Carnegie Mellon University

DATABASE SYSTEMS

Distributed Databases II

LECTURE #24 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #4 is due Sunday Dec 7th @ 11:59pm
→ Recitation Slides + Video (**@300**)

Homework #6 is due Sunday Dec 7th @ 11:59pm

Final Exam is on Thursday Dec 11th @ 1:00pm
→ Do not make travel plans before this date!

We are recruiting TAs for the next semester
→ Apply at: https://www.ugrad.cs.cmu.edu/ta/S26/

UPCOMING DATABASE TALKS

Apache Polaris (DB Seminar)
→ Monday Dec 1st @ 12:00pm
→ Zoom

Apache Fluss (DB Seminar)
→ Monday Dec 7th @ 12:00pm
→ Zoom

LAST CLASS

System Architectures
→ Shared-Nothing vs. Shared-Disk

Partitioning
→ Horizontal: Hash, Range, Round Robin

Replication
→ Primary-Replica vs. Multi-Primary

Transaction Coordination
→ Centralized vs. Decentralized

OLTP VS. OLAP

On-line Transaction Processing (OLTP):

→ Short-lived read/write txns.
→ Small footprint.
→ Repetitive operations.

On-line Analytical Processing (OLAP):

→ Long-running, read-only queries.
→ Complex joins.
→ Exploratory queries.

TODAY'S AGENDA

Atomic Commit Protocols

Consistency Issues (CAP / PACELC)

Distributed Join Algorithms

Shuffle

OBSERVATION

Recall that our goal is to have multiple physical nodes appear as a single logical DBMS.

We have not discussed how to ensure that all nodes agree to commit a txn and then to make sure it does commit if the DBMS decides it should.

→ What happens if a node fails?
→ What happens if messages show up late?
→ What happens if the system does not wait for every node to agree to commit?

IMPORTANT ASSUMPTION

We assume all nodes in a distributed DBMS are well-behaved and under the same administrative domain.

→ If we tell a node to commit a txn, then it will commit the txn (if there is not a failure).

If you do not trust the other nodes in a distributed DBMS, then you need to use a Byzantine Fault Tolerant protocol for txns (blockchain).

→ Blockchains are **not** good for high-throughput workloads.

Don't Do This!

ATOMIC COMMIT PROTOCOL

Coordinating the commit order of txns across nodes in a distributed DBMS.

→ Commit Order = State Machine
→ It does not matter whether the database's contents are replicated or partitioned.

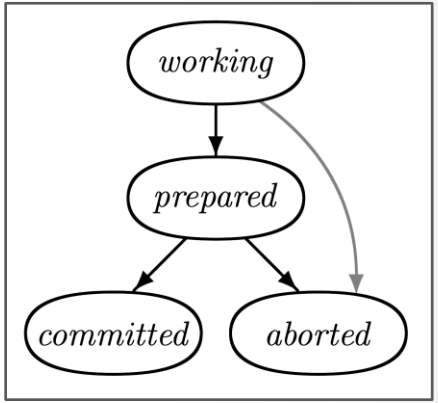
Examples:

→ Two-Phase Commit (1970s)
→ Three-Phase Commit (1983)
→ Viewstamped Replication (1988)
→ Paxos (1989)
→ ZAB (2008?)
→ Raft (2013)

ATOMIC COMMIT PROTOCOL

Resource Managers (RMs):

→ Execute on different nodes
→ Coordinate to decide fate of a txn.



Properties of the Commit Protocol:

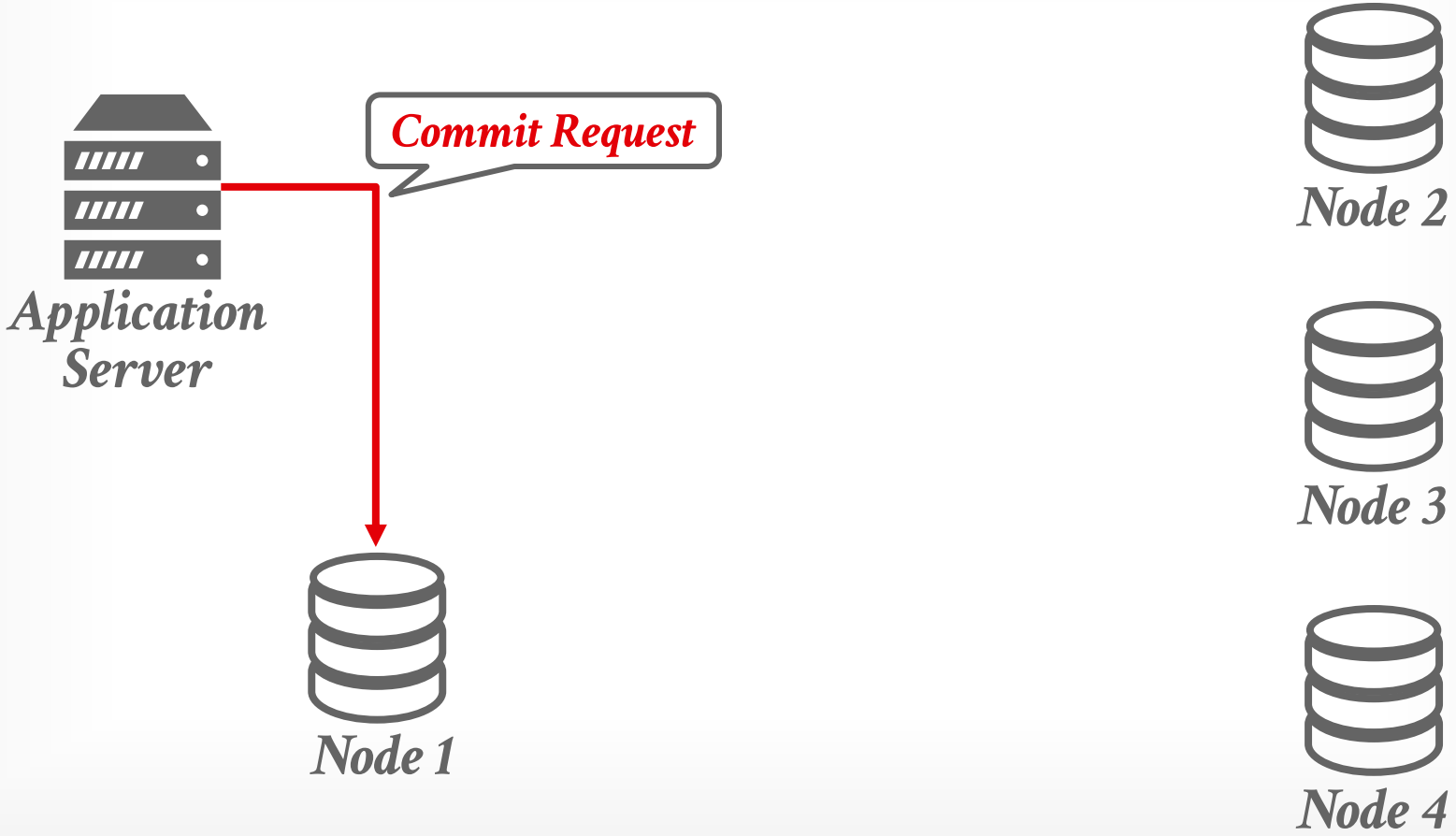
→ **Stability**: Once fate is decided, it cannot be changed.
→ **Consistency**: All RMs end in the same state.

Assumes Liveness:

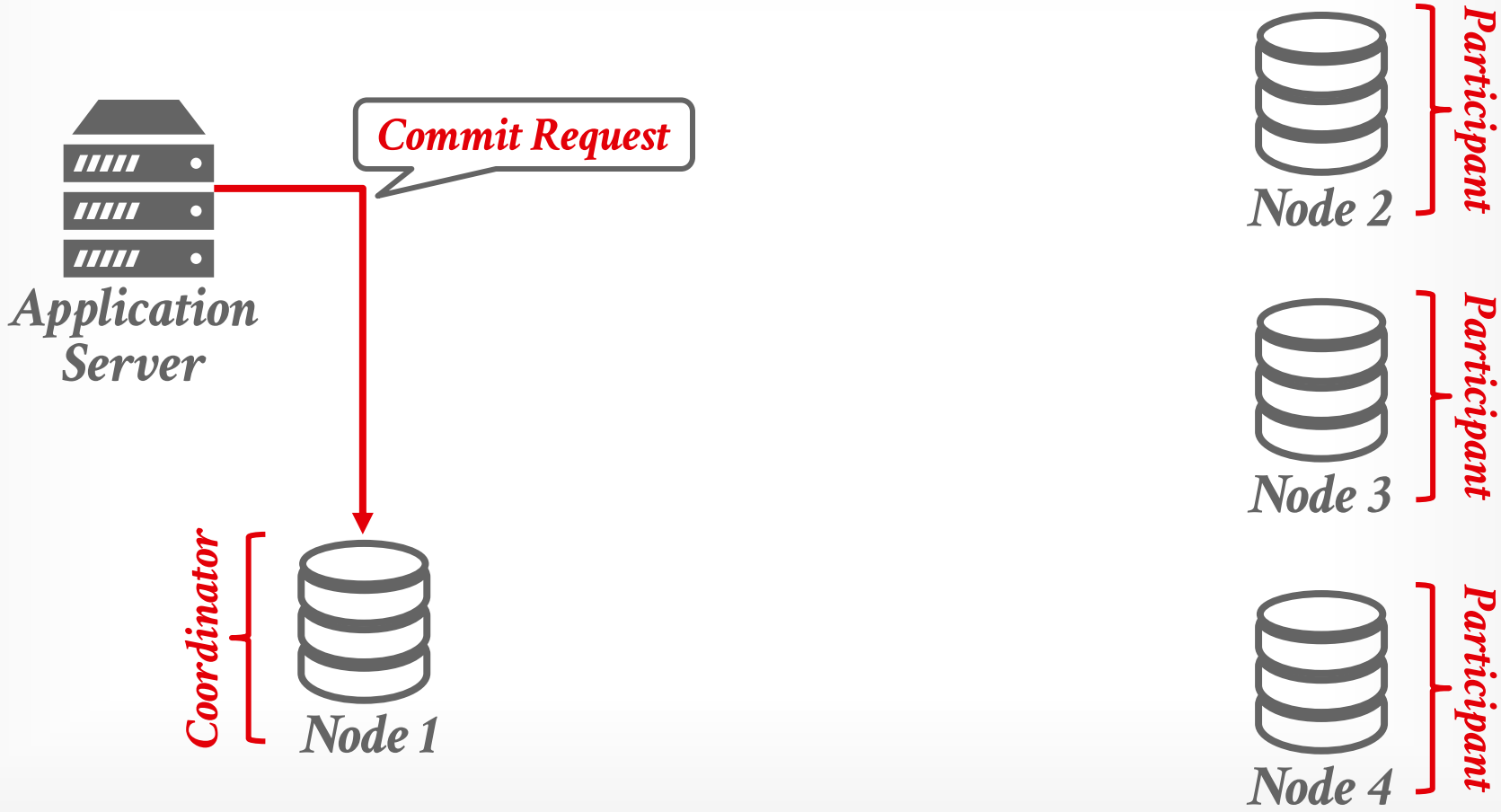
→ There is some way of progressing forward.
→ Enough nodes are alive and connected for the duration of the protocol.

Source: Jim Gray + Lesile Lamport

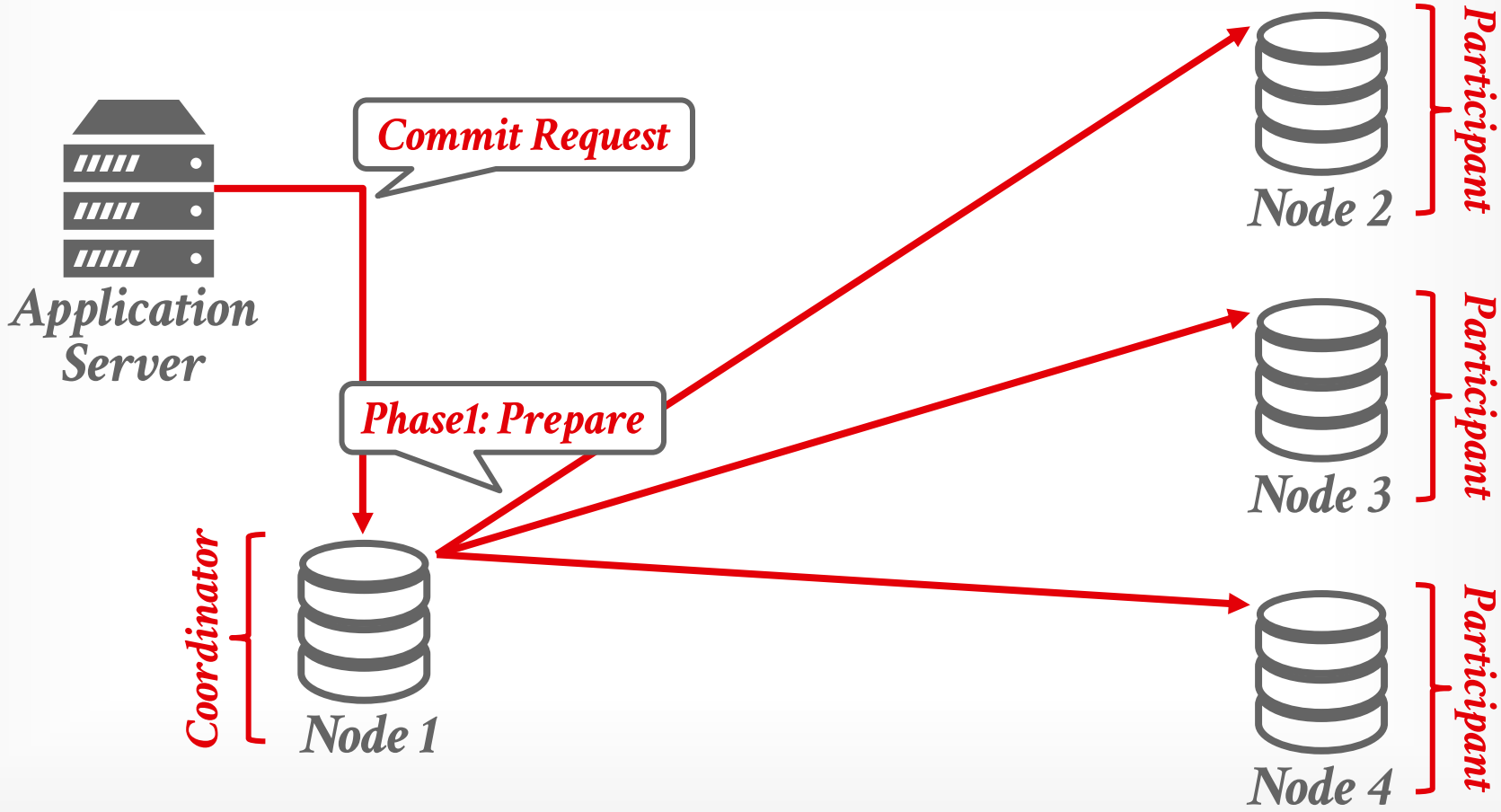
TWO-PHASE COMMIT (SUCCESS)



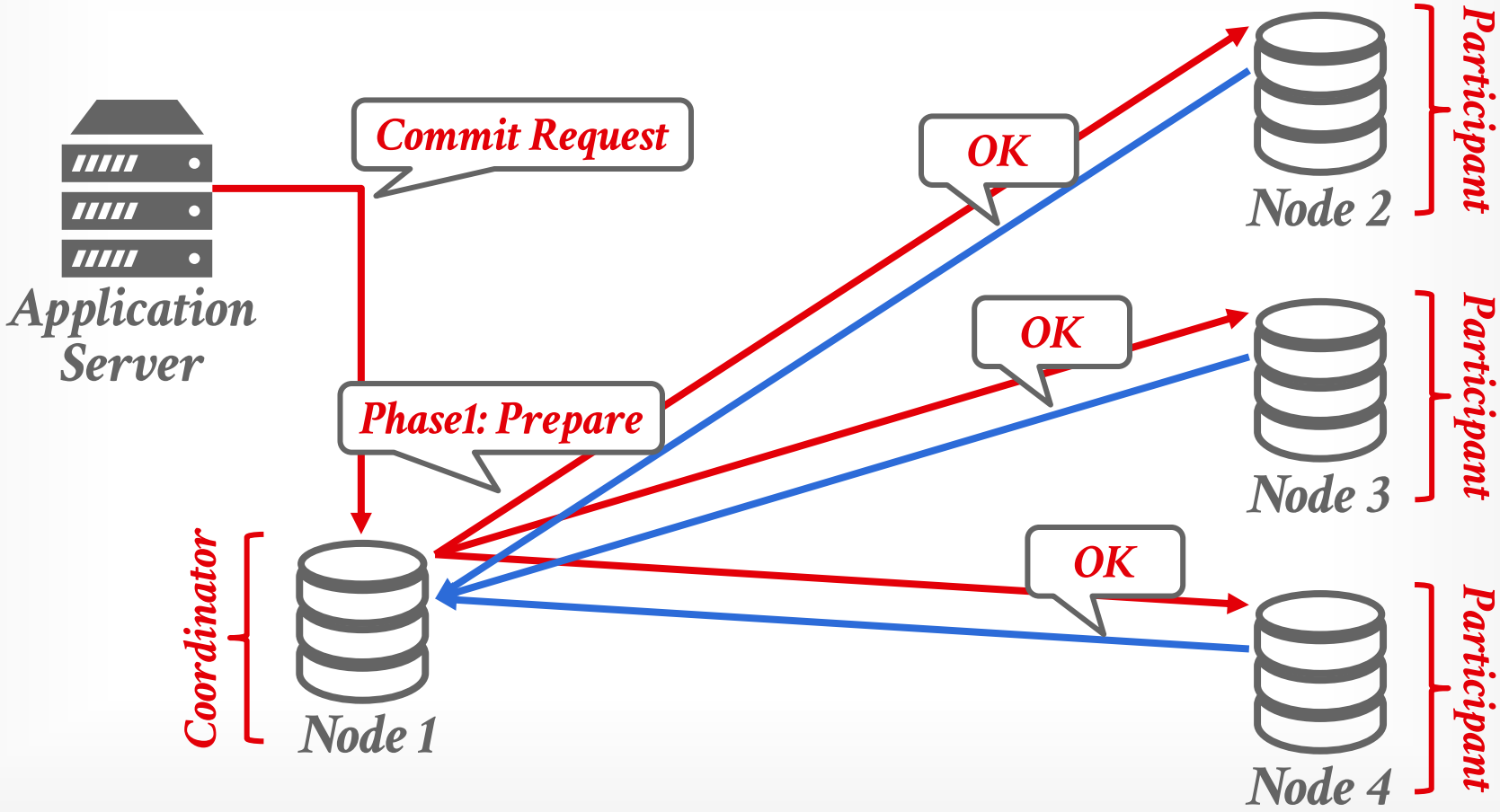
TWO-PHASE COMMIT (SUCCESS)



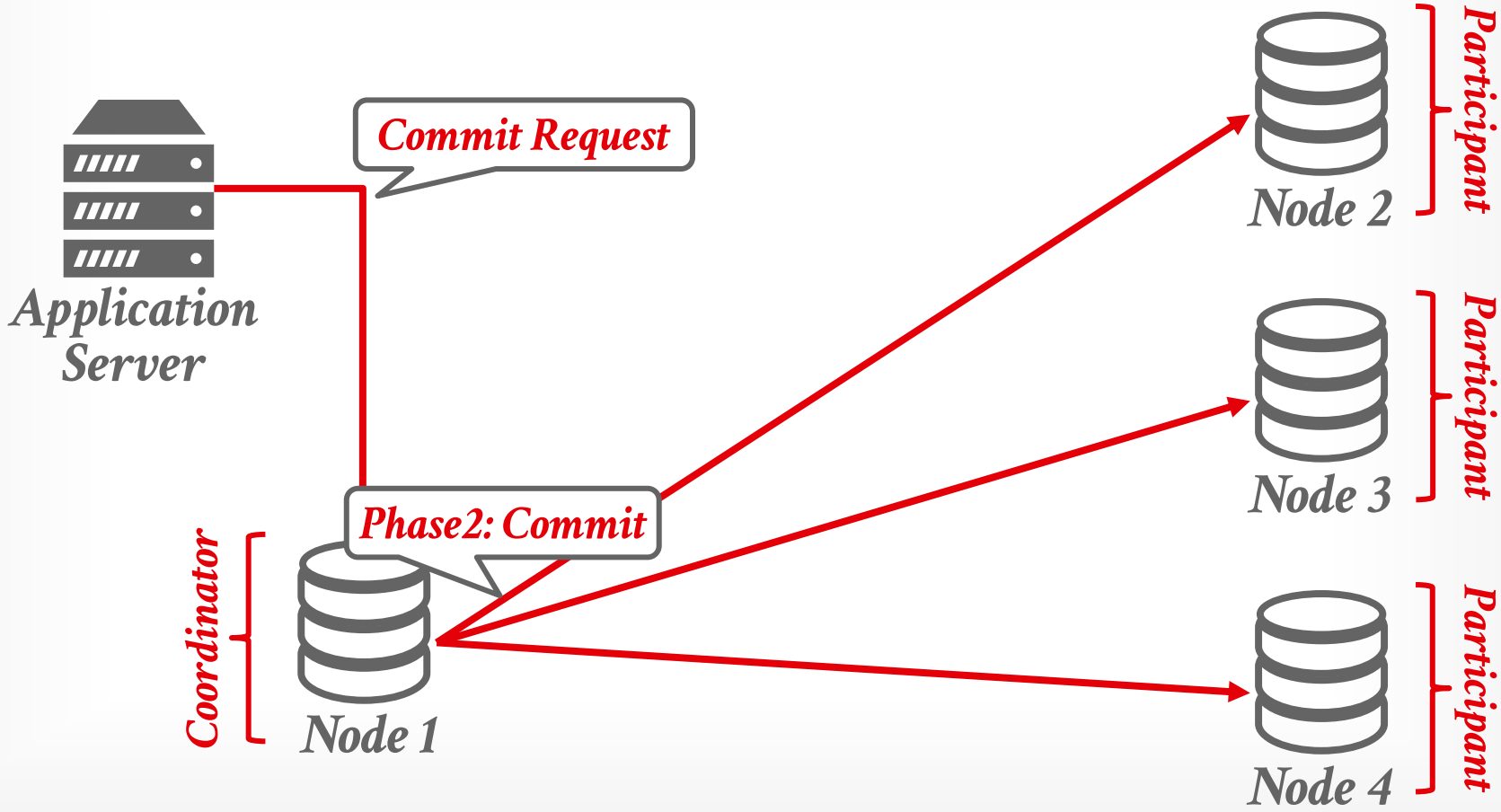
TWO-PHASE COMMIT (SUCCESS)



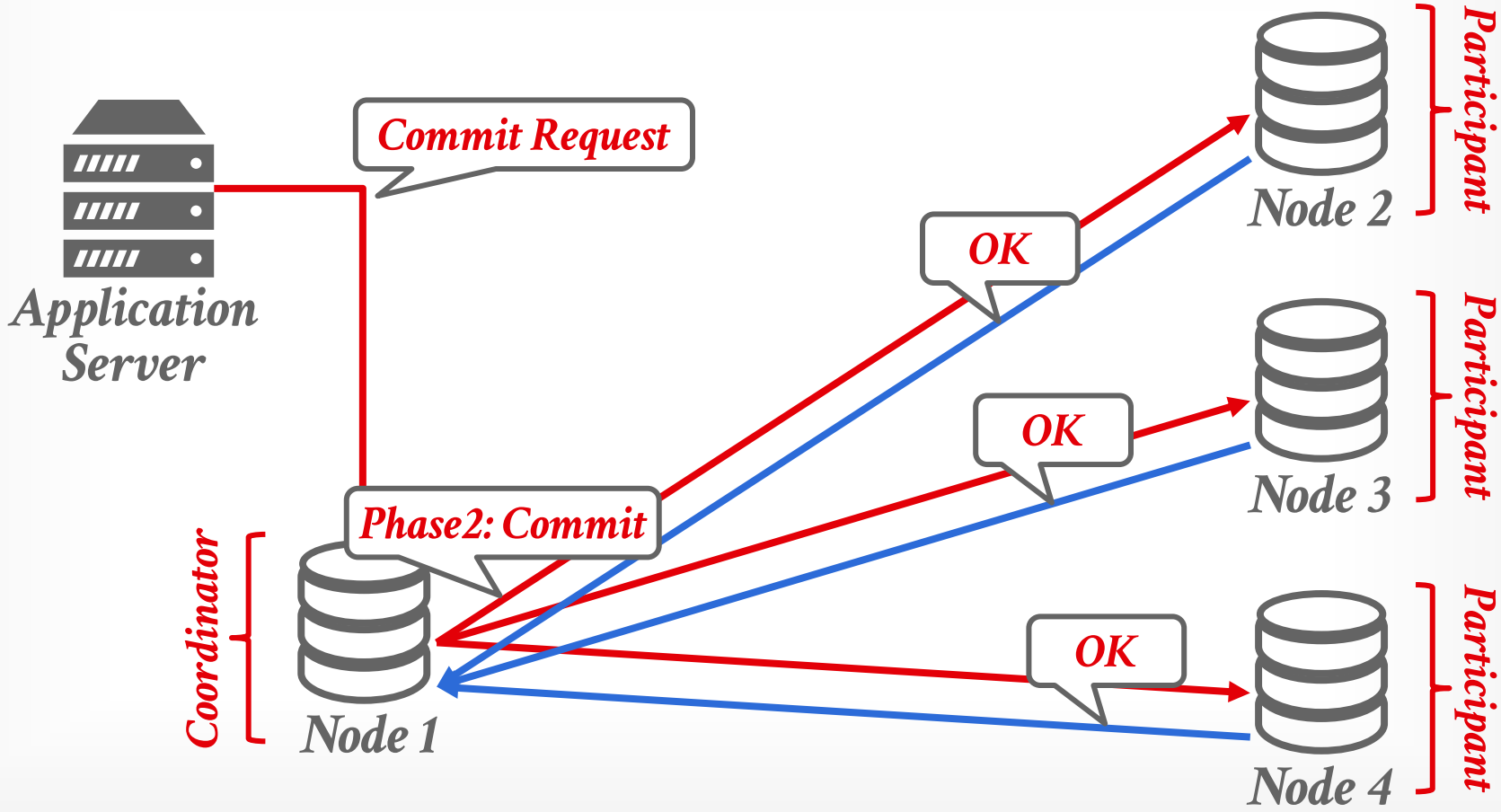
TWO-PHASE COMMIT (SUCCESS)



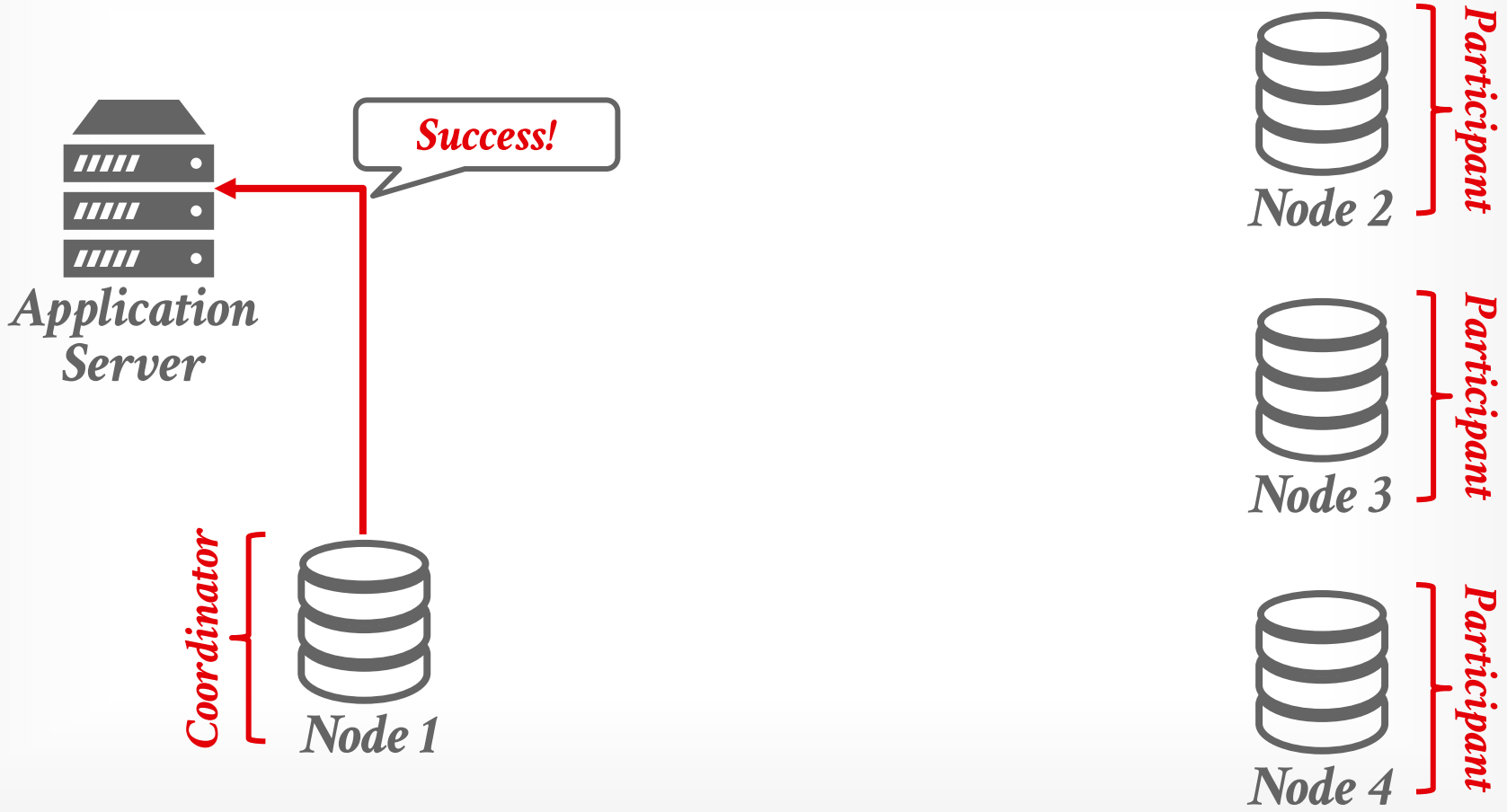
TWO-PHASE COMMIT (SUCCESS)



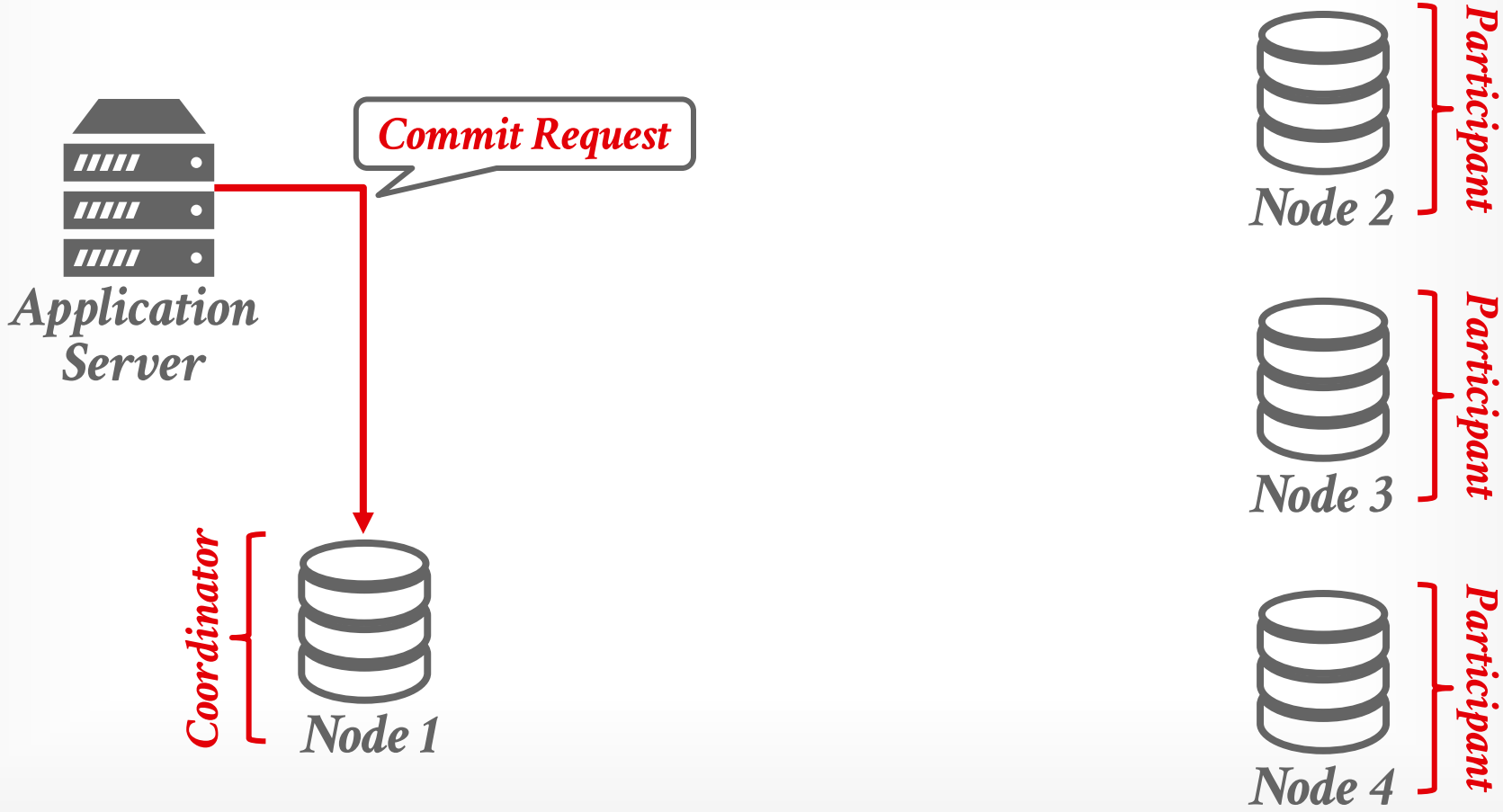
TWO-PHASE COMMIT (SUCCESS)



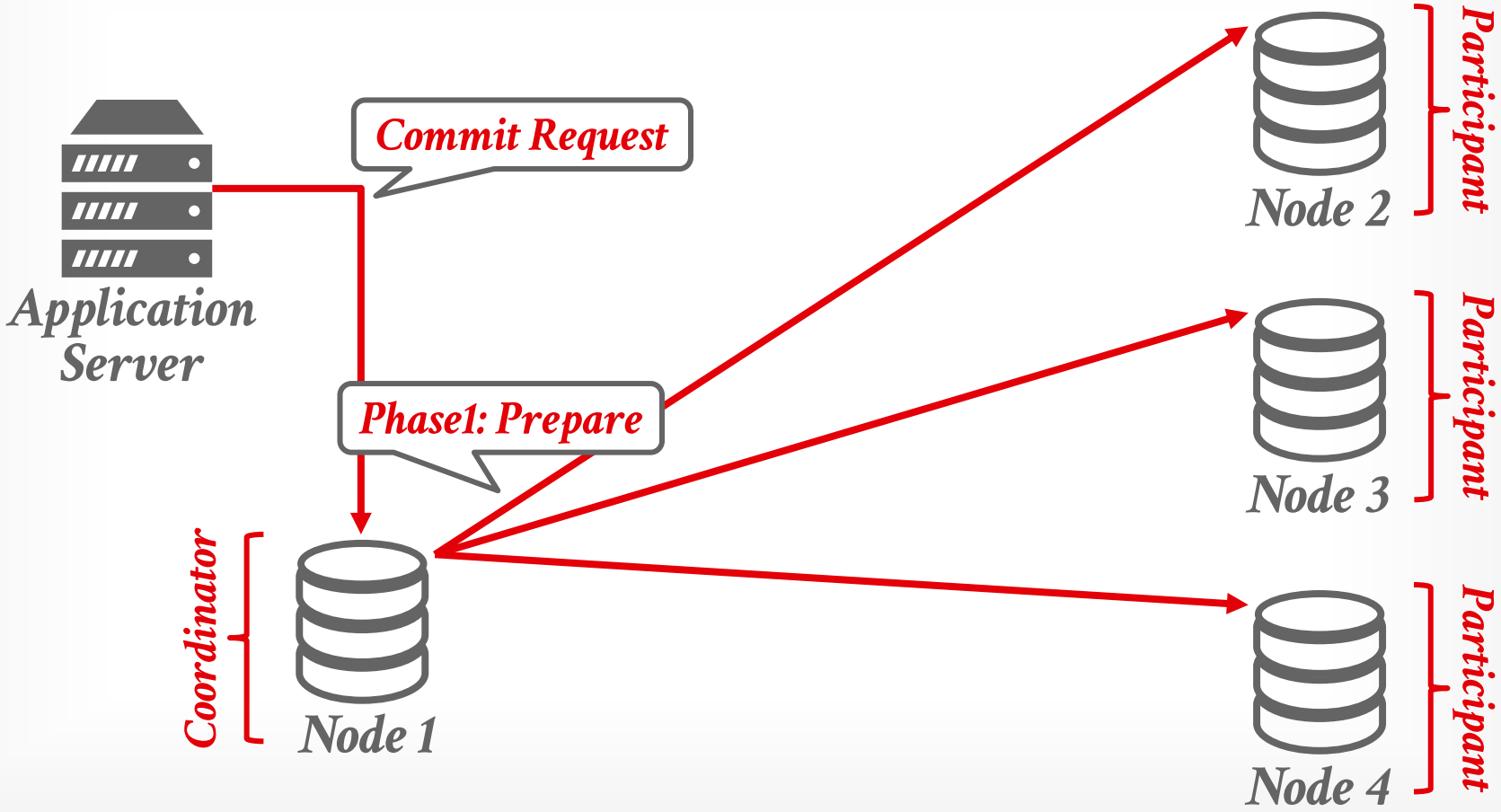
TWO-PHASE COMMIT (SUCCESS)



