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

Zhihan Guo, Xinru Zeng, Kan Wu, Wah-Chuen Hwang, Ziwei Ren, Xiangyao Ye, Mehmet Balatkinish, Philip A. Bernstein
University of Wisconsin-Madison, "Confluent, Inc., Microsoft Research. (zhihan.guo, kewu, wah-chuen.hwang, zewi@wisc.edu)

ABSTRACT

Three-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 round-trip latencies 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 stored highly available servers. One key observation is that dis-aggregated storage enables protocol innovations that can address both long latency and blocking problems.

We develop Corinus, an optimized 2PC protocol to achieve this goal. The only extra functionality Corinus requires is an atomic compare and swap capability in the storage layer, which many existing storage systems already support. Our performance is detailed with proofs and show how 2PC adds two limitations. We also show that Corinus outperforms 3PC, a well-established alternative for use in cloud environments. Experimental evaluations show that Corinus can achieve up to 1.5x latency reduction over conventional 2PC.

1 INTRODUCTION

Databases are migrating to the cloud because of desirable features such as simplicity, high availability, and cost competitiveness. Modern cloud-native databases feature a storage disaggregation architecture where the storage is decoupled from computation as a stateless service as shown in Figure 1. This architecture allows independent scaling and billing of computation and storage, which can improve resource utilization, reduce operational costs, and enable flexible cloud deployments with heterogeneous configurations. Many cloud-native database systems adopt such architecture for both OCP [22, 46, 52, 60] and EAP [17-1-22, 24, 31, 64]. Nowadays, as storage services offer essential functions such as fault tolerance, reliability, and security at a low cost, systems start to factor in their designs on the existing disaggregated storage services [23, 27].

This paper focuses on efficient deployment of the three-phase commit protocol on storage disaggregation architectures. The distributed three-phase commit protocol (2PC) is the most widely used atomic commit protocol, which enables the distributed execution of transactions over a set of distributed storage systems. 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. Prior work has demonstrated that the majority of a transaction’s execution time can be attributed to 2PC [29, 21, 24, 40, 30, 52, 64]. The second problems is blocking: 2PC blocking occurs if a coordinator

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

before notifying participants of the final decision. These 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 (29-31-24, 40, 46, 40, 50), or they mitigate the blocking problem by adding an extra phase and joining latency (25, 41, 55). Another line of research addresses both problems through eliminating the storage. Examples include Paxos Commit [29], TARM [10], MCC [46], and parallel commit in CockroachDB [27]. Existing solutions, however, are not applicable to general storage services because they require storage to support storage designs that perform conflict detection between transactions [6, 41, 37, 65] and/or need specific implementations protocols [19, 44, 65]. Therefore, they cannot be readily applied to most existing storage services.

In this paper, we also aim to maintain the flexibility brought by disaggregation without requiring customized APIs for the storage server. Therefore, a database can adopt existing highly optimized storage services and further avoid the expense of developing a new one, and still allow the storage to make new optimizations for storage use. Therefore, the following research question: What is the minimal requirement from the storage layer to enable 2PC optimizations under existing latency and locality? Our answer is that the only requirement is the ability to provide low-cost transactional, which means that for each transaction, only one operation of its state in the log is allowed. We show that log-visibility semantics can be achieved with a simple compare-and-set API, which is supported by almost every storage service today, including Redis [19], Microsoft Azure Storage [28], Amazon Dynamo [32], and Google BigTable [27].
Announcement

No lecture on Wednesday next week

Optional 10-min meeting to discuss your project with instructor

Signup sheet (access using your UW account)

– https://docs.google.com/spreadsheets/d/1HatkCJkKUD8Zl0zVe_xZ9OxhfthrY6YAgil9NX8uS9g/edit?usp=sharing

Meetings over zoom

– https://uwmadison.zoom.us/j/92584913804?pwd=NVdON0VjcWJLQTVwVkJ0NzdRSURyZz09
Outline

Cloud database
Storage disaggregation
Cornus protocol
Databases Moving to the Cloud

According to Gartner Report [1]

$39.2 billion, 49% of all DBMS revenue from cloud in 2021

On-prem database
$40.8B in 2021

Cloud database
$39.2B in 2021

Cloud vs. On-premises Revenue

Databases Moving to the Cloud

According to Gartner Report [1]
$39.2 billion, 49% of all DBMS revenue from cloud in 2021

