supports optional durability and can be used as a distributed, in-memory key-value database, cache, and message broker.

Compare-and-swap. Redis supports compare-and-swap operations through the "EVAL" command and Lua scripting [12]. It guarantees that Lua scripts run by "EVAL" commands are executed atomically. That is, the effect of a script is either all or none with respect to scripts and commands of other Redis clients. Listing 1 shows an implementation of *LogOnce()* using cpp_redis [7], an asynchronous multi-platform lightweight Redis client for C++.

Listing 1: Implementation of Synchronous *LogOnce()* on Redis

```
auto lua_script = R"(
        redis.call('set', KEYS[1], ARGV[1])
        redis.call('setnx', KEYS[2], ARGV[2])
        local state = tonumber(redis.call('get', KEYS[2]))
        return {tonumber(state)};
)";
string id = to_string(participant_id) + "-" + to_string(txn_id);
vector<string> keys = {"data-" + id, "state" + id};
vector<string> args = {data, to_string(VOTE_YES)};
client.eval(lua_script, keys, args, [](reply& reply) {
        // a callback function to execute when reply is ready
        uint64_t txn_state = response.as_array()[0].as_integer();
});
client.sync_commit();
```

Access Control. Redis Access Control List (ACL) [11] manages users' access to certain commands and/or keys patterns. We can configure each participant *n*'s access so that *n* has read-write access to transaction states of every participant but to the data log of only its own using the "ACL SETUSER" command. We do this using the "ACL SETUSER" command to set up read-write access to all keys starting with the pattern "data-<current participant id>" and "state-" if using the implementation in Listing 1.

4.2 Deployment on Microsoft Azure Blob Storage

Microsoft Azure Blob Storage is a scalable storage system that supports secure object storage for cloud-native workloads, archives, data lakes, high-performance computing, and machine learning.

Compare-and-swap. Azure Storage assigns an identifier to each stored object. The identifier is updated every time the object is updated. An HTTP GET request returns the identifier as part of the response using the Etag (entity tag) header defined within HTTP. To achieve the compare-and-swap functionality, a user updating an object can send in the original Etag with an "If-Match" conditional header so that the update will be performed only if the stored ETag matches the one passed in the request [13].

Listing 2: Implementation of Synchronous *LogOnce()* on Azure Blob with Separate ACLs

```
string id = to_string(participant_id) + "-" + to_string(txn_id);
if (data) {
    // request issued during normal execution (non-failure)
    storage::cloud_block_blob data_blob =
        container.get_block_blob_reference(U("data-" + id));
    auto t = data_blob.upload_text(U(data));
}
storage::cloud_block_blob state_blob =
    container.get_block_blob_reference(U("state-" + id));
try {
    storage::access_condition condition =
        storage::access_condition::generate_if_not_exists_condition();
    storage::blob_request_options options;
    storage::operation_context context;
    state_blob.upload_text(U(state), condition, options, context);
} catch (const std::exception &e) {
    // update failed, log already exist
    State state = (State) stoi(blob_status.download_text());
}
```

Access Control. Azure Blob Storage supports Azure attribute-based access control (Azure ABAC). It allows read access to blobs based on tags and custom security attributes. As there is no transaction support, we use two separate requests to log the transaction state after successfully logging the actual data as shown in Listing 2. In this case, Cornus shows no improvements over 2PC as will be shown in Figure 5e. Many applications do not require the data to be private to the corresponding partitions, in which case Cornus does not introduce this extra overhead.

4.3 Deployment on Key-Value Databases

Amazon DynamoDB [32] is a highly available key-value storage system with rich APIs [2]. The "putItem" and "updateItem" APIs provide conditional puts and updates respectively to achieve CAS functionality. The "TransactWriteItems" API supports submitting multiple actions in one request including the "putItem" and "updateItem" mentioned above. As DynamoDB offers item-level access control [1], transaction data and transaction states can be kept as

separate attributes in a table or in different tables, and the actions on both logs can be submitted in a single "TransactWriteItems" request.

Google Cloud Bigtable [29] is a Distributed Storage System for Structured Data. It supports conditional writes [8] so Cornus can implement *LogOnce()* on top. Google Cloud uses Identity Access Management (IAM) for access control. It is a role-based access control mechanism. For Bigtable, it can control accesses at the table level so that transaction states and log data can be kept in separate tables. Each participant can have read/write permission to its own log of transaction data stored in one table, and all transaction states stored in another table. However, Bigtable does not support writing to multiple tables in a batch, which, like Azure Storage, causes extra overhead when using it for Cornus.

5 EXPERIMENTAL EVALUATION

We evaluate the performance of Cornus compared with standard 2PC. We introduce the experimental setup in Section 5.1. We evaluate Cornus in a range of settings to demonstrate its efficacy in reducing latency, primarily in the commit phase.