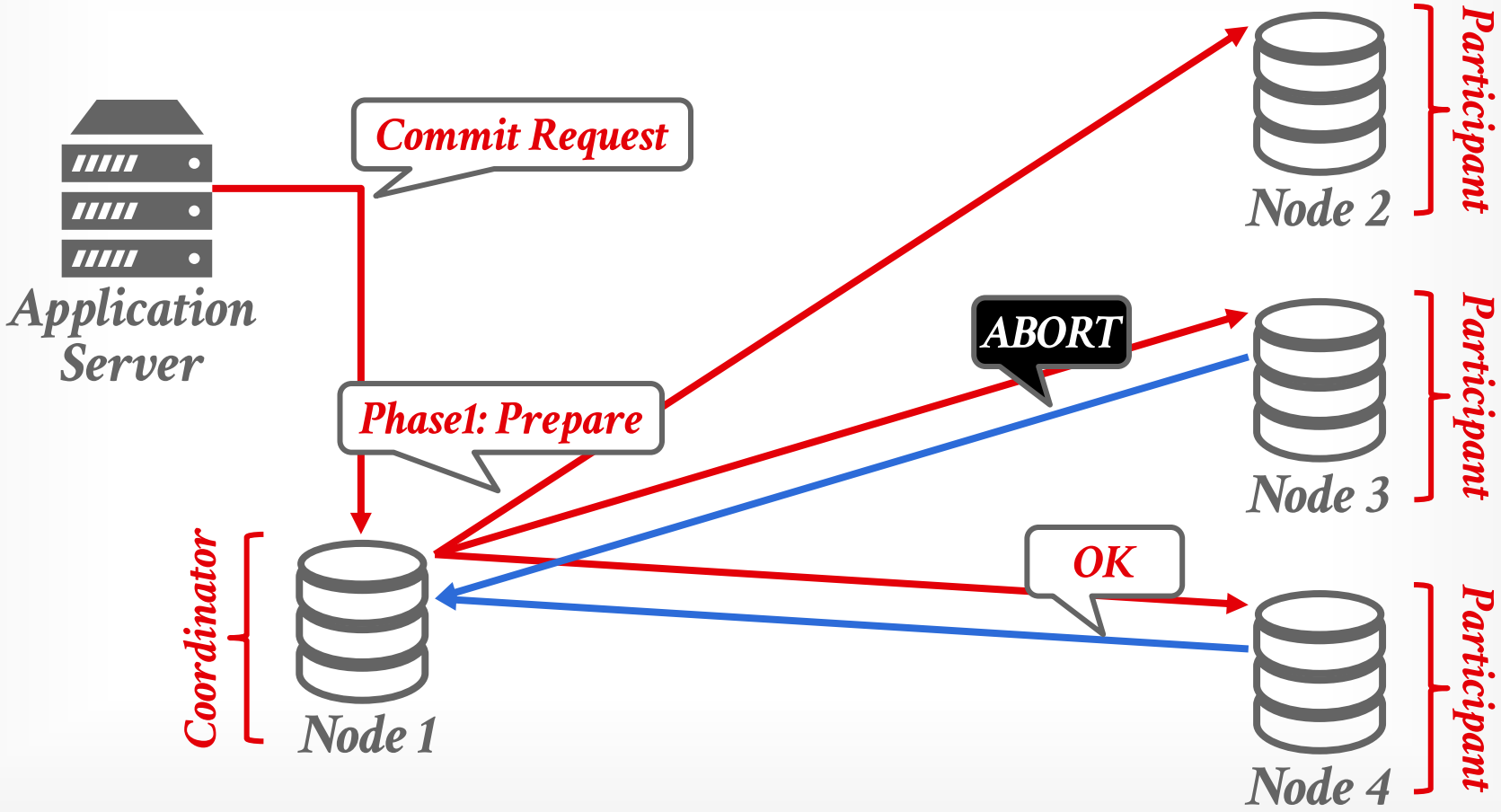
TWO-PHASE COMMIT (ABORT)



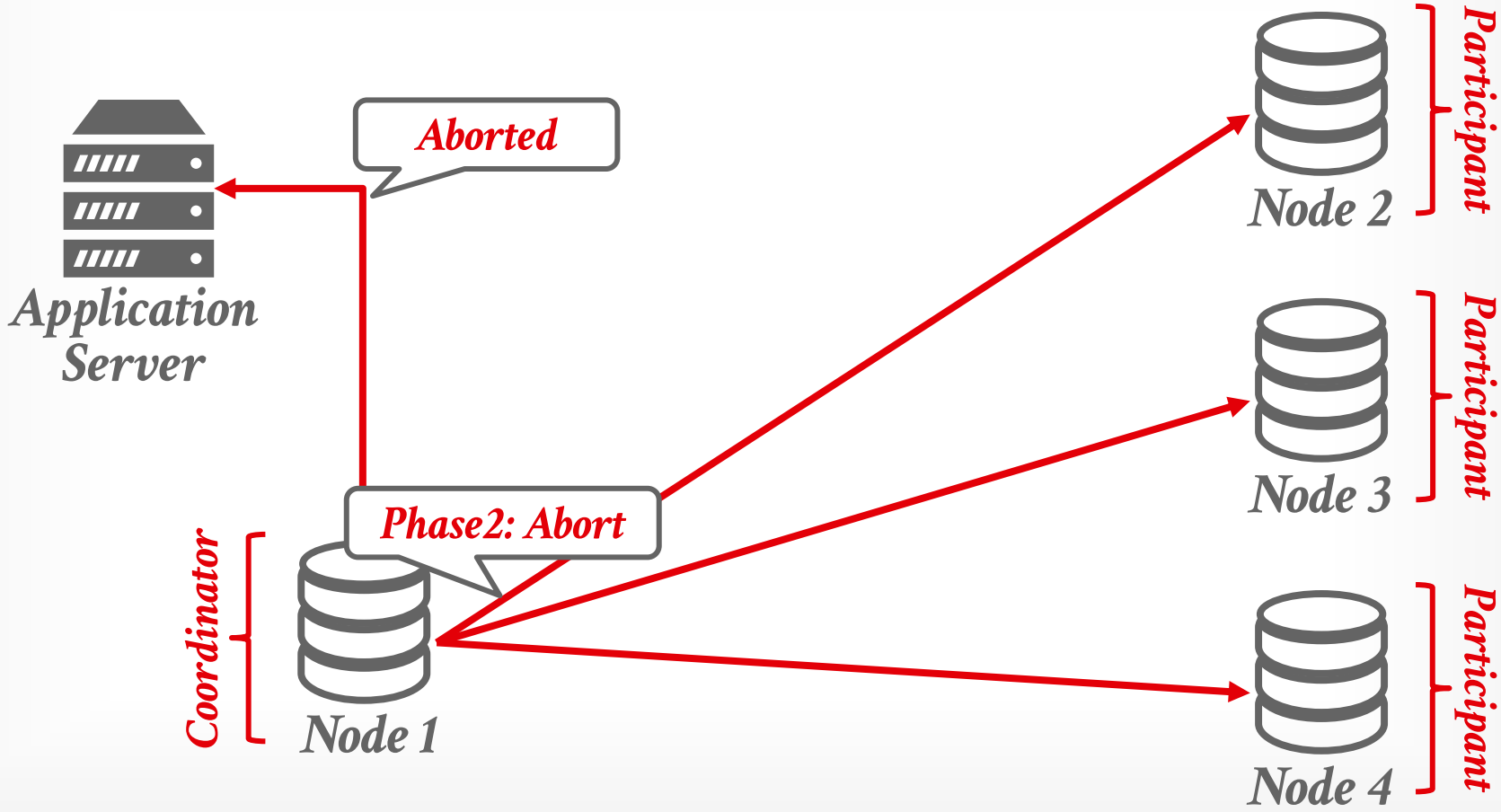
TWO-PHASE COMMIT (ABORT)



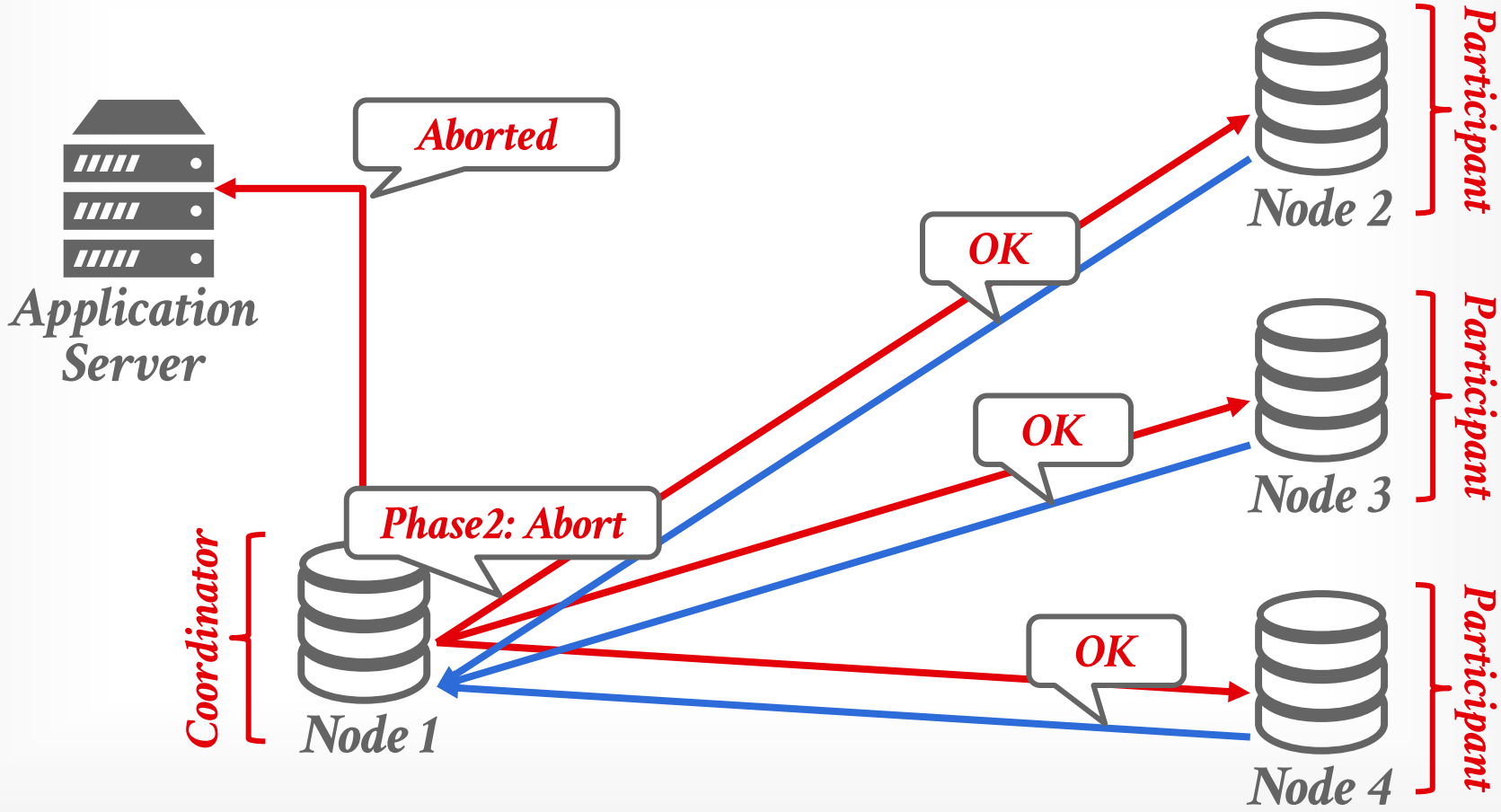
TWO-PHASE COMMIT (ABORT)



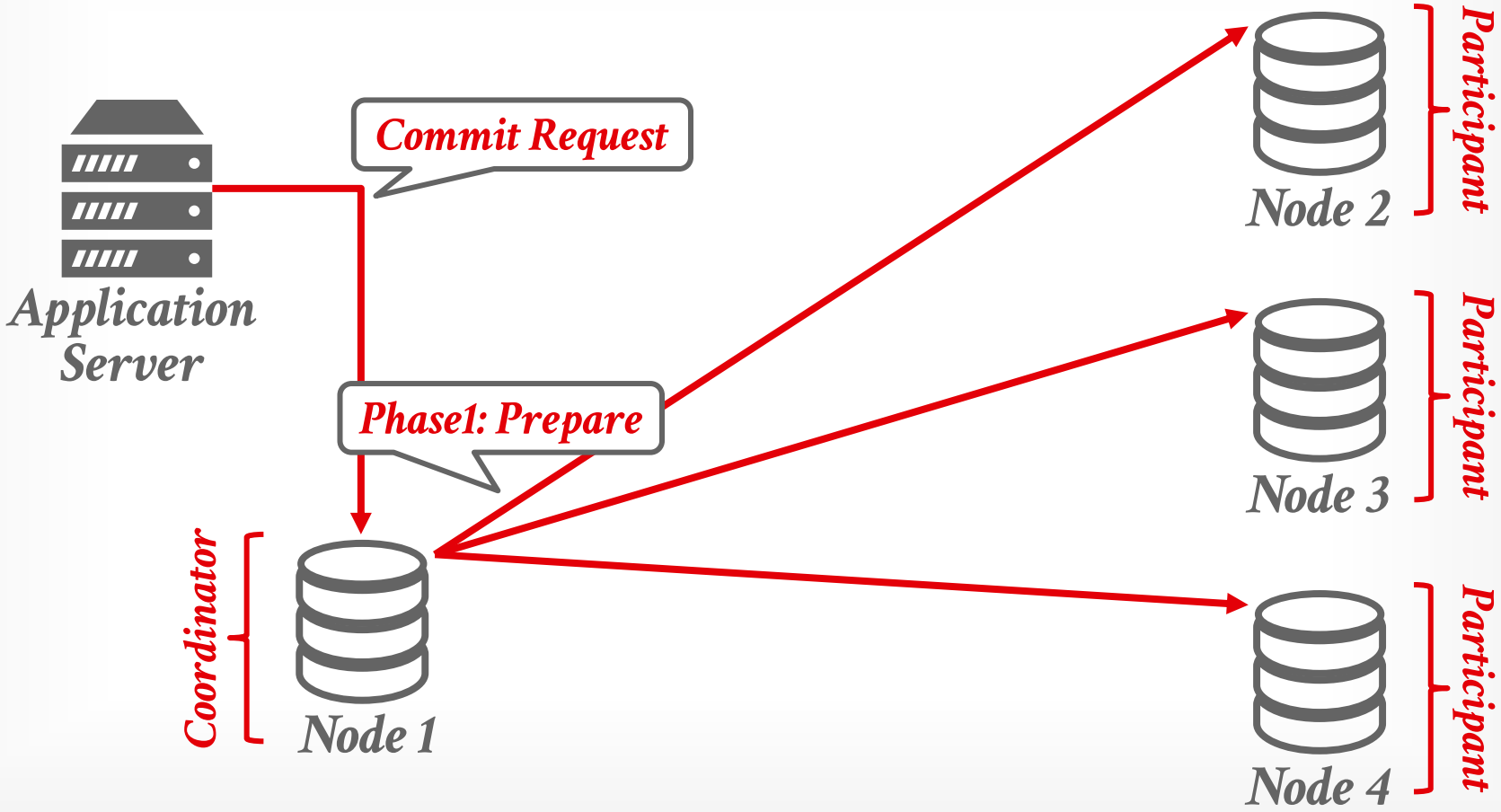
TWO-PHASE COMMIT (ABORT)



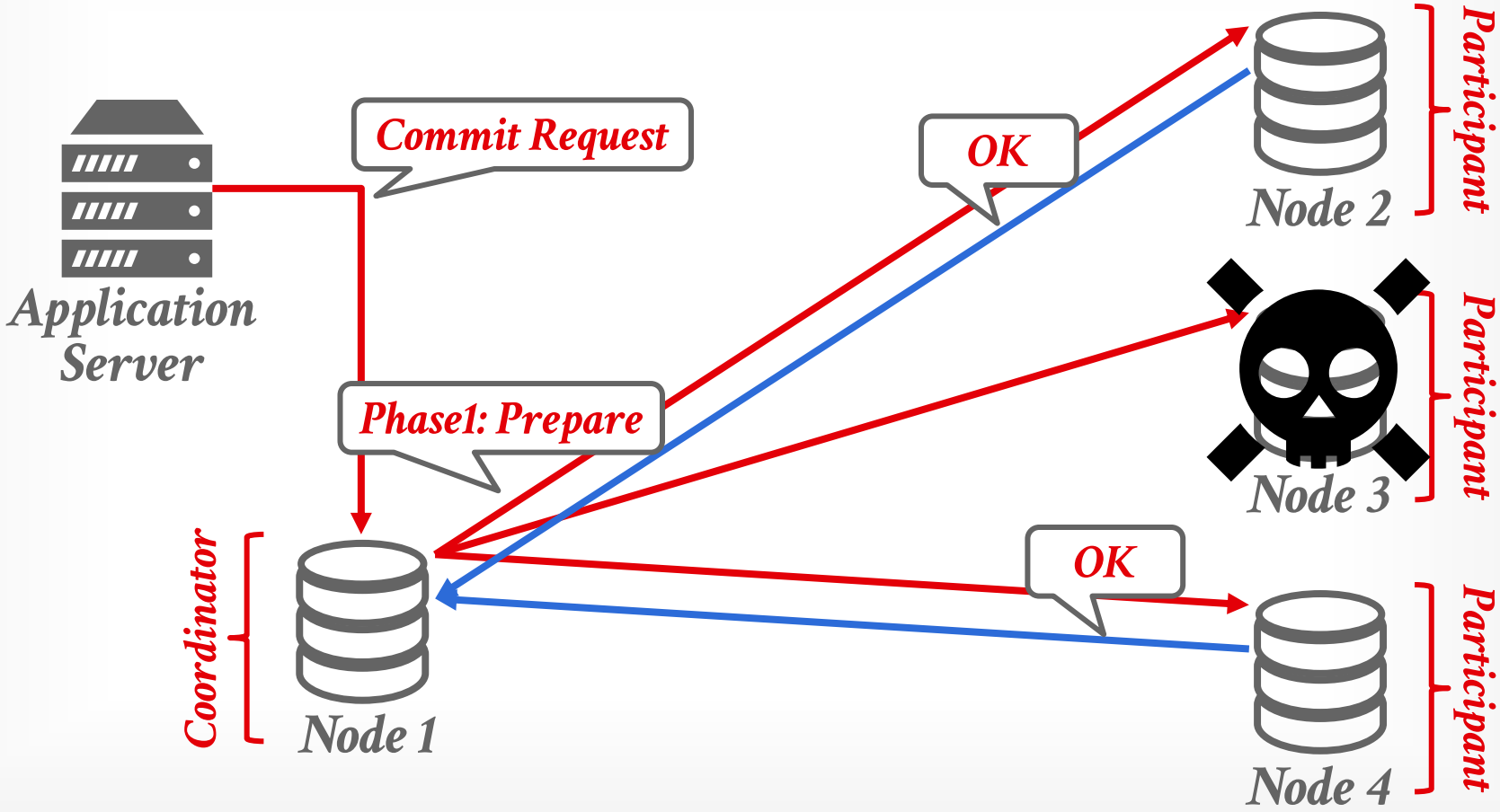
TWO-PHASE COMMIT (ABORT)



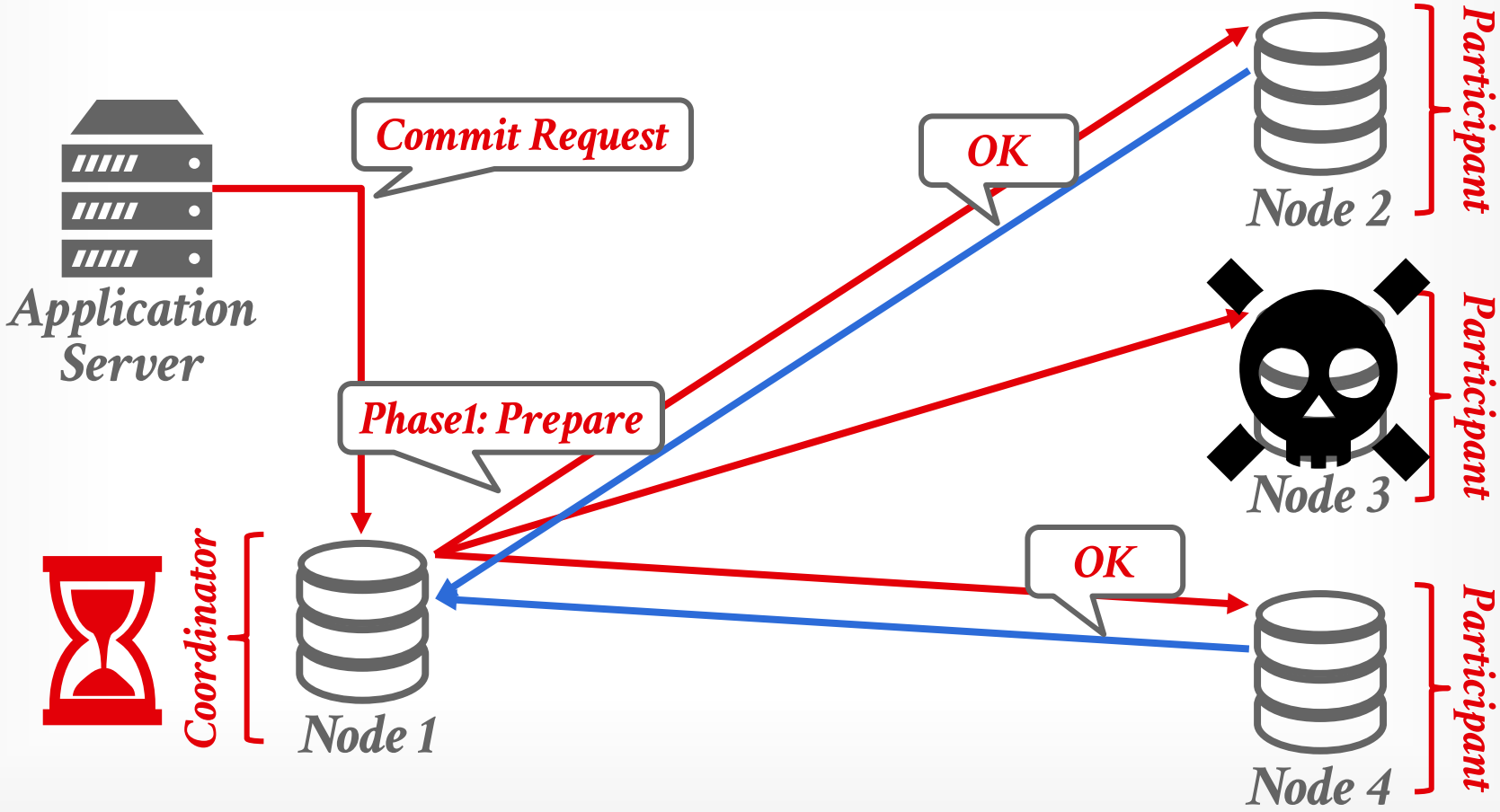
TWO-PHASE COMMIT (ABORT)



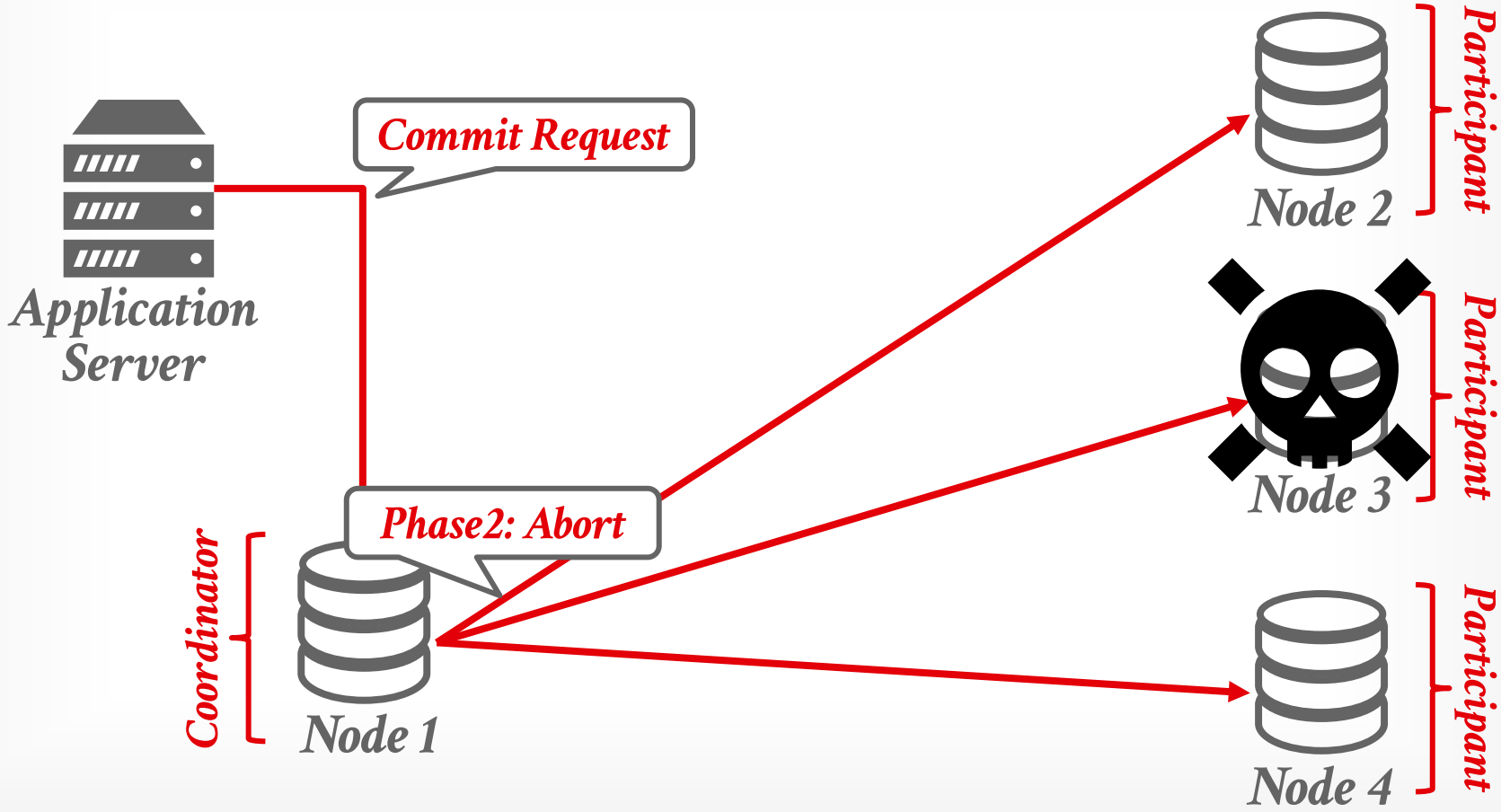
TWO-PHASE COMMIT (ABORT)



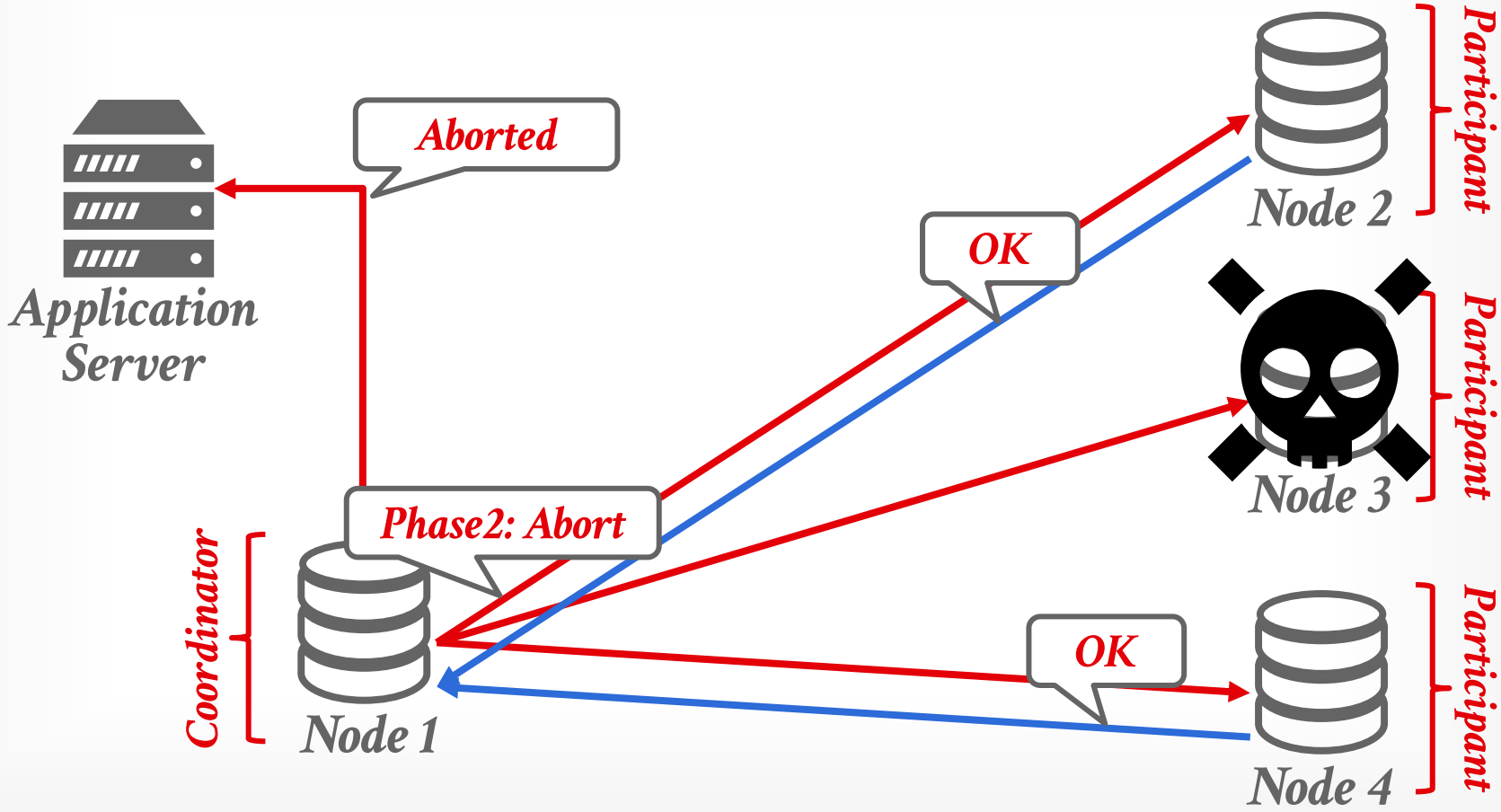
TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT

Each node records the inbound/outbound messages and outcome of each phase in a non-volatile storage log.

On recovery, examine the log for 2PC messages:

→ If local txn in prepared state, contact coordinator.
→ If local txn not in prepared, abort it.
→ If local txn was committing and node is the coordinator, send **COMMIT** message to nodes.

TWO-PHASE COMMIT FAILURES

What happens if the coordinator crashes?

→ Participants must decide what to do after a timeout
(this only applies if the participants know of all other participants).

→ System is not available during this time.

What happens if the participant crashes?

→ Coordinator assumes that it responded with an abort if it has not sent an acknowledgement yet.

→ Again, nodes use a timeout to determine whether a participant is dead.

2PC OPTIMIZATIONS

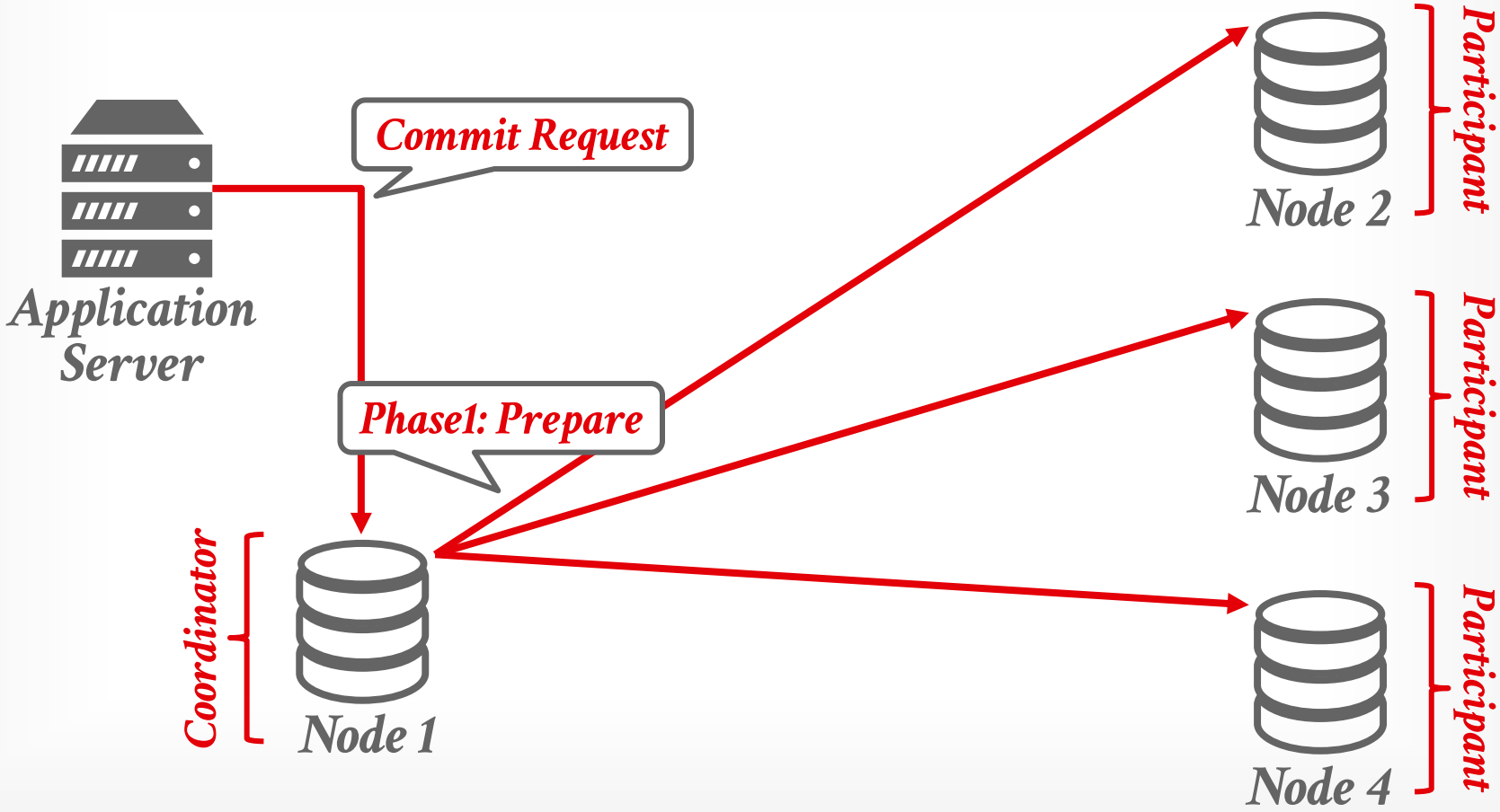
Early Prepare Voting *(Rare)*

→ If you send a query/request to a remote node that you know will be the last one to execute in this txn, then that node will also return their vote for the prepare phase with the query result.

