INFO-H515: BIG DATA: DISTRIBUTED MANAGEMENT

Lecture 5: NoSQL Databases

Dimitris Sacharidis

2023–2024

OUTLINE

Overview of NoSQL

Consistency - Availability - Partition

Concurrency Control Protocols

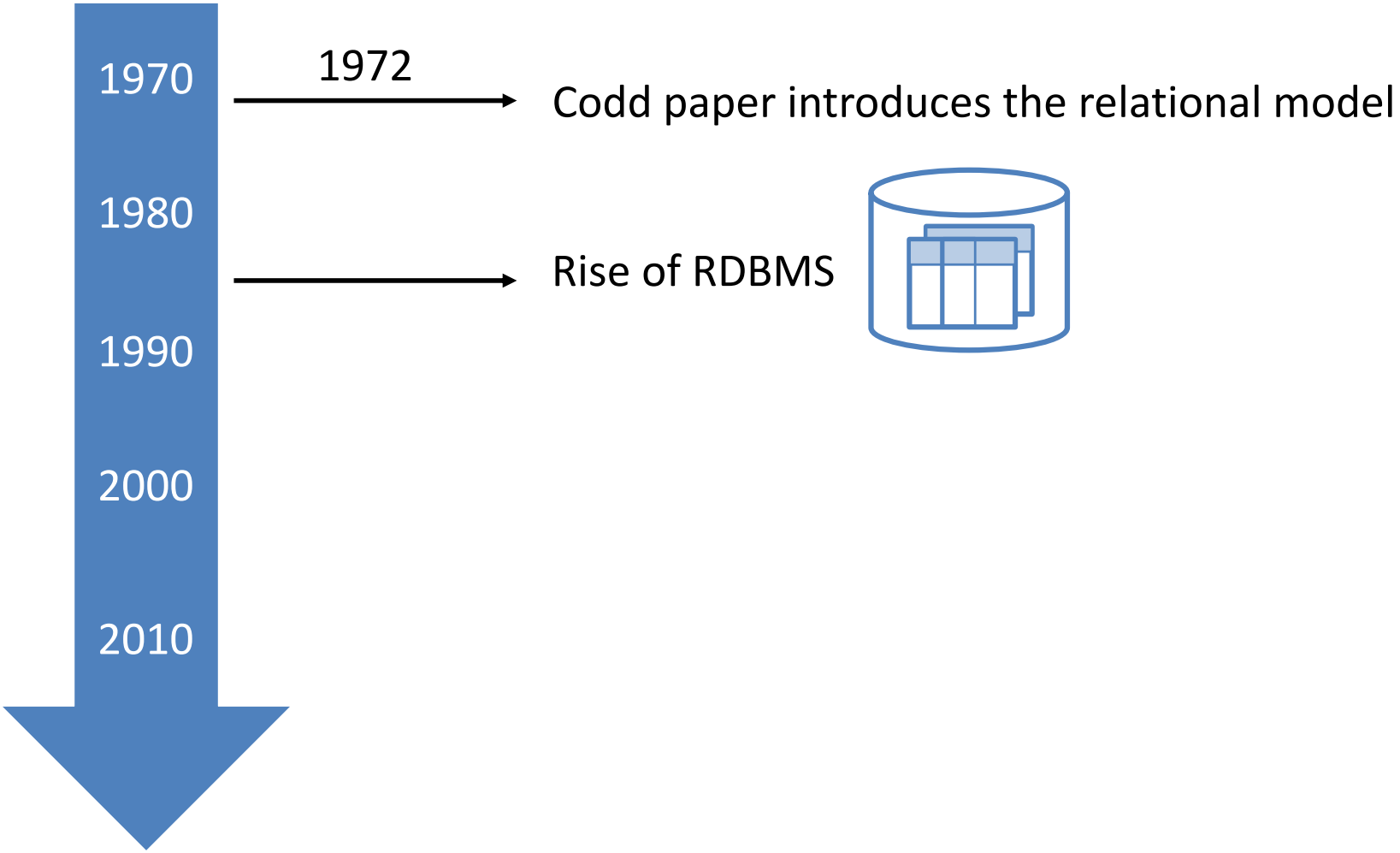
- The Two-Phase Commit Protocol (2PC)
- Multiversion Concurrency Control (MVCC)
- The Paxos Protocol

Key-Value Stores

NoSQL Rebuttal

OVERVIEW OF NOSQL

DBMS HISTORY



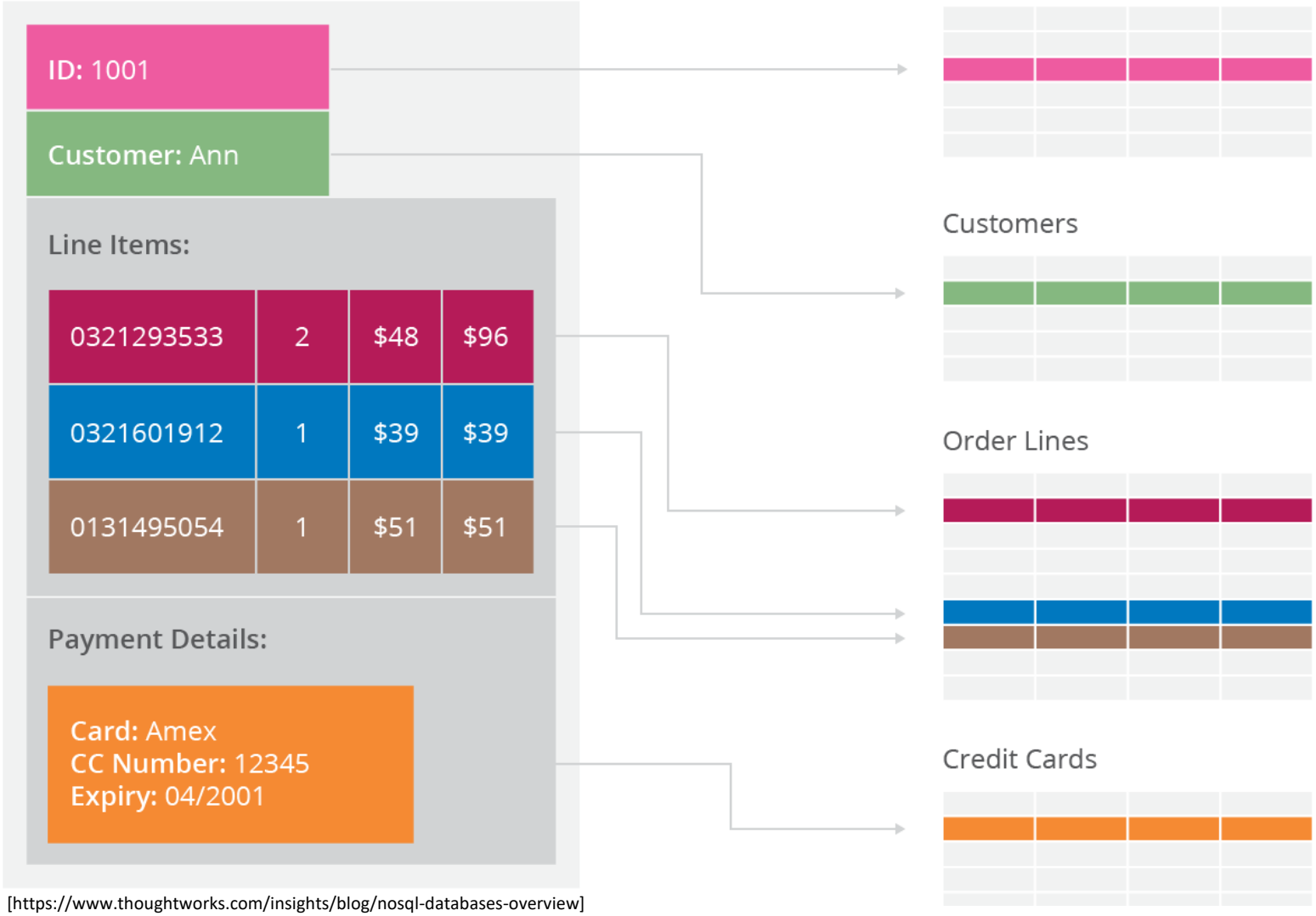
RDBMS - PROS

- A simple and intuitive model
- Often convenient for real-life data
  - Other models useful in some settings
  - XML for semi-structured data, RDF for semantic Web data
- An elegant mathematical foundation
  - Set and multi-set theory
  - Relational algebra and calculi
- Allows efficient algorithms
- ACID
- Industrial strength implementations are available