5.1 Experimental Setup

5.1.1 Architecture. We concentrate on systems comprised of multiple nodes in the compute layer that access a disaggregated storage service. The database data is partitioned, where each compute node runs a resource manager and has exclusive access to one partition. While a compute node executes transactions, it sends data access requests to the corresponding compute node. At commit time, the compute node coordinates the resource managers participated in the transaction to commit. Every compute node can write log records to the storage service, both its own log and those of other compute nodes.

We implemented Cornus on Sundial [64], an open-source distributed DBMS testbed. Our implementation is public[1]. Compute nodes communicate using gRPC [14], which can be either synchronous or asynchronous. Each node has a *gRPC* client for sending requests and a *gRPC* server that executes requests using a pool of server threads that it manages.

5.1.2 Compute Node Hardware and Storage Service. For compute nodes, we use a cluster of up to eight servers running Ubuntu 18.04 on Microsoft Azure. Each server has one Intel(R) Xeon(R) Platinum 8272CL CPUs (8 cores *times* 2 HT) and 64 GB of DRAM. The servers are connected by a 12.5 Gbps Gigabit Ethernet. Based on our measurements, a network round trip is 0.5 ms between two compute nodes.

Cornus can use any cloud storage service. We use the two below in three different configuration.

- **Microsoft Azure Blob Storage (Azure Blob)** [3]: We use a StorageV2 (general purpose v2) Azure storage account to store blobs, with geo-redundant storage (GRS) replication enabled. Data is stored in two regions. In the primary region, data is synchronously three-way replicated in one physical location. Then the data is asynchronously copied to a secondary region hundreds of miles from the primary. The average time for a conditional write request is 10.40 ms and for a plain write request is 10.29 ms.
- **Microsoft Azure Cache for Redis (Redis)** [4]: This experiment uses the Redis service provided by Microsoft Azure. We created a Premium P4 Redis instance of version 4.0.14. It uses master-slave replication with one master and one slave in the same region. Only the master node accepts reads and writes, which it applies to the slave node asynchronously. The average time for a conditional write request is 1.96 ms and for a plain write request is 1.84 ms.

5.1.3 Workloads. We use the Yahoo! Cloud Serving Benchmark (YCSB) [30] for performance evaluation. YCSB is a synthetic benchmark modeled on cloud services. It contains a single table partitioned across servers in a round-robin fashion. Each partition contains 10 GB data with 1 KB tuples. Each transaction accesses 16 tuples as a mixture of reads (50%) and writes (50%). The queries access tuples following a Zipfian power law distribution controlled by a parameter θ. By default, we use $\theta = 0$, which means data access is uniformly distributed. All transactions are executed as stored procedures that contain program logic intermixed with queries.

5.1.4 Implementation Details and Parameter Setup. Unless otherwise specified, we use the following parameter settings: We evaluate the system on up to eight compute nodes and a storage service. Eight worker threads per node execute the transaction logic, and eight worker threads per node execute the remote requests. The default concurrency control algorithm is NO-WAIT.

For each data point, we run five trials with 30 seconds per trial. We collect the latencies for distributed transactions — transactions involving more than one partition (node) — and take the result from the trial with median average latency. Running the experiments for a longer time does not change the conclusions.

For read-only transactions, we assume the coordinator of a transaction can learn that it is read-only at the end of the execution phase. Therefore, Cornus and 2PC can skip both the prepare and commit phases for read-only transactions [50].

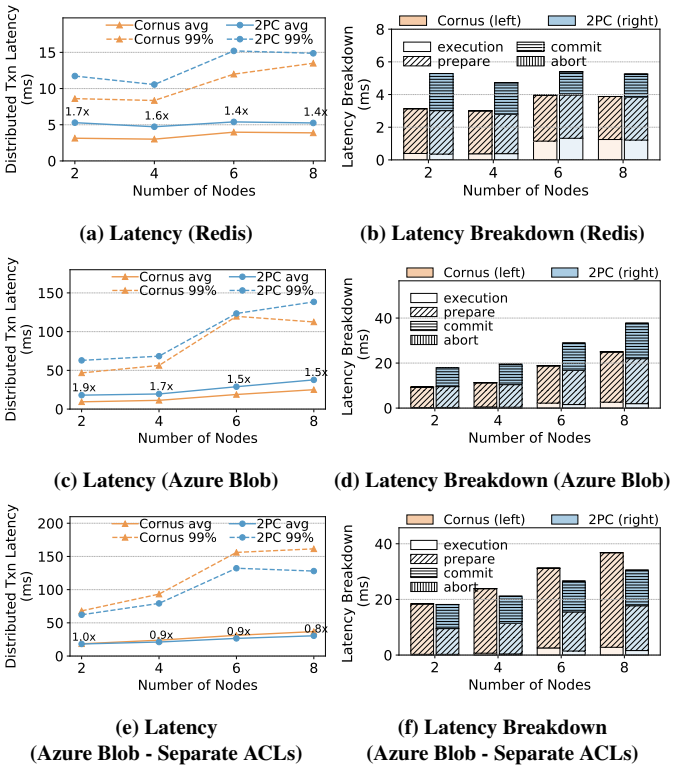
For the implementation of *LogOnce()* , we follow Listing 1 for Redis. As Azure Blob does not support batch update of two resources with separate access control, we implemented two versions. In the default version, we used the same access control for transaction data and transaction states for all experiments in this section. The second version with Azure Blob uses separate ACLs. Profiling shows that the second version increases the time for *LogOnce()* from an average of 10.40 ms to an average of 18.43 ms.