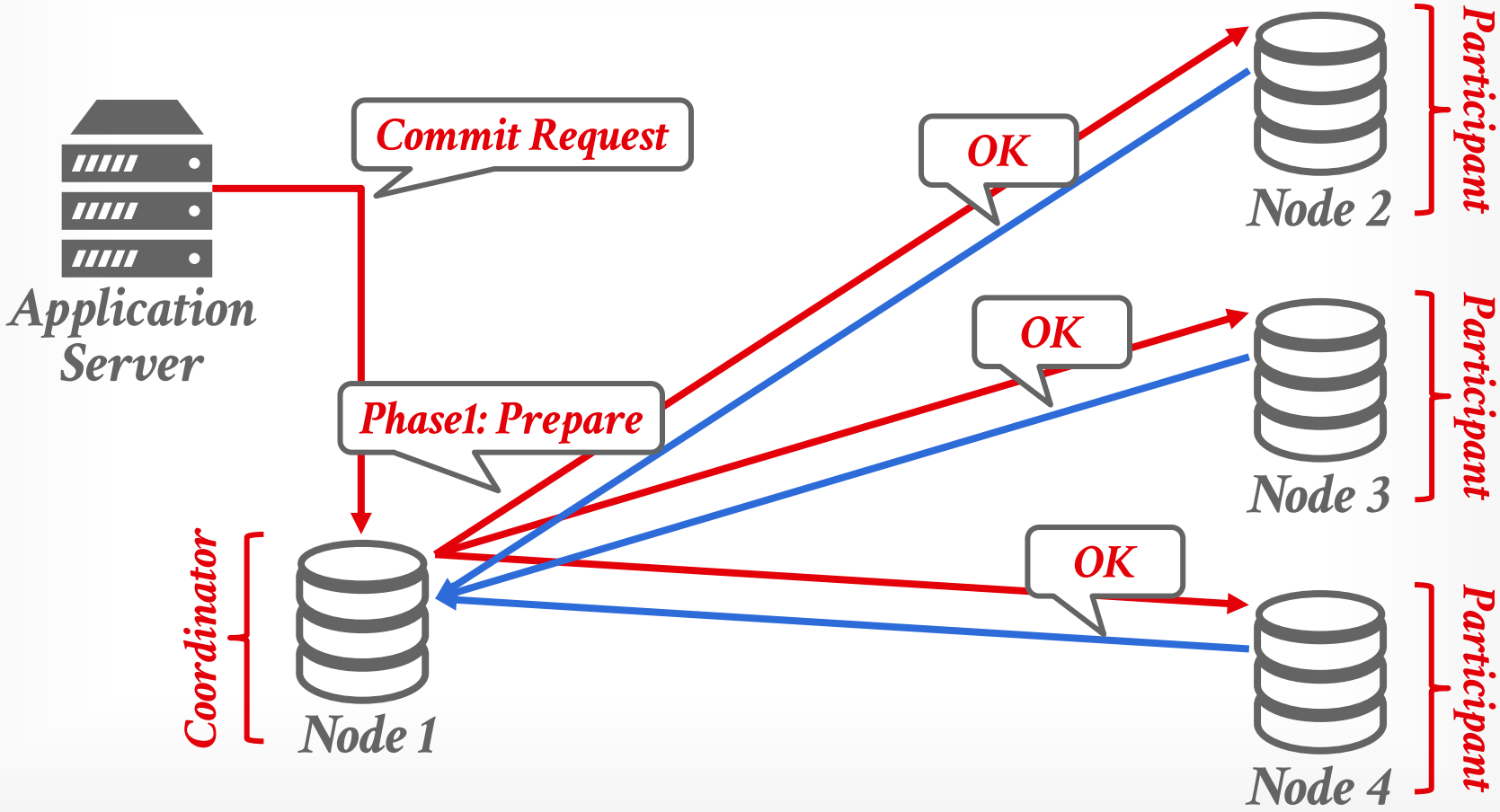
Early Ack After Prepare *(Common)*

→ If all nodes vote to commit a txn, the coordinator can send the client an acknowledgement that their txn was successful before the commit phase finishes.

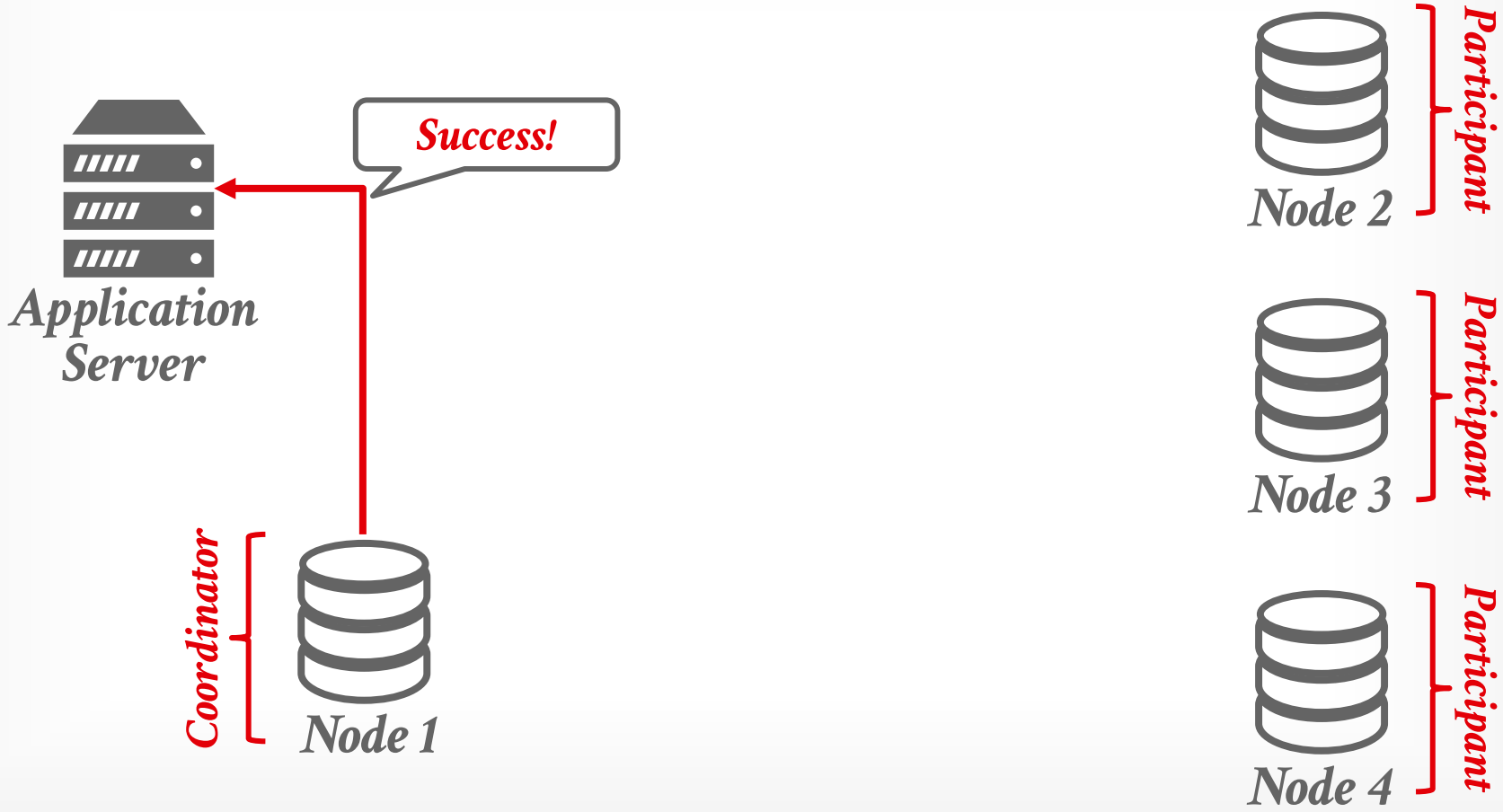
EARLY ACKNOWLEDGEMENT



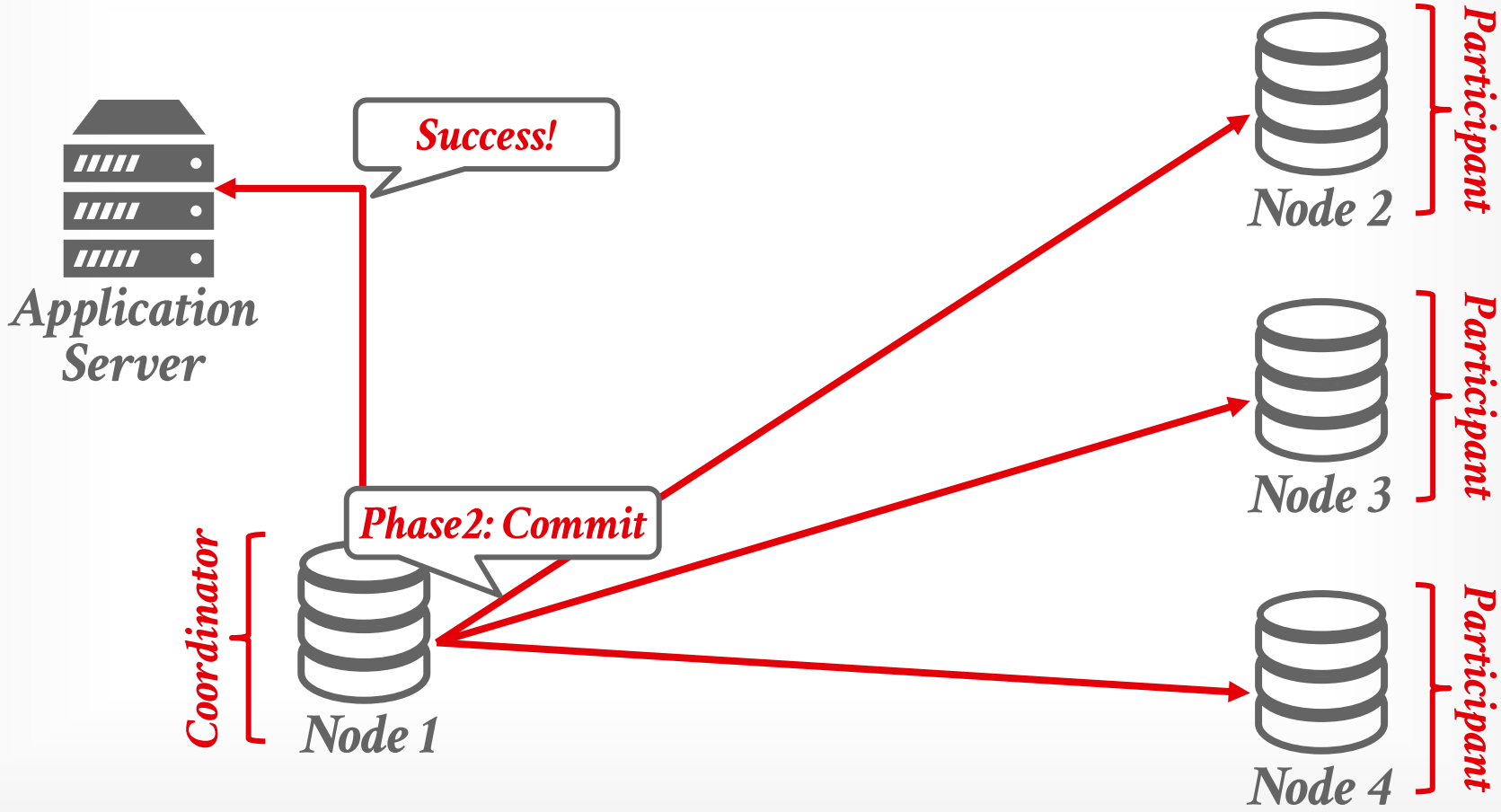
EARLY ACKNOWLEDGEMENT



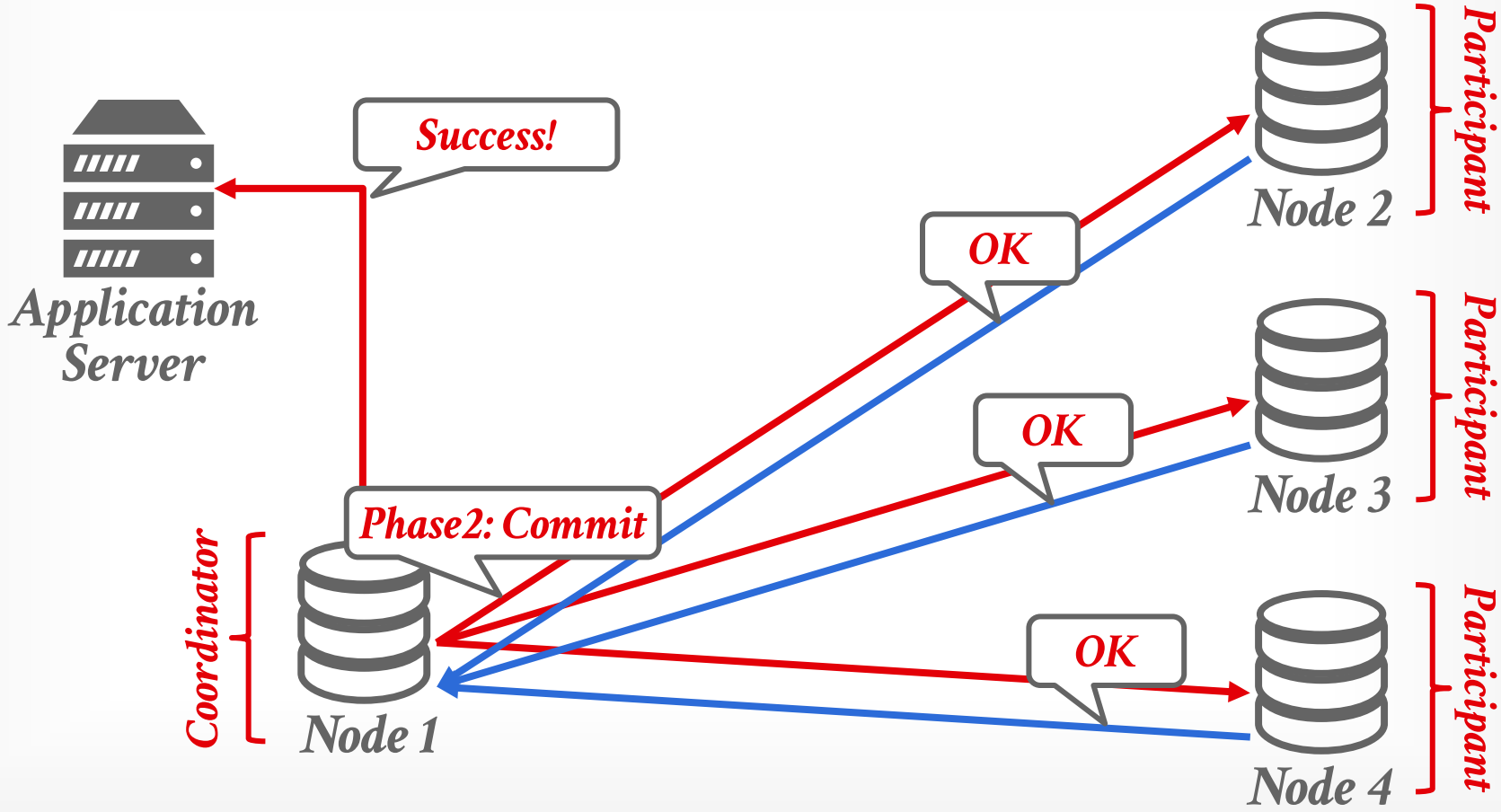
EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT

Application Server

Participant

Node 2

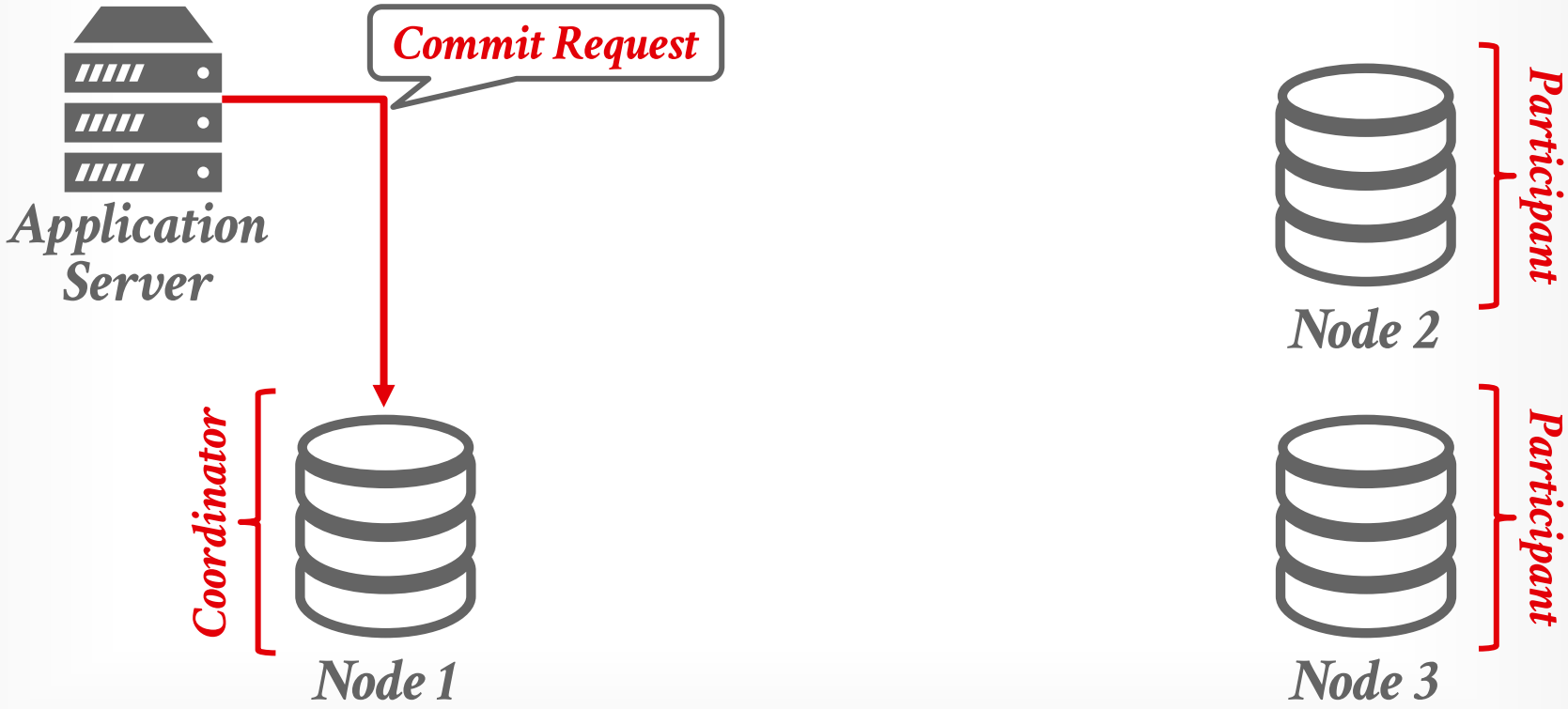
Coordinator

Node 1

Participant

Node 3

EARLY ACKNOWLEDGEMENT



PAXOS

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays in the best case.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

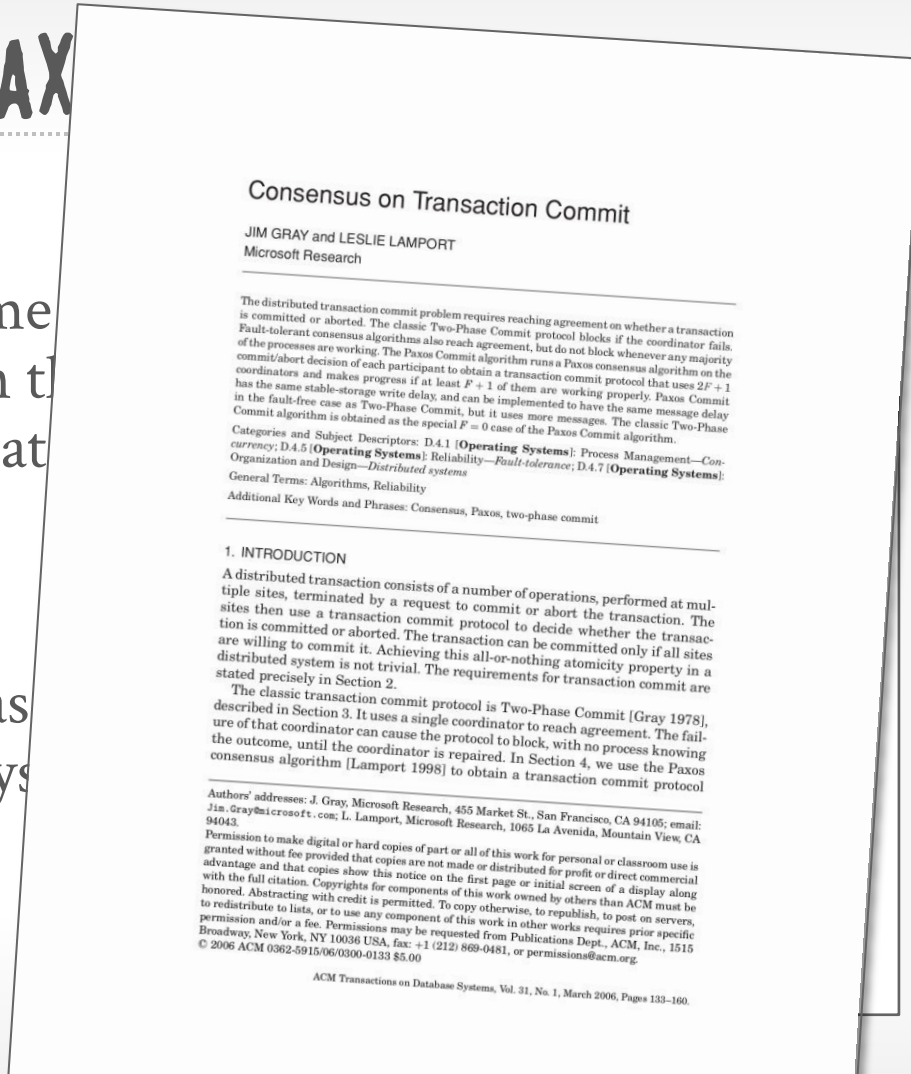
Keith Marzullo
University of California, San Diego

Authors' address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1998 ACM 0000-0000/98/0000-0000 $00.00

PAX

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then t participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays the best case.



Consensus on Transaction Commit

JIM GRAY and LESLIE LAMPORT
Microsoft Research

The distributed transaction commit problem requires reaching agreement on whether a transaction is committed or aborted. The classic Two-Phase Commit protocol blocks if the coordinator fails. Fault-tolerant consensus algorithms also reach agreement, but do not block whenever any majority of the processes are working. The Paxos Commit algorithm runs a Paxos consensus algorithm on the commit/abort decision of each participant to obtain a transaction commit protocol that uses $2F + 1$ coordinators and makes progress if at least $F + 1$ of them are working properly. Paxos Commit has the same stable-storage write delay, and can be implemented to have the same message delay in the fault-free case as Two-Phase Commit, but it uses more messages. The classic Two-Phase Commit algorithm is obtained as the special $F = 0$ case of the Paxos Commit algorithm.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Concurrency*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Consensus, Paxos, two-phase commit

1. INTRODUCTION

A distributed transaction consists of a number of operations, performed at multiple sites, terminated by a request to commit or abort the transaction. The sites then use a transaction commit protocol to decide whether the transaction is committed or aborted. The transaction can be committed only if all sites are willing to commit it. Achieving this all-or-nothing atomicity property in a distributed system is not trivial. The requirements for transaction commit are stated precisely in Section 2.

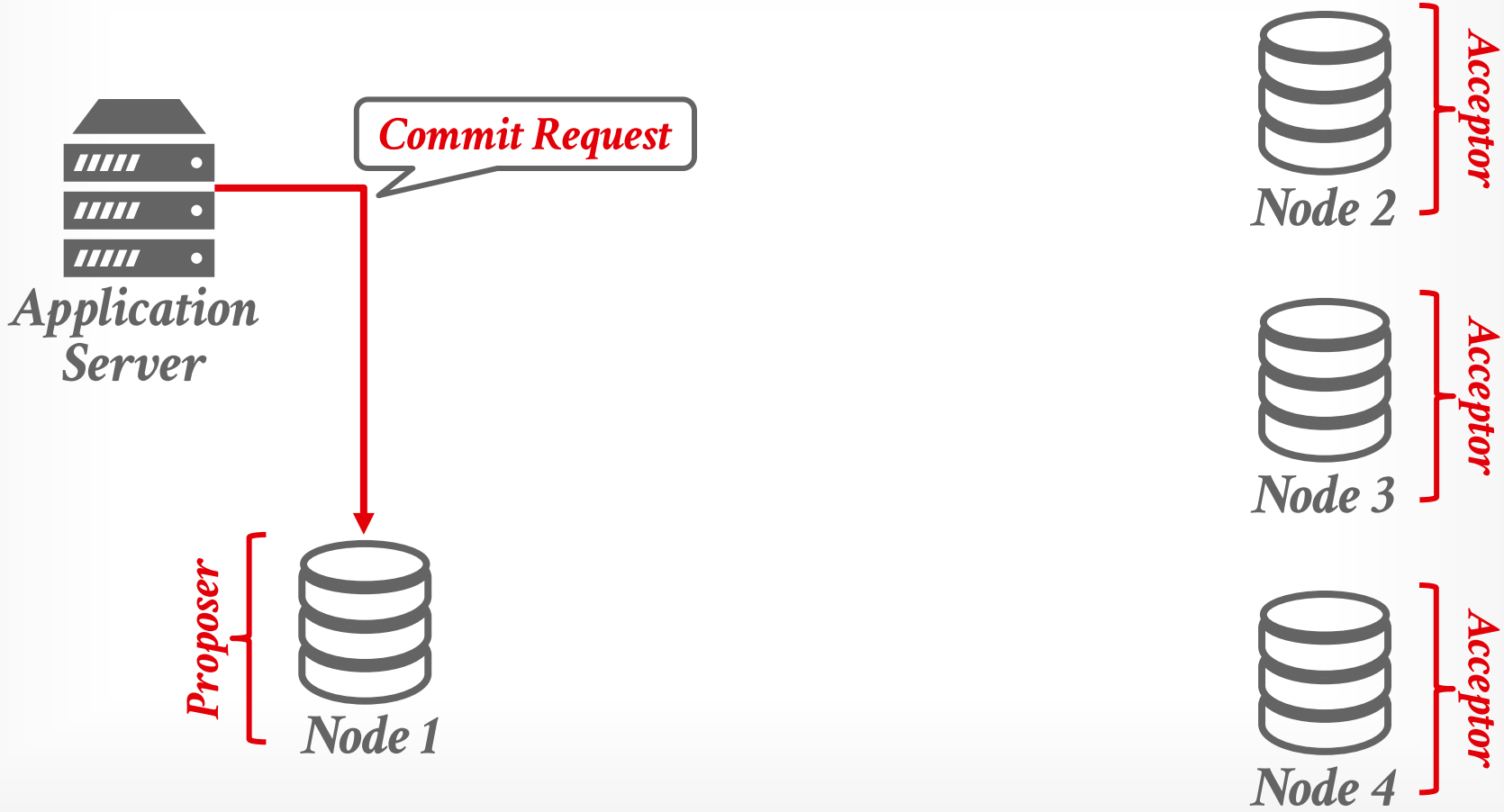
The classic transaction commit protocol is Two-Phase Commit [Gray 1978], described in Section 3. It uses a single coordinator to reach agreement. The failure of that coordinator can cause the protocol to block, with no process knowing the outcome, until the coordinator is repaired. In Section 4, we use the Paxos consensus algorithm [Lamport 1998] to obtain a transaction commit protocol

Authors' addresses: J. Gray, Microsoft Research, 455 Market St., San Francisco, CA 94105; email: Jim.Gray@microsoft.com; L. Lamport, Microsoft Research, 1065 La Avenida, Mountain View, CA 94043.

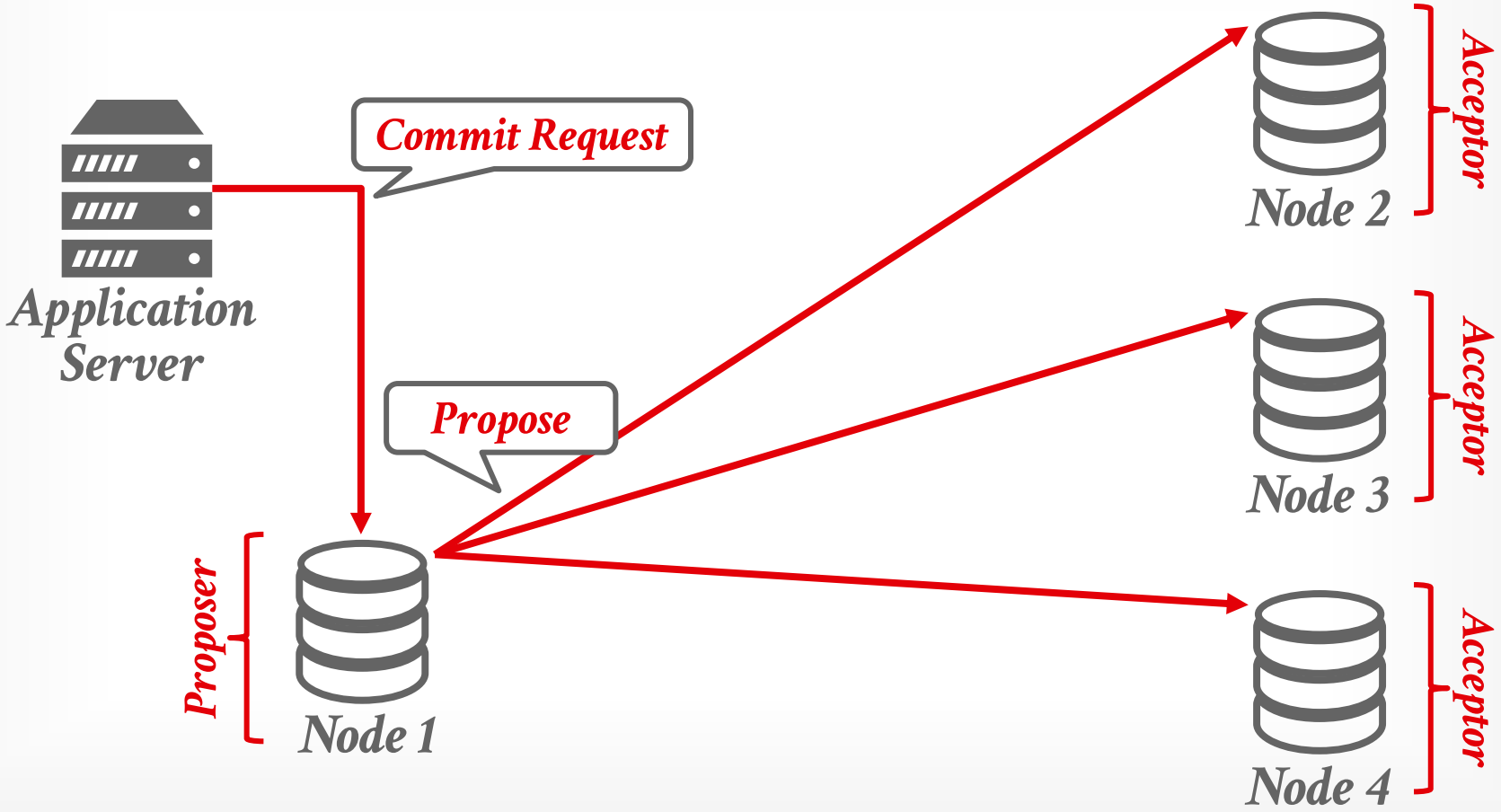
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0362-5915/06/0300-0133 $5.00

ACM Transactions on Database Systems, Vol. 31, No. 1, March 2006, Pages 133–160.

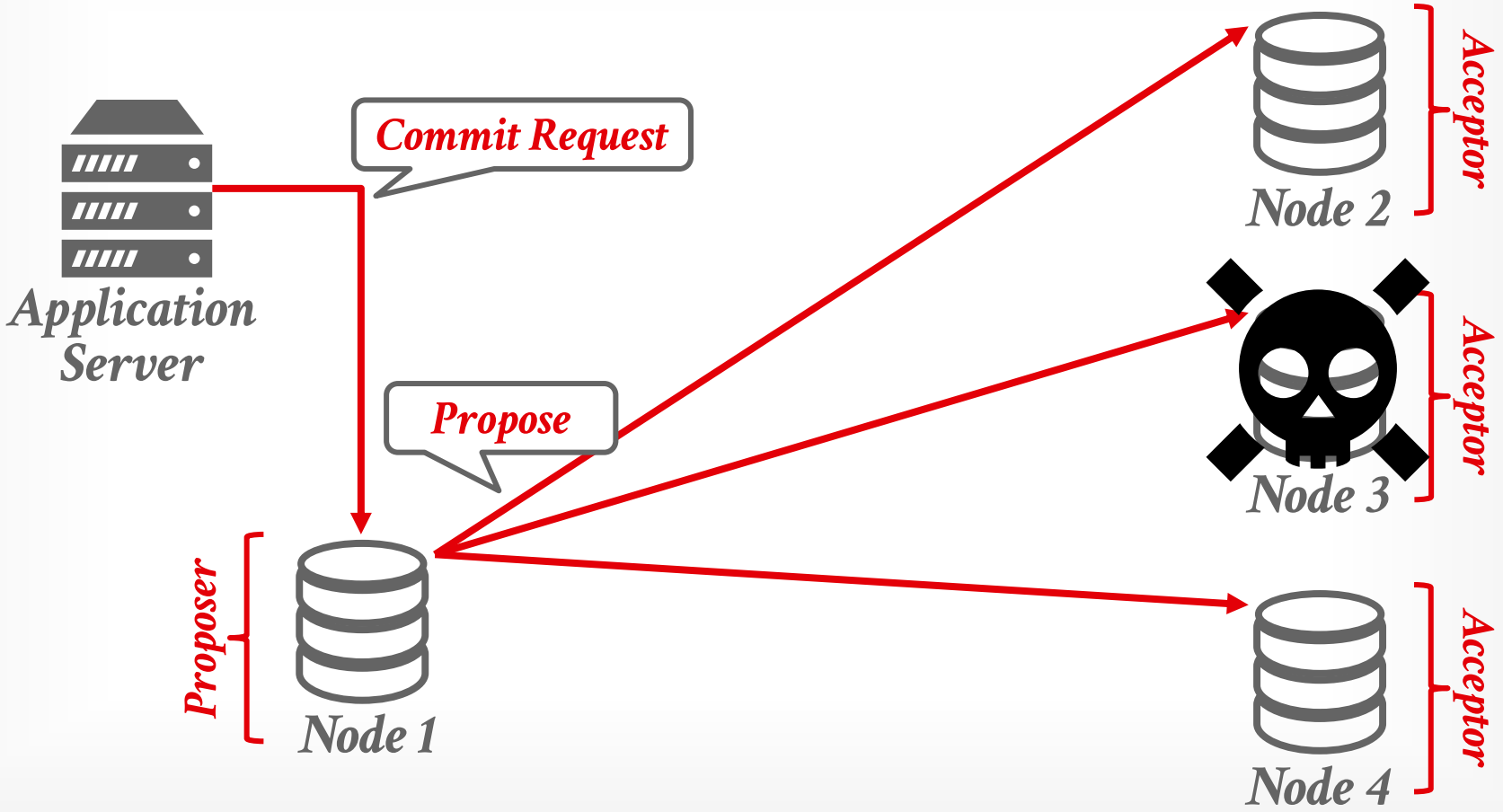
PAXOS


Commit Request
Application Server
Proposer
Node 1
Node 2
Acceptor
Node 3
Acceptor
Node 4
Acceptor

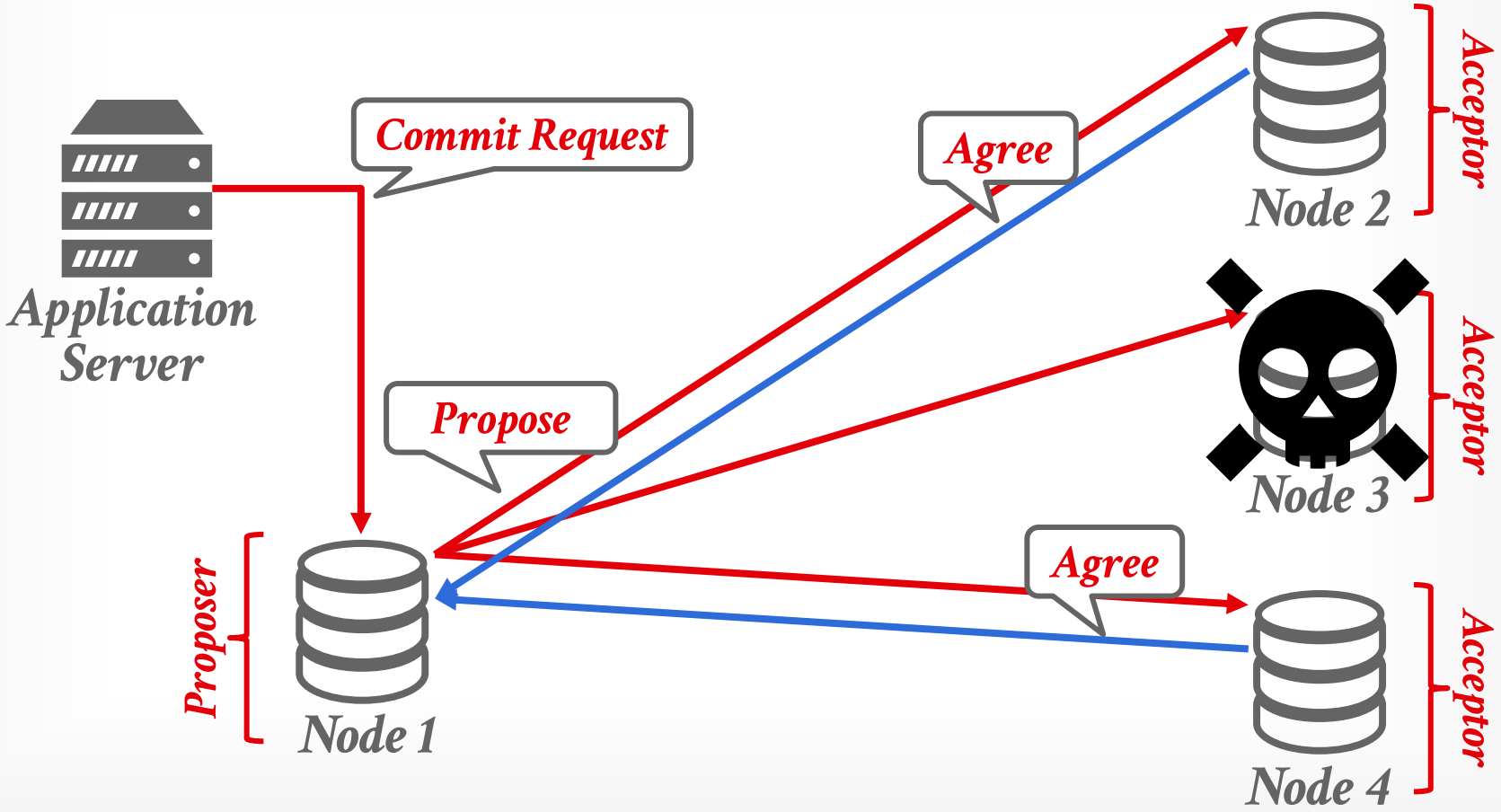
PAXOS


Commit Request
Application Server
Propose
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