Cloud vs. On-premises Revenue

On-prem database
$40.8B in 2021

Cloud database
$39.2B in 2021

Low Cost
Elasticity
Availability

Databases Moving to the Cloud

**Transactional DB**
- Amazon Aurora
- Polardb
- Orleans
- SQL Server
- GaussDB
- TiDB
- Cockroach DB
- YugabyteDB
- Azure Cosmos DB

**Analytical DB**
- Spark
- SQL
- Snowflake
- MongoDB
- Vertica
- Presto
- Amazon Athena
- Redshift
- BigQuery
- Trino
Cloud DB: Storage-Disaggregation

Database logic in computer cluster

Database states (e.g., tables and logs) in cloud storage service

Manage computation and storage as separate services
Cloud DB: Storage-Disaggregation

- Database logic in computer cluster
- Increasing network speed
- Database states (e.g., tables and logs) in cloud storage service

Manage computation and storage as separate services
Advantages of Storage-Disaggregation

**Advantage #1: Elasticity**

- Compute and storage resources can scale independently.
Advantages of Storage-Disaggregation

Advantage #1: Elasticity

- Compute and storage resources can scale independently

![Diagram of compute and storage nodes with labels: Compute-intensive workload and Storage-intensive workload]
Advantages of Storage-Disaggregation

Advantage #2: Low Cost

- Storage service can be much cheaper than compute servers

<table>
<thead>
<tr>
<th>S3 storage price</th>
<th>$0.02 per GB per month</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 vCPU Virtual Machine</td>
<td>$0.5 per hour per VM</td>
</tr>
</tbody>
</table>
Advantages of Storage-Disaggregation

Advantage #2: Low Cost

- Storage service can be much cheaper than compute servers

<table>
<thead>
<tr>
<th>Service</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3 storage price</td>
<td>$0.02 per GB per month</td>
</tr>
<tr>
<td>16 vCPU Virtual Machine</td>
<td>$0.5 per hour per VM</td>
</tr>
</tbody>
</table>
### Advantages of Storage-Disaggregation

#### Compute cluster

- **Data Center Network**
- **Storage as a Service (SaaS)**

#### Storage as a Service (SaaS)

- **Azure STORAGE**
- **RocksDB**
- **redis**
- **TiKV**
- **Cloud Bigtable**

#### Advantages of Storage

**Advantage #2: Low Cost**

- Storage service can be much cheaper than compute servers

<table>
<thead>
<tr>
<th>Service</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3 storage</td>
<td>$0.02 per GB per month</td>
</tr>
<tr>
<td>16 vCPU Virtual Machine</td>
<td>$0.5 per hour per VM</td>
</tr>
</tbody>
</table>

---

![Cost of previsioning for peak](chart)

**Cost of previsioning for peak**

- **Load**
- **Time**

---

14
Advantages of Storage-Disaggregation

Advantage #2: Low Cost

- Storage service can be much cheaper than compute servers

<table>
<thead>
<tr>
<th>Service</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3 storage price</td>
<td>$0.02 per GB per month</td>
</tr>
<tr>
<td>16 vCPU Virtual Machine</td>
<td>$0.5 per hour per VM</td>
</tr>
</tbody>
</table>

Load

Cost of previsioning for peak

Cost of elastic previsioning

Time
Advantages of Storage-Disaggregation

Advantage #2: Low Cost

- Storage service can be much cheaper than compute servers

<table>
<thead>
<tr>
<th>Service</th>
<th>Cost Per Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3 storage price</td>
<td>$0.02 per GB per month</td>
</tr>
<tr>
<td>16 vCPU Virtual Machine</td>
<td>$0.5 per hour per VM</td>
</tr>
</tbody>
</table>

No compute cost at zero load

Cost of previsioning for peak
Advantages of Storage-Disaggregation

Advantage #3: Availability

- Storage service provides high availability through geo-replication
- Simplifies fault tolerance in DB
Advantages of Storage-Disaggregation

Advantage #3: Availability
- Storage service provides high availability through geo-replication
- Simplifies fault tolerance in DB

Storage-disaggregation architecture widely deployed in cloud databases
The storage service can **scale horizontally**, has **built-in high availability**, and has **richer APIs**.
Distributed Atomic Commitment