5.2 Scalability

We first evaluate Cornus's scalability on YCSB as the number of compute nodes increases from 2 to 8. We set the parameter to default values described in Section 5.1.3. The results in Figure 5(a–d) show that both Cornus and 2PC scale well in YCSB with Redis or Azure Blob. As the number of nodes increases, the latency of both 2PC and Cornus increases linearly. The speedup of Cornus over 2PC on average latency slightly decreases as the number of nodes increases. This is due to the time spent on RPC calls in execution phase increases. Overall, the latency speedup is up to 1.9×.

Figure 5e and Figure 5f show the evaluation of an implementation on Azure Blob with separate access control for transaction data and

[1] https://gitfront.io/r/user-5630758/1e4cb8b6f88da5b05fe70c117873df37f4831b41/Sundial-Private/



(a) Latency (Redis)

(b) Latency Breakdown (Redis)

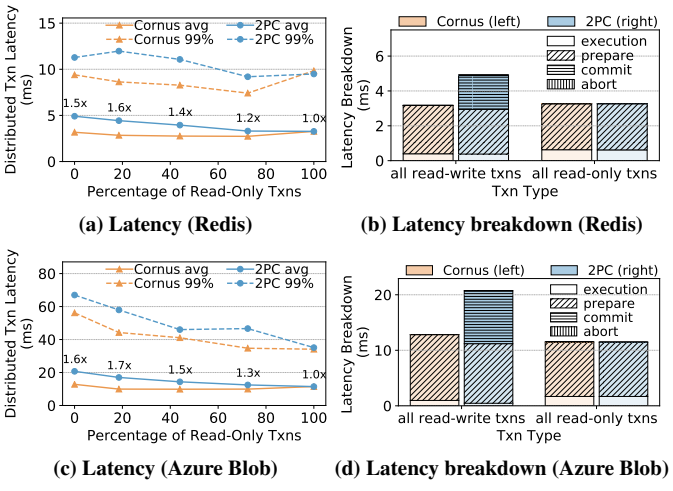
(c) Latency (Azure Blob)

(d) Latency Breakdown (Azure Blob)

(e) Latency (Azure Blob - Separate ACLs)

(f) Latency Breakdown (Azure Blob - Separate ACLs)

Figure 5: Scalability



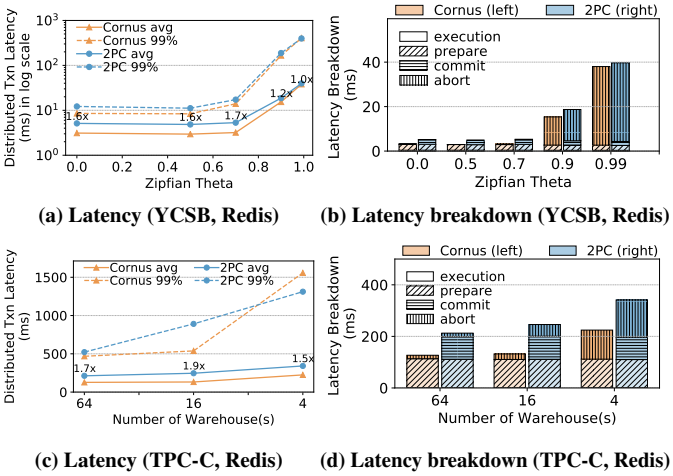
(a) Latency (Redis)

(b) Latency breakdown (Redis)

(c) Latency (Azure Blob)

(d) Latency breakdown (Azure Blob)

Figure 6: Varying percentage of read-only transactions



(a) Latency (YCSB, Redis)

(b) Latency breakdown (YCSB, Redis)

(c) Latency (TPC-C, Redis)

(d) Latency breakdown (TPC-C, Redis)

Figure 7: Varying workload contention

transaction states as introduced in Section 5.1.4. In 2PC, data and states are stored in a resource within the same access control group as both of them will not be accessed by other partitions. However, in Cornus, the data and states are stored in separate access control groups so that two remote logging requests instead of one must be used for *LogOnce()* . Thus, Cornus spent ~9.48 ms more time in the prepare phase for logging than 2PC (Figure 5f) and shows no improvement. We conclude that the current version of Azure Blob cannot benefit from Cornus for applications that want separate access control between data and transaction states.