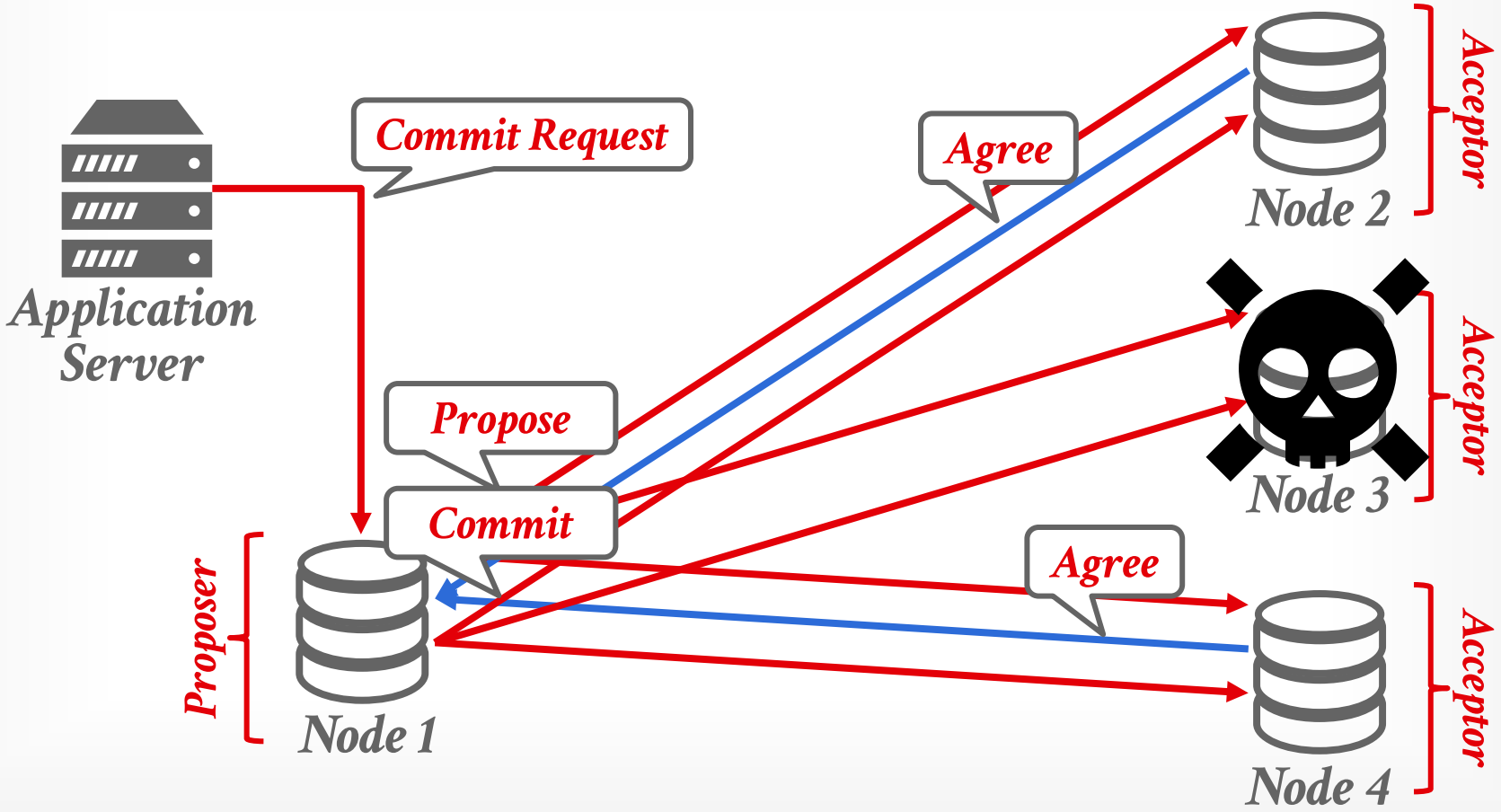
PAXOS


Commit Request
Application Server
Propose
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

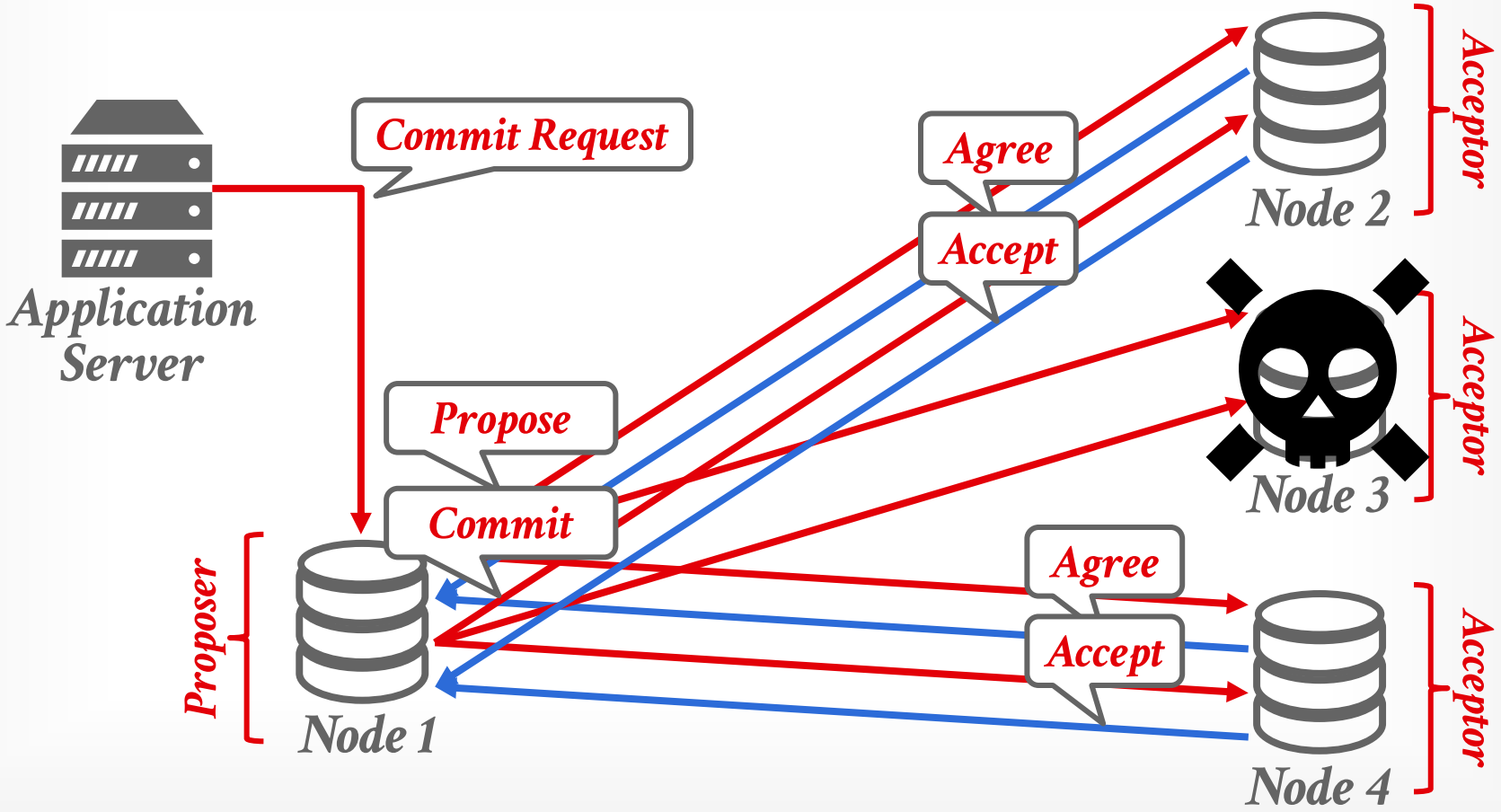
PAXOS


Application Server
Commit Request
Propose
Proposer
Node 1
Agree
Agree
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

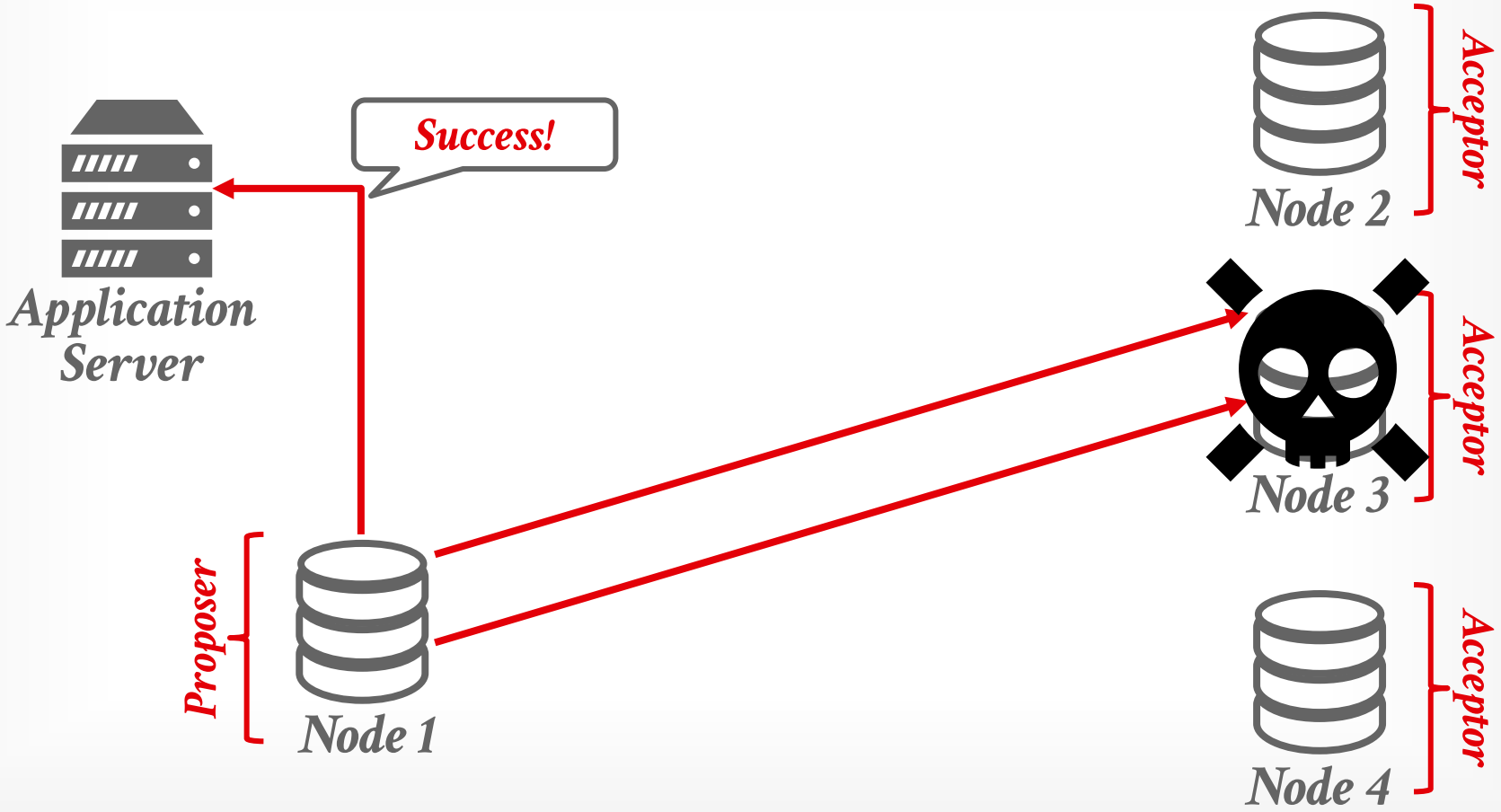
PAXOS


Application Server
Commit Request
Propose
Commit
Agree
Agree
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

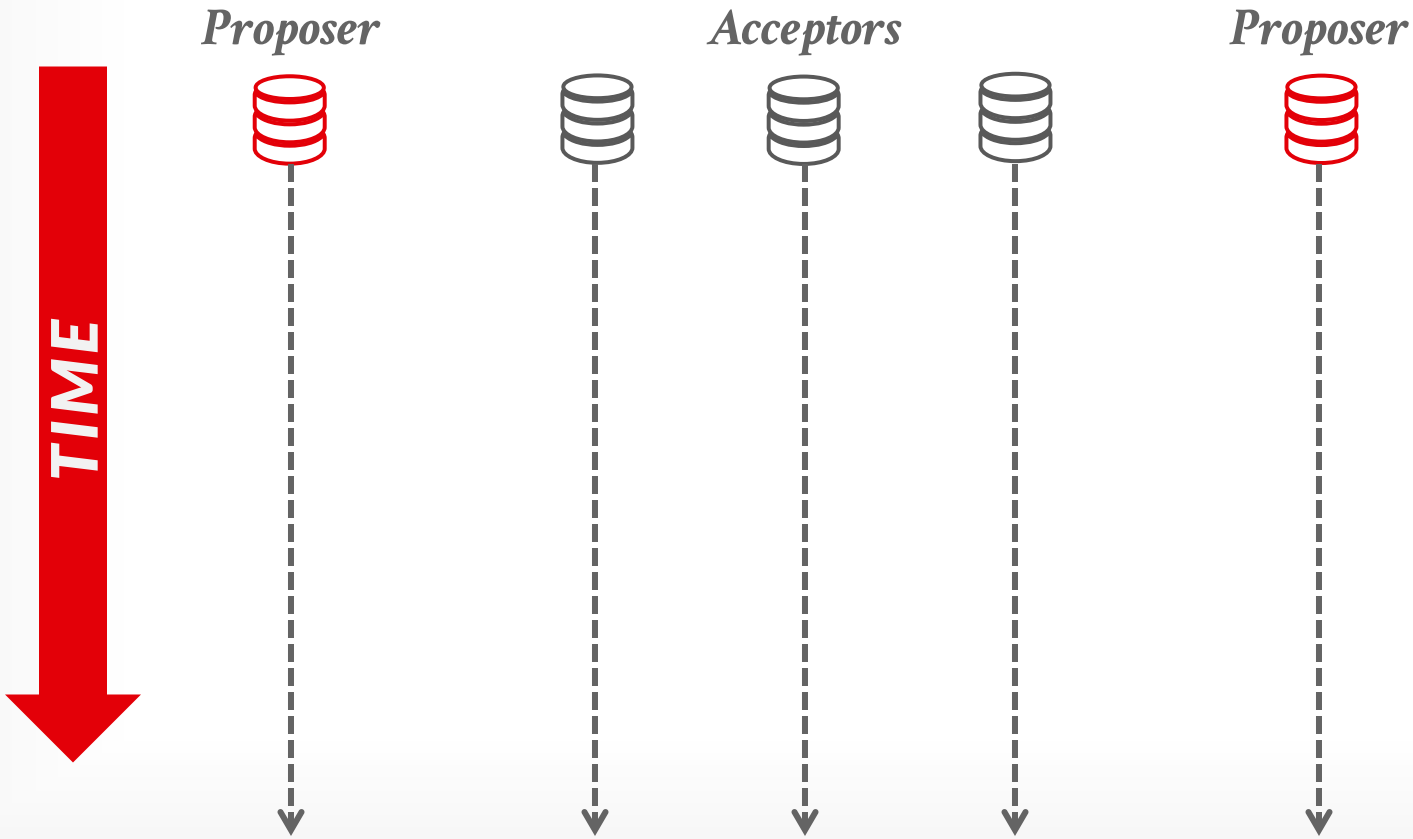
PAXOS


Application Server
Commit Request
Propose
Commit
Proposer
Node 1
Agree
Accept
Node 2
Acceptor
Node 3
Acceptor
Agree
Accept
Node 4
Acceptor

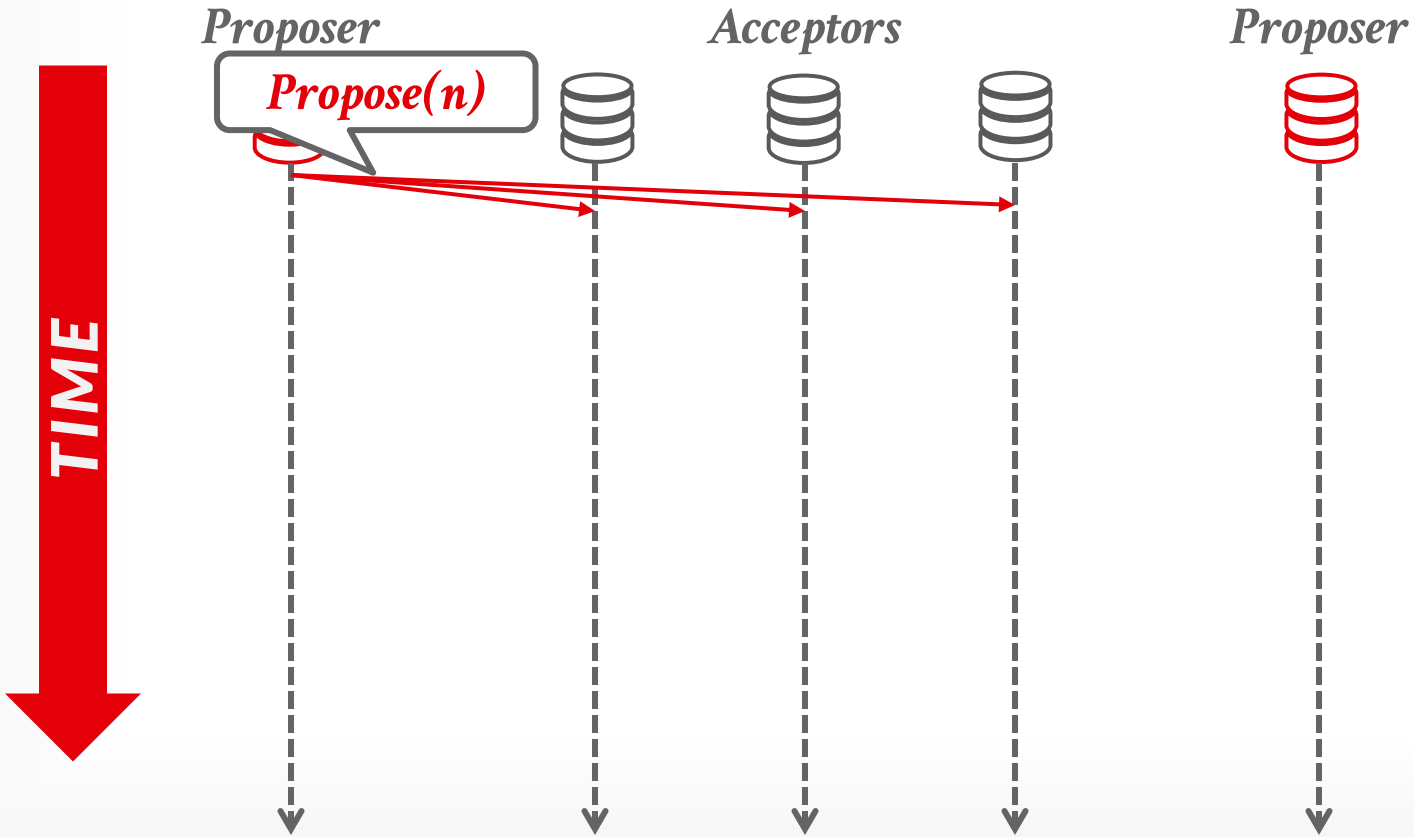
PAXOS


Success!
Application Server
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