Data partitioned across machines

Partition 1  Partition 2  Partition 3
Distributed Atomic Commitment

Data partitioned across machines

A transaction updates data across multiple partitions
Distributed Atomic Commitment

Data partitioned across machines

A transaction updates data across multiple partitions

**Atomic commitment** requires the transaction to commit in **all or none** of the involved partitions
Distributed Atomic Commitment

<table>
<thead>
<tr>
<th>Transaction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>write(A)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

With storage disaggregation, log files locate in the storage service.
Two-Phase Commit (2PC)

Coordinator initiates the 2PC protocol

The example assumes a committing transaction
# Two-Phase Commit (2PC)

<table>
<thead>
<tr>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

Transaction

Coordinator initiates the 2PC protocol
Two-Phase Commit (2PC)

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Coordinator</th>
<th>Participant 1</th>
<th>Participant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(A)</td>
<td>VOTE-YES</td>
<td>VOTE-YES</td>
<td>VOTE-YES</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each participant appends **VOTE-YES** to local log file
- Promise not to **unilaterally** abort
Two-Phase Commit (2PC)

Participants reply votes to coordinator
Two-Phase Commit (2PC)

Coordinator logs the final decision (e.g., `COMMIT` or `ABORT`)

The decision log record is the **ground truth** of the transaction outcome.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1.png" alt="CPU" /></td>
<td><img src="image2.png" alt="CPU" /></td>
<td><img src="image3.png" alt="CPU" /></td>
</tr>
<tr>
<td></td>
<td><img src="image4.png" alt="LOG" /></td>
<td><img src="image5.png" alt="LOG" /></td>
<td><img src="image6.png" alt="LOG" /></td>
</tr>
<tr>
<td>Coordinator</td>
<td><img src="image7.png" alt="Coordinator" /></td>
<td><img src="image8.png" alt="Participant 1" /></td>
<td><img src="image9.png" alt="Participant 2" /></td>
</tr>
<tr>
<td>VOTE-YES</td>
<td>VOTE-YES</td>
<td>VOTE-YES</td>
<td>VOTE-YES</td>
</tr>
</tbody>
</table>

Example:
- `write(A)`
- `write(B)`
- `write(C)`
Two-Phase Commit (2PC)

<table>
<thead>
<tr>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

Transaction Coordinator

- Participant 1: VOTE - YES
- Participant 2: VOTE - YES

Reply to user after writing the decision log record
Two-Phase Commit (2PC)

Coordinator sends the final decision to all participants

Transaction
write(A)  write(B)  write(C)

Coordination diagram:

back to user
Two-Phase Commit (2PC)

Coordinator sends the final decision to all participants

Participants log the decision
   – For independent recovery upon failure
Limitations of 2PC

**Limitation #1: Long latency**

- User experiences latency of two logging operations

Transaction

<table>
<thead>
<tr>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

Coordinator | Participant 1 | Participant 2
Limitations of 2PC

**Limitation #1: Long latency**
- User experiences latency of two logging operations

**Limitation #2: Blocking problem**
- Participants are blocked if the coordinator fails
### 2PC Limitations – Prior Solutions

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Example systems</th>
<th>Limitations in prior solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce latency</td>
<td>Coordinator log [1]</td>
<td>• Extra system or workload assumptions</td>
</tr>
<tr>
<td></td>
<td>Implicit yes vote [2]</td>
<td>• Violate site autonomy</td>
</tr>
<tr>
<td></td>
<td>Early prepare [3]</td>
<td></td>
</tr>
</tbody>
</table>

---

## 2PC Limitations – Prior Solutions

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Example systems</th>
<th>Limitations in prior solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce latency</td>
<td>Coordinator log [1]</td>
<td>• Extra system or workload assumptions</td>
</tr>
<tr>
<td></td>
<td>Implicit yes vote [2]</td>
<td>• Violate site autonomy</td>
</tr>
<tr>
<td></td>
<td>Early prepare [3]</td>
<td></td>
</tr>
<tr>
<td>Non-blocking</td>
<td>Three-phase commit (3PC) [4]</td>
<td>• Requires extra latency and/or network messages</td>
</tr>
</tbody>
</table>