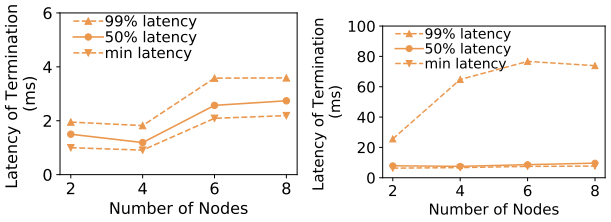
5.3 Percentage of Read-only Transactions

We evaluate the performance of Cornus under YCSB with different fractions of read-only transactions. We manage the percentage of read-only transactions by controlling the probability of each request being read in a transaction. With n requests per transaction and each request has a probability of p being read, the percentage of read-only transactions is expected to be n^p We expect Cornus to obtain a latency speedup solely for read-write transactions because both Cornus and 2PC omit the prepare and commit phases for read-only transactions, as discussed in Section 3.6 and Section 5.1.4.

The results shown in Figure 6a and Figure 6c match the expectation. The improvement of Cornus (relative to 2PC) grows as the percentage of read-only transactions decreases. When there are more read-write transactions, Cornus improves both average and P99 latency over 2PC by up to 1.7×. The pattern is consistent across different storage services. The result can be explained by the latency breakdown illustrated in Figure 6b and Figure 6d . Cornus improves latency for read-write transactions by eliminating the commit phase, which takes a significant amount of time especially in Azure Blob with geo-distribution and synchronous replication . Finally, we note that Cornus spends slightly more time in the prepare phase than 2PC due to the subtle differences between *Log()* and *LogOnce()* (Algorithm 1 Line 16).

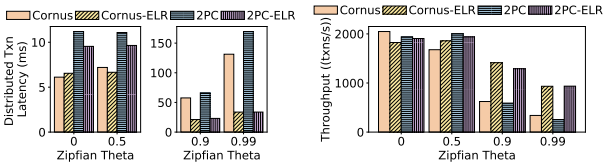
5.4 Contention

This section evaluates the performance of Cornus when serving workloads with varying contention. We use YCSB and TPC-C workloads. For YCSB, we adjust the Zipfian distribution of data accesses through θ. Increasing θ increases the level of contention. For TPC-C, we vary the number of warehouses; fewer warehouses indicate higher contention in the workloads. Figure 7 shows the results of both workloads; the x-axis from left to right indicates low to high contention for both YCSB and TPC-C. As shown in the figure, Cornus always improves transaction latency over 2PC



(a) Latency (Redis) (b) Latency (Azure Blob)

Figure 8: Time to terminate transactions on failure



(a) Latency (Redis) (b) Throughput (Redis)

Figure 9: Cornus and 2PC with speculative precommit

— the improvement is up to 1.8 × for YCSB, and 1.9 × for TPC-C workloads. Figure 7b and Figure 7d show that Cornus provides less improvement under high contention since the abort time dominates the total transaction elapsed time.

5.5 Time to Terminate Transactions on Failure

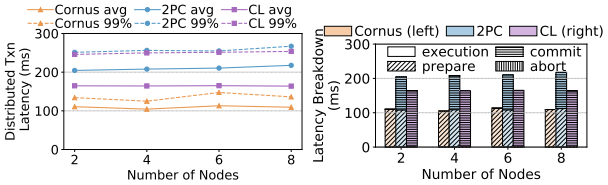
Figure 8 shows the time to run the termination protocol in Cornus once it is triggered. In 2PC, there is no bound on the time to run the termination protocol — a transaction that falls into uncertain states (due to coordinator failure before the decision is sent out to any participants) is blocked indefinitely, until the coordinator recovers. However, in Cornus, compute node failure does not lead to blocking. In this experiment, we assume the storage is still available and measure the time to terminate a transaction through the termination protocol while varying the number of nodes. As shown, for up to 8 nodes, Cornus always terminates a transaction within 4 ms with Redis and within 20 ms on average with Azure Blob. The tail latency of Azure Blob increases more than Redis as the number of nodes increases. This is due to the geo-distribution setup and some synchronous replication in Azure, while the two replicas of Redis are co-located in the same region and only perform asynchronous replication, as introduced in Section 5.1.2.

5.6 2PC Optimizations

In this section, we evaluate common 2PC optimizations and compare them with Cornus. These optimizations typically make different system/workload assumptions from the classic 2PC; therefore we conduct the comparisons here instead of in the main results.