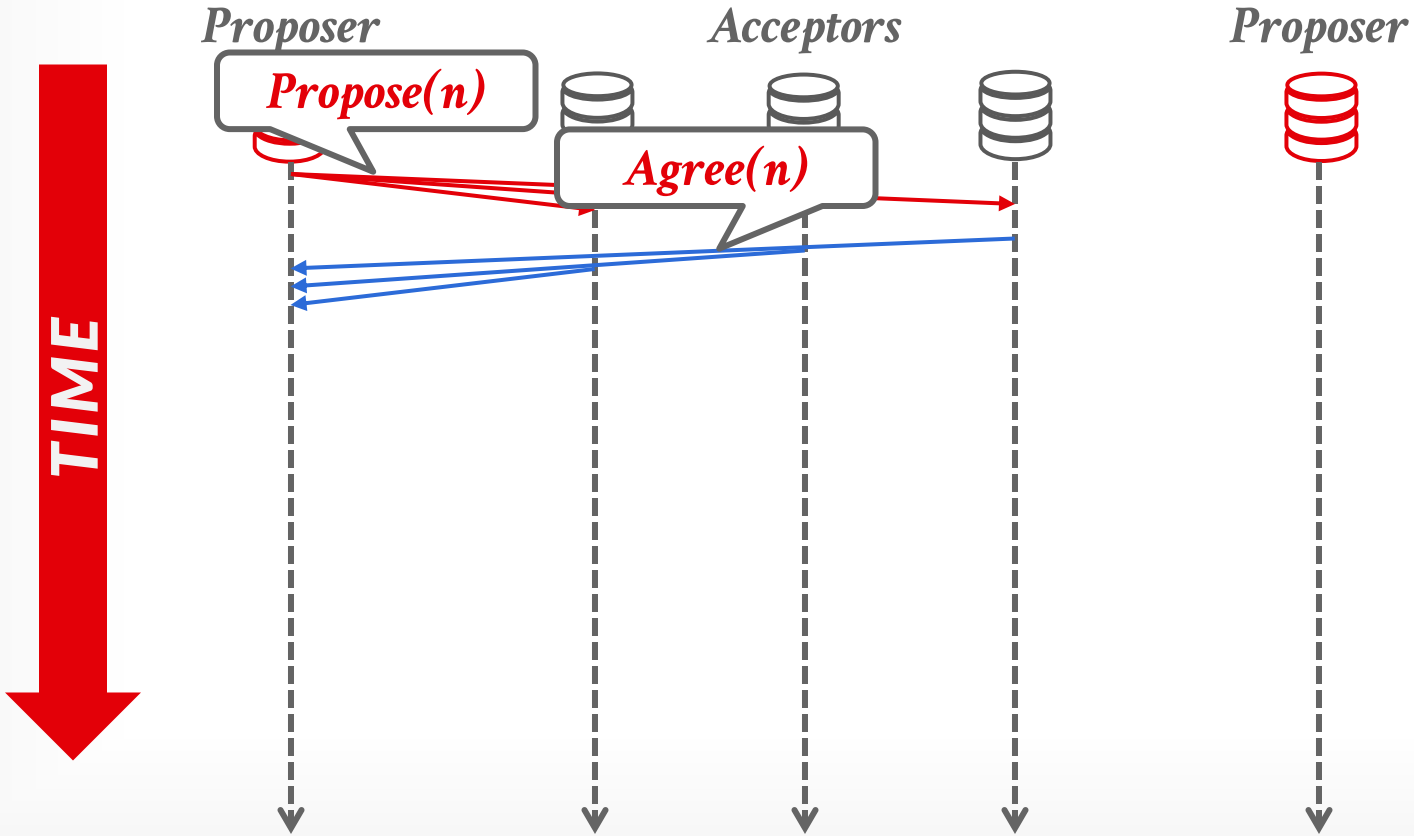
PAXOS



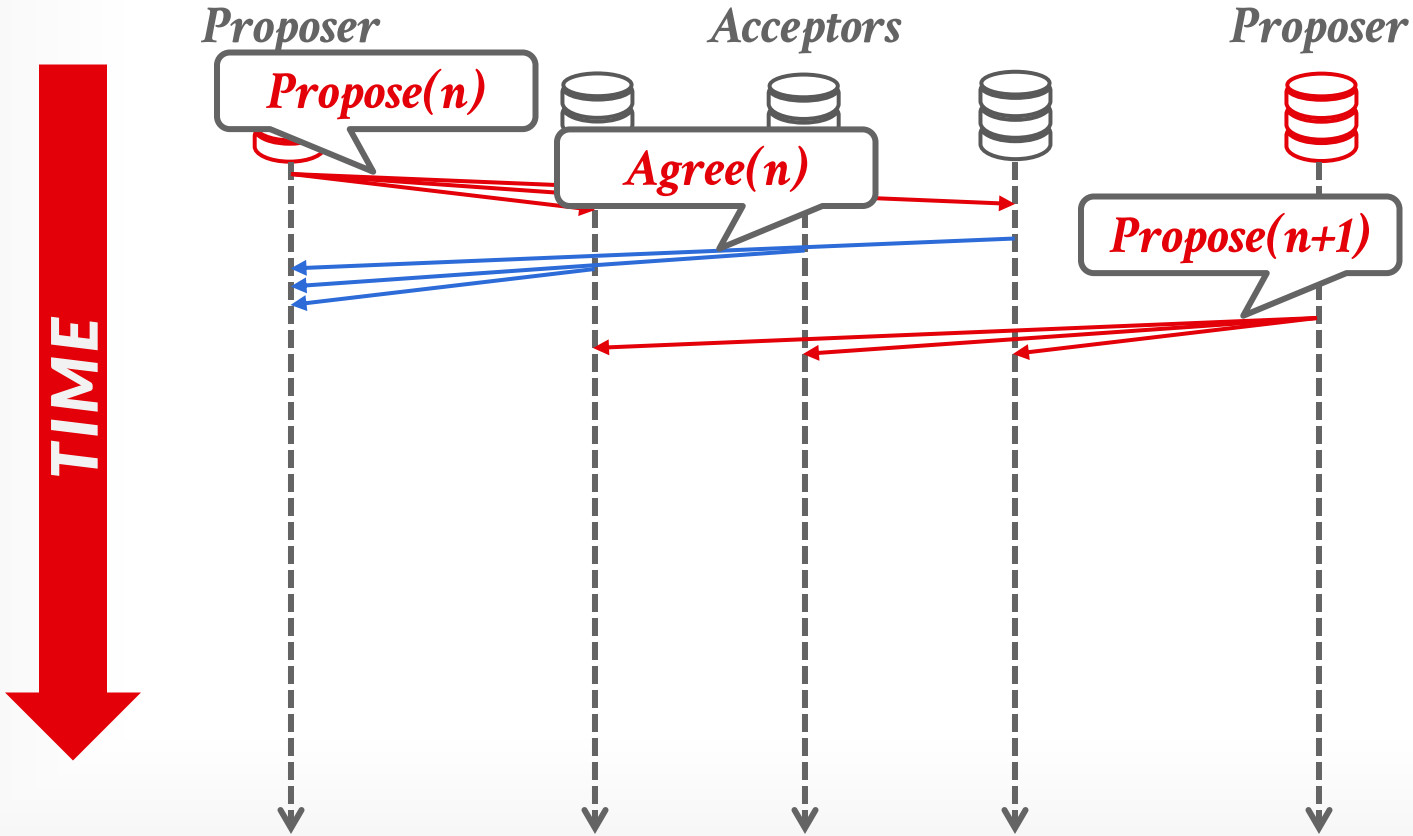
PAXOS



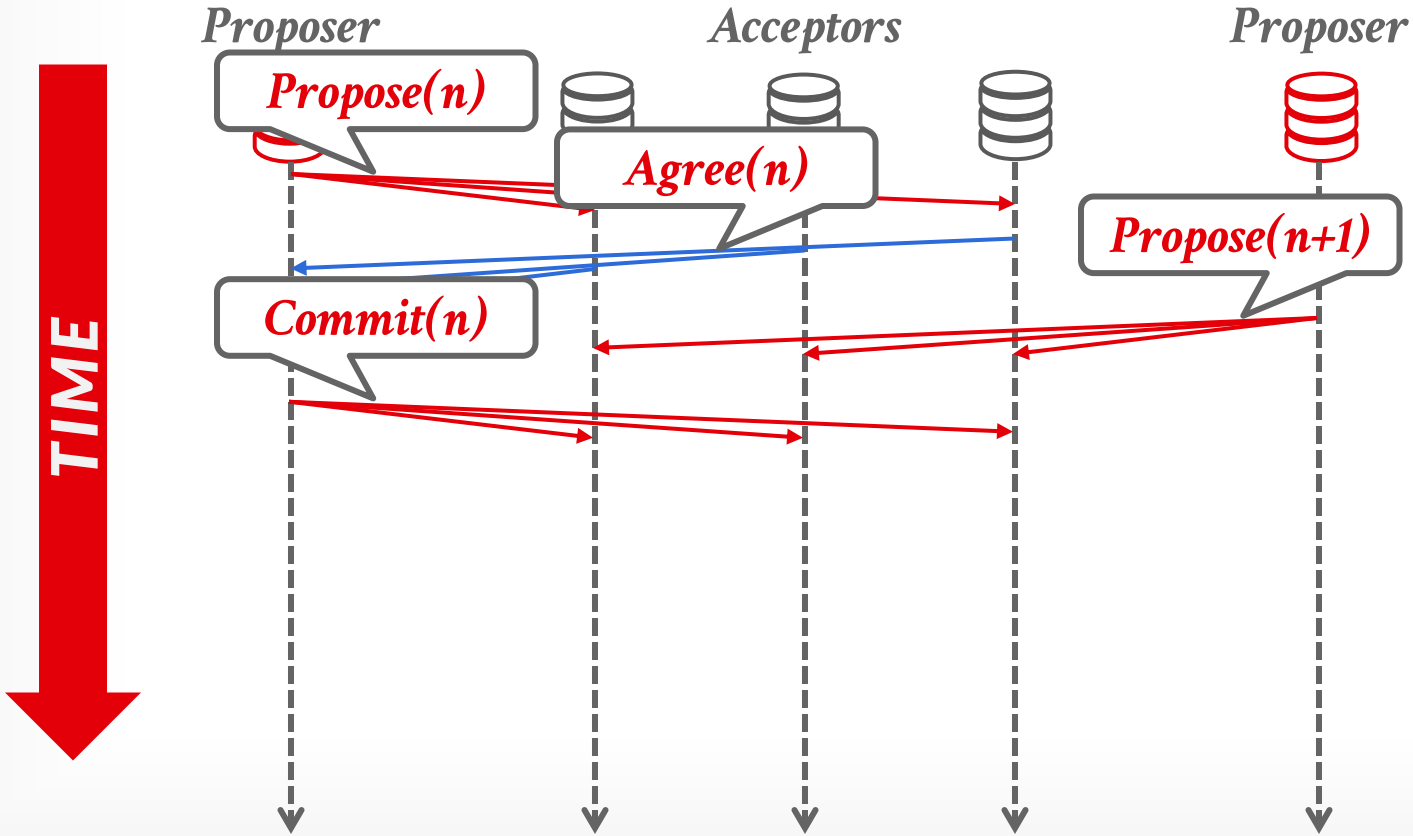
PAXOS



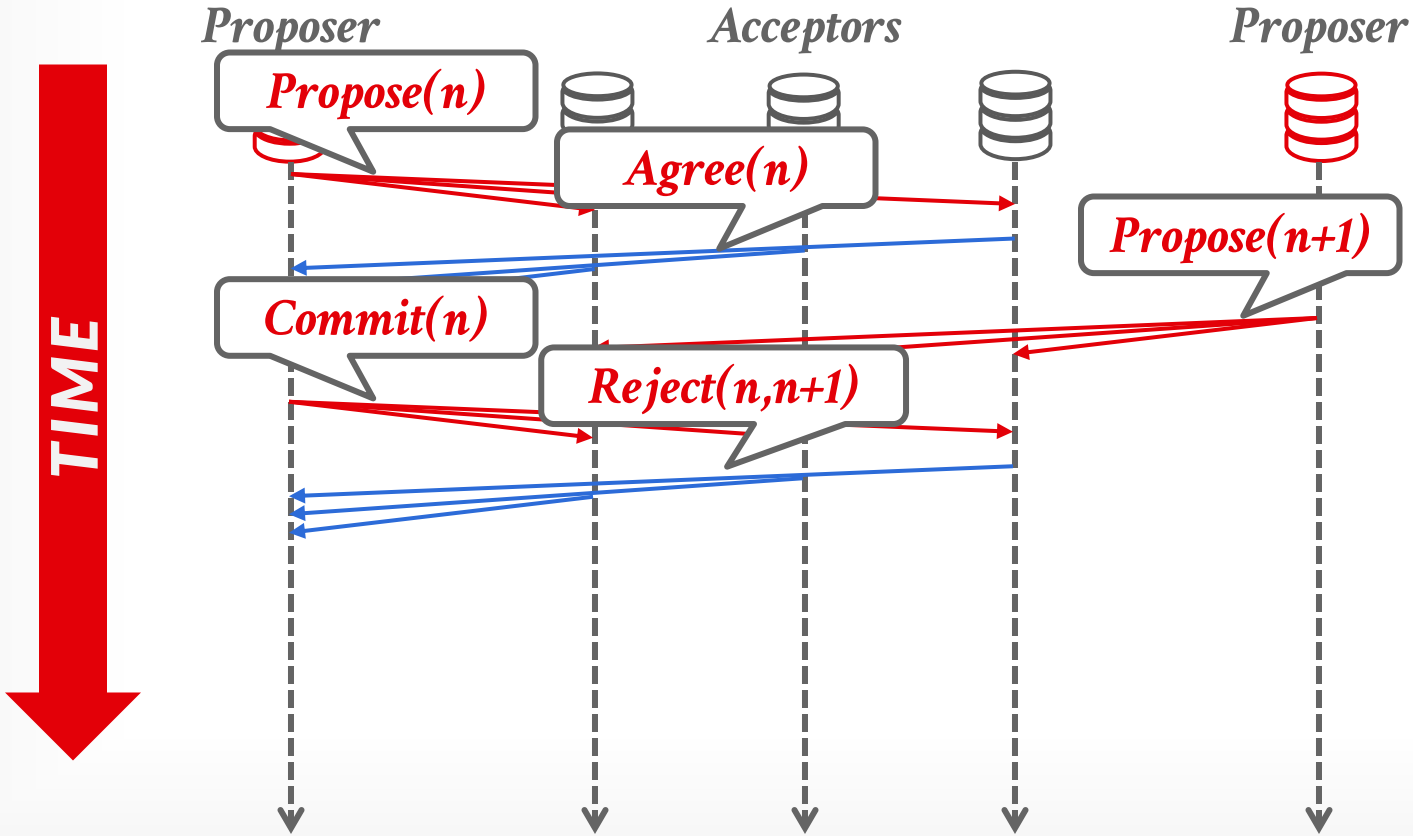
PAXOS


Proposer
Acceptors
Proposer
Propose(n)
Agree(n)
Propose(n+1)
TIME

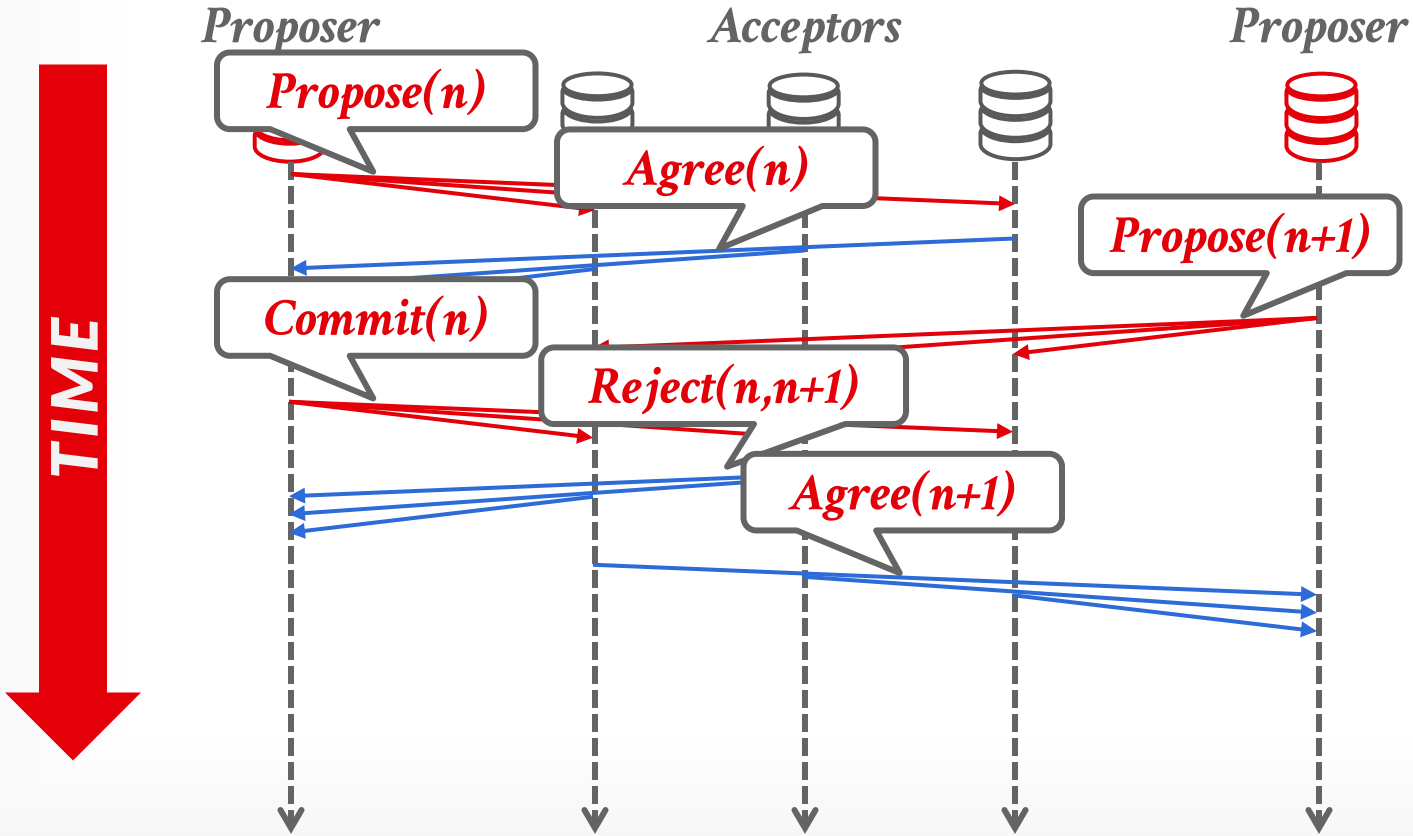
PAXOS


Proposer
Acceptors
Proposer
Propose(n)
Agree(n)
Propose(n+1)
Commit(n)
TIME

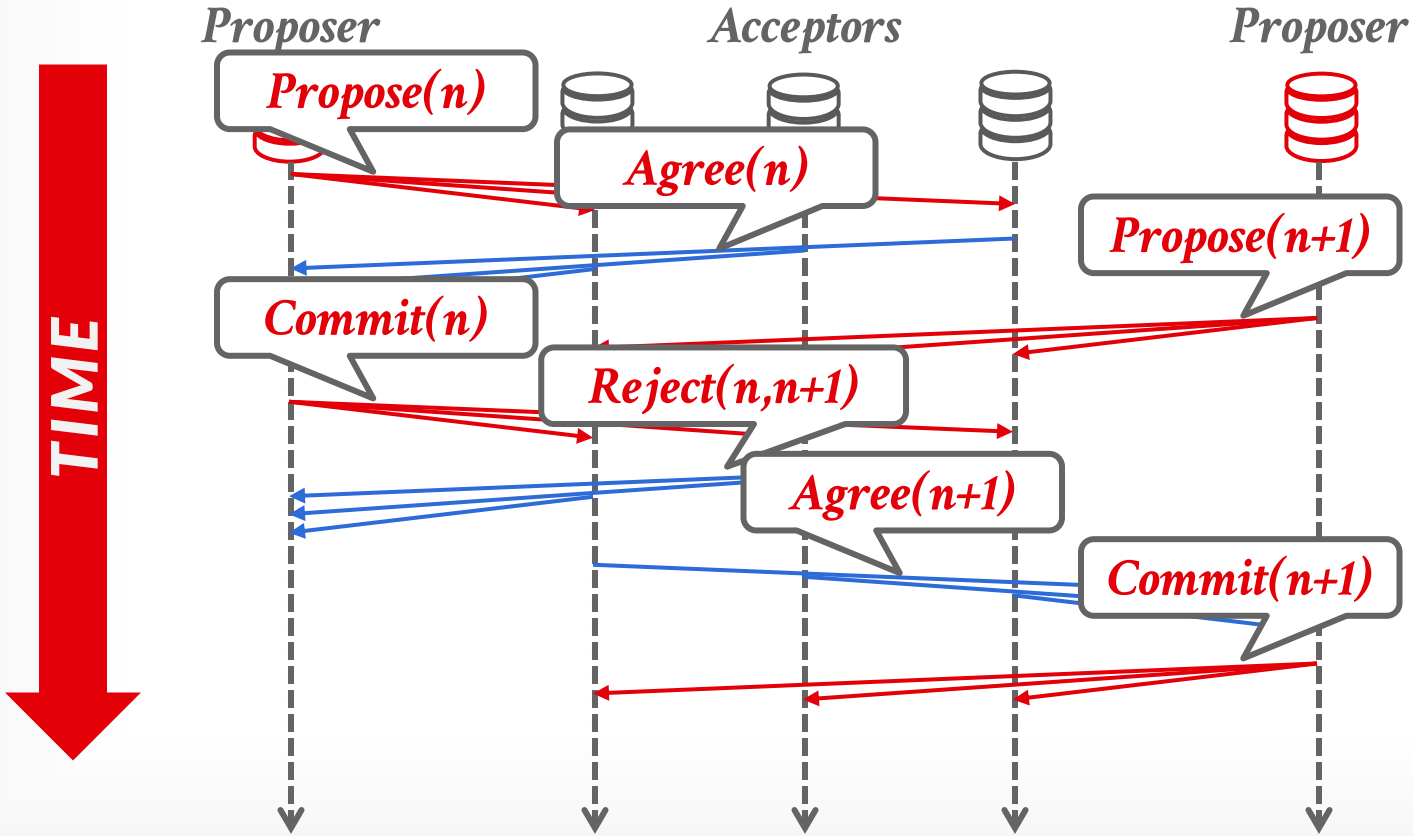
PAXOS


Proposer
Acceptors
Proposer
Propose(n)
Agree(n)
Propose(n+1)
Commit(n)
Reject(n,n+1)
TIME

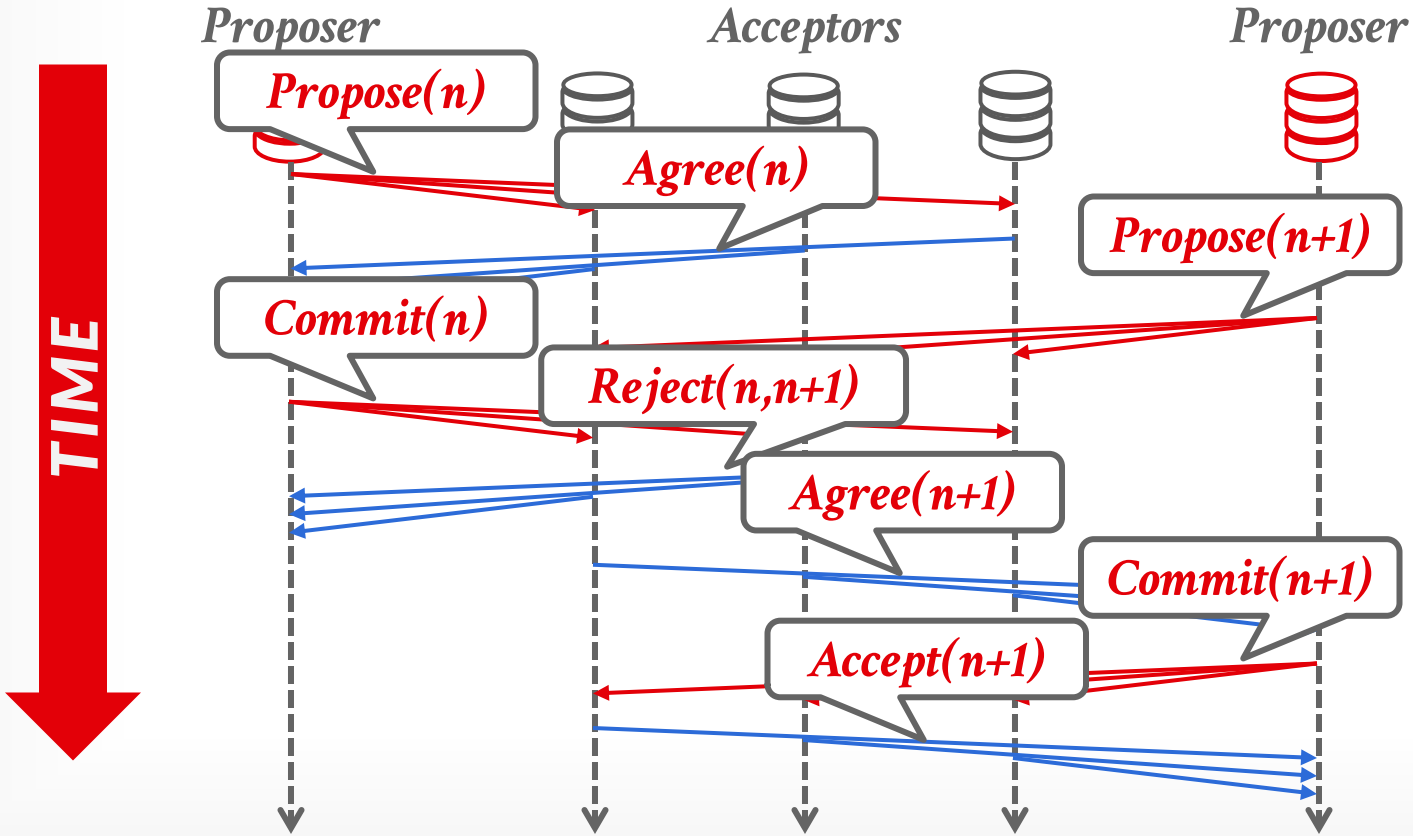
PAXOS


Proposer
Acceptors
Proposer
Propose(n)
Agree(n)
Propose(n+1)
Commit(n)
Reject(n,n+1)
Agree(n+1)
TIME

PAXOS


Proposer
Acceptors
Proposer
Propose(n)
Agree(n)
Propose(n+1)
Commit(n)
Reject(n,n+1)
Agree(n+1)
Commit(n+1)
TIME

PAXOS


Proposer
Acceptors
Proposer
TIME
Propose(n)
Agree(n)
Propose(n+1)
Commit(n)
Reject(n,n+1)
Agree(n+1)
Commit(n+1)
Accept(n+1)

MULTI-PAXOS

If the system elects a single leader that oversees proposing changes for some period, then it can skip the **Propose** phase.

→ Fall back to full Paxos whenever there is a failure.

The system periodically renews the leader (known as a *lease*) using another Paxos round.

→ Nodes must exchange log entries during leader election to make sure that everyone is up-to-date.

2PC VS. PAXOS VS. RAFT

Two-Phase Commit:

→ Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

Paxos:

→ Non-blocking if a majority participants are alive, provided there is a sufficiently long period without further failures.

Raft:

→ Similar to Paxos but with fewer node types.

→ Only nodes with most up-to-date log can become leaders.

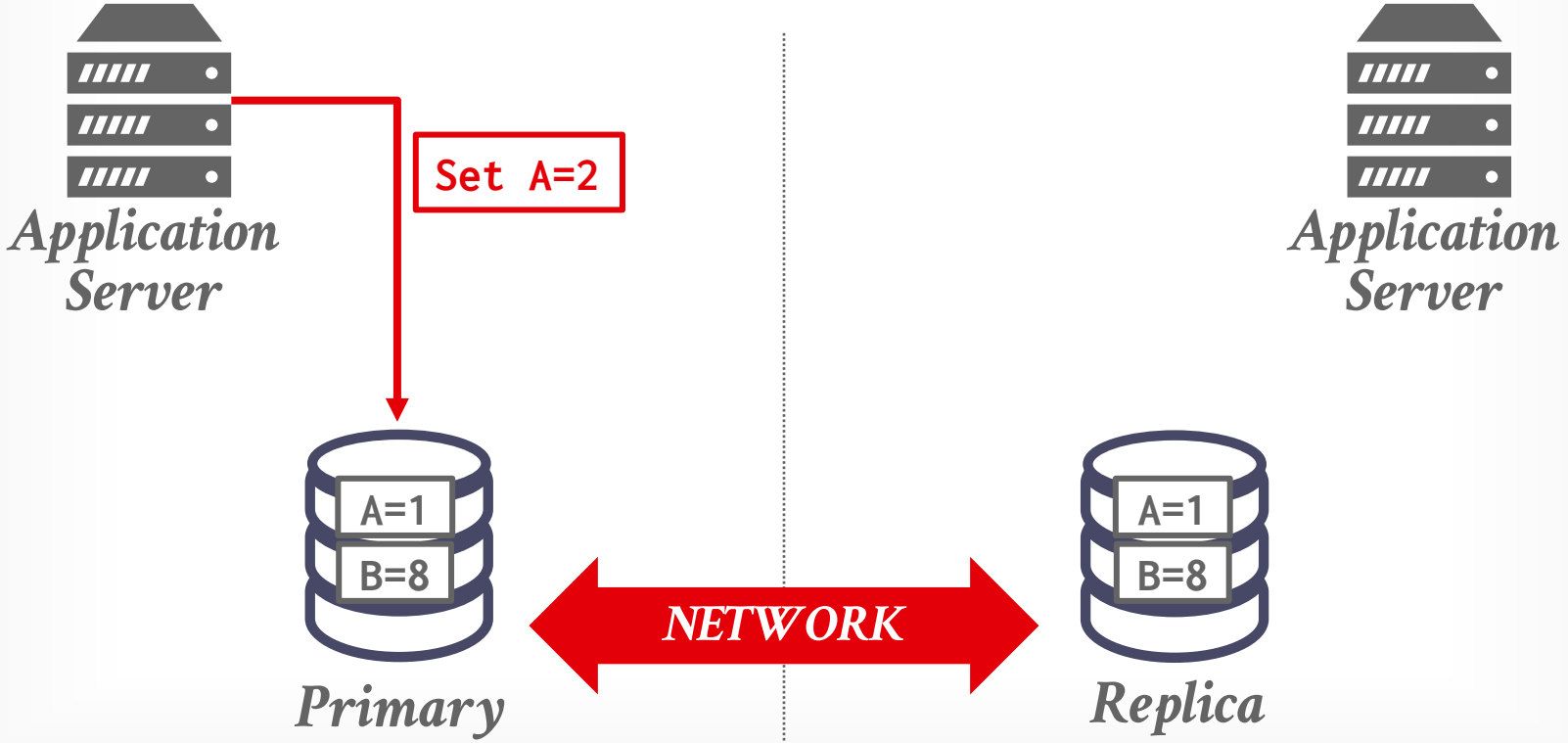
CAP THEOREM

Proposed in the late 1990s that is impossible for a distributed database to always be:

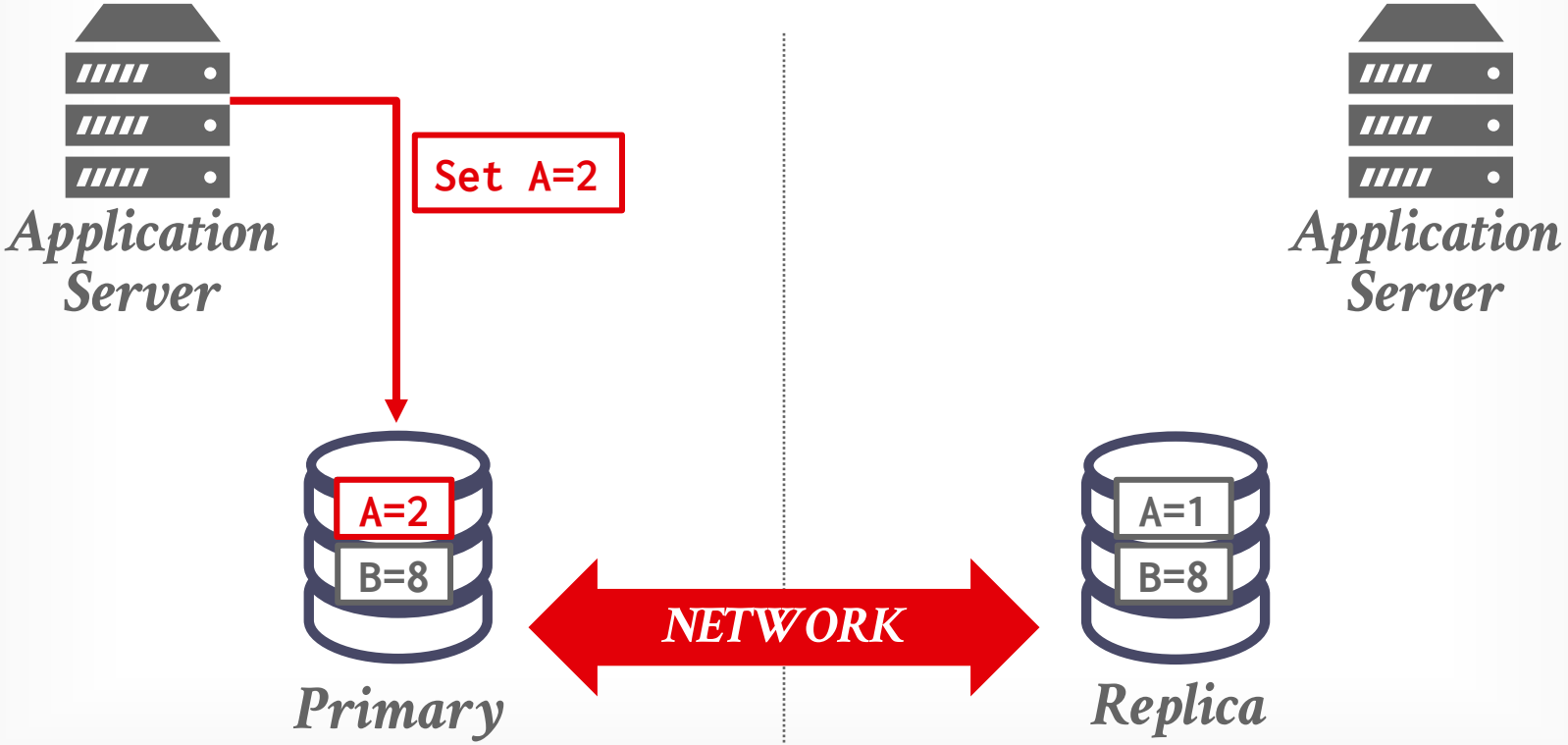
→ **C**onsistent
→ **A**lways Available
→ **N**etwork Partition Tolerant

Whether a DBMS provides **C**onsistency or **A**vailability during a **N**etwork partition.