RDBMS - CONS

- Impedance mismatch

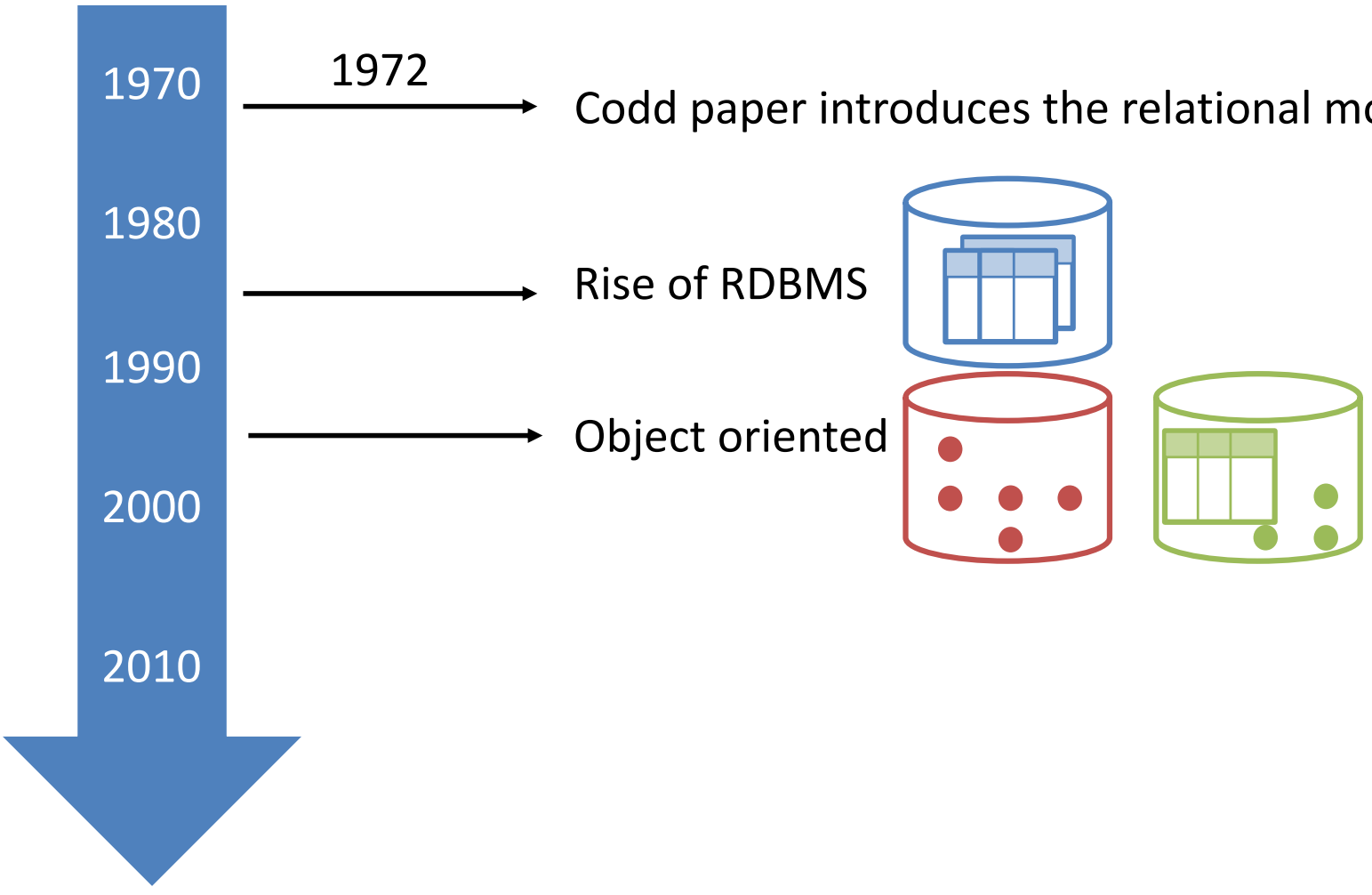
IMPEDANCE MISMATCH



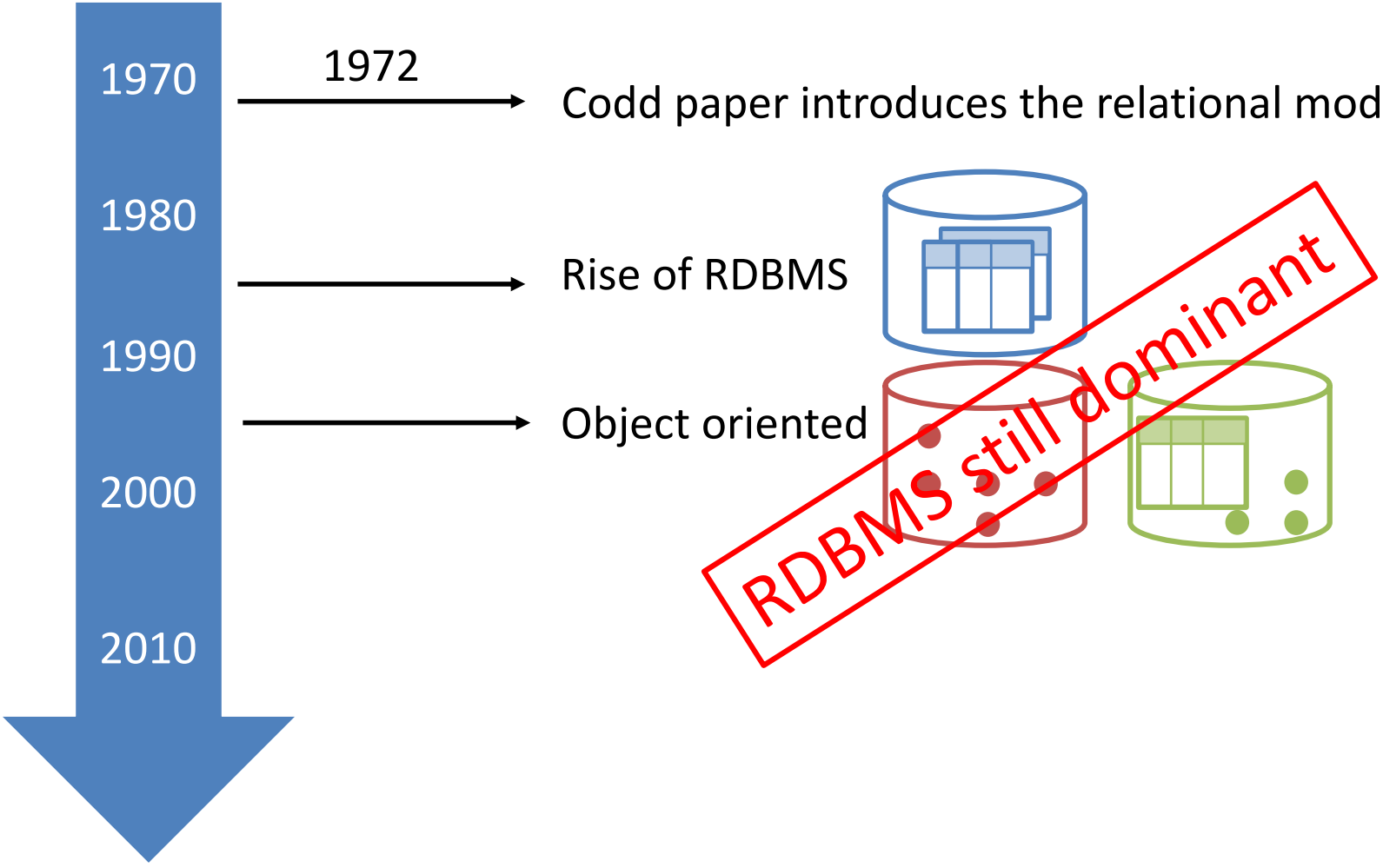
[https://www.thoughtworks.com/insights/blog/nosql-databases-overview]

what developers want ≠ what RDBMS offers

DBMS HISTORY



DBMS HISTORY

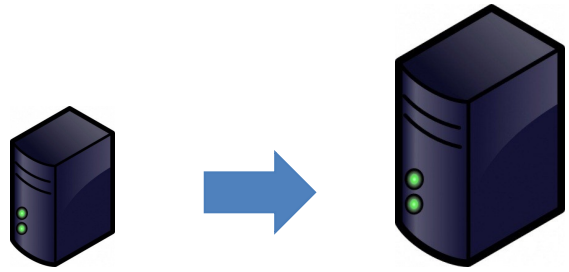


RDBMS - CONS

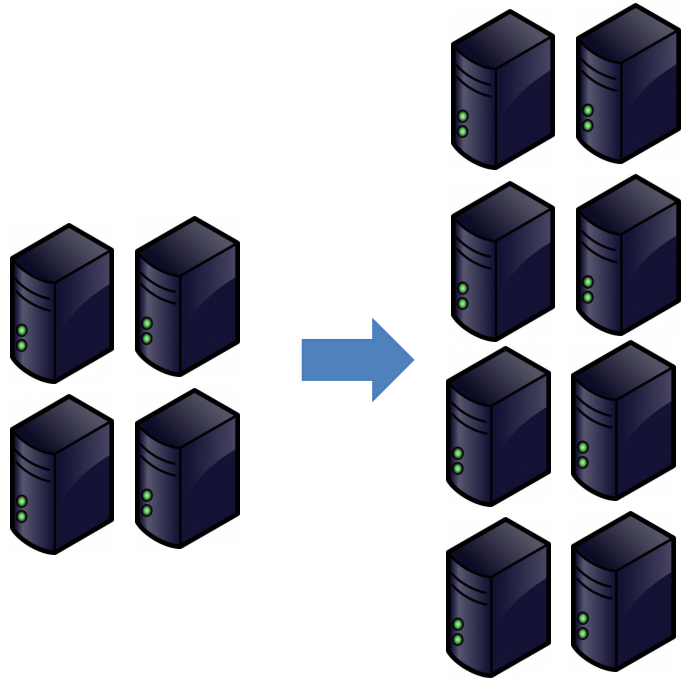
- Impedance mismatch
- Not built for distributed data management
  - Deploying on clusters is hard
- Single point of failure
- Performance
  - Not tailored for specialized applications
- Scaling
  - Can scale up
  - Cannot scale out

SCALE UP VS OUT

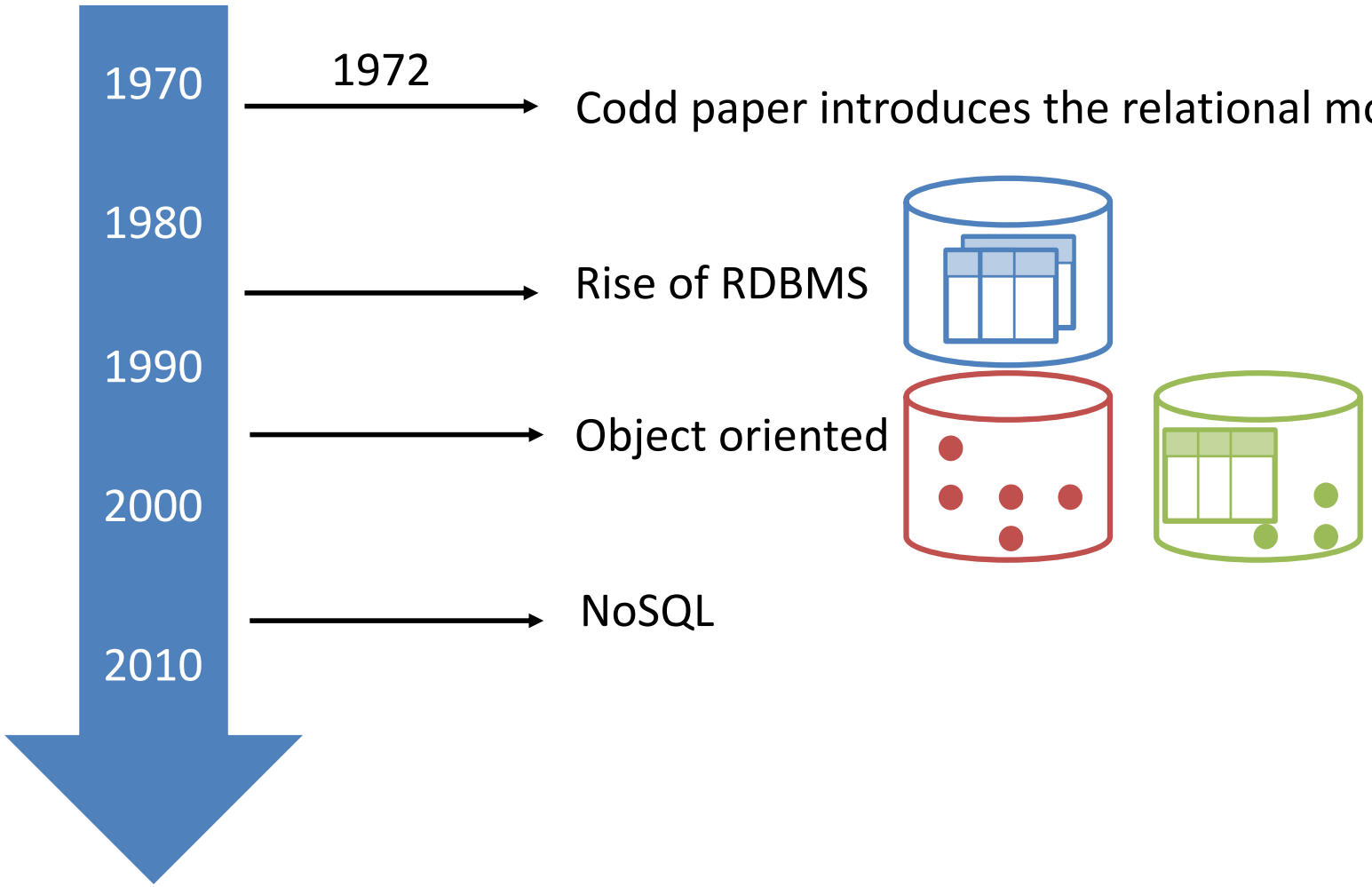
- Scale up
  - Grow capacity by getting a more powerful machine
  - Expensive



- Scale out
  - Grow capacity by adding more components
  - Buy some new faster machine, replace old



DBMS HISTORY



WHAT IS NOSQL?