Speculative Precommit

The first optimization is to speculatively presume commit in the prepare phase. This optimization makes the assumption that a transaction entering the prepare phase is unlikely to abort due to system crashes. Therefore, a transaction can allow others to read its



(a) Latency (Redis) (b) Latency (Redis)

Figure 10: Coordinator log

pre-committed data while waiting for the log to be persistent. Many previous papers [9, 34, 38, 43, 54] have studied this optimization.

This 2PC optimization is equally applicable to Cornus. We implement it in both Cornus (i.e., Cornus-ELR) and 2PC (i.e., 2PC-ELR) following a state-of-the-art protocol for distributed systems [40]. Our implementation is based on pessimistic concurrency control, and thus we refer it as Early Lock Release (ELR) in this paper. Figure 9 compares Cornus and 2PC with and without the speculative precommit in the YCSB workload. In this experiment, we use Azure Redis as the remote storage and vary the contention levels on the x-axis. As shown, ELR can significantly improve both 2PC and Cornus, particularly when workload contention is high. Specifically, Cornus and 2PC can improve by 175% and 267% with zipfian theta of 0.99 in throughput. With the optimization, the difference between Cornus and 2PC in throughput and latency decreases under high contention since contention becomes the dominating factor as shown in Figure 7b. In short, for systems that rarely crash during the prepare phase, this technique can be applied to Cornus to further improve the performance under high contention.

Coordinator Log

Another common 2PC optimization is to let the coordinator log on behalf of all participating nodes so that the transaction does not wait for other nodes to persist the logs. In our implementation, we ask the coordinator to log for all partitions during the prepare phase. At the same time, we send prepare requests to participants, which reply with their votes without logging. Upon receiving the votes, the coordinator makes decision and appends it to its log.

We implement this technique following the ideas in previous works [55, 56] and compare it with 2PC and Cornus quantitatively. In this experiment, we use Azure Redis as the storage service, where the latency of writing to storage is around 443ms. In Figure 10, we see that having the coordinator handle all logging (i.e., CL for Coordinator Log) outperforms the baseline 2PC by 33% in terms of average latency since it batches all the logs into one write request to storage. The improvement is significant especially with such high latency of writing to Redis. However, CL still underperforms Cornus by 50% since the coordinator needs to wait until both the data and the decision are logged by coordinator, while Cornus directly replies to the user without waiting for the commit decision to be logged.

Note that the coordinator log optimization has several limitations compared to 2PC or Cornus. First, it increases the size of acknowledgement messages. Second, it increases the complexity of recovery and raises security concerns. Specifically, it violates site autonomy which requires internal information concerning the local execution of transactions such as log records to remain private to a site and not

be exported [19, 20]. Although some works including Cornus allow other sites to access the log of transaction states, other sites will not access actual user data in Cornus (the second requirement discussed in Section 4).

Integration with Replication Protocol

While Cornus directly runs on top of a highly-available storage service through a consensus-agnostic interface, many prior works [39, 44, 57, 65] manage the replication on their own and co-design 2PC with replication protocols for further optimizations. Although these protocols cannot be directly applied to existing storage services, we still evaluate them to show the potential optimization space when having different assumptions.

We first performed theoretical evaluation on the expected latency for protocols combining with Paxos. Table 3 shows the number of Round Trip Times (RTTs) each protocol has on the critical path — from the time the coordinator starts the protocol until the decision can be returned to the user. We use A + B = C to show (A) the number of RTTs in the prepare phase, (B) the number of RTTs in the commit phase, and (C) the sum. The table also shows the requirements.

For 2PC and Cornus, we assume each process (coordinator/participant) runs an instance of Multi-Paxos in the underlying storage. When a participant sends a log request to the storage, it sends it to the leader of a Paxos instance, which refers to the proposer that has already run phase 1 to become a stable leader. The leader then initiates the second round of Paxos using one RTT and then gets back to the participant. Cornus can eliminate the coordinator logging from the critical path which corresponds to 2 RTTs (one for a participant communicating with the Paxos leader and one for the second round of Paxos).

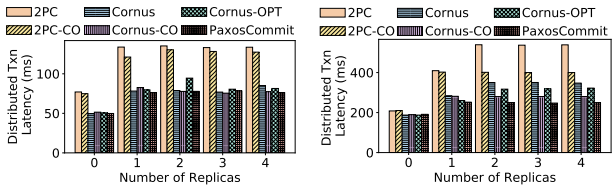
Cornus (optimization) refers to Cornus with optimization 1 discussed in Section 3.6. It can save the latency of logging acknowledgment from Paxos leader to participant (0.5 RTT) by forwarding the message to the coordinator. This optimization requires the storage to be able to send the message to an extra recipient but the replication details are still completely handled by the storage.