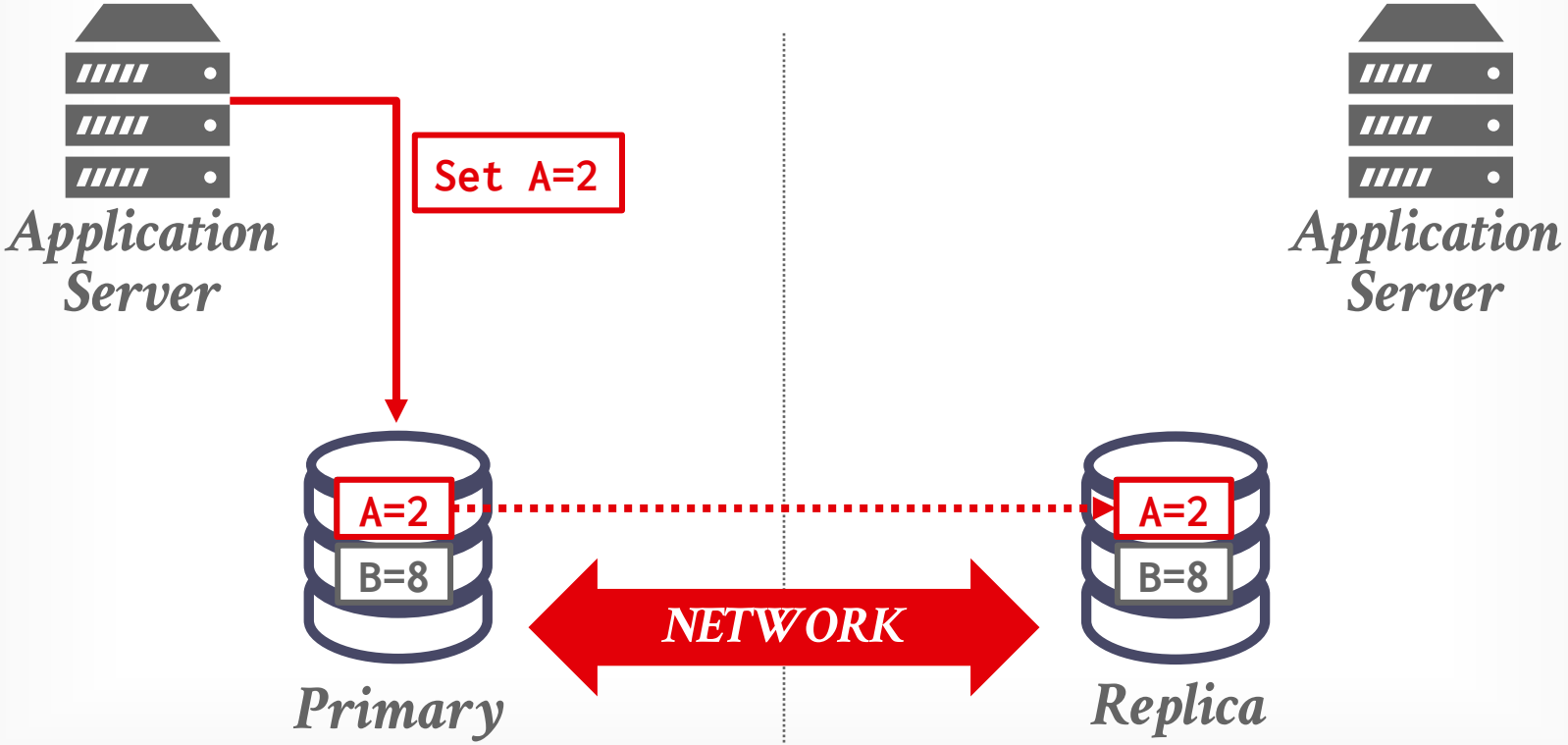
CONSISTENCY



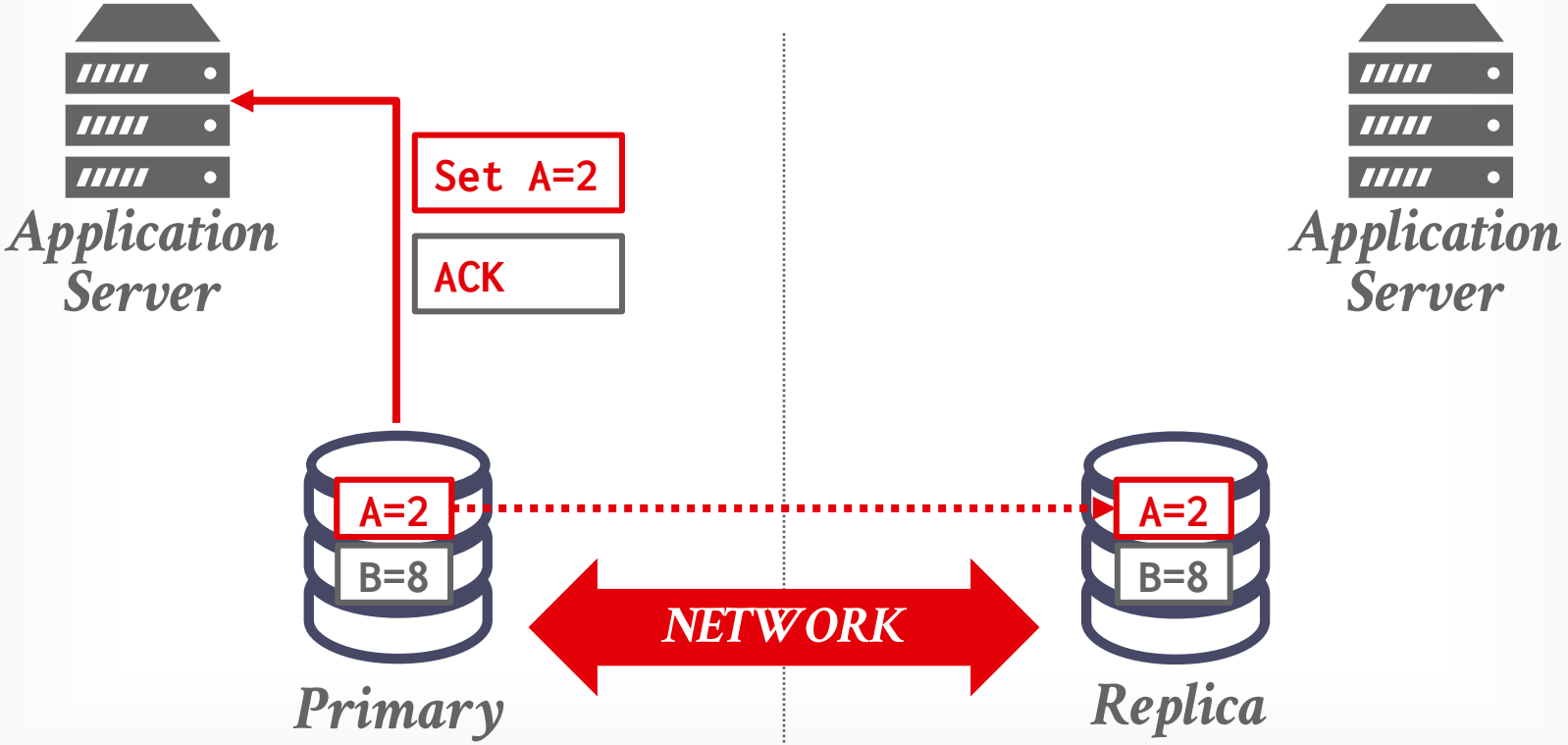
CONSISTENCY



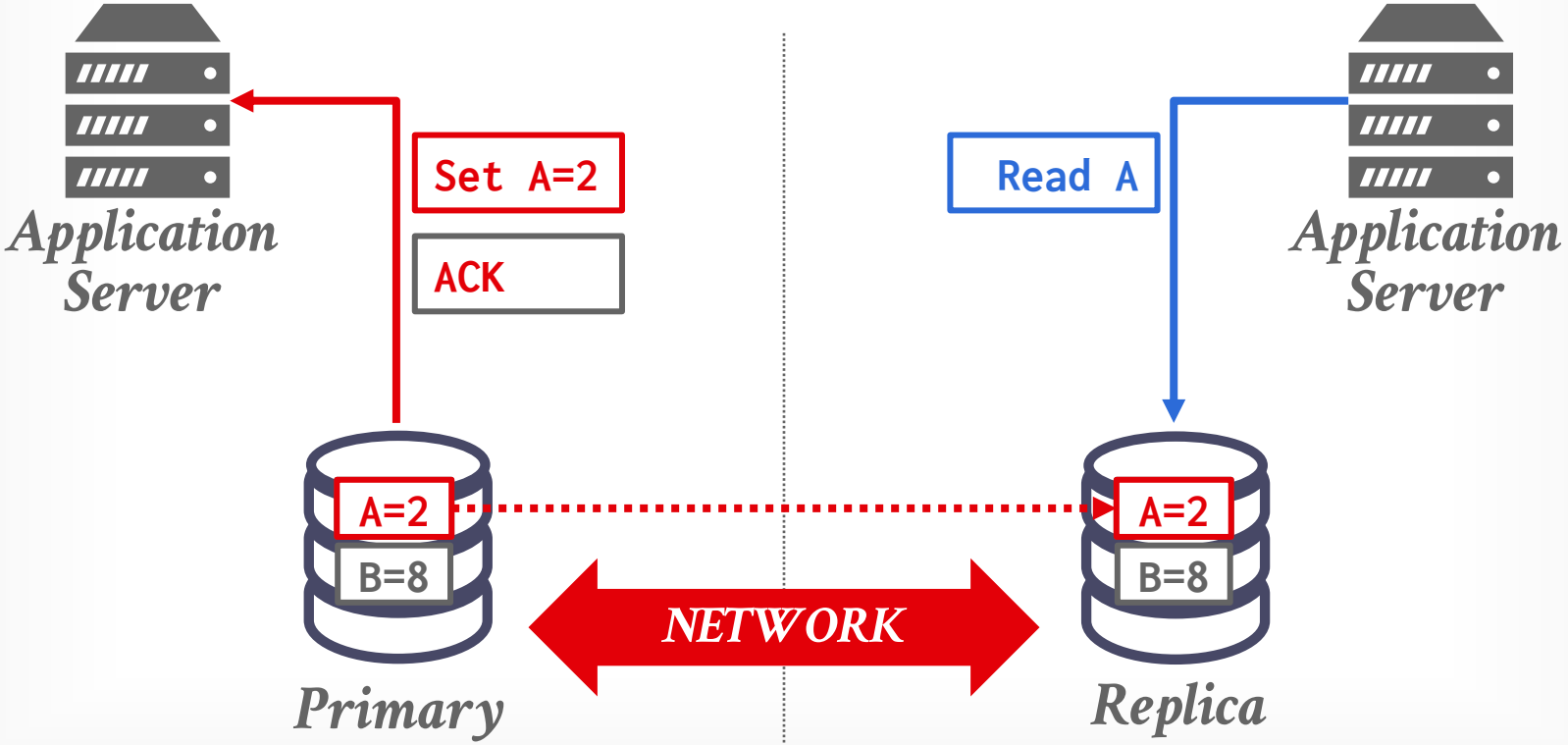
CONSISTENCY



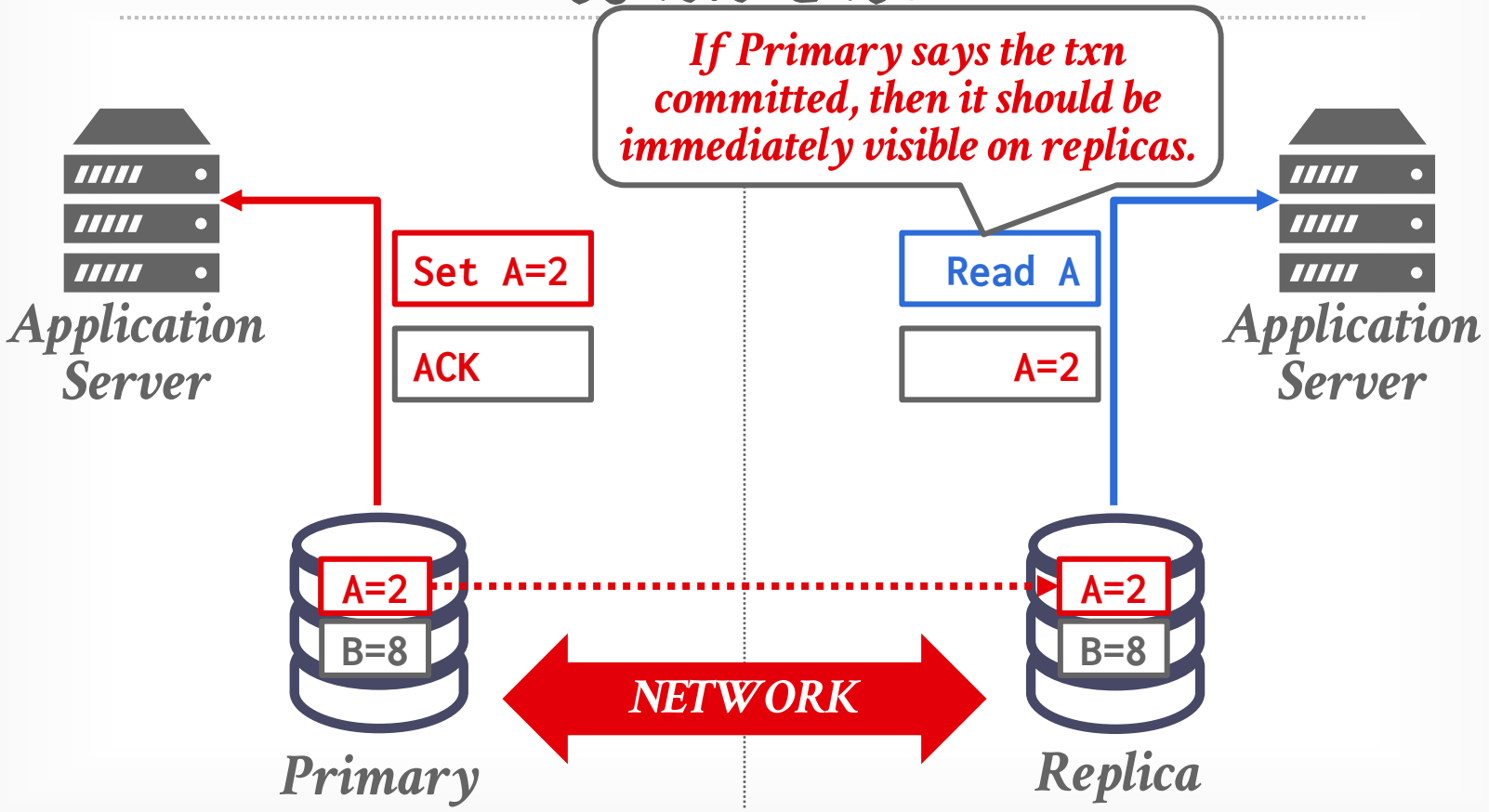
CONSISTENCY



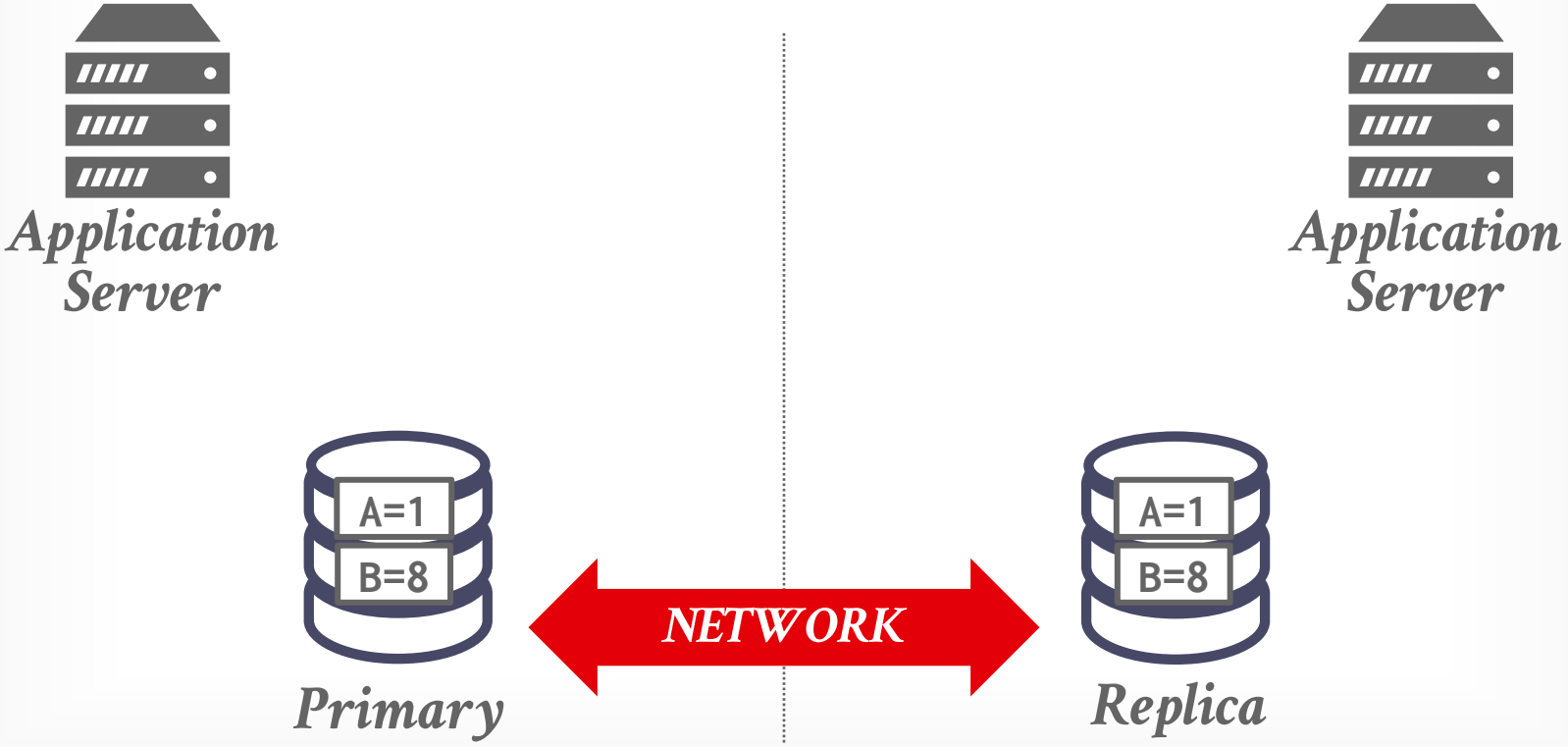
CONSISTENCY



CONSISTENCY

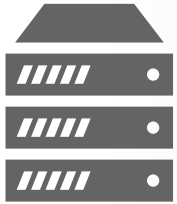


AVAILABILITY

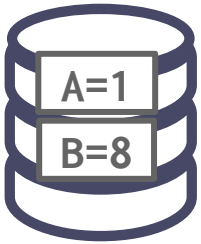


AVAILABILITY

Application Server



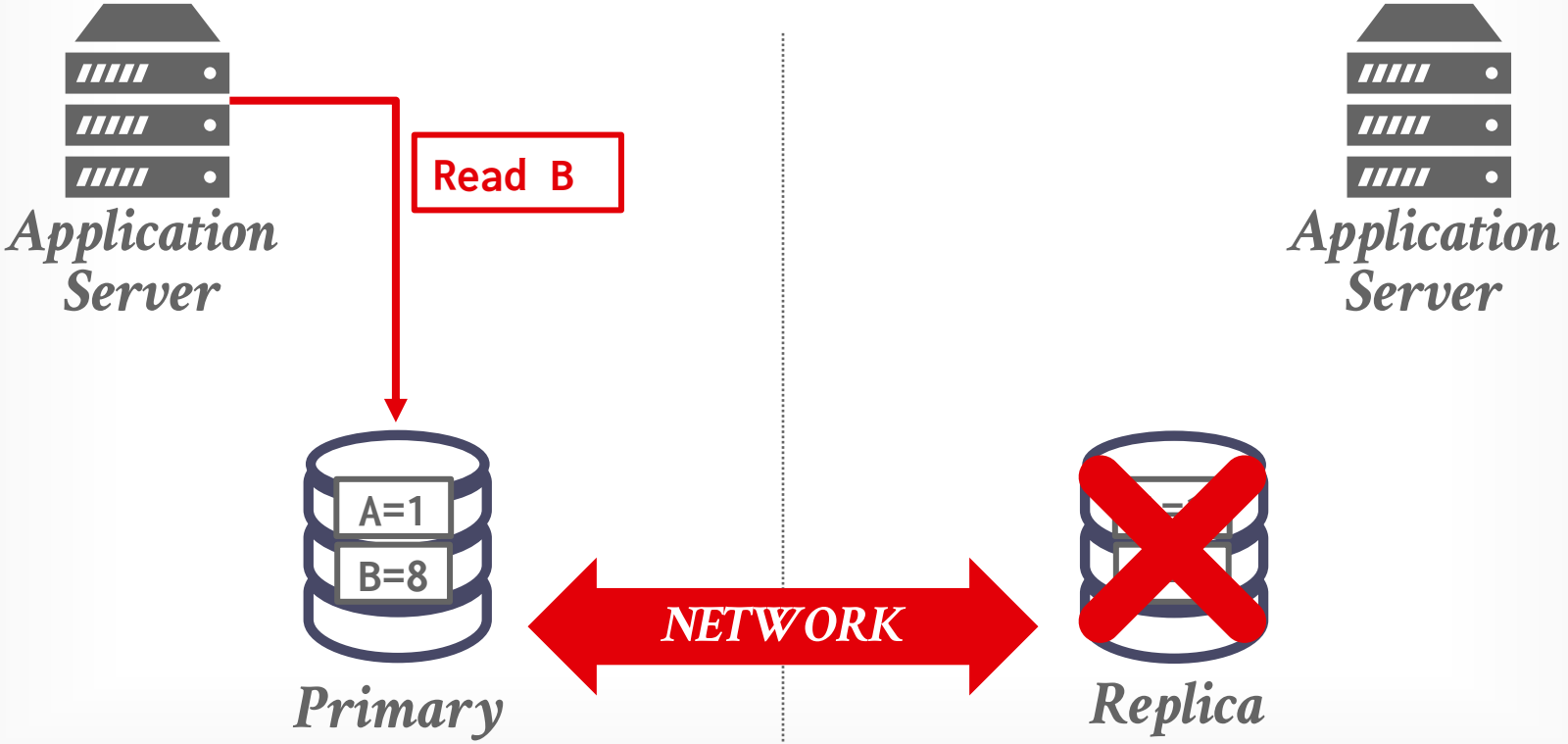
Application Server



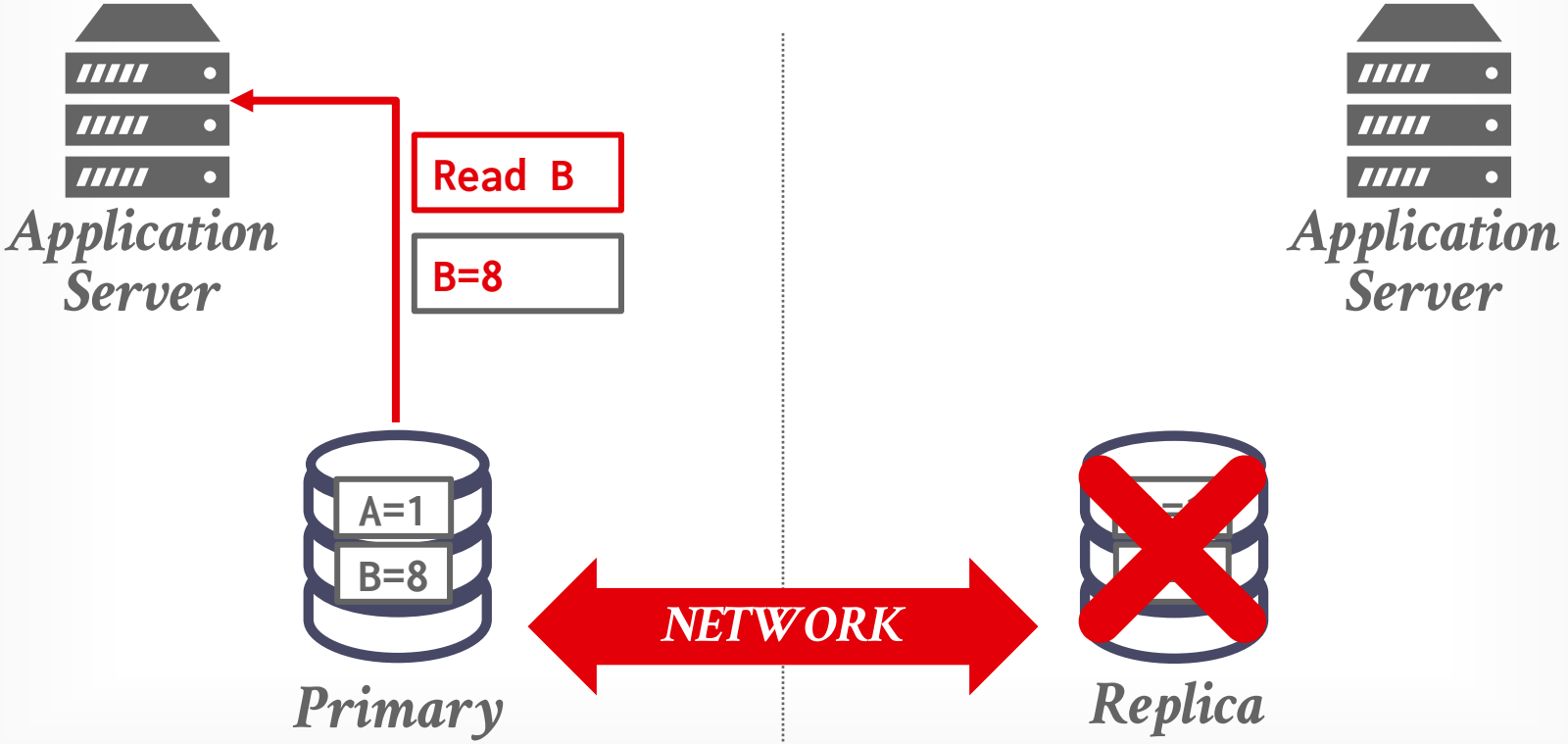
Primary

Replica

AVAILABILITY

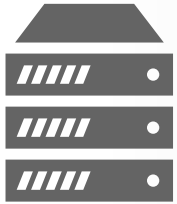


AVAILABILITY

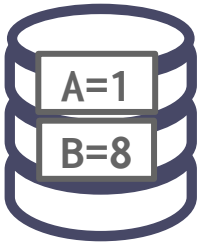


AVAILABILITY

Application Server



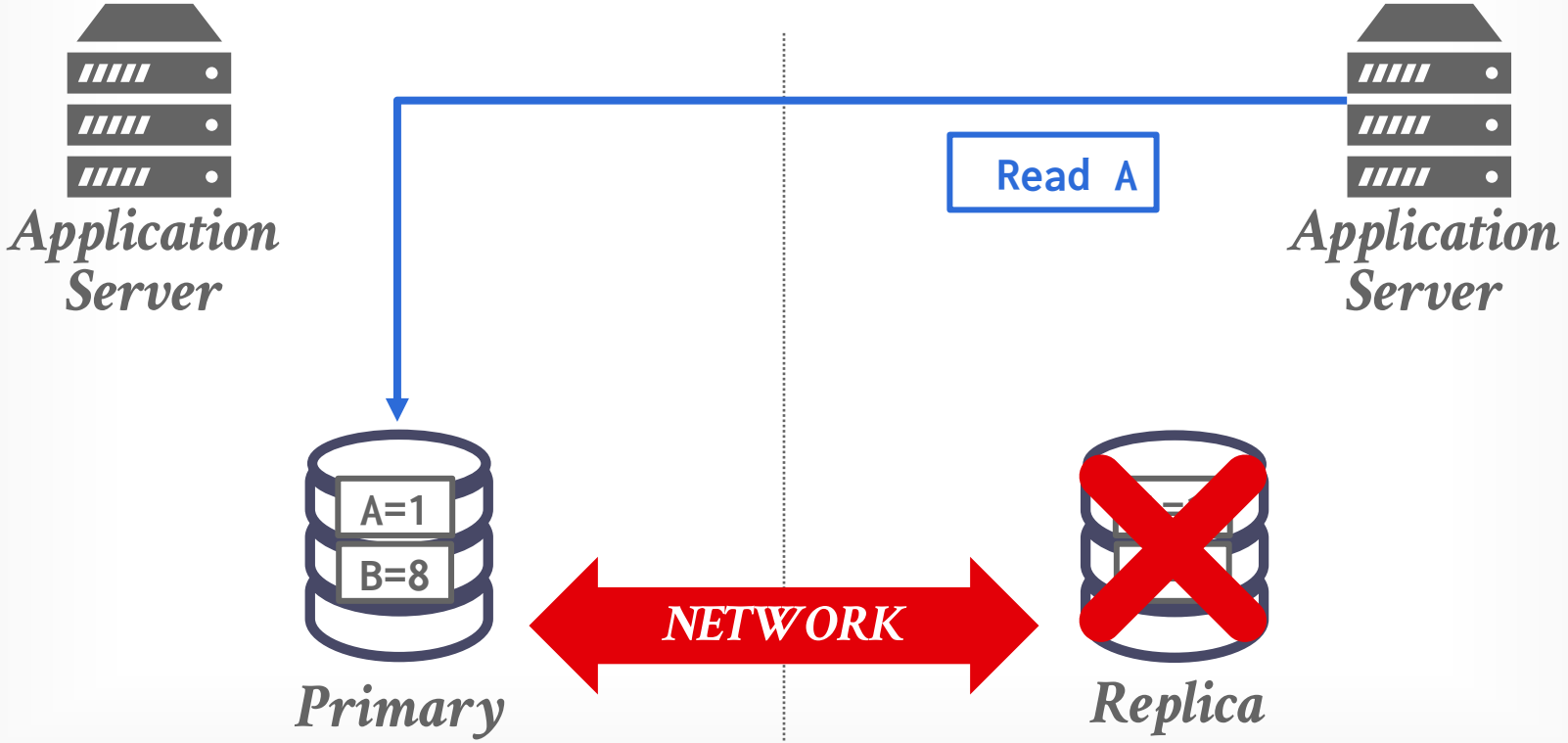
Application Server



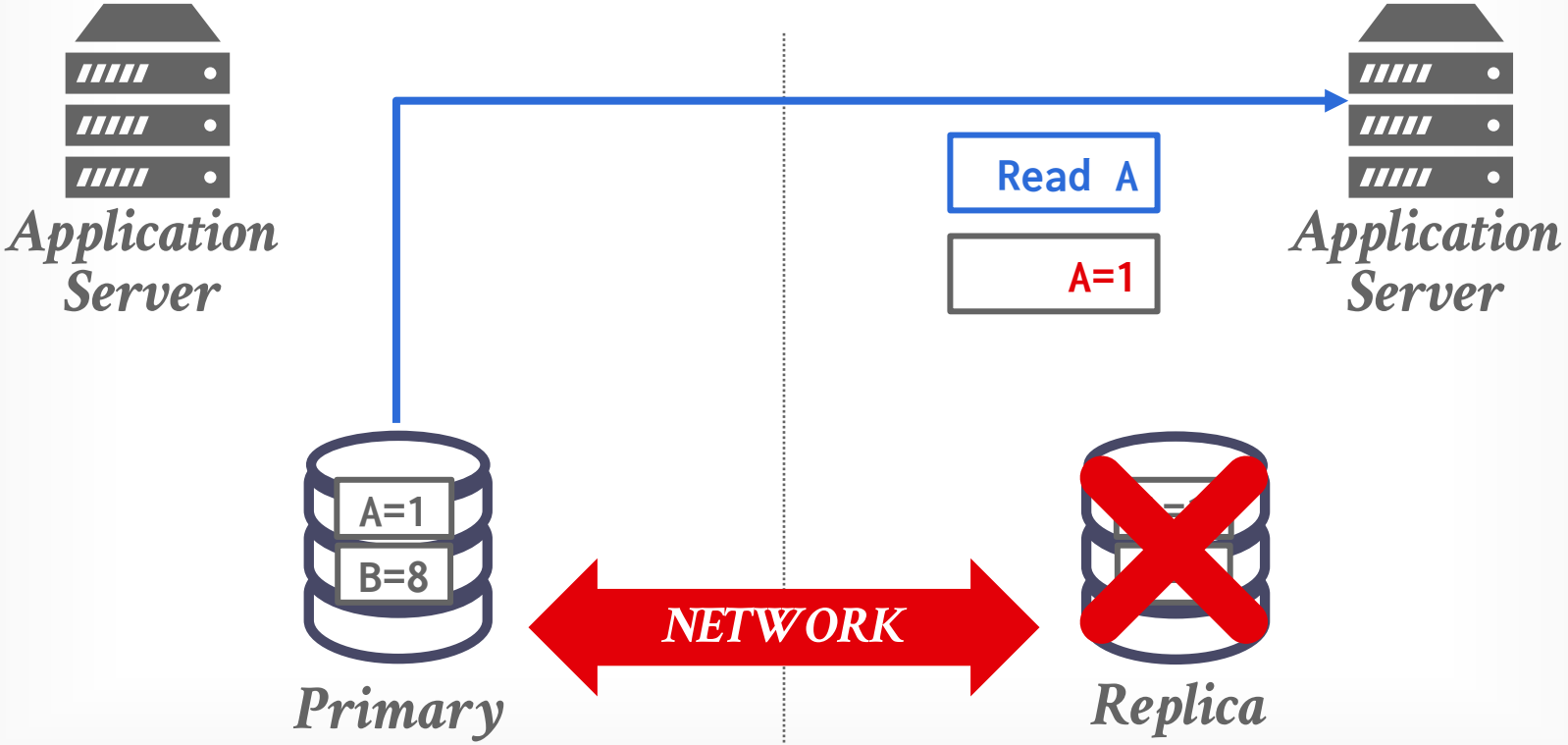
Primary

Replica

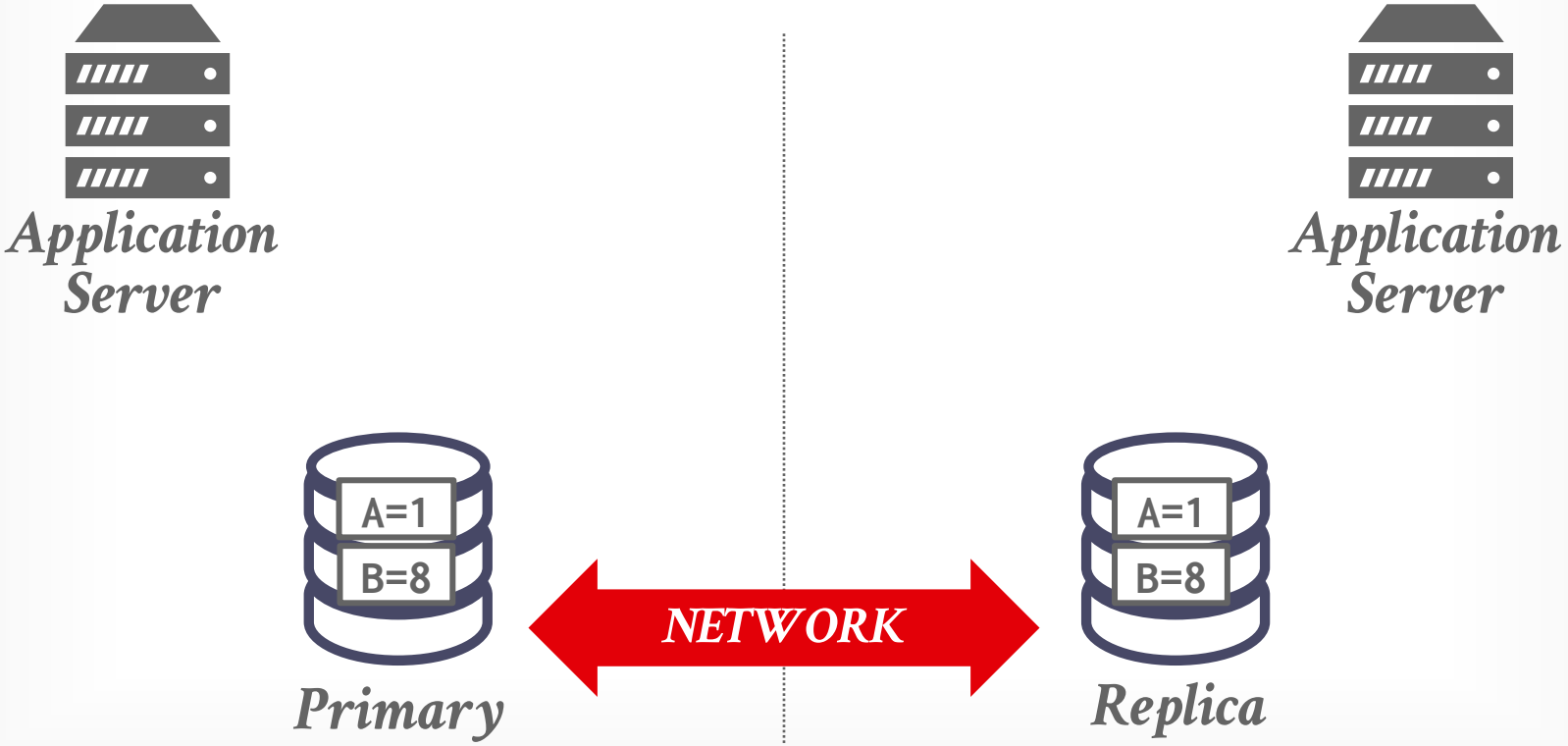
AVAILABILITY



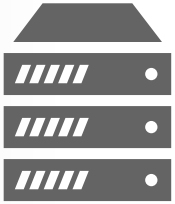
AVAILABILITY



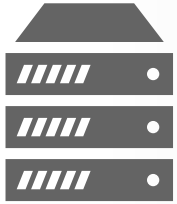
PARTITION TOLERANCE



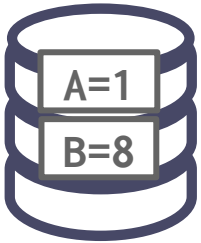
PARTITION TOLERANCE



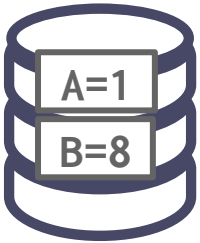
Application Server



Application Server

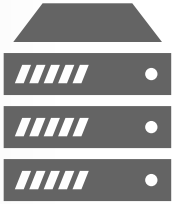


Primary

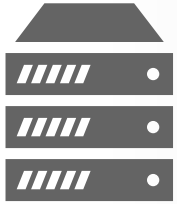


Replica

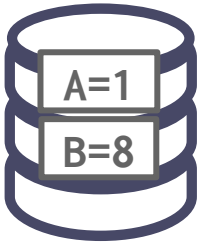
PARTITION TOLERANCE



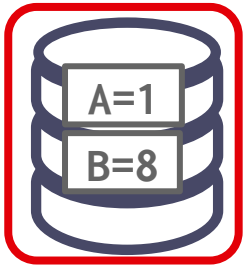
Application Server



Application Server

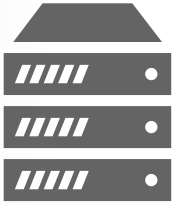


Primary

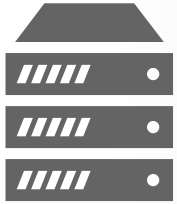


Replica

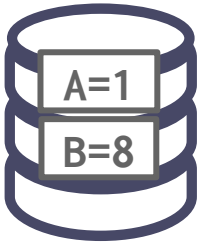
PARTITION TOLERANCE



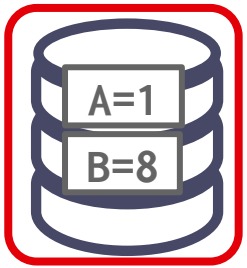
Application Server



Application Server



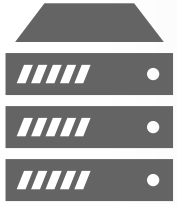
Primary



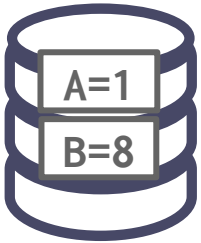
Primary

PARTITION TOLERANCE

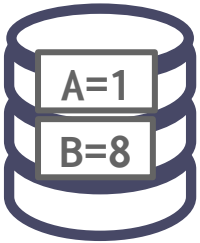
Application Server



Application Server

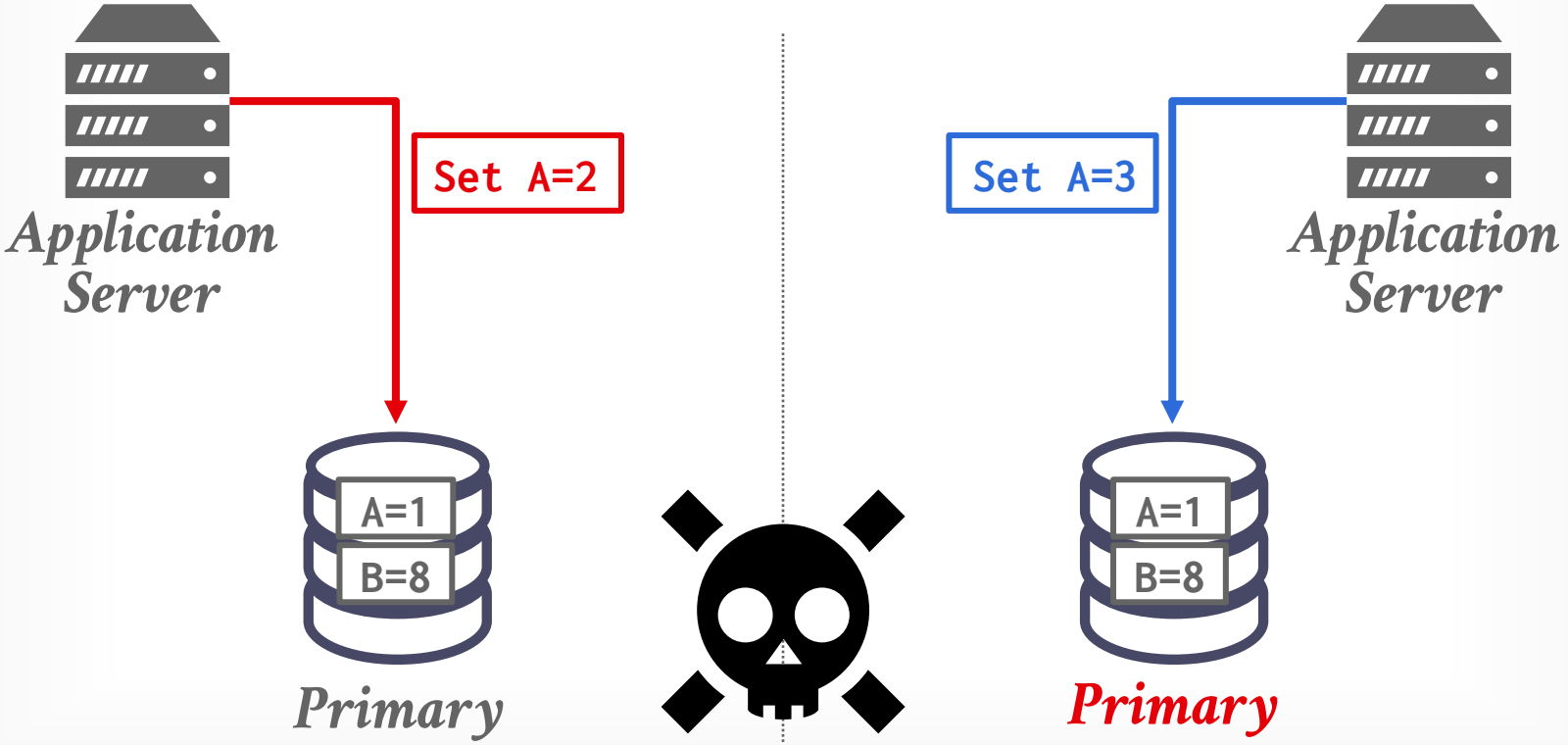


Primary

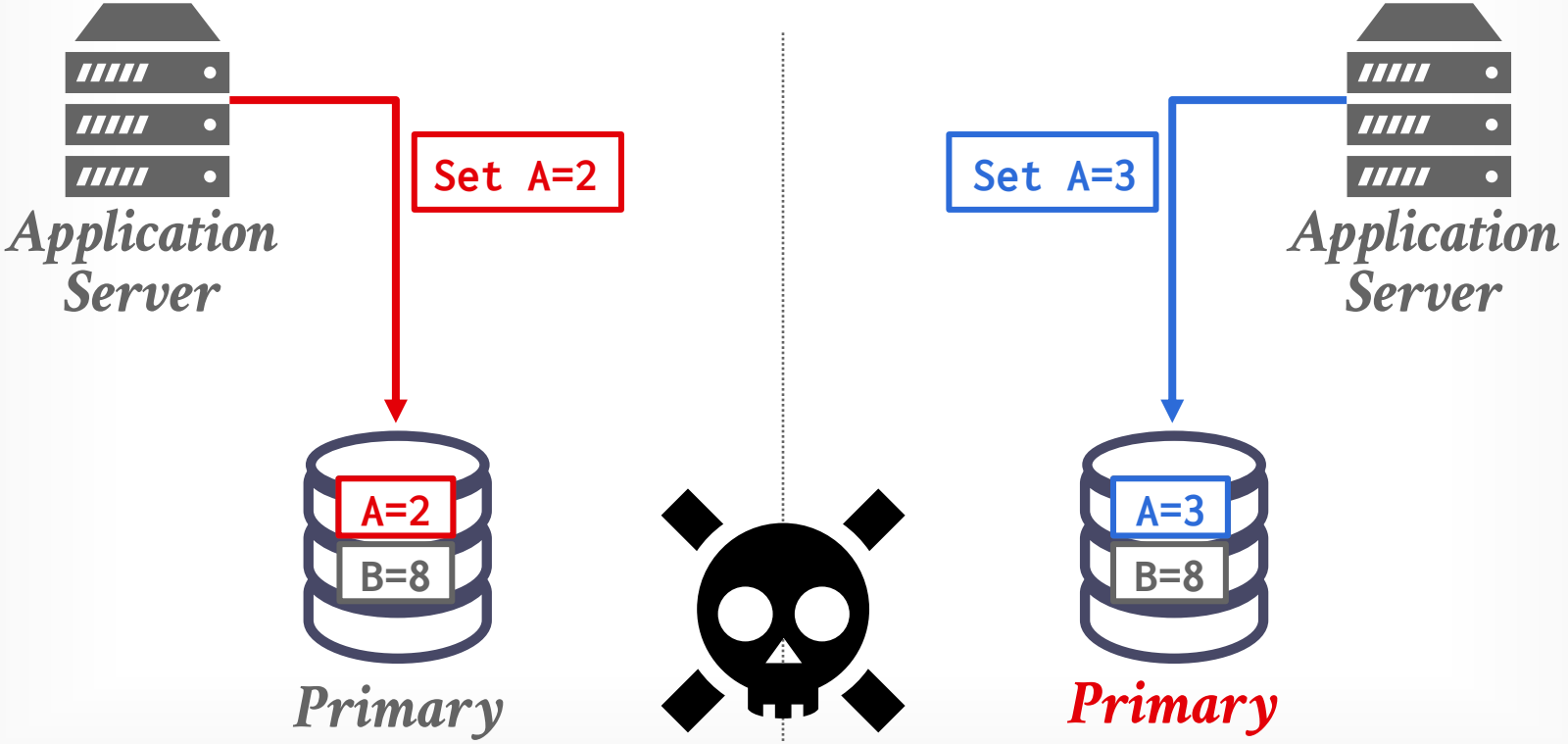


Primary

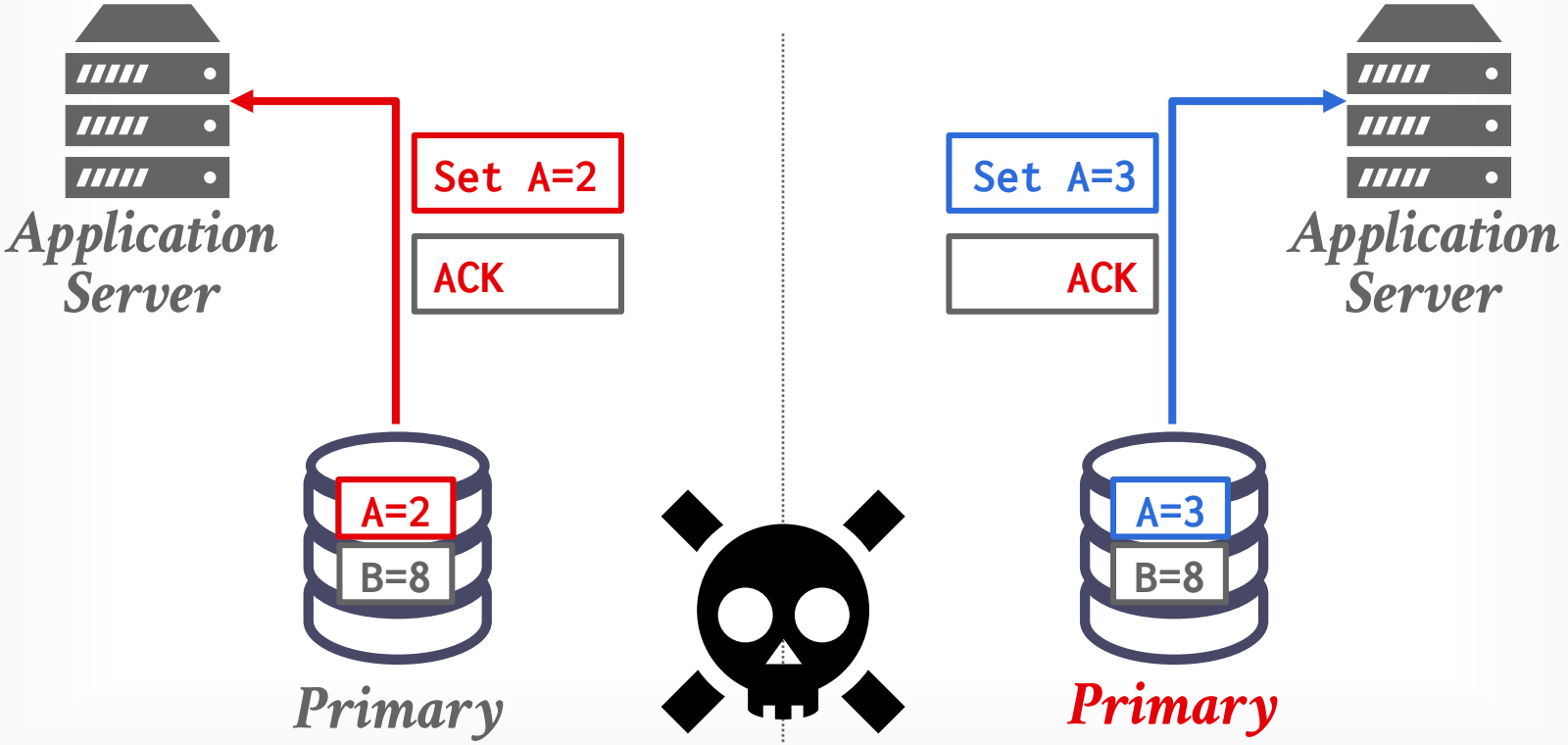
PARTITION TOLERANCE



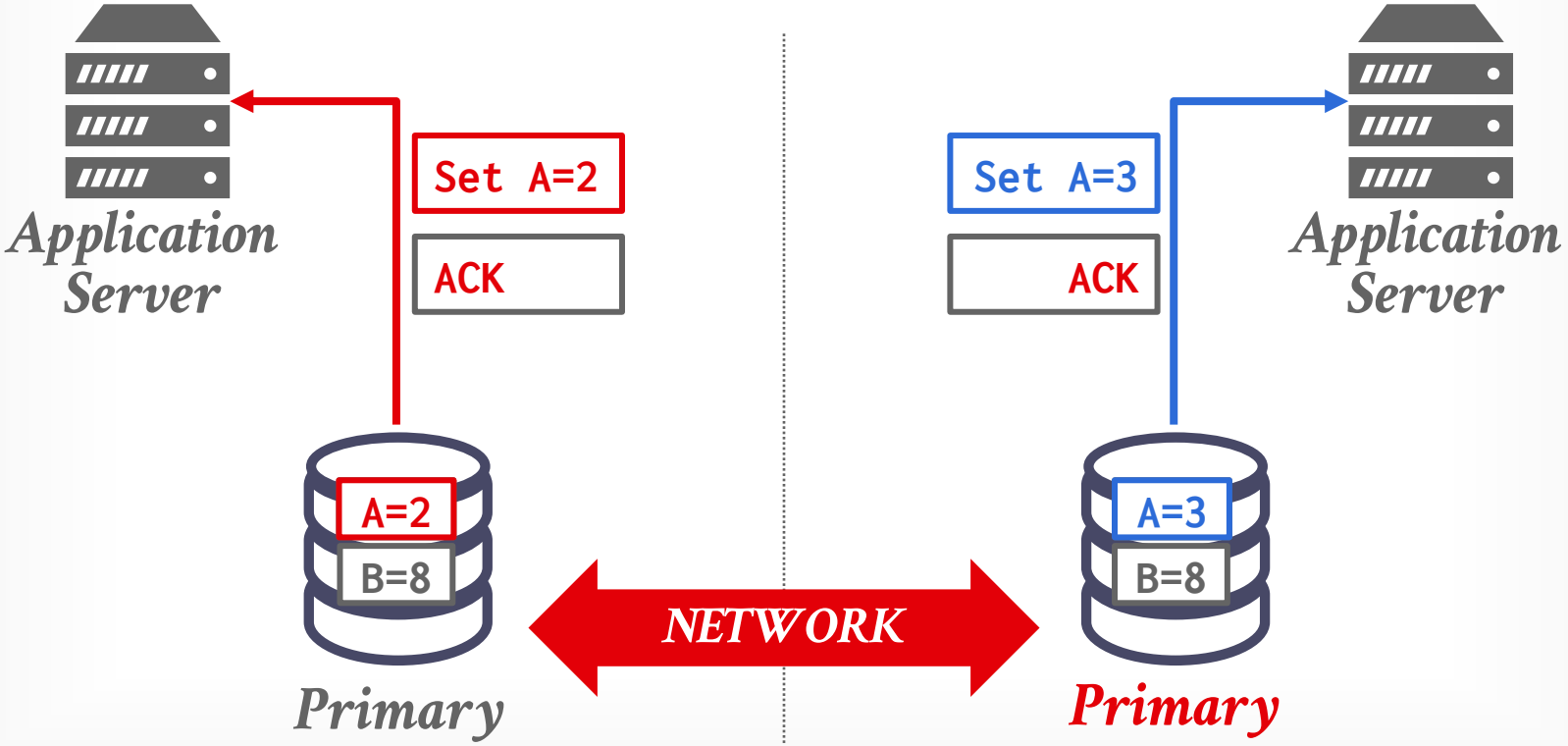
PARTITION TOLERANCE



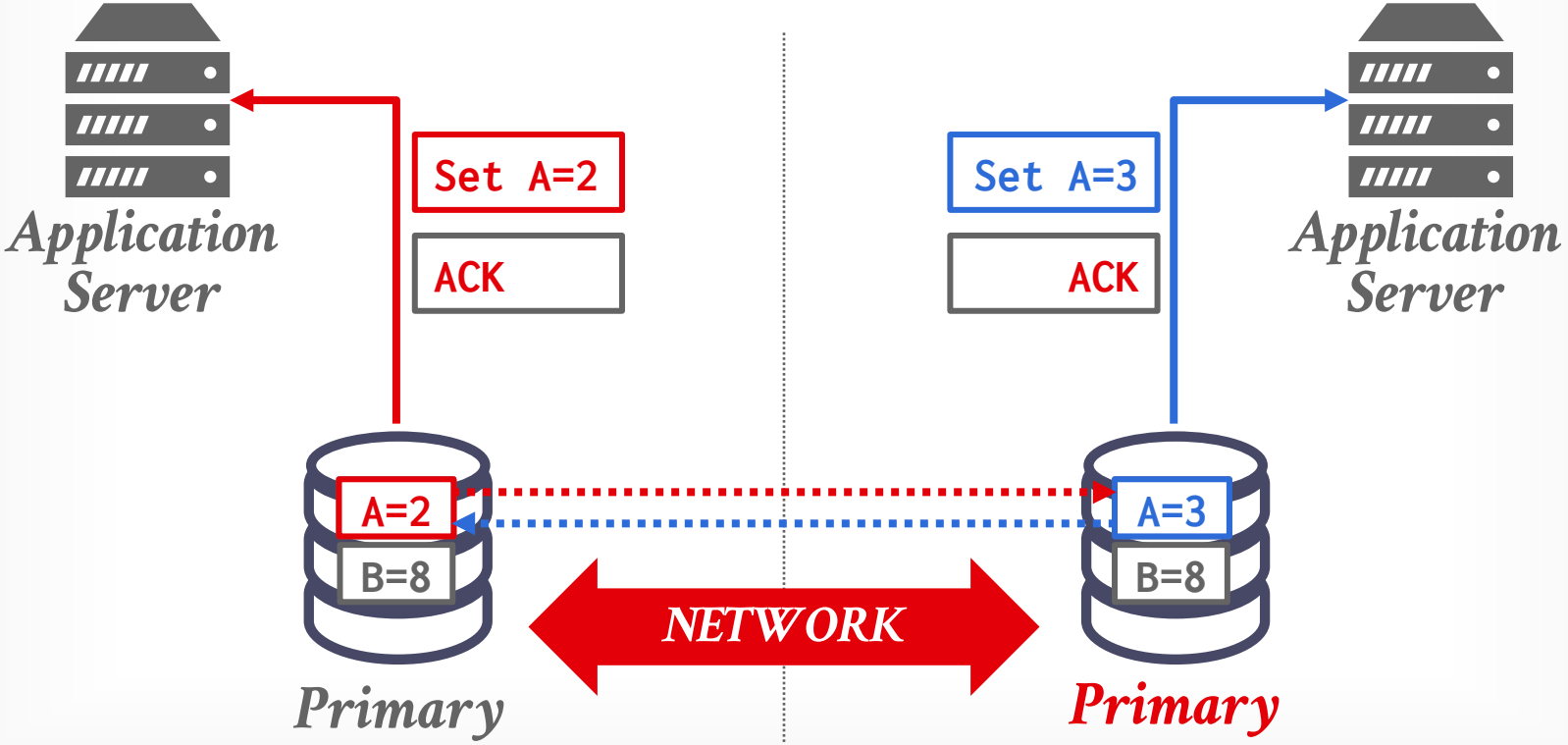
PARTITION TOLERANCE



PARTITION TOLERANCE



PARTITION TOLERANCE



PARTITION TOLERANCE

Choice #1: Halt the System

→ Stop accepting updates in any partition that does not have a majority of the nodes.