- Side note
  - Term introduced accidentally as a hash tag for a gathering
- Practically means "not only SQL"
- Hard to define – common characteristics
  - Non relational
  - Cluster-friendly (not all)
  - Open source (most)
  - Manage large-scale data – inspired by modern Web challenges
  - Schema-less

NOSQL LANDSCAPE

- Essentially four types
  - Key-value stores
  - Document stores
  - Column families
  - Graph databases

KEY-VALUE STORES

- Simple data model
  - Everything stored inside key-value pairs

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623



redis
DynamoDB
amazon
web services

DOCUMENT STORES

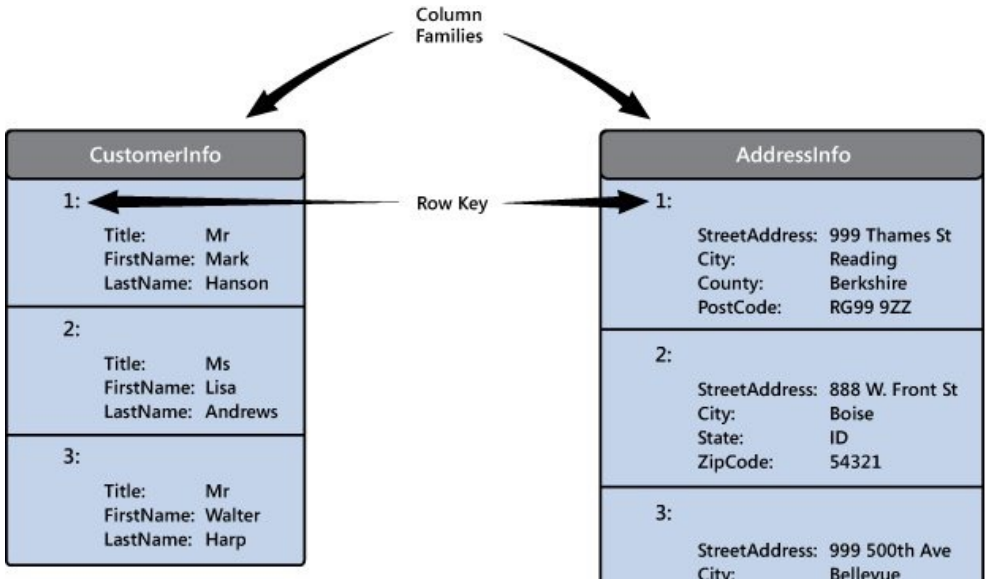
- Document-oriented data model
  - Store contents of every document
  - JSON

```
{“id” : 1001,
  “customer_id” : 7231,
  “items” : [{“product_id” : 4555, “quantity” : 10},
                {”product_id” : 1213, “quantity” : 1}]
}

{“id” : 1002,
  “customer_id” : 231,
  “items” : [{“product_id” : 55, “quantity” : 34}]
}
```

COLUMN FAMILIES

- Uses tables, rows, and columns
  - But names and format of the columns can vary from row to row in the same table



cassandra

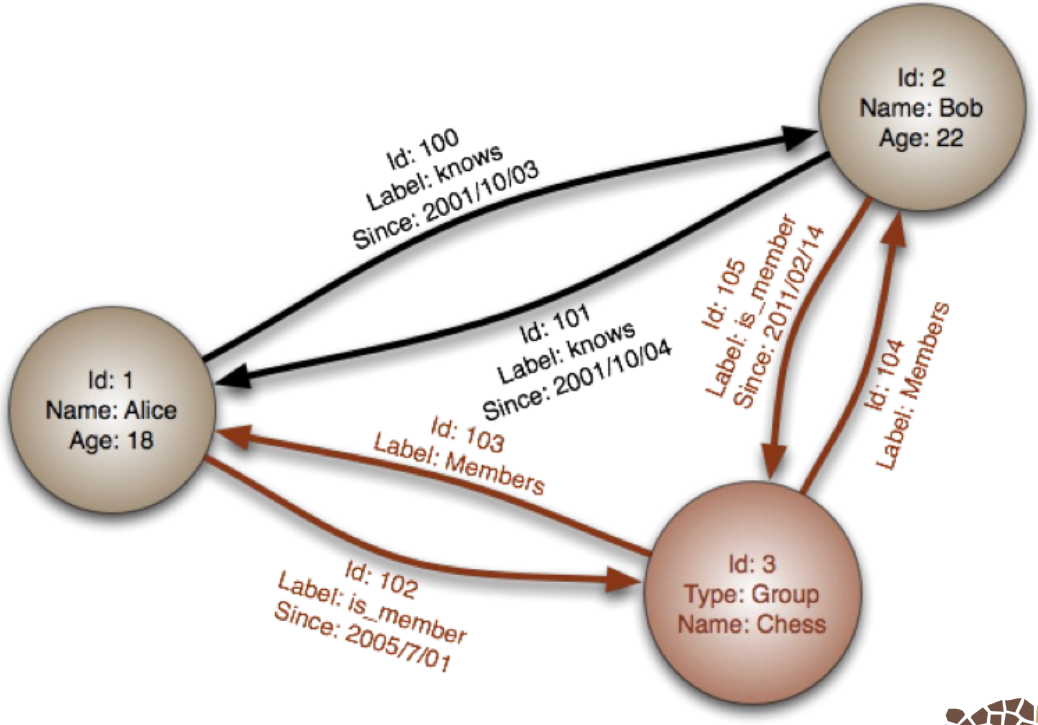
APACHE HBASE

DynamoDB

amazon web services

GRAPH DATABASES

- Graph data model
  - RDF

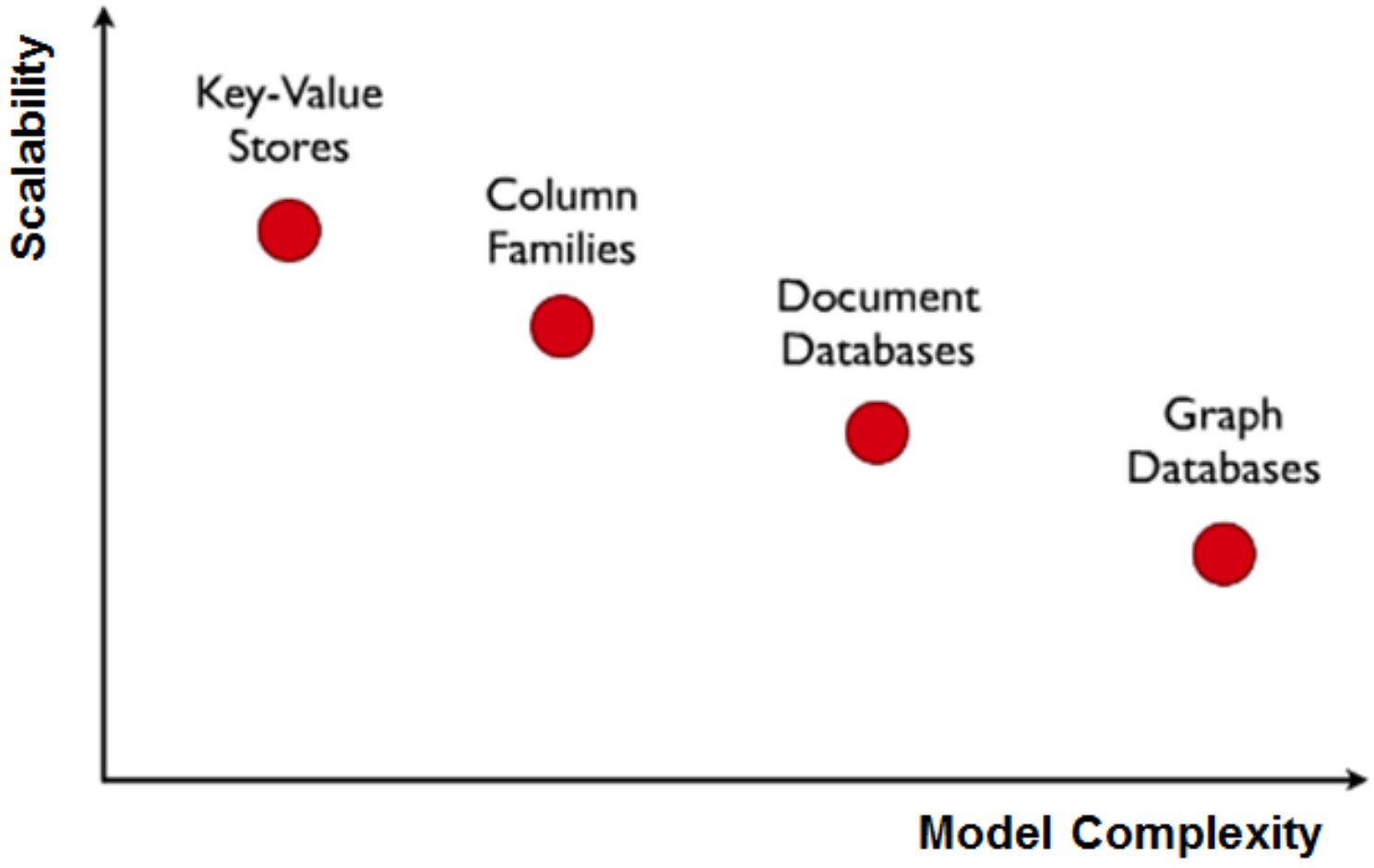


neo4j

Jena

APACHE GIRAPH

COMPARISON


Scalability
Key-Value Stores
Column Families
Document Databases
Graph Databases
Model Complexity

NOSQL AND RELATED SYSTEMS

Based on table made by BIll Howe, U. of Washington

Year	System/ Paper	Scale to 1000s	Primary Index	Secondary Index	Transactions	Joins/ Analytics	Integrity Constraints	Views	Language/ Algebra	Data model	My Label
1971	RDBMS	O	✓	✓	✓	✓	✓	✓	✓	tables	SQL-like
2003	memcached	✓	✓	O	O	O	O	O	O	key-val	lookup
2004	MapReduce	✓	O	O	O	✓	O	O	O	key-val	MR
2005	CouchDB	✓	✓	✓	record	MR	O	✓	O	document	filter/MR
2006	BigTable (Hbase)	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter/MR
2007	MongoDB	✓	✓	✓	EC, record	O	O	O	O	document	filter
2007	Dynamo	✓	✓	O	O	O	O	O	O	key-val	lookup
2007	SimpleDB	✓	✓	✓	O	O	O	O	O	ext. record	filter
2008	Pig	✓	O	O	O	✓	/	O	✓	tables	RA-like
2008	HIVE	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2008	Cassandra	✓	✓	✓	EC, record	O	✓	✓	O	key-val	filter
2009	Voldemort	✓	✓	O	EC, record	O	O	O	O	key-val	lookup
2009	Riak	✓	✓	✓	EC, record	MR	O			key-val	filter
2010	Dremel	✓	O	O	O	/	✓	O	✓	tables	SQL-like
2011	Megastore	✓	✓	✓	entity groups	O	/	O	/	tables	filter
2011	Tenzing	✓	O	O	O	O	✓	✓	✓	tables	SQL-like
2011	Spark/Shark	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2012	Spanner	✓	✓	✓	✓	?	✓	✓	✓	tables	SQL-like
2012	Accumulo	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter
2013	Impala	✓	O	O	O	✓	✓	O	✓	tables	SQL-like

- Scale to 1000s = Scale to that many machines
- Primary index = Can look-up by key using index
- Secondary index = Look-up by non-key attribute
- Transactions = supports (kind of) transactions
- Joins/Analytics = Relative complex computations
- Integrity Constraints = Hard schema
- Views = Multiple views on same data
- Language = Higher-level query language