Cornus (co-location) and 2PC (co-location) represents the designs that co-locates a participant with the Paxos leader, i.e., the participant initiates the second round of Paxos by talking to all the replicas directly instead of asking the leader to initiate the process. Compared with naive Cornus and 2PC, this optimization can save the 1 RTT of Paxos leader communicating with other acceptors. However, this assumes that participants manage the replication process explicitly.

Paxos Commit [39] and MDCC [44] are two related works introduced in Section 2 and Section 6. MDCC-Classic refers to a protocol in the paper following the framework of Paxos Commit. These protocols apply all the optimizations discussed above. Specifically, during the prepare phase, a participant directly talks to all acceptors to log and all acceptors forward the logging acknowledgment to the coordinator. The coordinator then learns all the votes from the quorum. It can save 1 RTT for inter-replica communication and 0.5 RTT for logging acknowledgment sent from Paxos to coordinator compared with Cornus. However, it also has all the corresponding requirements including participant/coordinator coordinating replication and acceptors forwarding logging acknowledgement to the coordinator.

Protocol	# RTT	Extra Requirements
2PC	3 + 2 = 5	-
Cornus	3 + 0 = 3	Storage supports conditional write
Cornus (optmization)	2.5 + 0 = 2.5	Leader of Paxos can forward a message to coordinator
2PC (co-location)	2 + 1 = 3	Participant coordinates replication
Cornus (co-location)	2 + 0 = 2	Participant coordinates replication
Paxos Commit / MDCC-Classic	1.5 + 0 = 1.5	Participant coordinates replication; Acceptors forward messages to coordinator to learn from quorum

Table 3: Time complexity for protocols integrating with Paxos or its variations



(a) same region (RTT = 0.46 ms) **(b) across region (RTT = 68 ms)**

Figure 11: Cornus and 2PC with customized storage

Besides the theoretical analysis, we also perform quantitative evaluation of these protocols with a self-implemented disaggregated storage and vary the number of replicas in the underlying storage. We run the experiment in two situations — one with all replicas in the same region (Figure 11a) and one with replicas distributed across regions from US. East to US. West (Figure 11b). Overall, the experiment results confirm the theoretical computation in Table 3 — the relative performance maps to the expected differences in expected count of RTTs and the absolute improvement increases as RTT increases. The performance difference also increases with more replicas. This set of experiments demonstrates how much performance may be sacrificed when treating the storage as a black box with a consensus-agnostic abstraction, and the potential optimization space with co-designing 2PC with consensus to different degrees.

6 RELATED WORK

This section describes related works on optimizing 2PC. We categorize prior works into three categories: techniques reducing latency, techniques addressing blocking, and codesign of 2PC and replication to address both problems.

6.1 Techniques Reducing Latency

Centralized Logging. Centralized logging asks the coordinator to log for all participants on behalf of them to reduce latency. A line of previous work follows this idea including *coordinator log* [55, 56], *implicit yes vote* [21], and Lee and Yeom's protocol [46]. The main idea is to have each participant send its log with its acknowledgment of its prepare request to the coordinator. The coordinator force writes

those acknowledgments to its log along with its commit decision so that it eliminates the logging latency in prepare phase. However, as discussed in Section 5.6, these designs increase the size of acknowledgment, increase the complexity of recovery, and raise security concerns as they violate site autonomy.

Early Prepare During Execution. Another technique to reduce 2PC latency is to let participants prepare during execution so that a transaction can skip the prepare phase at commit time. Many previous works incorporate this idea to reduce the commit latency [21, 46, 55, 56]. For example, in *Early Prepare* (EP) [55], each participant forces a prepare record before replying to the coordinator on receiving a work request during execution. It also requires the coordinator to record the identity of the participant in the log before sending the work request. However, if the transaction accesses more than one partition or the work requests for each partition cannot be batched into one request, EP needs to pay more logging in the critical path than 2PC.

To address this issue, subsequent works try to combine this technique with centralized logging or decentralized decision. For example, implicit yes vote [21] applies both centralized log and early prepare. It asks participant to piggyback its log in the acknowledgement of every work request and treats the acknowledgement as an implicit yes vote. However, it still suffers from the limitations of centralized log discussed above and additionally restrict the design of concurrency control. For example, protocols like optimistic concurrency control and two-phase locking with wound-wait [26] need to be carefully redesigned since a participant may abort after sending the acknowledgement at the end of execution phase [20].

Speculative Pre-Commit. If a transaction that has entered the prepare phase is unlikely to abort due to system crashes, the database can speculatively presume commit in the prepare phase. Specifically, a transaction can let others read its pre-committed data while waiting for the log to be persistent. Many previous works [9, 34, 38, 43, 54] have studied this optimization known as early lock release [43] and controlled lock violation [38]. In Section 5.6, we have shown this optimization can be applied to both 2PC and Cornus leading to similar throughput improvement and latency reduction.