Choice #2: Allow Split, Reconcile Changes

→ Allow each side of partition to keep accepting updates.
→ Upon reconnection, perform reconciliation to determine the "correct" version of any updated record
→ Server-side: Last Update Wins
→ Client-side: Vector Clocks

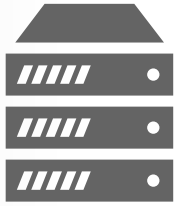
Don't Do This!

PACELC THEOREM

Extension to CAP proposed in 2010 to include consistency vs. latency trade-offs:

→ **P**artition Tolerant
→ **A**lways Available
→ **C**onsistent
→ **E**lse, choose during normal operations
→ **L**atency
→ **C**onsistency

LATENCY VS. CONSISTENCY



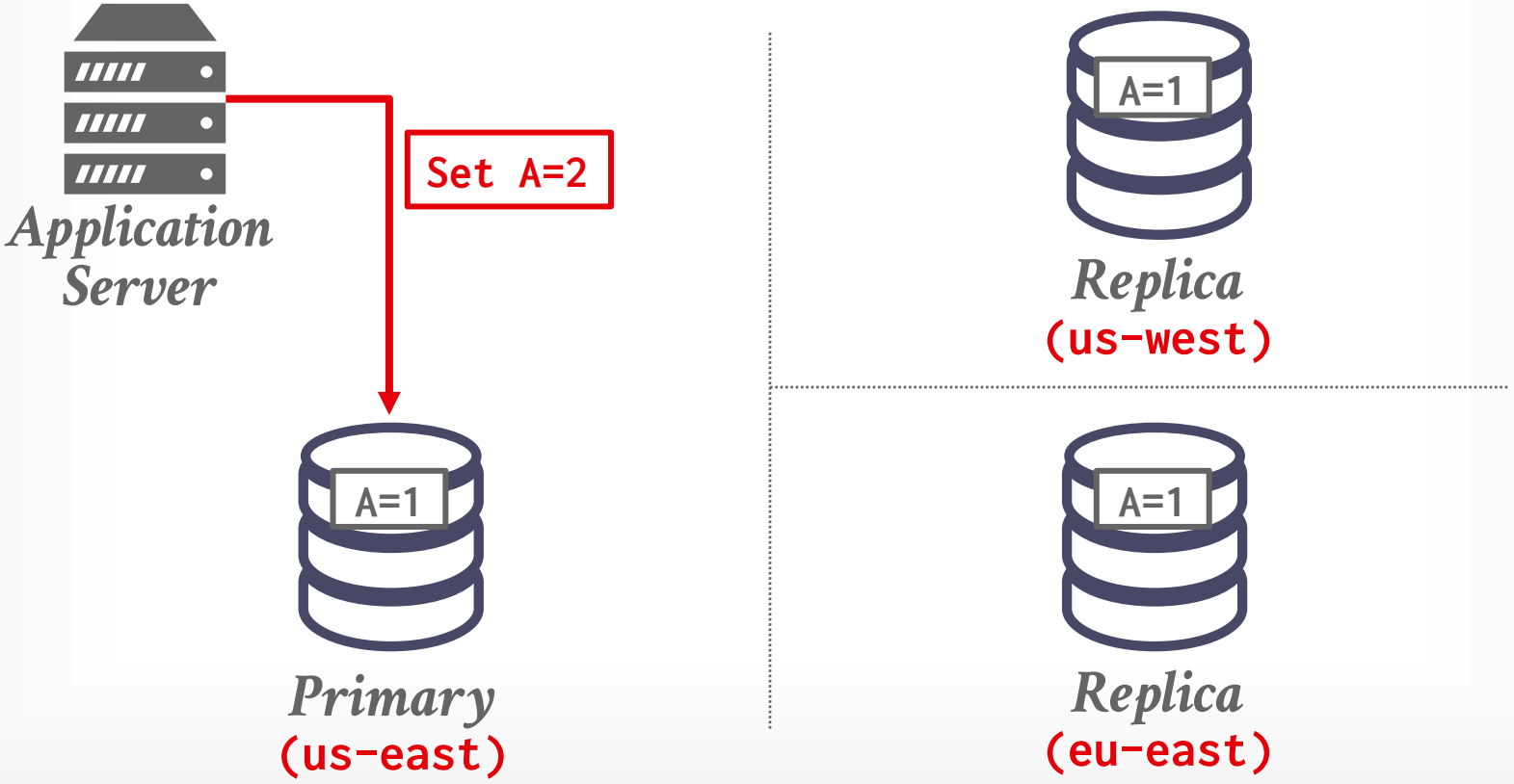
Application Server

Replica
(us-west)

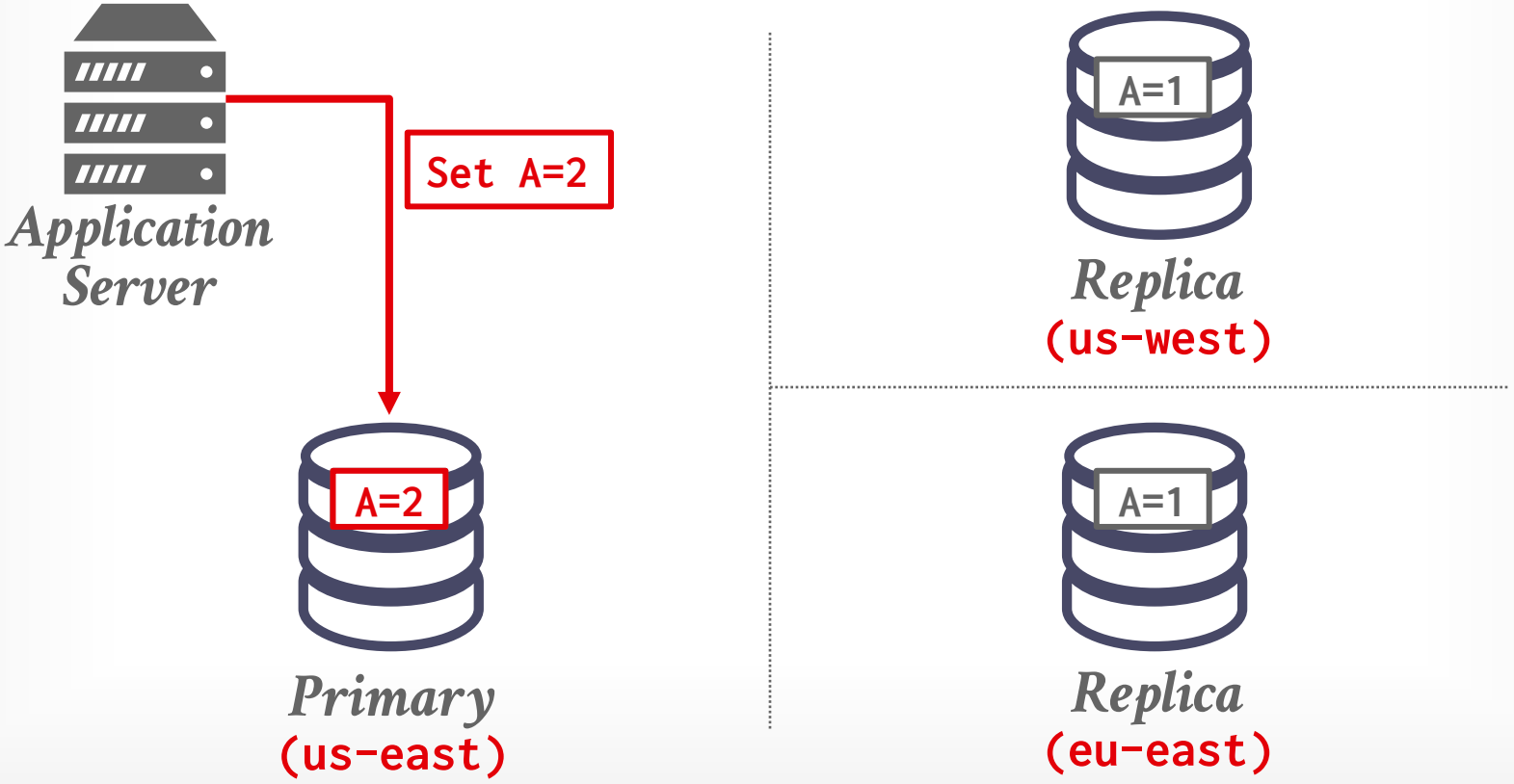
Primary
(us-east)

Replica
(eu-east)

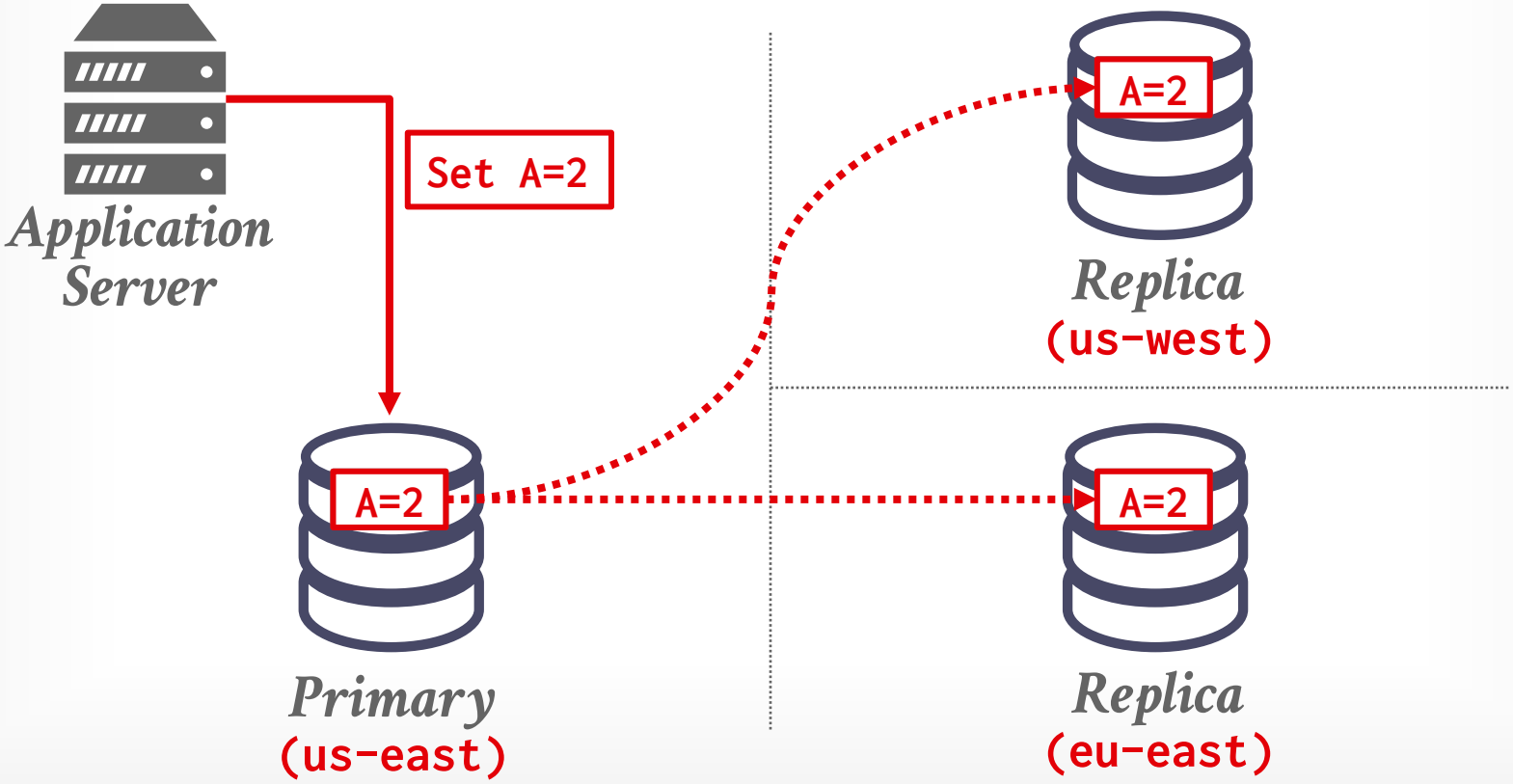
LATENCY VS. CONSISTENCY



LATENCY VS. CONSISTENCY



LATENCY VS. CONSISTENCY



LATENCY VS. CONSISTENCY

Application Server

Primary
(us-east)

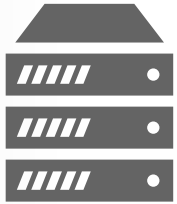
ACK

Replica
(us-west)

ACK

Replica
(eu-east)

LATENCY VS. CONSISTENCY



Application

Trade-off between how long to wait for acknowledgements and the latency of the DBMS.

Primary
(us-east)

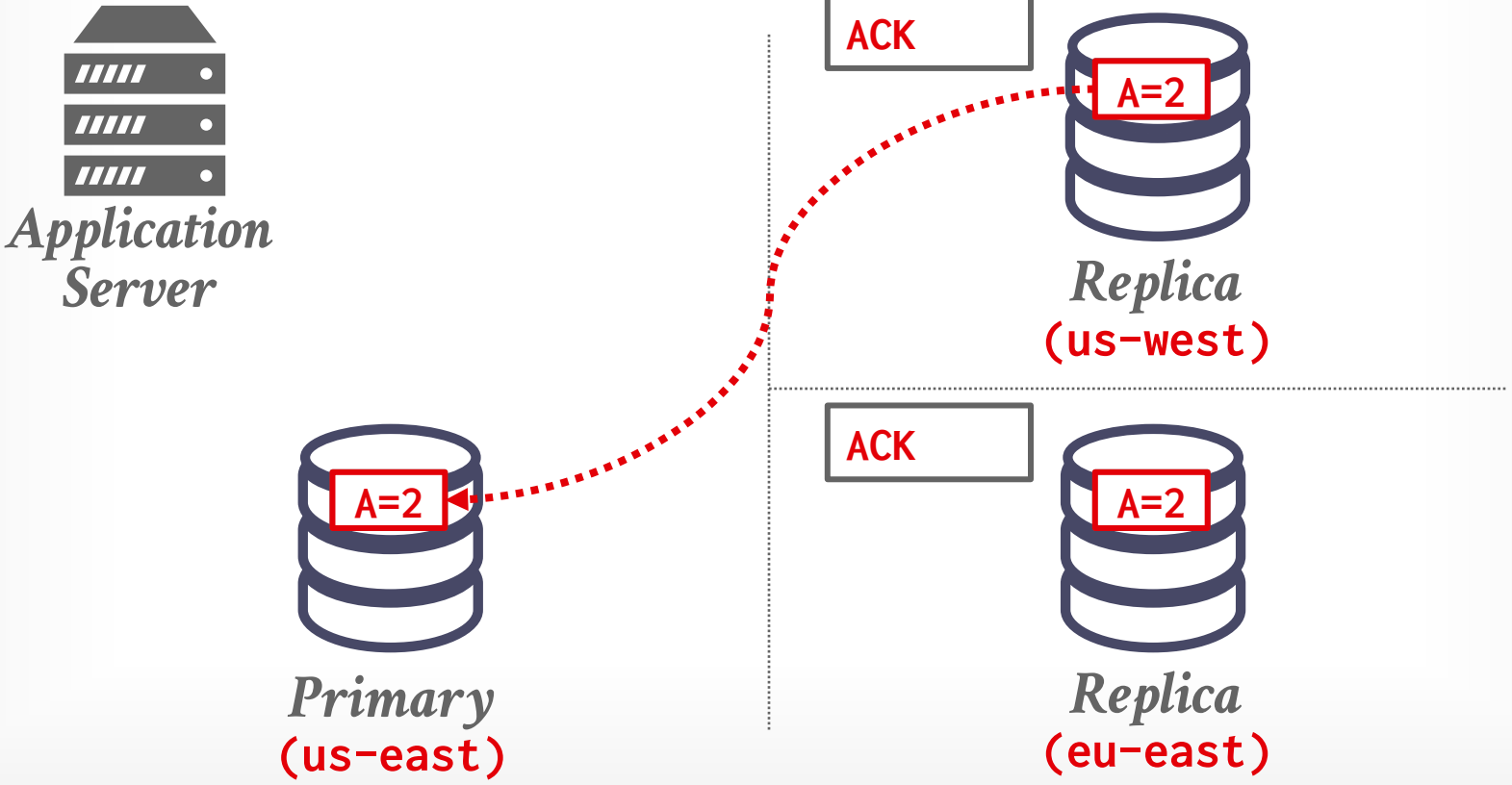
ACK

Replica
(us-west)

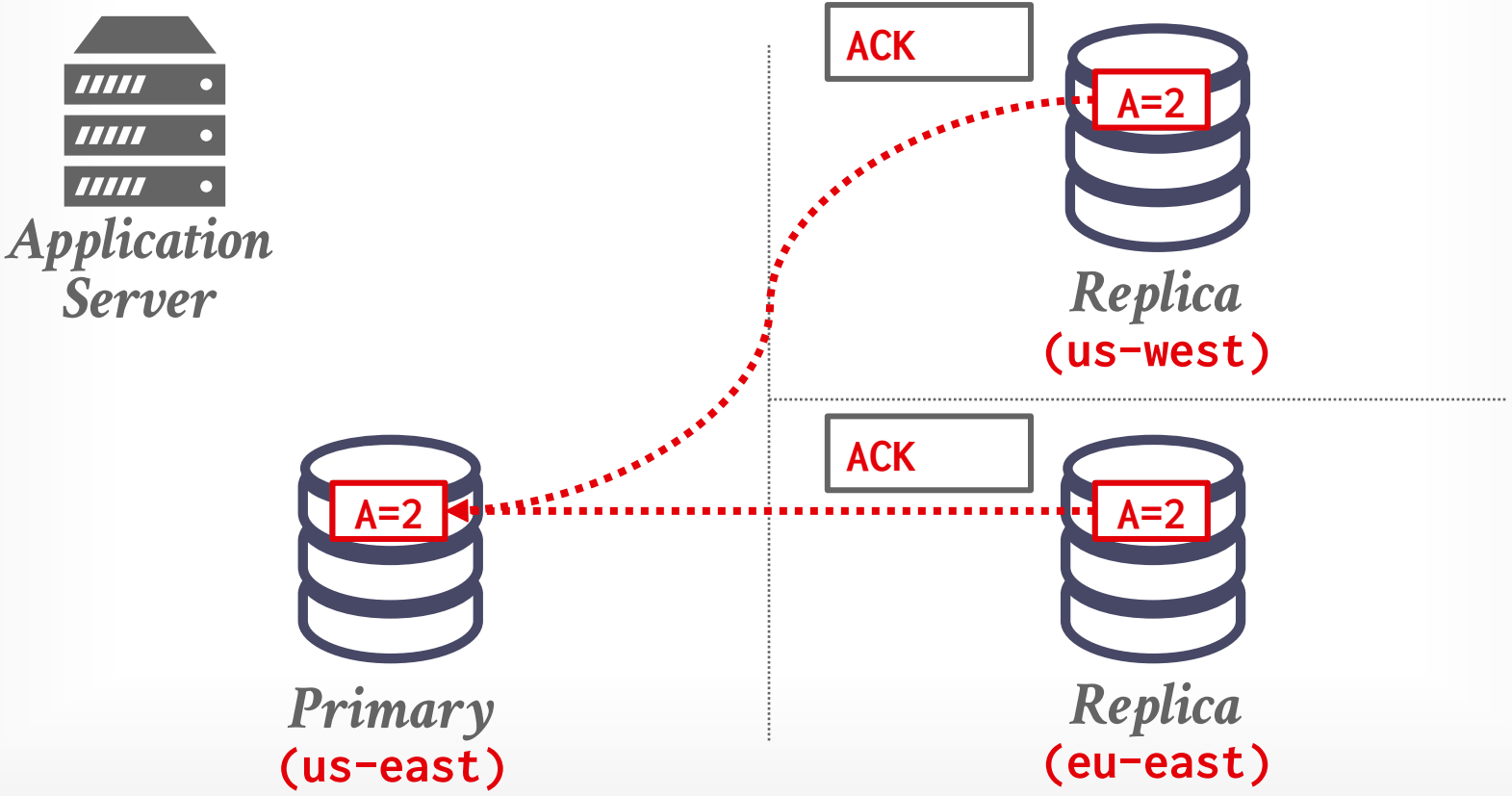
ACK

Replica
(eu-east)

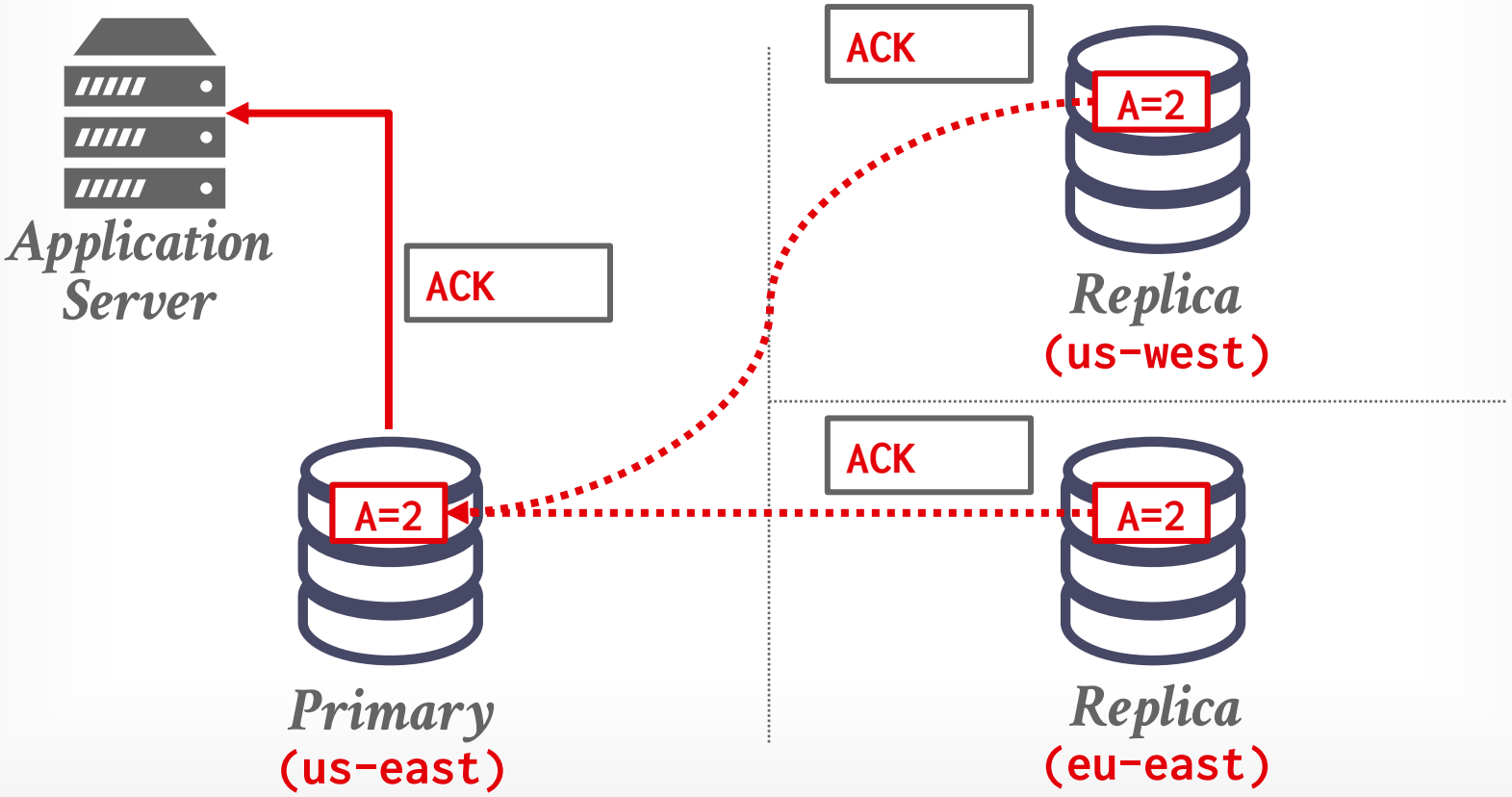
LATENCY VS. CONSISTENCY



LATENCY VS. CONSISTENCY



LATENCY VS. CONSISTENCY



OBSERVATION

The efficiency of a distributed join depends on the target tables' partitioning schemes.

One approach is to put entire tables on a single node and then perform the join.

→ You lose the parallelism of a distributed DBMS.

→ Costly data transfer over the network.

DISTRIBUTED JOIN ALGORITHMS

To join tables **R** and **S**, the DBMS needs to get the proper tuples on the same node.

Once the data is at the node, the DBMS then executes the same join algorithms that we discussed earlier in the semester.

→ Need to produce the correct answer as if all the data is located in a single node system.

SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

Each node joins its local data in parallel and then sends their results to a coordinating node.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

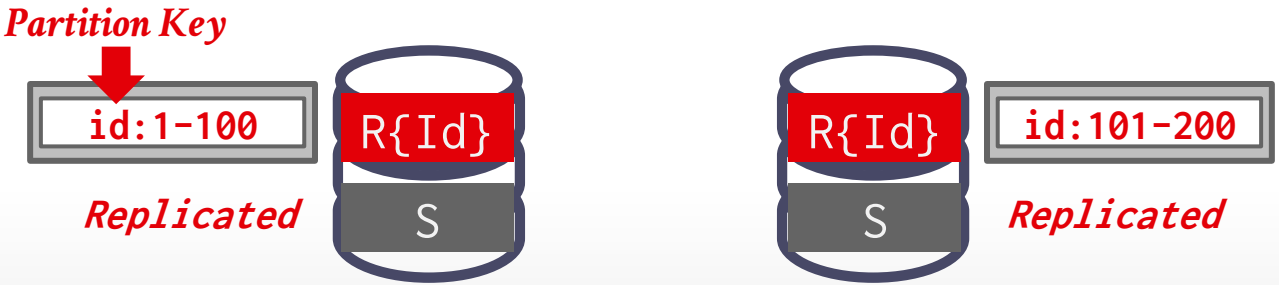
Each node joins its local data in parallel and then sends their results to a coordinating node.

SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

Each node joins its local data in parallel and then sends their results to a coordinating node.

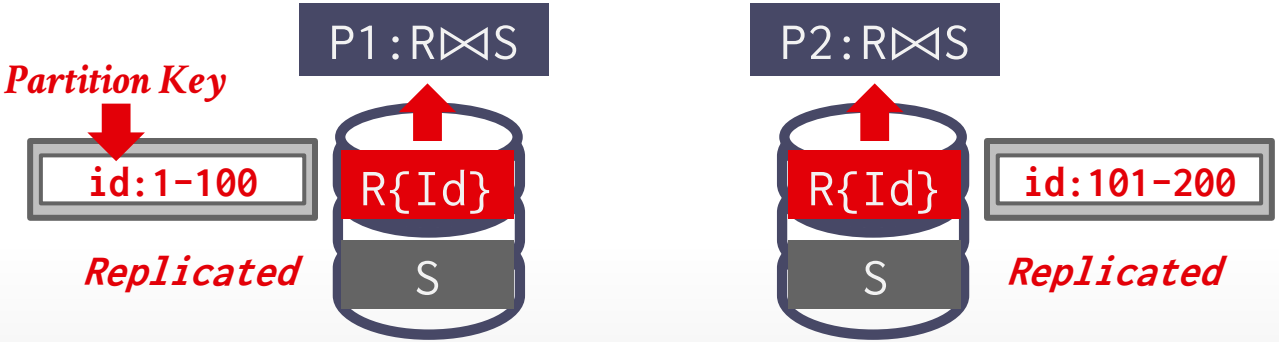


SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

Each node joins its local data in parallel and then sends their results to a coordinating node.

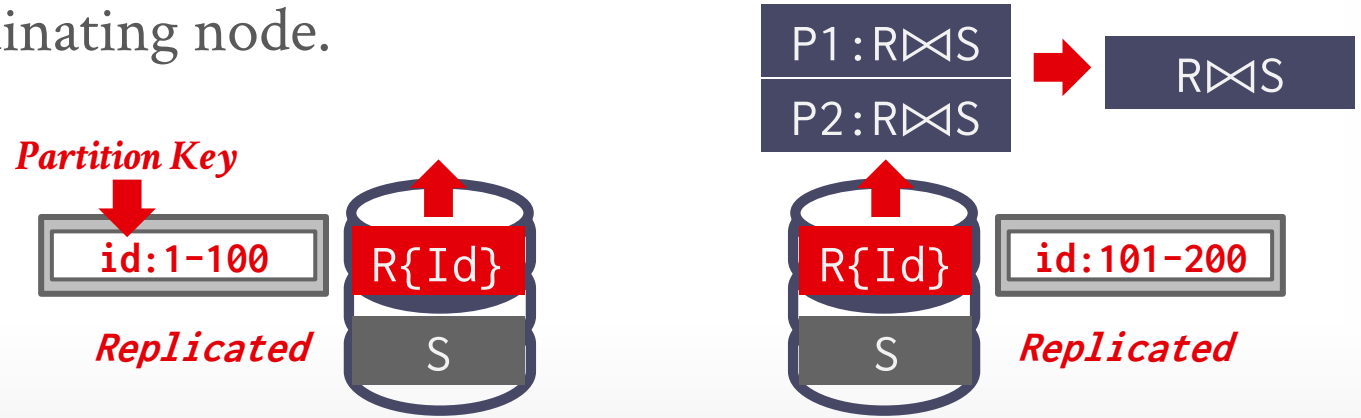


SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

Each node joins its local data in parallel and then sends their results to a coordinating node.