## 2PC Limitations – Prior Solutions

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Example systems</th>
<th>Limitations in prior solutions</th>
</tr>
</thead>
</table>
| Reduce latency | Coordinator log [1]  
Implicit yes vote [2]  
Early prepare [3] | • Extra system or workload assumptions  
• Violate site autonomy |
| Non-blocking | Three-phase commit (3PC) [4] | • Requires extra latency and/or network messages |
| Codesign 2PC with replication | Paxos commit [5]  
MDCC [6]  
Parallel commit [7]  
TAPIR [8] | • Extra design complexity  
• Custom-designed consensus protocol |

## 2PC Limitations – Prior Solutions

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Example systems</th>
<th>Limitations in prior solutions</th>
</tr>
</thead>
</table>
| Reduce latency                    | Coordinator log [1]  
Implicit yes vote [2]  
Early prepare [3]                                                      | • Extra system or workload assumptions  
• Violate site autonomy                                                  |
| Non-blocking                      | Three-phase commit (3PC) [4]                                                  | • Requires extra latency and/or network messages                      |
| Codesign 2PC with replication     | Paxos commit [5]  
MDCC [6]  
Parallel commit [7]  
Tapir [8]                                                              | • Extra design complexity  
• Custom-designed consensus protocol                                     |

**Research Question**: What is the **minimal requirement** from the storage service to enable 2PC optimizations addressing **high latency and blocking**?
An optimized two-phase commit protocol for a cloud database with storage disaggregation
Cornus Overview

An optimized two-phase commit protocol for a cloud database with storage disaggregation

2PC Limitation 1: **Long latency**
⇒ Cornus reduces 2 logging events to 1 logging event

2PC Limitation #2: **Blocking problem**
⇒ Cornus is **non-blocking**
Cornus Overview

An optimized two-phase commit protocol for a cloud database with storage disaggregation

2PC Limitation 1: Long latency
⇒ Cornus reduces 2 logging events to 1 logging events

2PC Limitation #2: Blocking problem
⇒ Cornus is non-blocking

Only new storage-layer function is `LogOnce()` which can be implemented using `compare-and-swap`
Cornus Key Ideas

Key idea #1: **Remove decision logging**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

- **CPU**
- **LOG**

**Coordinator**  
**Participant 1**  
**Participant 2**

```
write(A)  
write(B)  
write(C)  
```

```
Transaction Coordinator Participant 1 Participant 2
VOTE - YES VOTE - YES VOTE - YES
```

back to user
Cornus Key Ideas

Key idea #1: Remove decision logging

**Ground truth:** collective votes in all participants logs

- Uncertain node can directly read all votes
Cornus Key Ideas

**Key idea #1: Remove decision logging**

**Ground truth:** collective votes in all participants logs
- Uncertain node can directly read all votes

Enabled by storage disaggregation through
- Highly available storage service
- Shared across compute nodes
Key idea #2: LogOnce() storage API
Key idea #2: LogOnce() storage API

Avoid blocking by directly updating log files of unresponsive nodes
- Only first LogOnce() request can succeed
Key idea #2: LogOnce() storage API

**Avoid blocking** by directly updating log files of unresponsive nodes
- Only first LogOnce() request can succeed

LogOnce() can be implemented using CAS-like APIs (e.g., Etags)
Cornus Key Ideas

Transaction

<table>
<thead>
<tr>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

Key idea #2: `LogOnce()` storage API

Enabled by storage disaggregation through
– Rich APIs of storage service
Cornus Failure Example

<table>
<thead>
<tr>
<th>Transaction</th>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

Coordinator fails

Coordinator | Participant 1 | Participant 2

VOTE-YES | VOTE-YES | VOTE-YES
Cornus Failure Example

Coordinator fails

Timeout in participant 1 waiting for coordinator’s message

<table>
<thead>
<tr>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

Transaction

Coordinator
Participant 1
Participant 2

write(A) write(B) write(C)
Cornus Failure Example

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Coordinator</th>
<th>Participant 1</th>
<th>Participant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Use LogOnce() to write ABORT to other nodes’ log files
Cornus Failure Example

Use LogOnce() to write ABORT to other nodes’ log files

\textit{VOTE-YES} already exists, LogOnce() does not modify log content
Cornus Failure Example

| write(A) | write(B) | write(C) |

Transaction Coordinator
Participant 1
Participant 2

Storage service returns **VOTE-YES** without updating the logs

Participant 1 logs the **COMMIT** decision

**write(A)**
**write(B)**
**write(C)**

**timeout**

fail
### Cornus Failure Example

<table>
<thead>
<tr>
<th>write(A)</th>
<th>write(B)</th>
<th>write(C)</th>
</tr>
</thead>
</table>