NOSQL AND RELATED SYSTEMS

Year	System/ Paper	Scale to 1000s	Primary Index	Secondary Index	Transactions	Joins/ Analytics	Integrity Constraints	Views	Language/ Algebra	Data model	My Label
1971	RDBMS	O	✓	✓	✓	✓	✓	✓	✓	tables	SQL-like
2003	memcached	✓	✓	O	O	O	O	O	O	key-val	lookup
2004	MapReduce	✓	O	O	O	✓	O	O	O	key-val	MR
2005	CouchDB	✓	✓	✓	record	MR	O	✓	O	document	filter/MR
2006	BigTable (Hbase)	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter/MR
2007	MongoDB	✓	✓	✓	EC, record	O	O	O	O	document	filter
2007	Dynamo	✓	✓	O	O	O	O	O	O	key-val	lookup
2007	SimpleDB	✓	✓	✓	O	O	O	O	O	ext. record	filter
2008	Pig	✓	O	O	O	✓	/	O	✓	tables	RA-like
2008	HIVE	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2008	Cassandra	✓	✓	✓	EC, record	O	✓	✓	O	key-val	filter
2009	Voldemort	✓	✓	O	EC, record	O	O	O	O	key-val	lookup
2009	Riak	✓	✓	✓	EC, record	MR	O			key-val	filter
2010	Dremel	✓	O	O	O	/	✓	O	✓	tables	SQL-like
2011	Megastore	✓	✓	✓	entity groups	O	/	O	/	tables	filter
2011	Tenzing	✓	O	O	O	O	✓	✓	✓	tables	SQL-like
2011	Spark/Shark	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2012	Spanner	✓	✓	✓	✓	?	✓	✓	✓	tables	SQL-like
2012	Accumulo	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter
2013	Impala	✓	O	O	O	✓	✓	O	✓	tables	SQL-like

- RDBMs have many features, but did not scale well beyond 10s of machines
- The problem was not so much read operations, but large workloads of many small updates

TYPES OF NOSQL DATABASES

Year	System/ Paper	Scale to 1000s	Primary Index	Secondary Index	Transactions	Joins/ Analytics	Integrity Constraints	Views	Language/ Algebra	Data model	My Label
1971	RDBMS	O	✓	✓	✓	✓	✓	✓	✓	tables	SQL-like
2003	memcached	✓	✓	O	O	O	O	O	O	key-val	lookup
2004	MapReduce	✓	O	O	O	✓	O	O	O	key-val	MR
2005	CouchDB	✓	✓	✓	record	MR	O	✓	O	document	filter/MR
2006	BigTable (Hbase)	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter/MR
2007	MongoDB	✓	✓	✓	EC, record	O	O	O	O	document	filter
2007	Dynamo	✓	✓	O	O	O	O	O	O	key-val	lookup
2007	SimpleDB	✓	✓	✓	O	O	O	O	O	ext. record	filter
2008	Pig	✓	O	O	O	✓	/	O	✓	tables	RA-like
2008	HIVE	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2008	Cassandra	✓	✓	✓	EC, record	O	✓	✓	O	key-val	filter
2009	Voldemort	✓	✓	O	EC, record	O	O	O	O	key-val	lookup
2009	Riak	✓	✓	✓	EC, record	MR	O			key-val	filter
2010	Dremel	✓	O	O	O	/	✓	O	✓	tables	SQL-like
2011	Megastore	✓	✓	✓	entity groups	O	/	O	/	tables	filter
2011	Tenzing	✓	O	O	O	O	✓	✓	✓	tables	SQL-like
2011	Spark/Shark	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2012	Spanner	✓	✓	✓	✓	?	✓	✓	✓	tables	SQL-like
2012	Accumulo	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter
2013	Impala	✓	O	O	O	✓	✓	O	✓	tables	SQL-like

Clustering from:

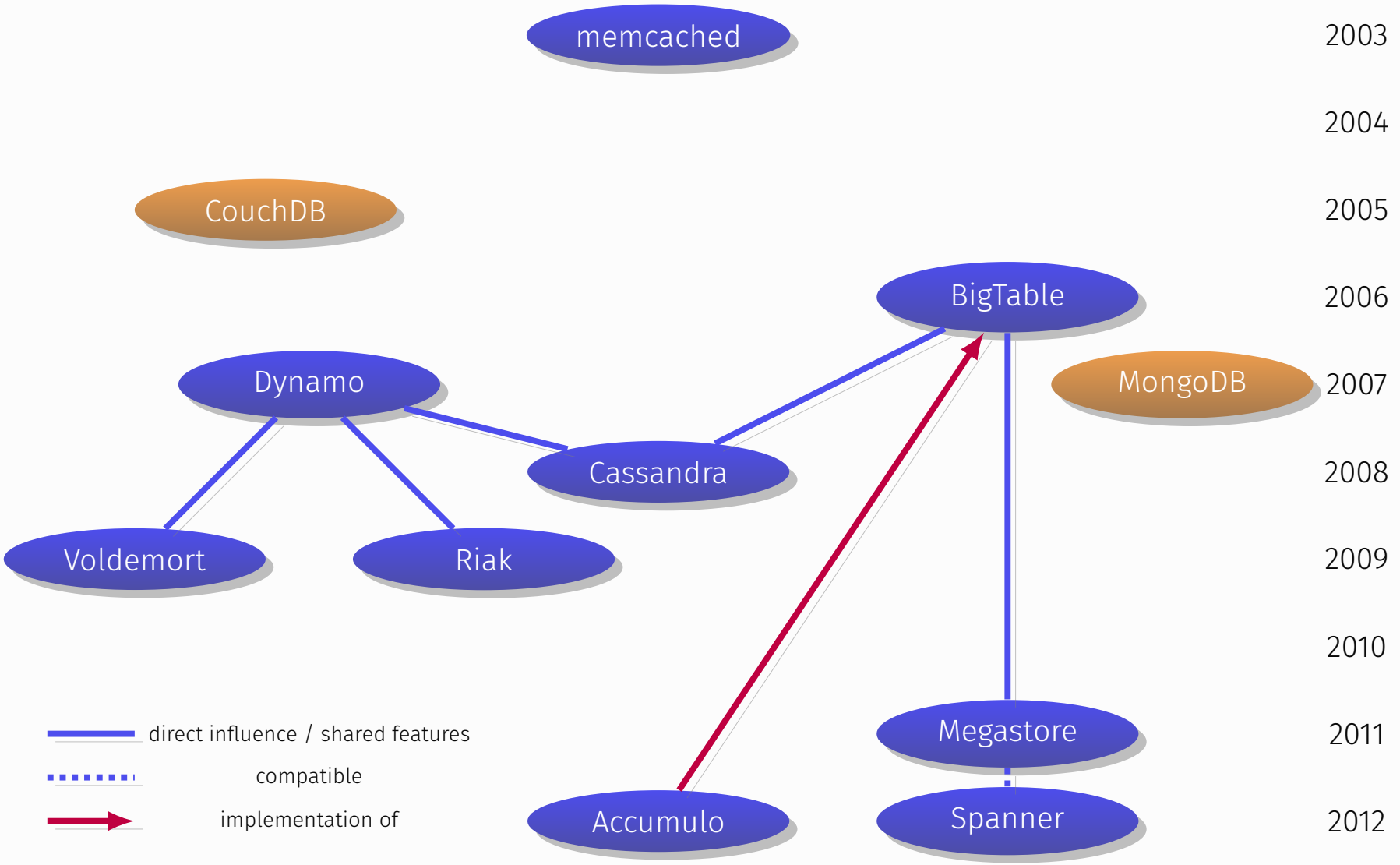
Rick Cattell. 2011. *Scalable SQL and NoSQL data stores.* SIGMOD Rec. 39, 4 (May 2011), 12-27. DOI: https://doi.org/10.1145/1978915.1978919