6.2 Techniques Addressing Blocking

Extra Network Roundtrip. Skeen [53] gave necessary and sufficient conditions for a correct non-blocking commit protocol, known as *the fundamental nonblocking theorem*. It proved that a fifth state in addition to initial, wait, abort, and commit can be added to avoid blocking with the same assumptions made in 2PC. Adding this state requires one more network round trip, resulting in a three-phase commit (3PC) protocol. Although it solves the blocking issue, 3PC magnifies the problem of latency delay in 2PC.

Extra Message Count. Some protocols decrease the chance of blocking by asking each site to broadcast the messages upon receiving. EasyCommit [41] solves the blocking problem by requiring each participant to forward the decision to other participants before logging it. However, the protocol satisfies the atomic commit properties only if at least one participant receives the forwarded message before the sender flushes the decision to its log. It also introduces extra messages during normal execution and increases the complexity when failure happens. Babaoglu and Toueg [25] propose a non-blocking atomic commit protocol based on 2PC. It applies three strategies: (1) synchronizing clocks on different sites so that out-of-time messages can be ignored; (2) having participants forward the decision to other participants upon receiving the message from the coordinator; and (3) presuming abort instead of running a termination protocol upon timeout. The first two strategies enable the last to avoid blocking. However, the protocol introduces more communication in the absence of failure, and the algorithm relies on synchronized clocks, a non-trivial requirement for real-world applications.

6.3 Co-design of 2PC and Replication

Many prior works optimize 2PC through the co-design of 2PC and replication protocols (e.g., Paxos) to solve latency and blocking problems at the time. Gray and Lamport [39] proposed Paxos Commit, which is a theoretical framework for optimizing 2PC and Paxos with a space of optimizations. It proposed some Paxos-specific optimizations such as pre-paring acceptors and piggybacking messages of 2PC and Paxos. Several implementations followed the spirit of Paxos Commit and made adaption for their scenarios [44, 65].

Kraska et al. [44] introduced Multi Data Center Consistency (MDCC). It assumes resource manager and Paxos leader are co-located on the same site so that it can piggyback the messages of Paxos with 2PC requests to save message roundtrips. They also propose a leaderless version combining 2PC and Fast Paxos. This version assumes that each acceptor can perform conflict detection for optimistic concurrency control independently and produce the same validation result. It can further reduce latency when conflicts are rare. Moreover, it optimizes for commutative operations by using update intents ("options") instead of the actual updates.

A recent work of parallel commit protocol [18, 61] in CockroachDB (CRDB) [57] uses primary copy replication based on Raft to determine whether the commit operation is completed. It co-locates the leader of Raft with participants (resource manager) while Cornus elevates this check to an external wrapper over cloud storage, whose replication algorithm is a black box.

Zhang et al. [65] introduced TAPIR. It used a customized replication protocol, Inconsistent Replication, to relax consistency in storage replicas and relies on application protocols to resolve inconsistencies. Mahmoud et al. [48] proposed a Replicated Commit protocol that runs 2PC at different data centers. It uses Paxos to reach consensus across data centers to determine if a transaction should commit. Yan et al.'s Carousel [63] runs 2PC in parallel with the execution phase and state replication, but assumes the read and write key sets are known in advance. Fan et al. [37] combines concurrency control, transaction commit, and replication in a single protocol to better utilize data locality and reduce cross-data center networks.

Finally, deterministic databases [35, 36, 47, 51, 58, 59] take a drastically different approach to handle 2PC and replication. Instead of treating computation nodes and storage service as horizontally separated layers, a deterministic database vertically partitions the cluster into replicas, and ensures consistency across replicas by running the same input transactions deterministically in all the replicas such that they produce identical results. Deterministic databases also simplify 2PC since only the inputs of transactions are made persistent and no logging happens during transaction execution. Compared to 2PC

and Cornus, deterministic databases have several limitations. For example, transactions need to be one-shot (i.e., cannot support multiple interactions with the DBMS); transactions must run in batches such that a single long-running transaction prolongs response time for the entire batch; most deterministic databases (except Aria [47]) require knowing the read/write sets of transactions before execution.

7 CONCLUSION

We proposed Cornus, a protocol optimizing 2PC for storage disaggregation, an architecture widely used by modern cloud databases. Cornus solves both the *long latency* and the *blocking problem* in 2PC by leveraging the new features provided by the architecture. We formally proved the correctness of Cornus and evaluated it on top of practical storage services including Redis and Azure Blob Storage. Our evaluations on YCSB show a speedup of up to 1.9× in latency.

REFERENCES

[1] [n.d.]. Amazon DynamoDB: Allows item-level access to DynamoDB based on an Amazon Cognito ID. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_dynamodb_items.html.

[2] [n.d.]. Amazon DynamoDB API Operations. https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Operations_Amazon_DynamoDB.html.