- **Transaction**

  **Coordinator**
  
  **Participant 1**
  
  **Participant 2**

**Storage service returns** *VOTE-YES* **without updating the logs**

**Participant 1 logs the** *COMMIT* **decision**

**Same process can happen for other participants (e.g., Participant 2)**
Cornus vs. 2PC Summary

Cornus

Two-Phase Commit

Commit Case

back to user
Cornus vs. 2PC Summary

**Cornus**

**Commit Case**

VOTE-YES  VOTE-YES  VOTE-YES

back to user

**Failure Case**

fail  timeout  VOTE-YES  VOTE-YES

Non-Blocking!

**Two-Phase Commit**

**Commit Case**

VOTE-YES  VOTE-YES  VOTE-YES

back to user

**Failure Case**

fail  timeout  timeout  VOTE-YE

Blocking!
Cornus vs. 2PC Summary

Key idea #1: No decision logging
Key idea #2: LogOnce() storage API

Enabled by storage disaggregation through
- Highly available storage service
- Shared across compute nodes
- Rich APIs of storage service
Performance Evaluation (on Redis)

**Hardware**: 8 core (Intel Xeon 8272CL × 8), 64 GB DRAM

**Workload**: 10GB YCSB data set, 16 accesses per txn, reads/updates = 50/50, no skew

**Storage service**: Premium P4 Redis instance on Azure. One master node + one slave node.

Cornus reduces latency by up to $1.9\times$ compared to 2PC
Further Optimizations

Optimization #1: Storage service responds to both the requesting participant and coordinator
- Save one network hop
- Requires changes in storage API

Prepare in Cornus

Optimization #1
Further Optimizations

Optimization #2: Storage service responds to coordinator and all participants
- Save one more network hot
- Incurs more network traffic
- Requires changes in storage API
Further Optimizations

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

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

Further optimizations require the co-design of 2PC and consensus
Check out Our VLDB’22 Paper

• Pseudo-code of Cornus
• Analysis of failure and recovery
• Proof of correctness
• Deployment over Redis and Azure blob store
• More performance evaluation

Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation
Zhihan Guo
Xinyu Zeng
Kai Wu
University of Wisconsin-Madison
zhihan@cs.wisc.edu
xzeng@cs.wisc.edu
kaiwu@cs.wisc.edu
Malakhi Balakrishnan
Conference, Inc.
malakhi.balakrishnan@conference.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 major log entries on the critical path, and blocking of queries when a coordinator fails. In this paper, we introduce the concept of pseudocode and present several pseudocode innovations that can address both the long latency and blocking problems. We design Cornus, an optimized 2PC protocol to achieve this goal. The only extra functionality Cornus requires is an atomiccompare-and-wrap capability in the storage layer, which many existing storage services already support. We present Cornus in detail and show how it alleviates the two limitations. We also deploy it on real storage services including Azure Red-Storage and Redis. Empirical evaluations show that Cornus can achieve up to 1.6X latency reduction over conventional 2PC.

PYLDB Reference Format:
DOI: 10.14778/3593061.3593087

PYLDB Article Availability:
The source code, data, and other artifacts have been made available at https://github.com/Conference/Cornus.

1 INTRODUCTION
Databases are expected to scale as the cloud because of attributes such as scalability, high availability, and cost competitiveness. Modern cloud-native databases feature a storage-disaggregation architecture where the storage is decoupled from computation as a

This work is licensed under the Creative Commons BY-NC-SA 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-sa/4.0/ to view a copy of this license. For more information about the license, please visit https://creativecommons.org/about. Permission is granted to copy, distribute, and/or modify this document under the terms of the license. Please credit the author.
DOI: 10.14778/3593061.3593087

Figure 1: Shared-nothing vs. Storage-disaggregation.
Q/A – Cornus

What implementation of 2PC used for comparison?
Cornus on a shared-nothing architecture?
Consensus algorithm like Paxos or Raft used for replication?
Completely decouple compute sharding from storage sharding?
In storage disaggregation, any strength to partition keys? Why not to run one transaction only in one node?
Consistency required from underlying storage service?
How does storage implement compare-and-swap?
Next Lecture