column families

document stores

key-value stores

RELATIONSHIPS BETWEEN NOSQL DATABASES

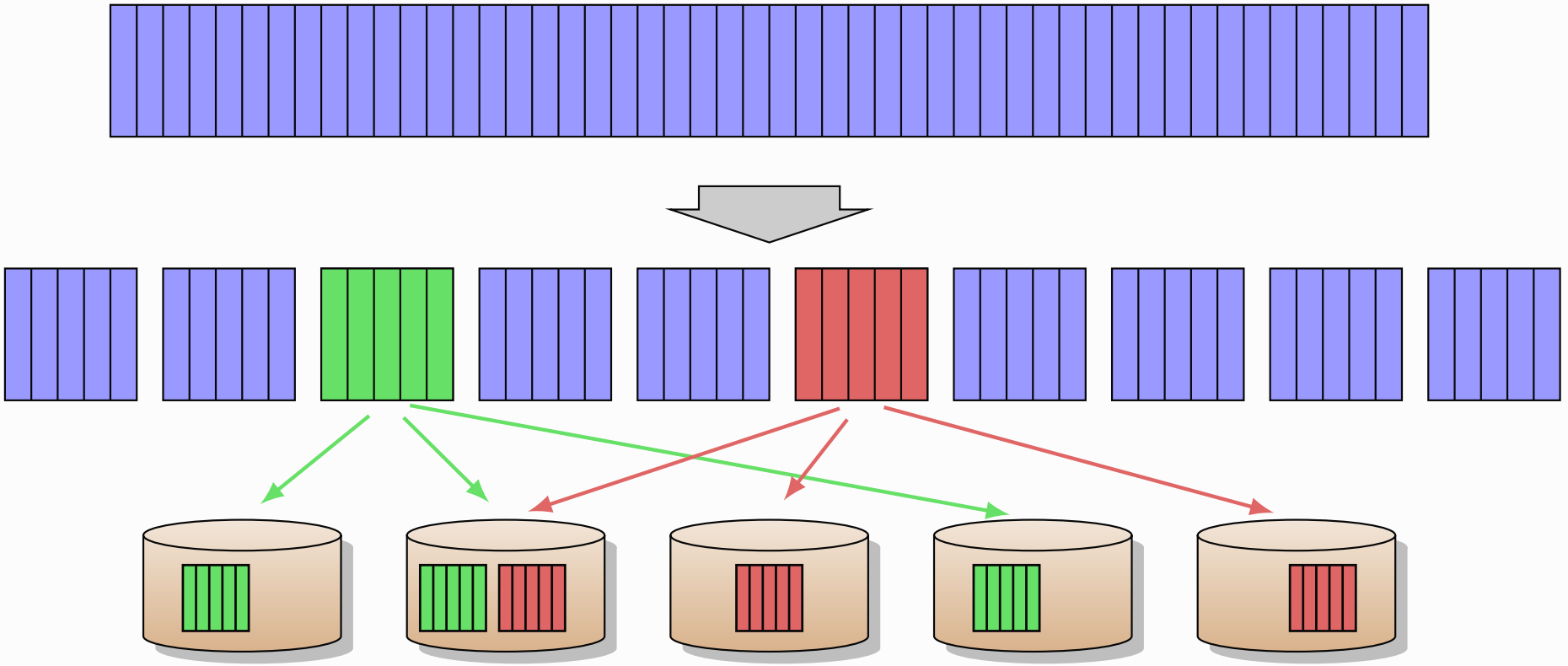


THE DRIVERS BEHIND NOSQL TECHNOLOGY

Year	Source	System/ Paper	Scale to 1000s	Primary Index	Secondary Index	Transactions	Joins/ Analytics	Integrity Constraints	Views	Language/ Algebra	Data model	My Label
1971	many	RDBMS	O	✓	✓	✓	✓	✓	✓	✓	tables	SQL-like
2003	other	memcached	✓	✓	O	O	O	O	O	O	key-val	lookup
2004	Google	MapReduce	✓	O	O	O	✓	O	O	O	key-val	MR
2005	Couchbase	CouchDB	✓	✓	✓	record	MR	O	✓	O	document	filter/MR
2006	Google	BigTable (Hbase)	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter/MR
2007	10gen	MongoDB	✓	✓	✓	EC, record	O	O	O	O	document	filter
2007	Amazon	Dynamo	✓	✓	O	O	O	O	O	O	key-val	lookup
2007	Amazon	SimpleDB	✓	✓	✓	O	O	O	O	O	ext. record	filter
2008	Amazon	Pig	✓	O	O	O	✓	/	O	✓	tables	RA-like
2008	Facebook	HIVE	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2008	Facebook	Cassandra	✓	✓	✓	EC, record	O	✓	✓	O	key-val	filter
2009	other	Voldemort	✓	✓	O	EC, record	O	O	O	O	key-val	lookup
2009	basho	Riak	✓	✓	✓	EC, record	MR	O			key-val	filter
2010	Google	Dremel	✓	O	O	O	/	✓	O	✓	tables	SQL-like
2011	Google	Megastore	✓	✓	✓	entity groups	O	/	O	/	tables	filter
2011	Google	Tenzing	✓	O	O	O	O	✓	✓	✓	tables	SQL-like
2011	Berkeley	Spark/Shark	✓	O	O	O	✓	✓	O	✓	tables	SQL-like
2012	Google	Spanner	✓	✓	✓	✓	?	✓	✓	✓	tables	SQL-like
2012	Accumulo	Accumulo	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter
2013	Cloudera	Impala	✓	O	O	O	✓	✓	O	✓	tables	SQL-like

CONSISTENCY - AVAILABILITY - PARTITION

DISTRIBUTION AND REPLICATION



- We want high availability $\Rightarrow$ Replication
- We want consistency $\Rightarrow$ Update propagation

EXAMPLE

User: Sue
Friends: Joe, Kai, ...
Status: "Headed to new Bond flick"
Wall: "...", "...."

User: Joe
Friends: Sue, ...
Status: "I'm sleepy"
Wall: "...", "...."

User: Kai
Friends: Sue, ...
Status: "Done for tonight"
Wall: "...", "...."

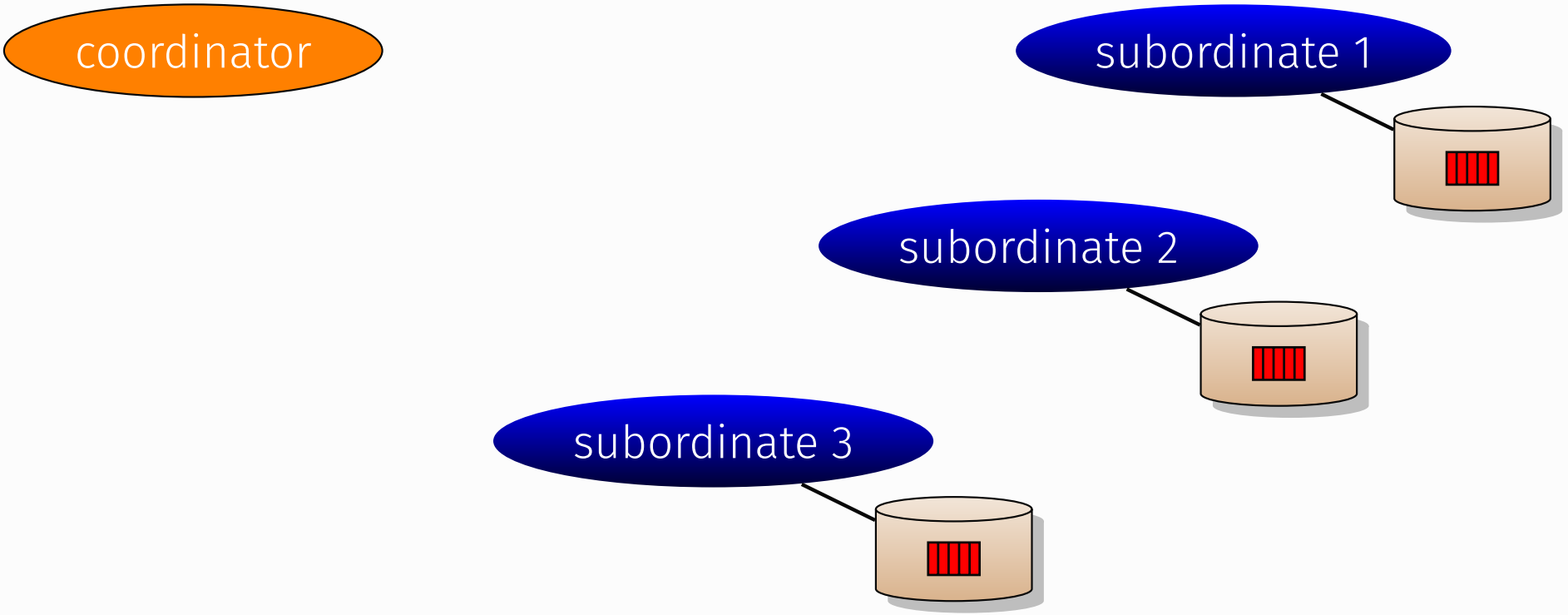
- **write operation:** Update Sue's status. Who sees the new status, who sees the old one?
  - **SQL Database:** *"Everyone must see the same thing, either old or new, no matter how long it takes."*
  - **NoSQL Database:** *"For large applications, we cannot afford to wait that long, and it may not matter that much anyway."*

ACID TRANSACTIONS

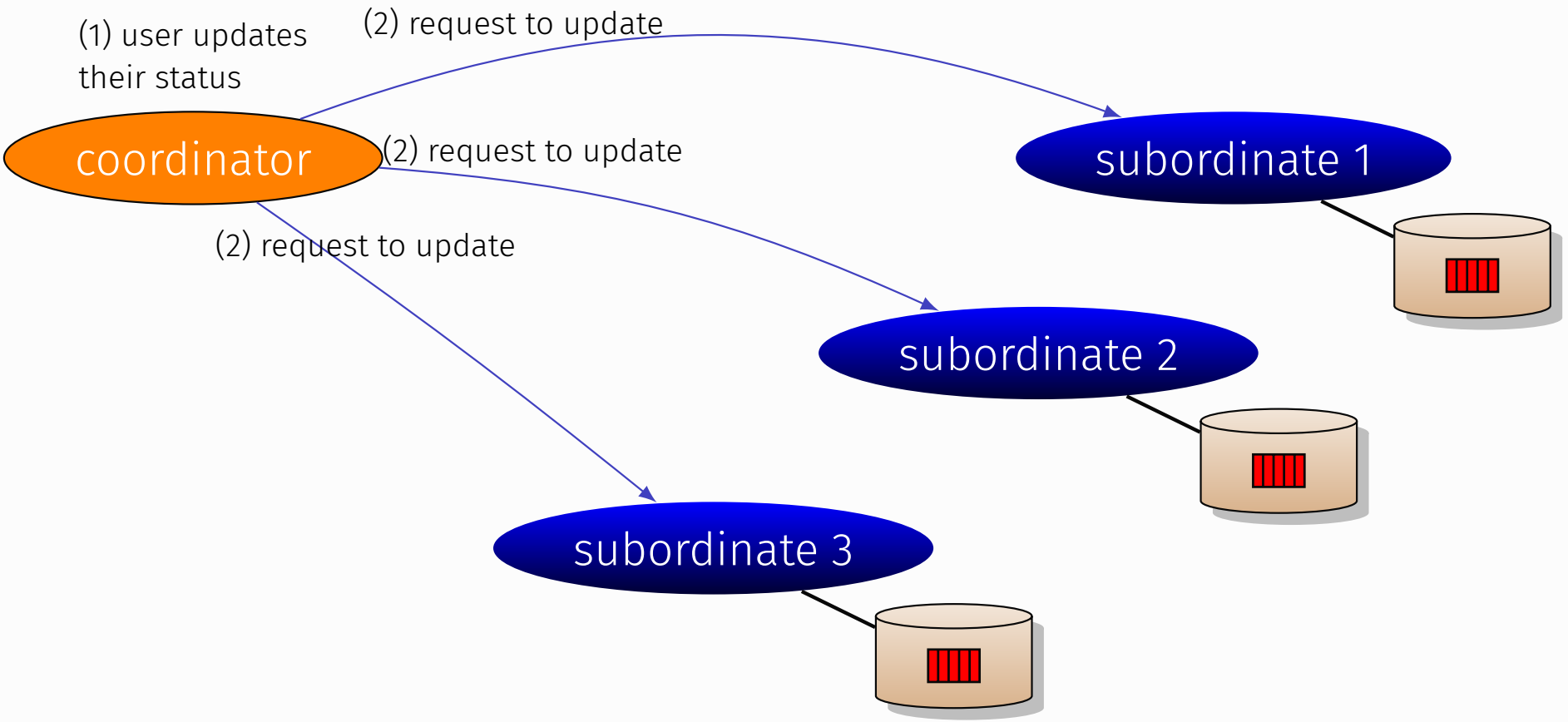
- The golden standard for transaction management in Relational DBMSs.
- Recall what ACID stands for:
  - **Atomicity:** Either all effects of the transaction are recorded in the database, or none are.
  - **Consistency:** After the transaction, the database must satisfy all validity rules.
  - **Isolation:** A transaction must have the impression it is the only one running, in terms of values it has read and the resulting state of the database.
    - This is also referred to as **serializability**, i.e., the answers to queries and the final state of the database must be as if the transactions were serially executed.
  - **Durability:** Once a transaction has been committed, it stays committed.
- How to achieve this for large distributed datasets?