[3] [n.d.]. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[4] [n.d.]. Azure Cache for Redis. https://azure.microsoft.com/en-us/services/cache/.

[5] [n.d.]. Azure Storage redundancy. https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy.

[6] [n.d.]. CockroachDB. https://www.cockroachlabs.com.

[7] [n.d.]. cpp_redis. https://github.com/cpp-redis/cpp_redis.

[8] [n.d.]. Google Cloud BigTable — Writes. https://cloud.google.com/bigtable/docs/writes#conditional.

[9] [n.d.]. H-Store: A Next Generation OLTP DBMS. http://hstore.cs.brown.edu.

[10] [n.d.]. Redis. https://redis.io.

[11] [n.d.]. Redis ACL. https://redis.io/topics/acl.

[12] [n.d.]. Redis EVAL script. https://redis.io/commands/eval.

[13] 2014. Managing Concurrency in Microsoft Azure Storage. https://azure.microsoft.com/en-us/blog/managing-concurrency-in-microsoft-azure-storage-2/.

[14] 2015. gRPC: A high performance, open-source universal RPC framework. https://grpc.io/.

[15] 2018. Amazon Athena — Serverless Interactive Query Service. https://aws.amazon.com/athena.

[16] 2018. Amazon Redshift. https://aws.amazon.com/redshift.

[17] 2018. Presto. https://prestodb.io.

[18] 2020. *Parallel Commits*. https://www.cockroachlabs.com/docs/v20.2/architecture/transaction-layer.html#parallel-commits

[19] Maha Abdallah. 1997. A non-blocking single-phase commit protocol for rigorous participants. In *In Proceedings of the National Conference Bases de Donnes Avances*. Citeseer.

[20] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. 1998. One-phase commit: does it make sense?. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250)*. IEEE, 182–192.

[21] Y Al-Houmaily and P Chrysanthis. 1995. Two-phase commit in gigabit-networked distributed databases. In *Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*.

[22] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[23] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12, 3411–3424. https://doi.org/10.14778/3415478.3415560

[24] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*.

[25] Ozalp Babaoglu and Sam Toueg. 1993. Understanding non-blocking atomic commitment. *Distributed systems* (1993).

[26] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley New York.

[27] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a Database on S3. 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, 251–264. https://doi.org/10.1145/1376616.1376645

[28] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 143–157.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2, 1–26.

[30] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[31] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[33] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*. 54–70.

[34] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud*. Technical Report.

[35] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *PVLDB* (2015), 1190–1201.

[36] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017), 613–624.

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

[38] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 85–96.

[39] Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Trans. Database Syst.* 31, 1 (March 2006), 133–160. https://doi.org/10.1145/1132863.1132867

[40] Hua Guo, Xuan Zhou, and Le Cai. 2021. Lock Violation for Fault-tolerant Distributed Database System. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1416–1427.

[41] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol.. In *EDBT*. 157–168.

[42] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *VLDB* (2017), 553–564.

[43] Hideaki Kimura, Goetz Graefe, and Harumi A Kuno. 2012. Efficient locking techniques for databases on modern hardware.. In *ADMS@ VLDB*. 1–12.

[44] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 113–126.

[45] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[46] Inseon Lee and Heon Young Yeom. 2002. A single phase distributed commit protocol for main memory database systems. In *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 8–pp.

[47] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. (2020).

[48] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment* 6, 9 (2013), 661–672.

[49] Dahlia Malkhi and Jean-Philippe Martin. 2013. Spanner's concurrency control. *ACM SIGACT News* 44, 3 (2013), 73–77.

[50] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[51] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication.. In *EDBT*. 73–84.

[52] George Samaras, Kathryn Britton, Andrew Citron, and C Mohan. 1993. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 520–529.

[53] Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 133–142.

[54] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial strictness in two-phase locking. In *International Conference on Database Theory*. Springer, 139–147.

[55] James W Stamos and Flaviu Cristian. 1990. A low-cost atomic commit protocol. In *Proceedings Ninth Symposium on Reliable Distributed Systems*. IEEE, 66–75.

[56] James W Stamos and Flaviu Cristian. 1993. Coordinator log transaction execution protocol. *Distributed and Parallel Databases* 1, 4 (1993), 383–408.

[57] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[58] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 70–80.

[59] 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 *SIGMOD*. 1–12.

[60] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive — A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*.

[61] Nathan VanBenschoten. 2019. *Parallel Commits: An Atomic Commit Protocol For Globally Distributed Transactions*. https://www.cockroachlabs.com/blog/parallel-commits/

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

[63] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*. 231–243.

[64] 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.

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

[66] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*. 2653–2666.

[67] 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*. Association for Computing Machinery, New York, NY, USA, 2653–2666. https://doi.org/10.1145/3448016.3457559