CS 764: Topics in Database Management Systems

Lecture 21: Cornus

Xiangyao Yu
11/16/2022
Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation (Extended Version)
Zhixiao Gao, Xinyu Zeng, Ken Wu, Wah-Chuen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein
University of Wisconsin-Madison, "C"oreTrust, Inc., Microsoft Research
zhixiaogao@cs.wisc.edu, xinyuzeng@wisc.edu, kenwu@cs.wisc.edu

ABSTRACT
Every node commit (2C) is widely used in distributed databases to ensure atomicity of distributed transactions. Conventional 2C was originally designed for the shared-nothing architecture and has two limitations: low latency due to the two phase log commit 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 cloud highly available service. One key observation is that disaggregated storage enables protocol innovation that can address both low latency and blocking problems. We develop Cornus, an optimized 2C 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 systems already support. We present Cornus in detail with proofs and show how to add this extra functionality. We also provide the implementation of the storage layer of an in-memory key-value store. Empirical evaluations show that Cornus can achieve up to 1.5x latency reduction over conventional 2C.

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 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 OpenCF [22, 49, 62, 63] and DBAF [15-17, 24, 34, 64]. However, as storage services often offer essential features such as fault tolerance, availability, and scalability at a low cost, systems that support these features on the existing disaggregated storage services [25, 27].

This paper focuses on efficient deployment of the two-phase commit protocol (2C) for cloud storage systems (CSS). The major contribution of this work is to show for the first time that 2C is the most widely used atomic commit protocol, which enables its adoption in cloud storage systems. We evaluate the performance of the two-phase commit transaction (2C) in the most widely used atomic commit protocol, which we implement as a cloud storage service. We evaluate the use of the involved data partitions. 2C was originally designed for the shared-nothing architecture and suffers from two major limitations. The first is low latency. 2C requires two round-trip network messages and associated logging operations. Per user work has demonstrated that the majority of a transaction's execution time can be attributed to 2C [25, 26, 51, 52, 53, 62]. The second problem is blocking. 2C blocking occurs if a coordinator

2VLDB 2022
Figure 1: Shared-Nothing vs. Storage-Disaggregation.

before modifying participants of the final decision. These two problems greatly limit the performance of 2C, especially in a storage disaggregation architecture.

Various techniques have been proposed to address these two problems with 2C. Some proposed optimizations target the shared-nothing architecture and do not solve both problems simultaneously. These proposals either reduce latency by making strong assumptions about the workload and system that are not always practical for disaggregated storage [19–21, 26, 49, 51, 62], or they mitigate the blocking problems by adding an extra phase and joining latency [25, 51]. Another line of research addresses both problems through optimizing the storage. Examples include Paxon-Cornus (19), TAPR (18), MDCS (16), and parallel commit in CockroachDB (17). Existing solutions, however, are not applicable to general storage services because they require significant changes to storage designs that perform conflict detection between transactions [6, 41, 57, 65] and/or need specific replication protocols (19, 44, 65). Therefore, they cannot be readily applied to most existing storage services.

In this paper, we aim to maintain the flexibility brought by scalability 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 also allow the storage to exploit new functionalities that are not in the current 2C design. The following research question: What is the minimal requirement from the storage layer to enable 2C optimization and extra high availability and scalability? Our answer is to add only a new component to the existing COW and Atomicity. One specific feature is the ability to provide key consistency. To achieve this, we add metadata for each transaction, and the new operation is to update the old entry in the log to a new state. The limitations of this operation is to use the same storage service, and no optimization at the storage layer is needed. A simple implementation is to use a single registered API, which is supported by almost every storage service today, including Redis (16), Microsoft Azure Storage (26), Apache Dynomite (22), and Google BigTable (29).
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/1HatkJkKUD8ZI0zVe_xZ9OxhftthY6YAgil9NX8uS9g/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1HatkJkKUD8ZI0zVe_xZ9OxhftthY6YAgil9NX8uS9g/edit?usp=sharing)