ACID IN DISTRIBUTED DATASETS

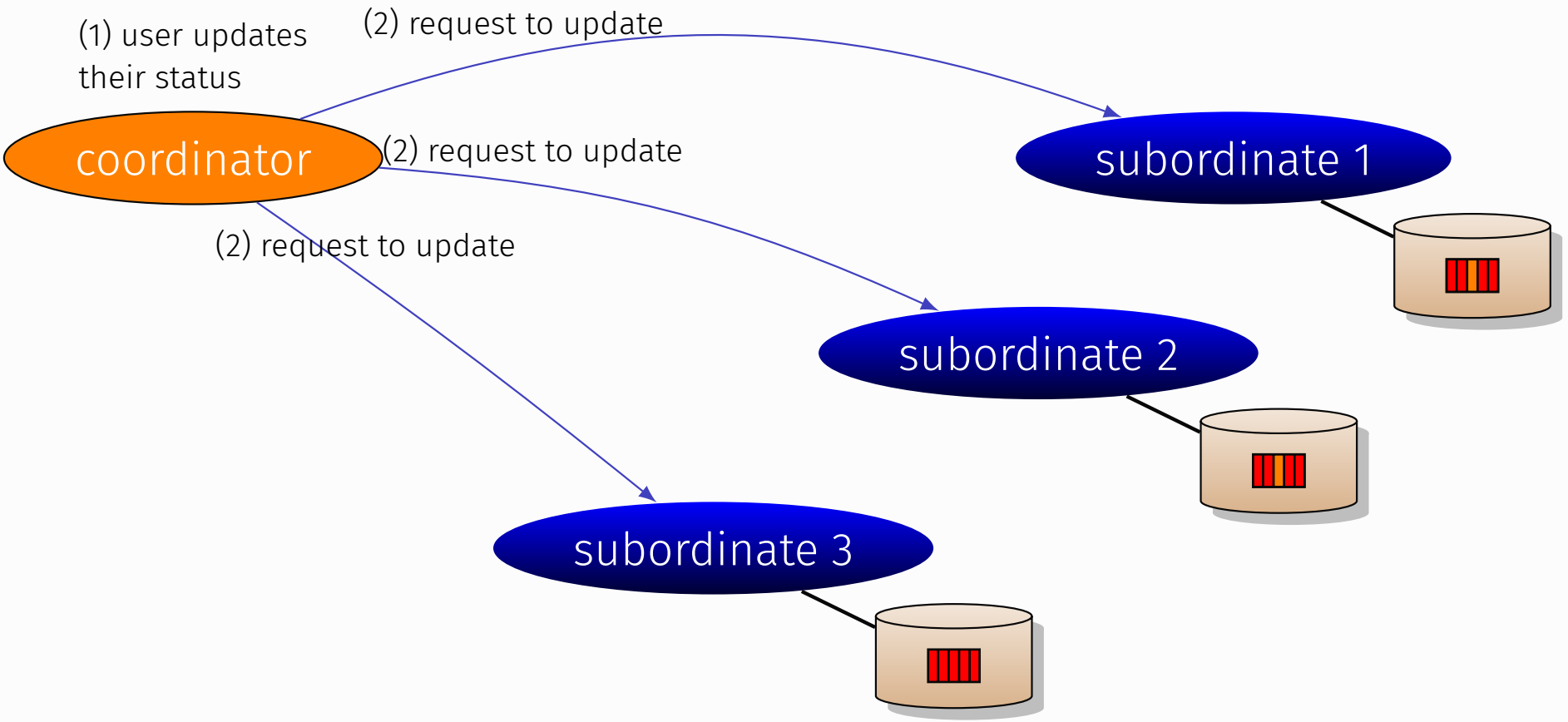
(1) user updates
their status



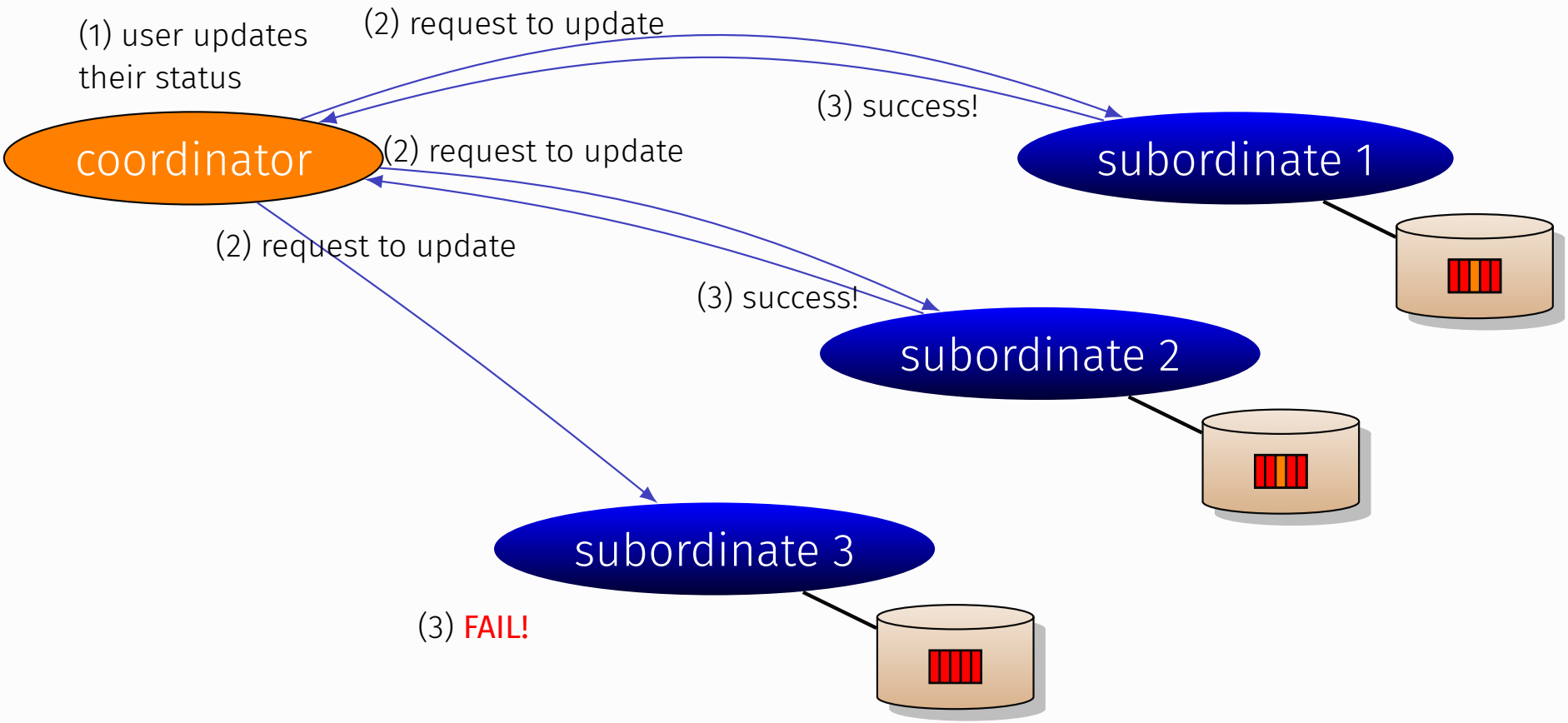
ACID IN DISTRIBUTED DATASETS



ACID IN DISTRIBUTED DATASETS



ACID IN DISTRIBUTED DATASETS



- Ends in an inconsistent state.

CONSISTENCY IN THE WEB ERA

- ACID properties are always desirable
- But, web applications have different needs from applications that RDBMS were designed for
  - Low and predictable response time (latency)
  - Scalability & elasticity (at low cost!)
  - High availability
  - Flexible schemas and semi-structured data
    Geographic distribution (multiple data centers)
- Web applications can (usually) do without
  - Transactions, strong consistency, integrity
  - Complex queries

CAP THEOREM

- Consistency
  - The system is in a consistent state after an operation
  - All clients see the same data
  - Strong consistency (ACID) vs. eventual consistency (BASE)
- Availability
  - The system is “always on”, no downtime
  - Node failure tolerance – all clients can find some available replica
  - Software/hardware upgrade tolerance
- Partition tolerance
  - The system continues to function even when split into disconnected subsets, e.g., due to network errors or addition/removal of nodes
  - Not only for reads, but writes as well!

- CAP Theorem (E. Brewer, N. Lynch)
  - In a “shared-data system”, at most 2 out of the 3 properties can be achieved at any given moment in time.

CAP THEOREM

- CA
  - Single site clusters (easier to ensure all nodes are always in contact) e.g., 2PC
  - When a partition occurs, the system blocks
- CP
  - Some data may be inaccessible (availability sacrificed), but the rest is still consistent/accurate
  - E.g., sharded database
- AP
  - System is still available under partitioning, but some of the data returned may be inaccurate
  - I.e., availability and partition tolerance are more important than strict consistency
  - E.g., DNS, caches, Master/Slave replication
  - Need some conflict resolution strategy

CAP THEOREM

- Conventional databases assume no partitioning – clusters were assumed to be small and local
- NoSQL systems may sacrifice consistency

Original papers

- Eric A. Brewer. 2000. *Towards robust distributed systems (abstract).* In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (PODC '00). ACM, New York, NY, USA, 7-. DOI=http://dx.doi.org/10.1145/343477.343502
- Seth Gilbert and Nancy Lynch. 2002. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.* SIGACT News 33, 2 (June 2002), 51-59. DOI: https://doi.org/10.1145/564585.564601

EVENTUAL CONSISTENCY

- The term originates in the context of distributed operation systems.
- It was there not seen as a compromise, but actually the best way of dealing with update conflicts.

Original reference where term was coined

B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. *Managing update conflicts in Bayou, a weakly connected replicated storage system.* SIGOPS Oper. Syst. Rev. 29, 5 (December 1995), 172-182. DOI: http://dx.doi.org/10.1145/224057.224070

> *"We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications."*
>
> *"Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application."*

EVENTUAL CONSISTENCY

The meaning:

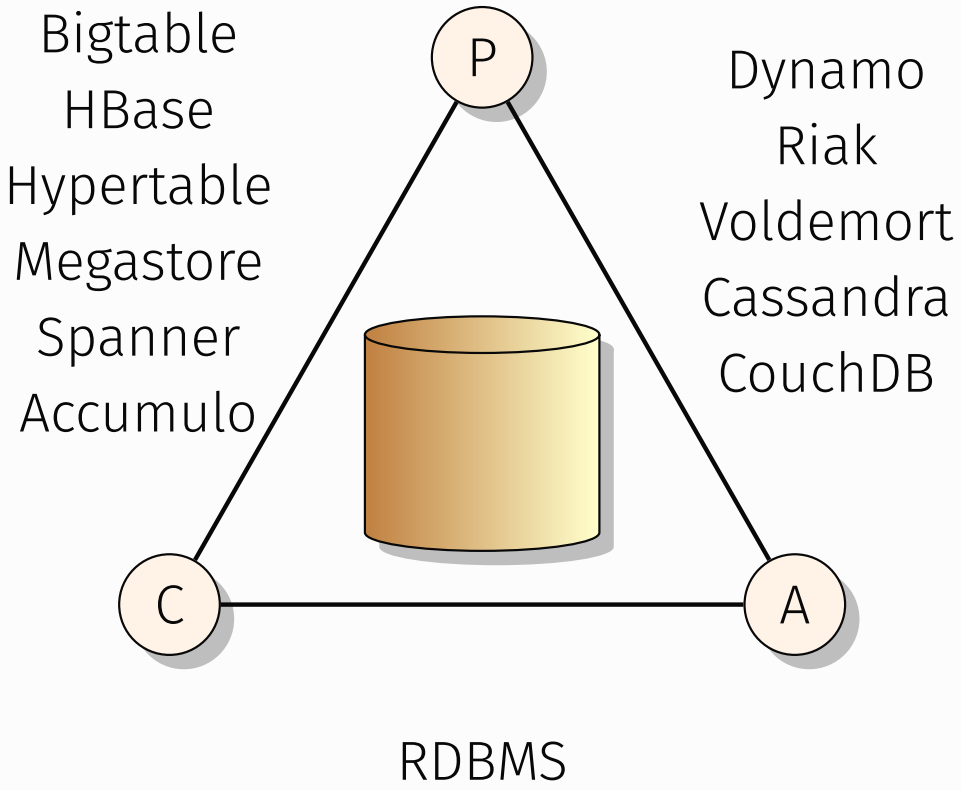
- In absence of updates, all replicas converge towards identical copies.
- What the application sees in the meantime is sensitive to replication mechanics and difficult to predict.
- Contrast with RDBMS: Immediate (or “strong”) consistency, but there may be deadlocks.