SCENARIO #2

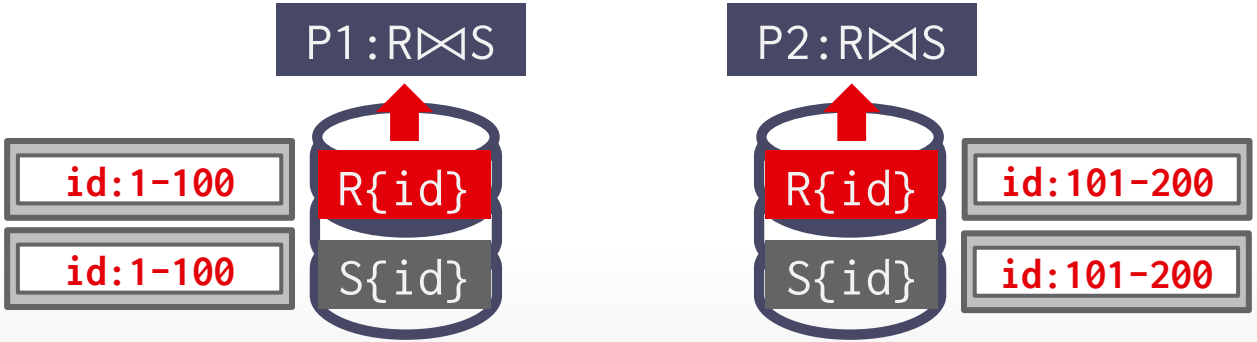
Both data sets are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

SCENARIO #2

Both data sets are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

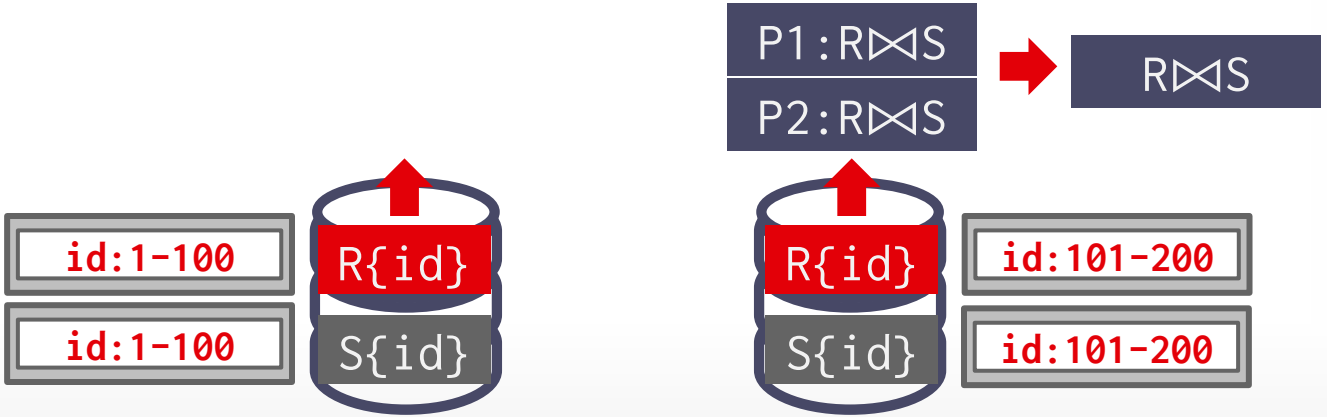
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #2

Both data sets are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3: BROADCAST JOIN

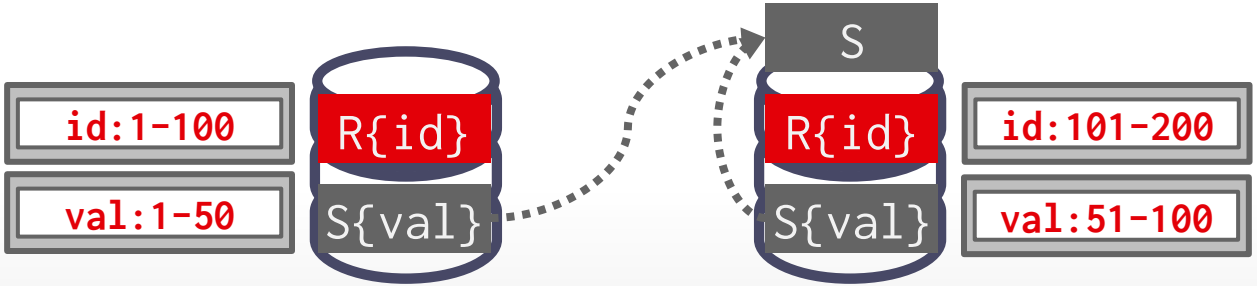
Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

SCENARIO #3: BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

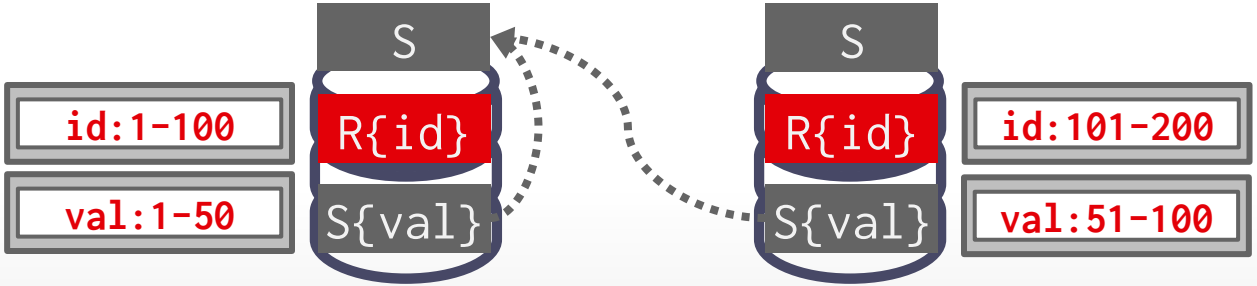
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3: BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

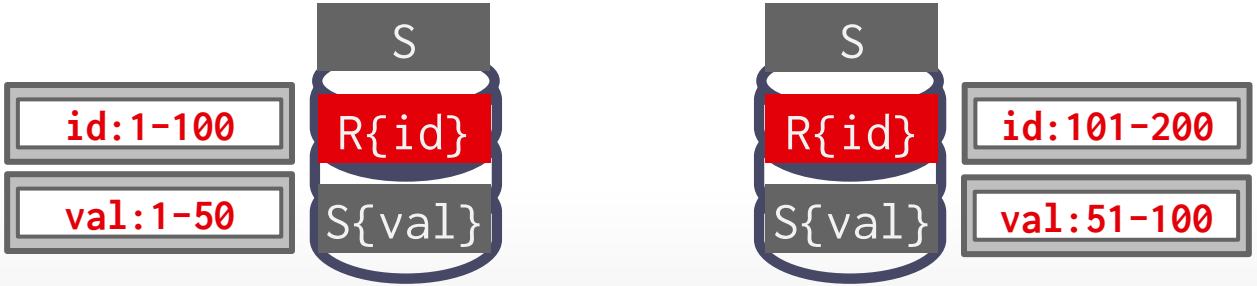
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3: BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

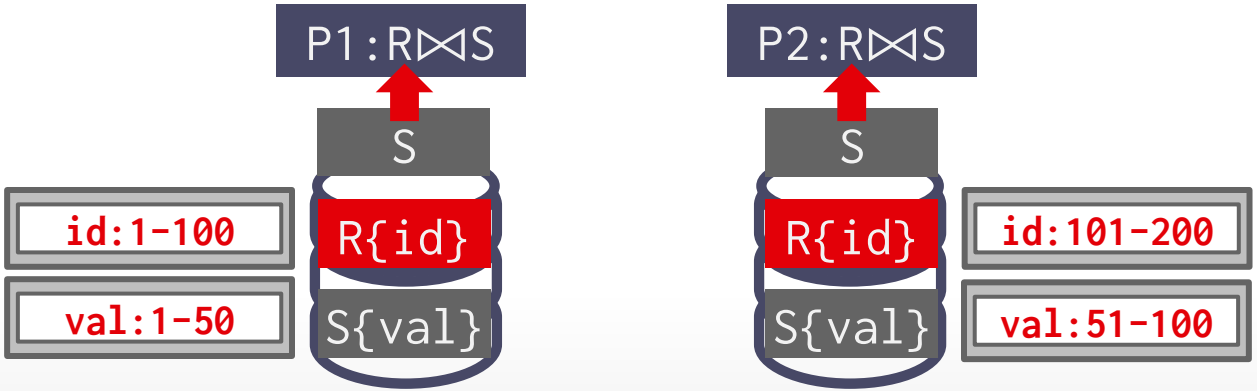
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3: BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

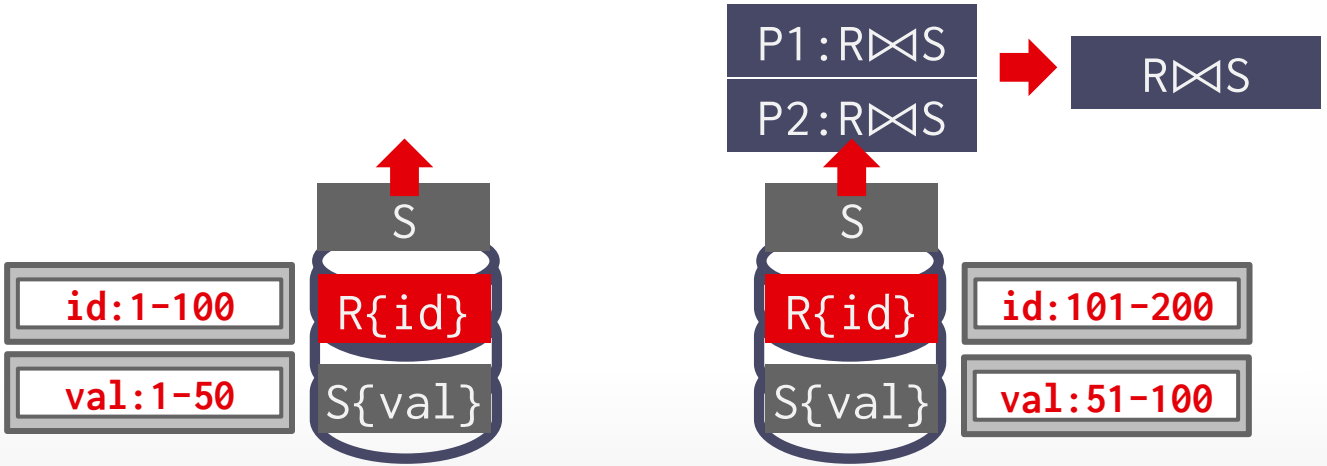
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3: BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4: SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

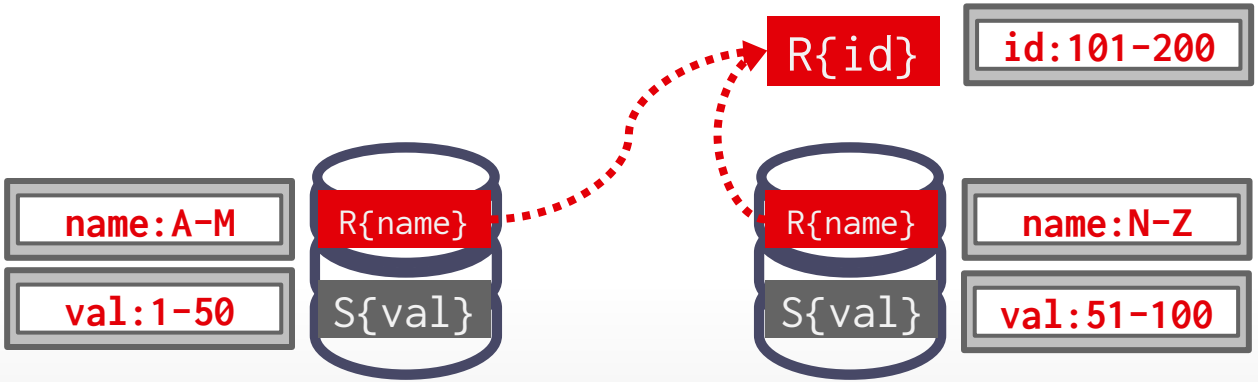
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

SCENARIO #4: SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

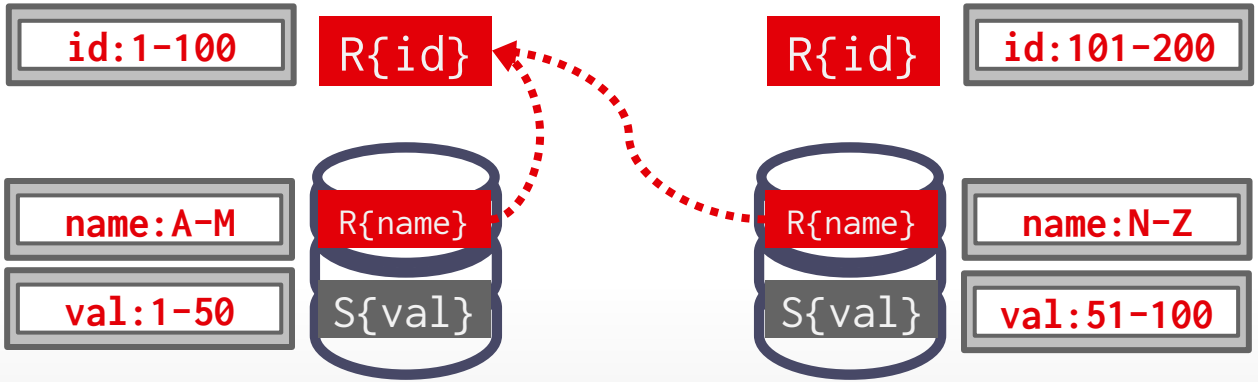


SCENARIO #4: SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

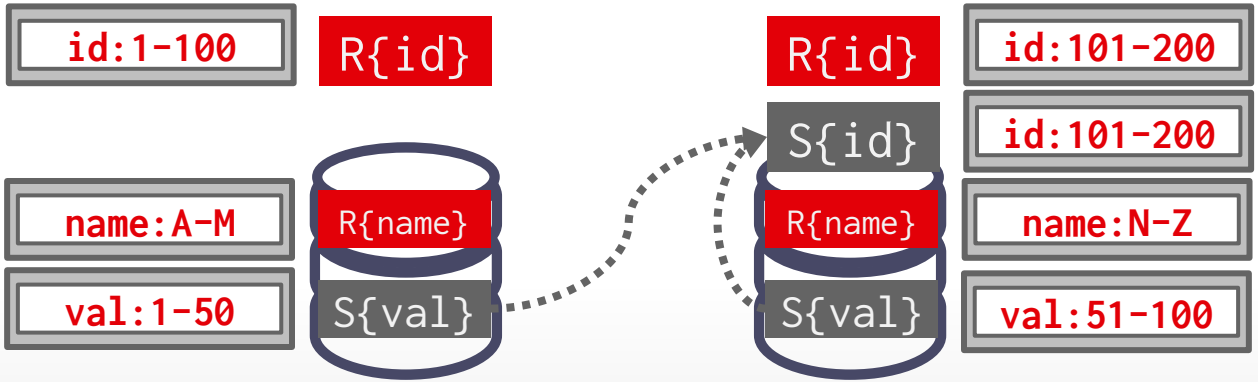


SCENARIO #4: SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

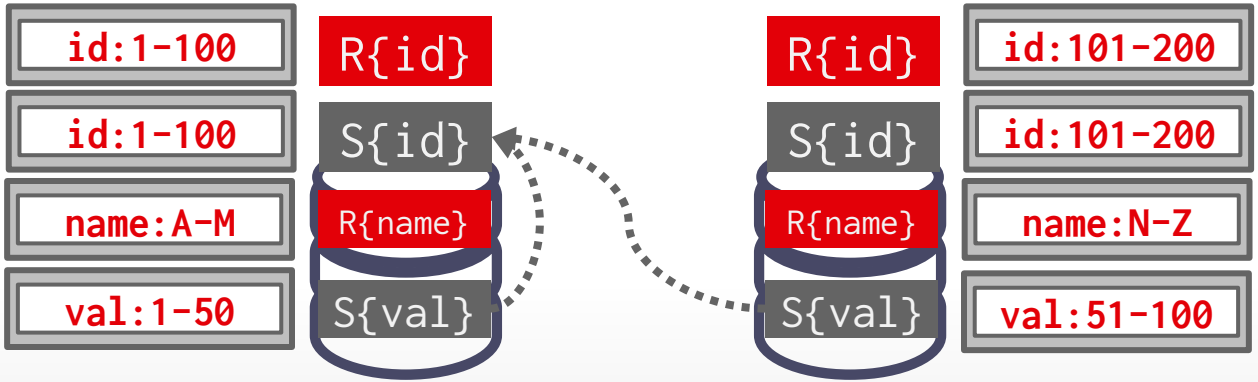


SCENARIO #4: SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

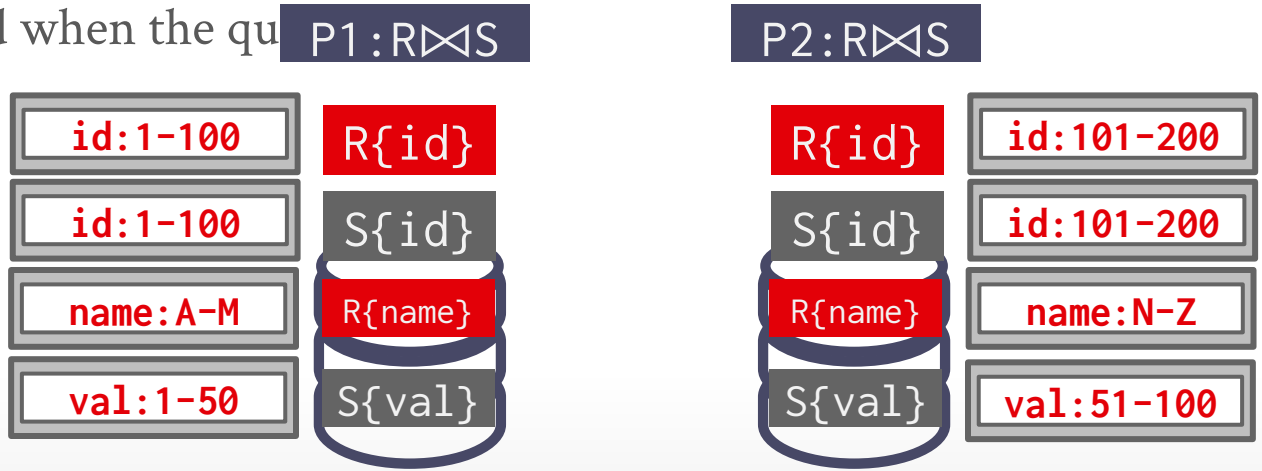


SCENARIO #4: SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the qu

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

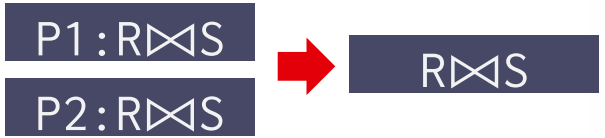


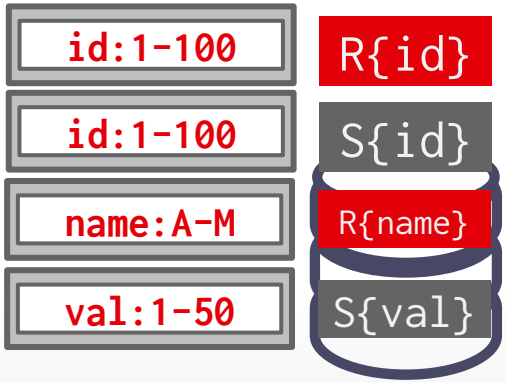
SCENARIO #4: SHUFFLE JOIN

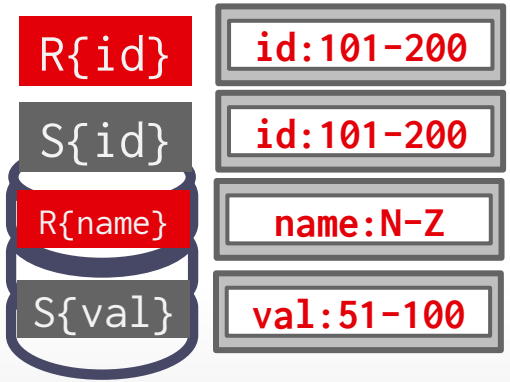
Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```







SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$\mathbf{Dim_{semi} = \Pi_{id} (\sigma_{zip = 15213} Dim)}$$

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$\mathbf{Dim_{semi} = \Pi_{id} (\sigma_{zip = 15213} Dim)}$$

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$\mathbf{Dim_{semi} = \Pi_{id} (\sigma_{zip = 15213} Dim)}$$

$$\mathbf{F\text{-}small = Fact \bowtie Dim_{semi}}$$

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

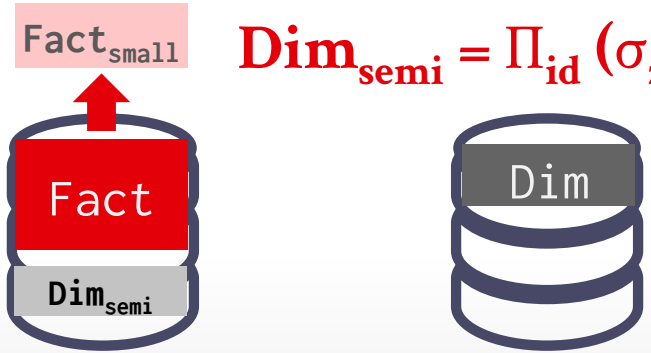
→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$\text{Dim}_{\text{semi}} = \Pi_{\text{id}} (\sigma_{\text{zip} = 15213} \text{Dim})$

$\text{F-small} = \text{Fact} \bowtie \text{Dim}_{\text{semi}}$



SEMI-JOIN OPTIMIZATION

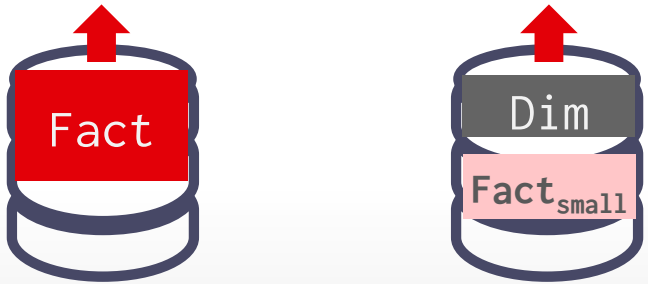
Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$\textbf{Result} = \Pi_{\textbf{price}}(\textbf{Dim} \bowtie \textbf{Fact}_{\textbf{small}})$$



OBSERVATION

Direct communication between compute nodes means the DBMS knows which nodes will participate in query execution ahead of time.

But data skew can cause imbalances…

A better approach is to dynamically adjust compute resources on the fly as a query executes.

SHUFFLE PHASE

Redistribute of intermediate data across nodes between query plan pipelines.

→ Can repartition / rebalance data based on observed data characteristics.

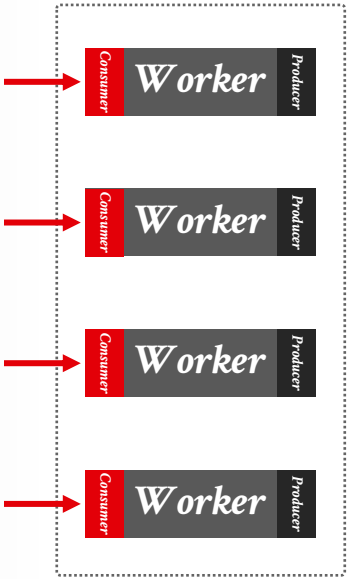
Google Big Query

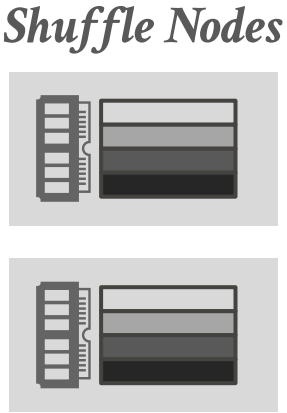
Some DBMSs support standalone fault-tolerant shuffle services.

→ Example: You can replace Spark's built-in in-memory shuffle implementation or replace it with a separate service.

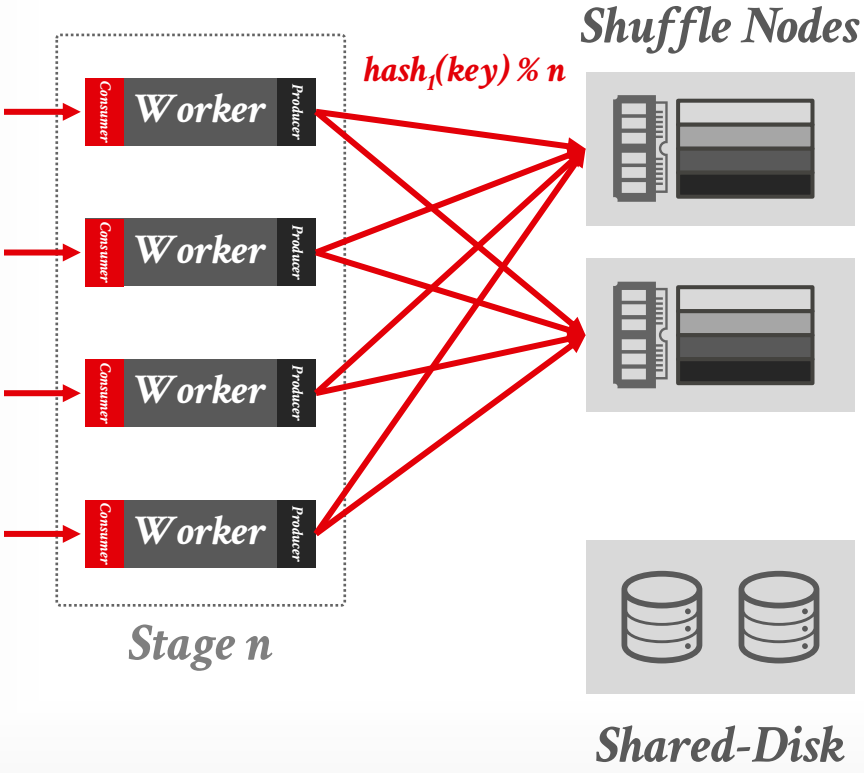
Apache Uniffle

SHUFFLE PHASE

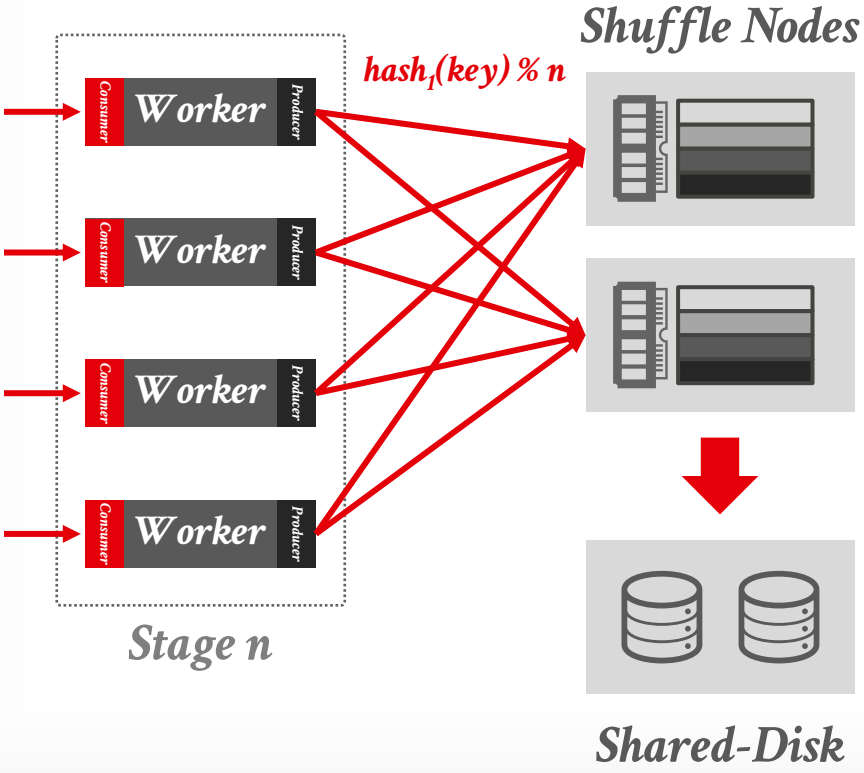




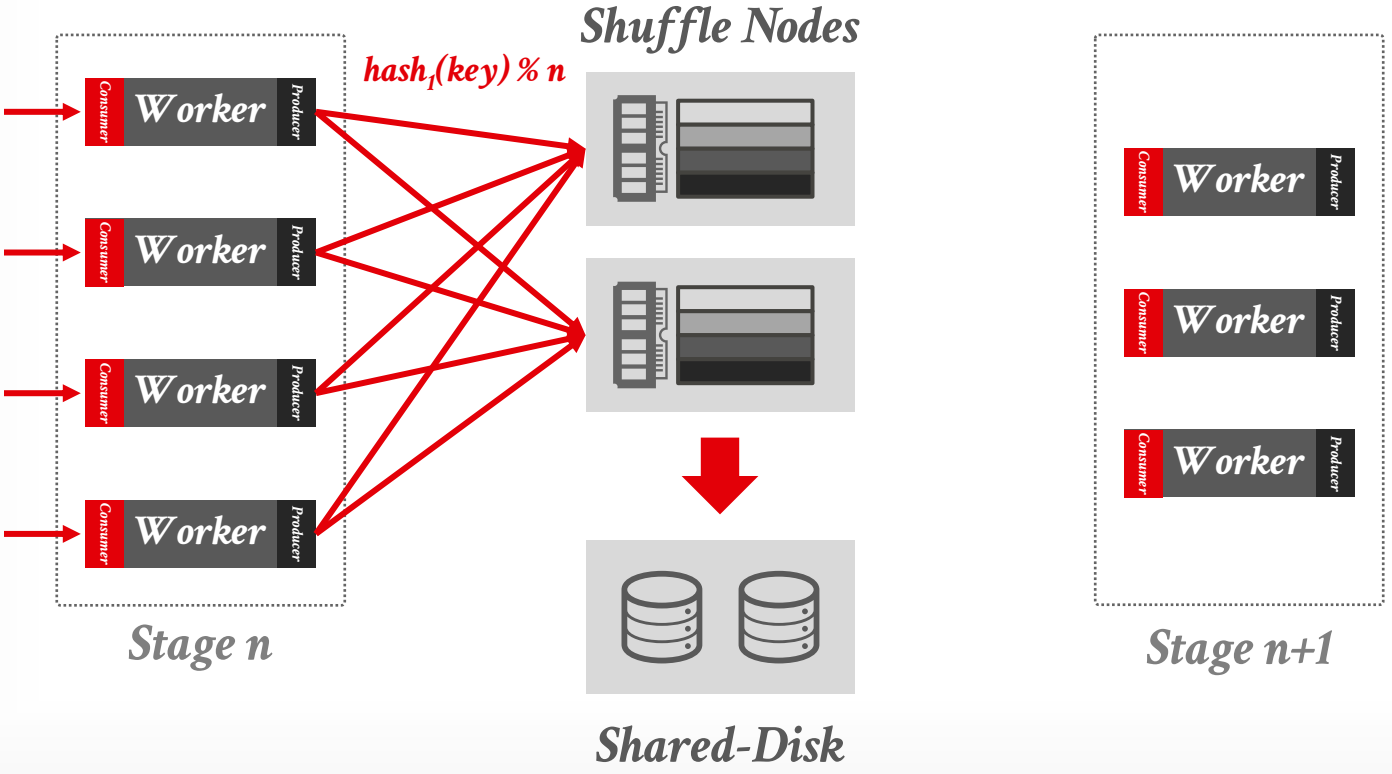
SHUFFLE PHASE



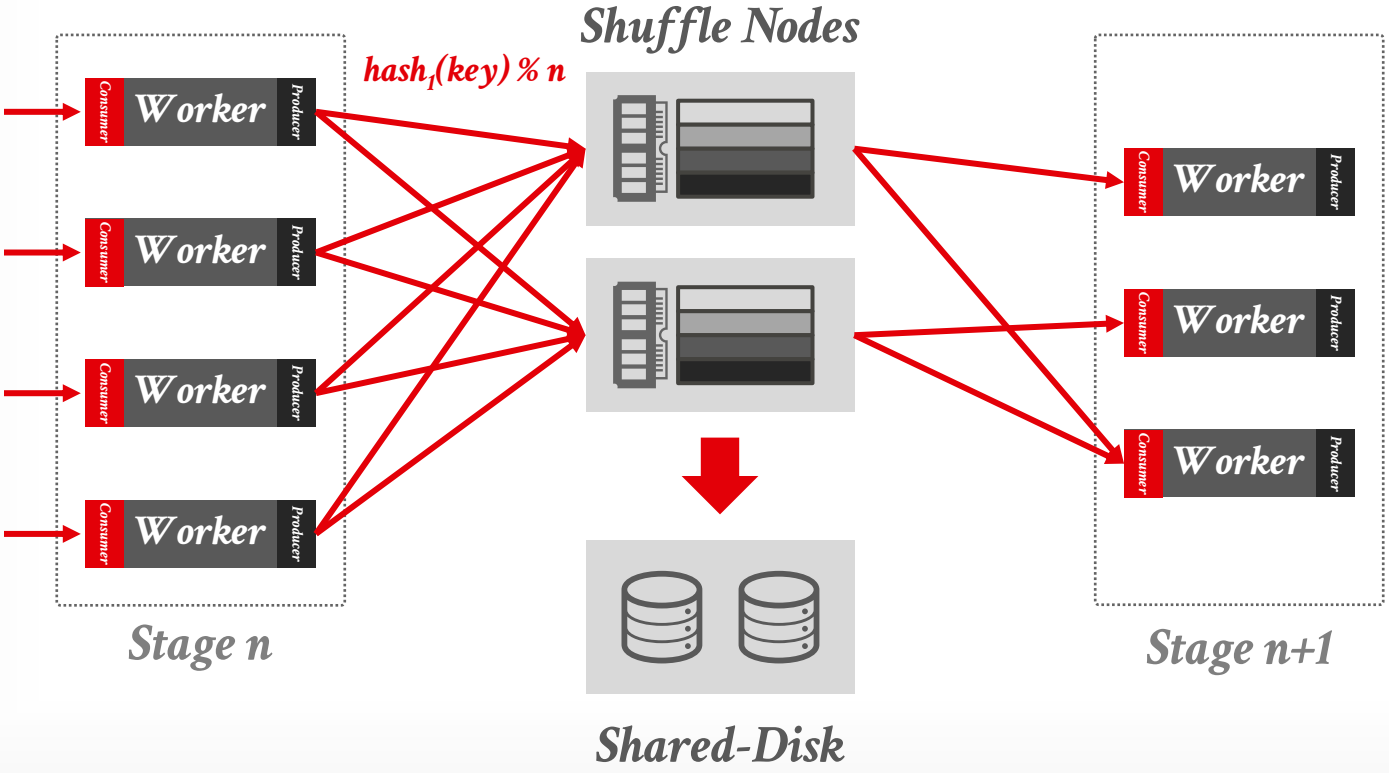
SHUFFLE PHASE



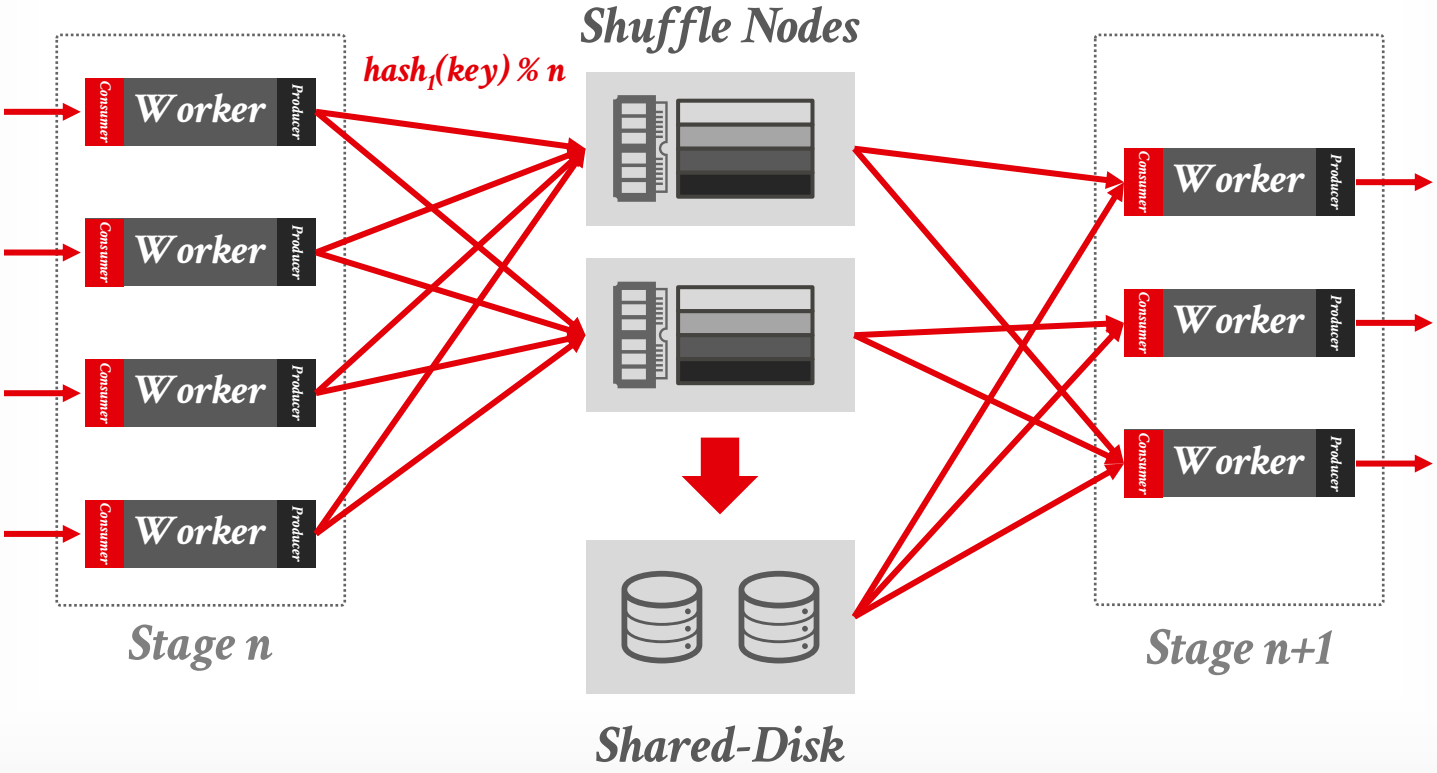
SHUFFLE PHASE



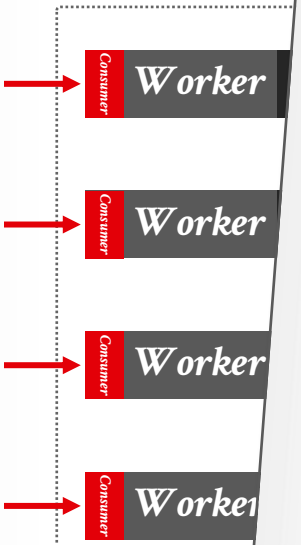
SHUFFLE PHASE



SHUFFLE PHASE



SHUFFLE PHASE

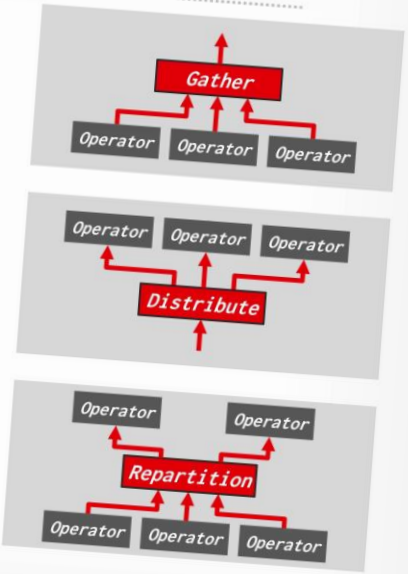


EXCHANGE OPERATOR

Exchange Type #1 – Gather
→ Combine the results from multiple workers into a single output stream.

Exchange Type #2 – Distribute
→ Split a single input stream into multiple output streams.

Exchange Type #3 – Repartition
→ Shuffle multiple input streams across multiple output streams.
→ Some DBMSs always perform this step after every pipeline (e.g., Google BigQuery).



Source: Craig Freedman

DATABASE SYSTEMS (FALL 2025)

Shared-Disk

CONCLUSION

Maintaining transactional consistency across multiple nodes is hard. Bad things will happen.
→ Don't let the "unwashed masses" go without txns!

2PC / Paxos / Raft are the most common protocols to ensure correctness in a distributed DBMS.

Moving data during distributed joins is expensive.

NEXT CLASS

Final Review

15-721 in a single lecture!