Meetings over zoom

- [https://uwmadison.zoom.us/j/92584913804?pwd=NVdON0VjcWJLOTVwVk9UNzdRSURyZz09](https://uwmadison.zoom.us/j/92584913804?pwd=NVdON0VjcWJLOTVwVk9UNzdRSURyZz09)
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

Cloud vs. On-premises Revenue

Cloud database
$39.2B in 2021

On-prem database
$40.8B in 2021

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

Databases Moving to the Cloud

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

**Analytical DB**
- Spark
- SQL
- Snowflake
- MongoDB
- Hive
- 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 showing the separation of compute and storage resources.](image)

- **Storage-intensive workload**
- **Compute-intensive workload**

- Data Center Network
- Compute cluster
- Storage as a Service (SaaS)
Advantages of Storage-Disaggregation

Advantage #2: Low Cost

- Storage service can be much cheaper than compute servers

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

Compute cluster

Data Center Network

Storage as a Service (SaaS)

- S3 storage price
- 16 vCPU Virtual Machine
Advantages of Storage-Disaggregation

**Compute cluster**

**Data Center Network**

**Storage as a Service (SaaS)**

**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>

Load

Time
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>

**Load vs. Time**

- Cost of provisioning for peak

---

Data Center Network

- Compute cluster

Storage as a Service (SaaS)

- S3
- 16 vCPU Virtual Machine
- $0.5 per hour per VM

---

**Time**

**Load**
Advantages of Storage-Disaggregation

Advantage #2: Low Cost

- Storage service can be much cheaper than compute servers

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

No compute cost at zero load
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

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(A)</td>
<td>write(B)</td>
<td>write(C)</td>
</tr>
</tbody>
</table>

**Transaction**

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

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(A)</td>
<td>write(B)</td>
<td>write(C)</td>
</tr>
</tbody>
</table>

- **Transaction**

---

22
Distributed Atomic Commitment

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)

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

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

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

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

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

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

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

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

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

```
Coordinator
Participant 1
Participant 2
```

```
VOTE-YES
VOTE-YES
VOTE-YES
```
Two-Phase Commit (2PC)

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

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

Coordinator sends the final decision to all participants
Two-Phase Commit (2PC)

Coordinator sends the final decision to all participants

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

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

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

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

```
Coordinator  Participant 1  Participant 2
```

```
VOTE-YES  VOTE-YES  VOTE-YES
```

Back to user
Limitations of 2PC

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

**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>
<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>
<tr>
<td>Codesign 2PC with replication</td>
<td>Paxos commit [5]</td>
<td>• Extra design complexity</td>
</tr>
<tr>
<td></td>
<td>MDCC [6]</td>
<td>• Custom-designed consensus protocol</td>
</tr>
<tr>
<td></td>
<td>Parallel commit [7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TAPIR [8]</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>
<tr>
<td>Codesign 2PC with replication</td>
<td>Paxos commit [5]</td>
<td>• Extra design complexity</td>
</tr>
<tr>
<td></td>
<td>MDCC [6]</td>
<td>• Custom-designed consensus protocol</td>
</tr>
<tr>
<td></td>
<td>Parallel commit [7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tapir [8]</td>
<td></td>
</tr>
</tbody>
</table>

**Research Question**: What is the **minimal requirement** from the storage service to enable 2PC optimizations addressing **high latency and blocking**?
Cornus Overview

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

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

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

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

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

Coordinator | Participant 1 | Participant 2

- VOTE - YES
- VOTE - YES
- VOTE - YES

back to user
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: \texttt{LogOnce()} storage API

Avoid blocking by directly updating log files of unresponsive nodes

– Only first \texttt{LogOnce()} request can succeed

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

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

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

Enabled by storage disaggregation through
- Rich APIs of storage service

back to user
# Cornus Failure Example

A transaction consists of `write(A)`, `write(B)`, and `write(C)` operations. The `Coordinator` initially votes `YES` for each operation, but fails before the completion of the transaction.

## Coordinator Failure

- **Transaction Operations**:
  - `write(A)`
  - `write(B)`
  - `write(C)`

- **Coordinator Actions**:
  - Initially votes `YES` for each operation.