TRANSACTION SUPPORT IN NOSQL DATASTORES

Year	System/ Paper	Scale to 1000s	Primary Index	Secondary Index	Transactions	Joins/ Analytics	Integrity Constraints	Views	Language/ Algebra	Data model	My Label
2003	memcached	✓	✓	O	O	O	O	O	O	key-val	lookup
2005	CouchDB	✓	✓	✓	record	MR	O	✓	O	document	filter/MR
2006	BigTable (Hbase)	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter/MR
2007	MongoDB	✓	✓	✓	EC, record	O	O	O	O	document	filter
2007	Dynamo	✓	✓	O	O	O	O	O	O	key-val	lookup
2008	Cassandra	✓	✓	✓	EC, record	O	✓	✓	O	key-val	filter
2009	Voldemort	✓	✓	O	EC, record	O	O	O	O	key-val	lookup
2009	Riak	✓	✓	✓	EC, record	MR	O			key-val	filter
2011	Megastore	✓	✓	✓	entity groups	O	/	O	/	tables	filter
2012	Spanner	✓	✓	✓	✓	?	✓	✓	✓	tables	SQL-like
2012	Accumulo	✓	✓	✓	record	compat. w/MR	/	O	O	ext. record	filter

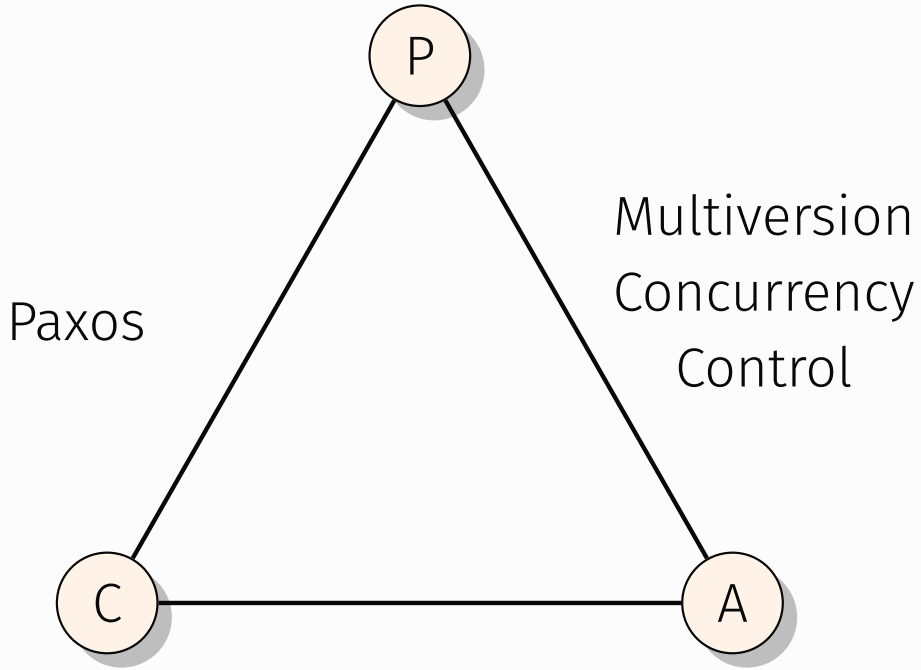
- “record” = strong consistency within records
- “EC” = eventual consistency
- “entity groups” = consistent within predefined sets of related records

THE CAP TRIANGLE



CONCURRENCY CONTROL PROTOCOLS

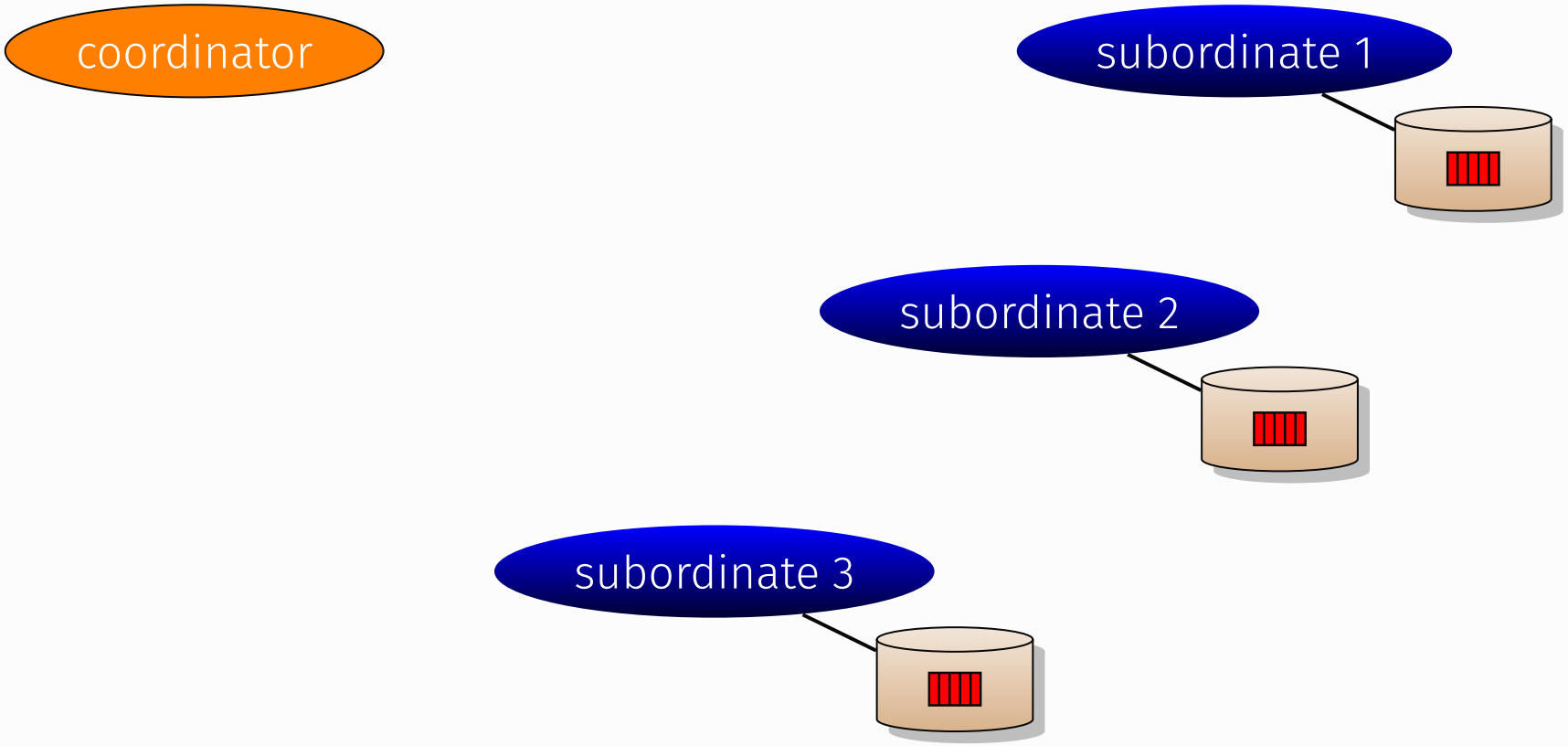
CONCURRENCY CONTROL PROTOCOLS



THE TWO-PHASE COMMIT PROTOCOL

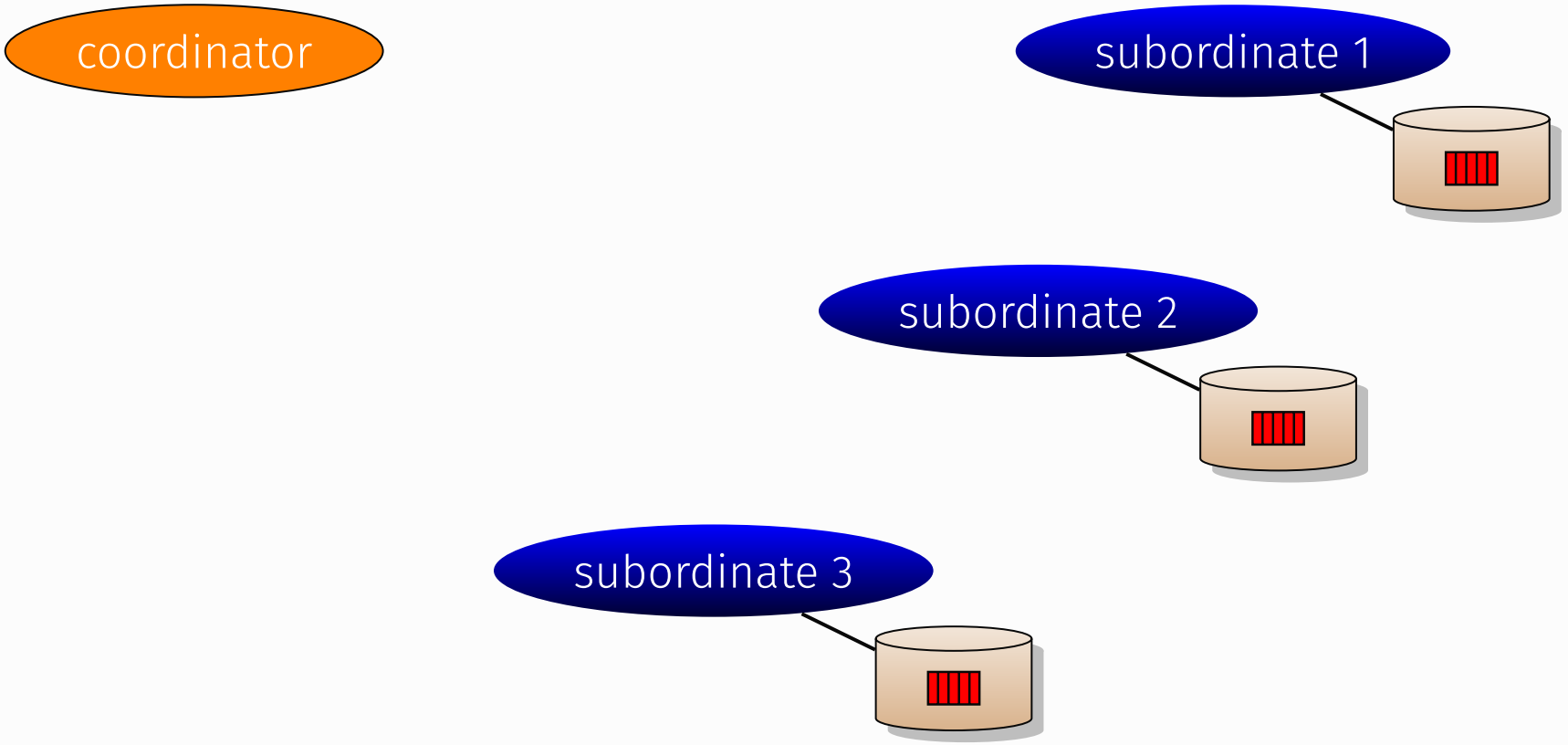
- Phase 1:
  - Coordinator sends Prepare to Commit
  - Subordinates make sure they can do so, no matter what
    - and write the decision to the local log to tolerate failure after this point
  - Subordinates reply Ready to Commit or Not able to Commit
- Phase 2:
  - If all subordinates have sent Ready to Commit,
    - then the coordinator sends Commit to all
    - else it sends Abort to all
  - In both cases the coordinator writes the decision to the local log.
  - If a subordinate receives Commit or Abort, then it
    - writes the received message to the local log and
    - does the update, if it received Commit, and releases the resources