- **Participant Actions**:
  - Participant 1 votes `YES` after `write(A)`.
  - Participant 2 votes `YES` after `write(B)`.
  - Participant 3 votes `YES` after `write(C)`.

- **Coordination Failure**:
  - Coordinator fails before receiving all `YES` votes.

- **Log Files**:
  - Coordinator log file.
  - Participant 1 log file.
  - Participant 2 log file.
  - Participant 3 log file.

- **Final Status**:
  - Transaction fails due to the coordinator failure.
## Cornus Failure Example

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

### Coordinator fails

#### Timeout in participant 1 waiting for coordinator’s message
## Cornus Failure Example

### Transaction Log

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

Use `LogOnce()` to write `ABORT` to other nodes' log files

### Diagram

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

The diagram illustrates the voting process with `VOTE-YES` and handling failure with `fail` and `timeout`.
Use LogOnce() to write ABORT to other nodes’ log files

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

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

Transaction

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

Participant 1 logs the **COMMIT** decision
**Cornus Failure Example**

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

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**

- VOTE-YES
- VOTE-YES
- VOTE-YES

**Two-Phase Commit**

- VOTE-YES
- VOTE-YES
- VOTE-YES

Commit Case

VOTE-YES

VOTE-YES

VOTE-YES

VOTE-YES
Cornus vs. 2PC Summary

**Commit Case**
- **Cornus**
  - VOTE-YES
  - VOTE-YES
  - VOTE-YES
  - back to user

- **Two-Phase Commit**
  - VOTE-YES
  - VOTE-YES
  - VOTE-YES
  - back to user

**Failure Case**
- **Cornus**
  - VOTE-YES
  - VOTE-YES
  - VOTE-YES
  - fail
  - timeout
  - VOTE-YES
  - Non-Blocking!

- **Two-Phase Commit**
  - VOTE-YES
  - VOTE-YES
  - VOTE-YES
  - fail
  - timeout
  - timeout
  - timeout
  - 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× 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
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

Further optimizations require the codesign of 2PC and consensus

<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>(optimization)</td>
<td>2.5 + 0 = 2.5</td>
<td>Leader of Paxos can forward a message to coordinator</td>
</tr>
<tr>
<td>2PC (colocation)</td>
<td>2 + 1 = 3</td>
<td>Participant coordinates replication</td>
</tr>
<tr>
<td>Cornus (colocation)</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 co-ordinator to learn from quorum</td>
</tr>
</tbody>
</table>

Table 3: Time complexity for protocols integrating with Paxos or its variations
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
Zhihao Guo
Xinyu Zeng
Kui Wu
University of Wisconsin-Madison
zhilhao@wisc.edu
xin@cs.wisc.edu
kuwu@cs.wisc.edu
Maldev Bhalukshinan
Confient.io
mbhalukshinan@confient.io

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 writes on the critical path, and blocking of queries when a coordinator fails. Conventional 2PC implementations are moving to storage disaggregation architecture whose storage is a shared highly-available service. Conventional 2PC implementations of 2PC fail to 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 a strong consistency and log capability in the storage layer, which many existing storage services already support. We present Cornus in detail and show how it silences the two limitations. We also deploy it on real storage services including Azure Web-Storage and Redis. Experimental evaluations show that Cornus can achieve up to 1.6x latency reduction over conventional 2PC.

PYLDB Reference Format:
doi:10.14778/3402975.3403077

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

1 INTRODUCTION

Databases are evolving to the cloud because of attractive 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

This work is licensed under the Creative Commons CC-BY-4.0 International License. You have the right to share and adapt this work but must give appropriate credit to the original work and provide a link to the license. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/. For details and suggestions on how to credit the original work, visit https://creativecommons.org/learn/credit. For more information on the license, visit https://creativecommons.org/licenses/by/4.0/.

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

Yi Lu, et al., Aria: A Fast and Practical Deterministic OLTP Database. VLDB, 2020