2PC EXAMPLE ONE

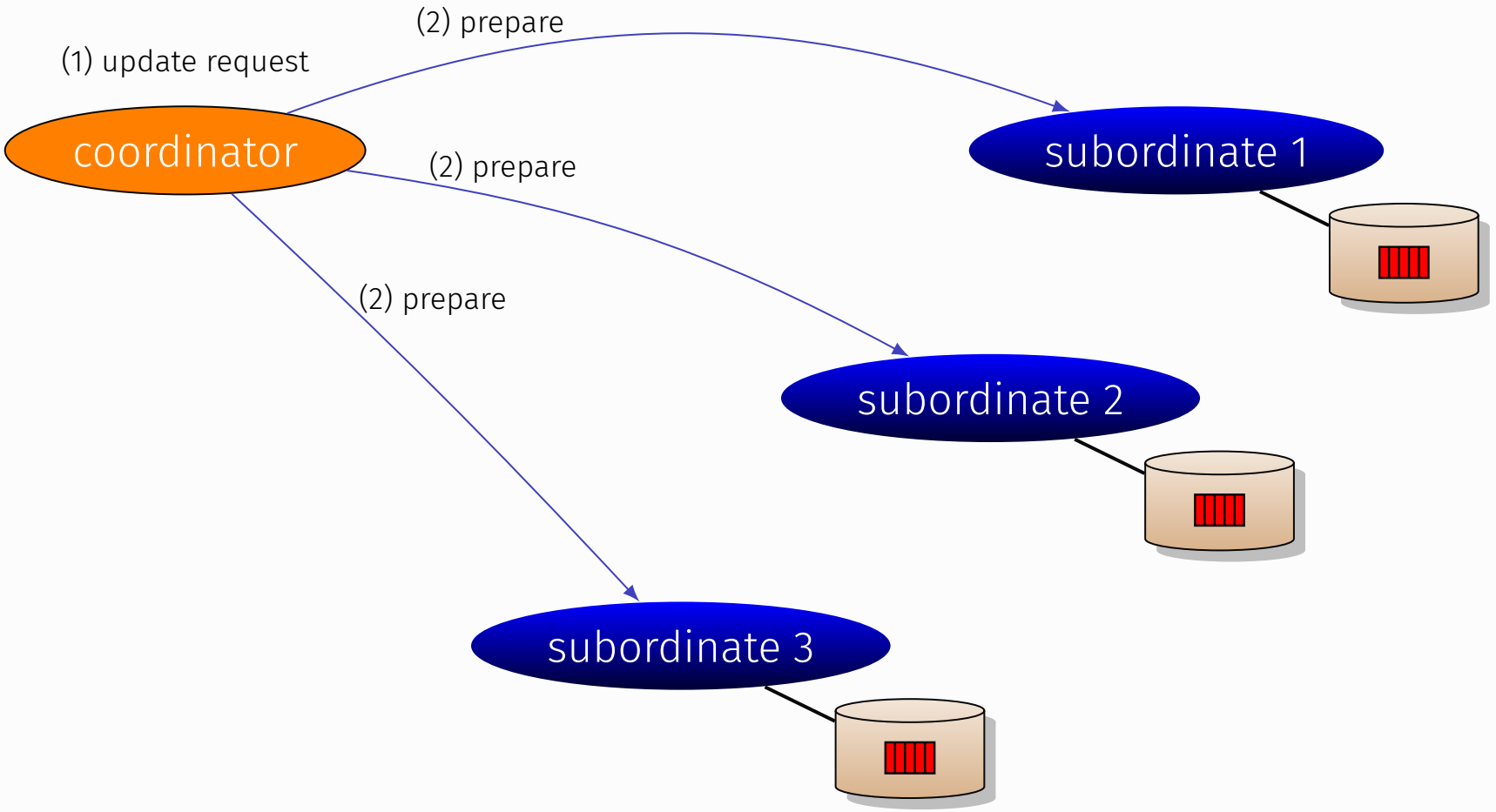


2PC EXAMPLE ONE

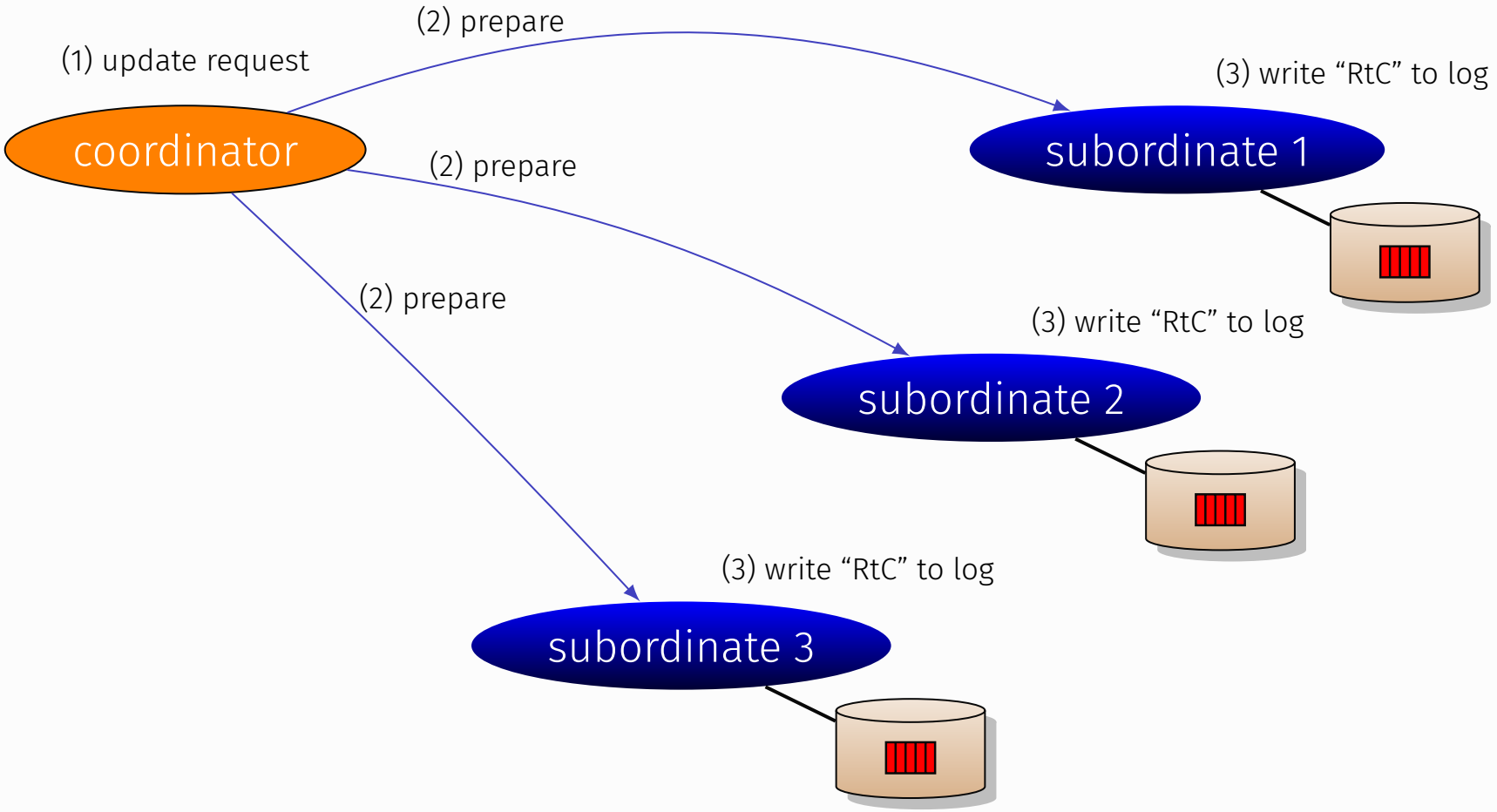
(1) update request



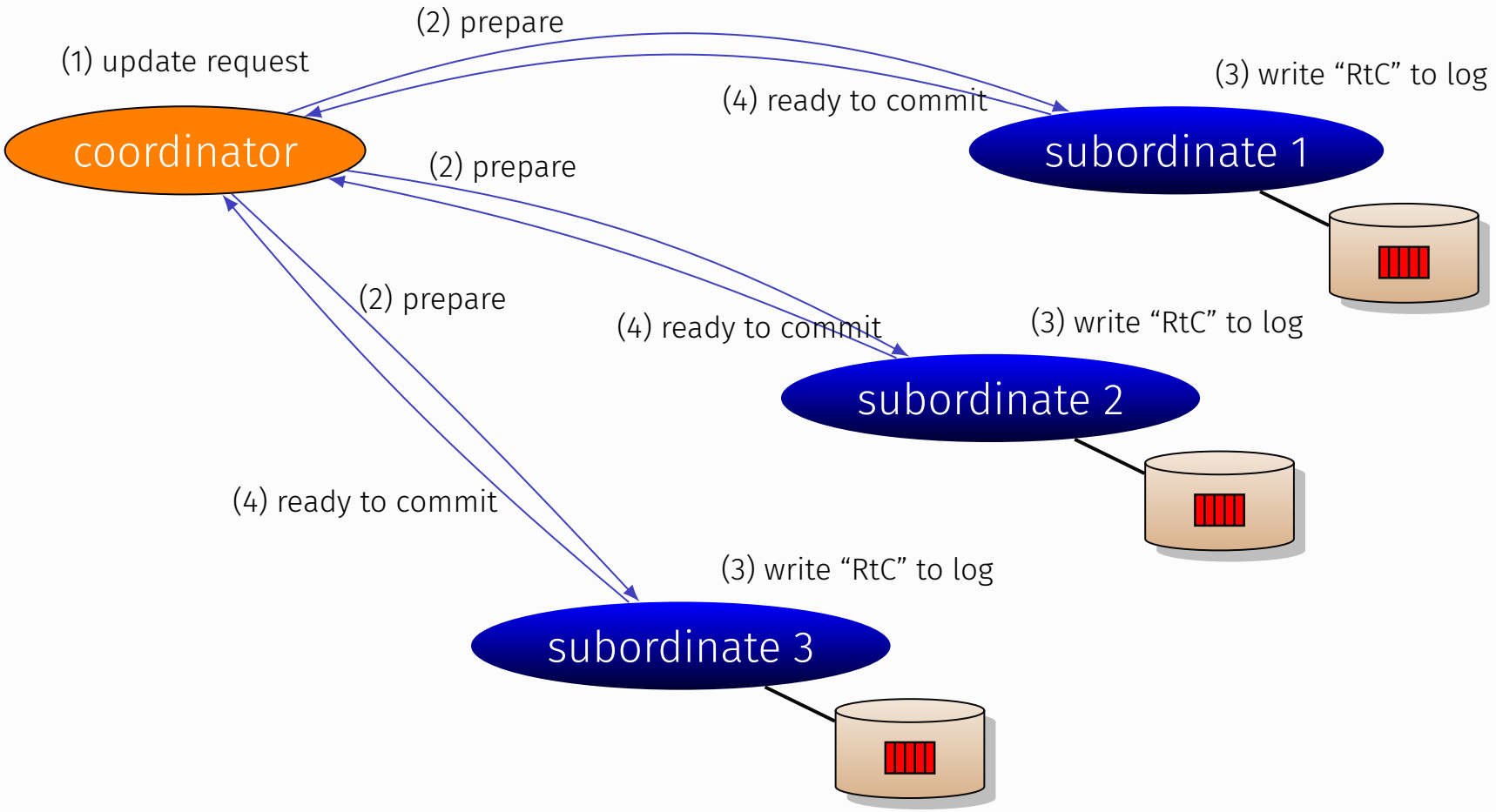
2PC EXAMPLE ONE



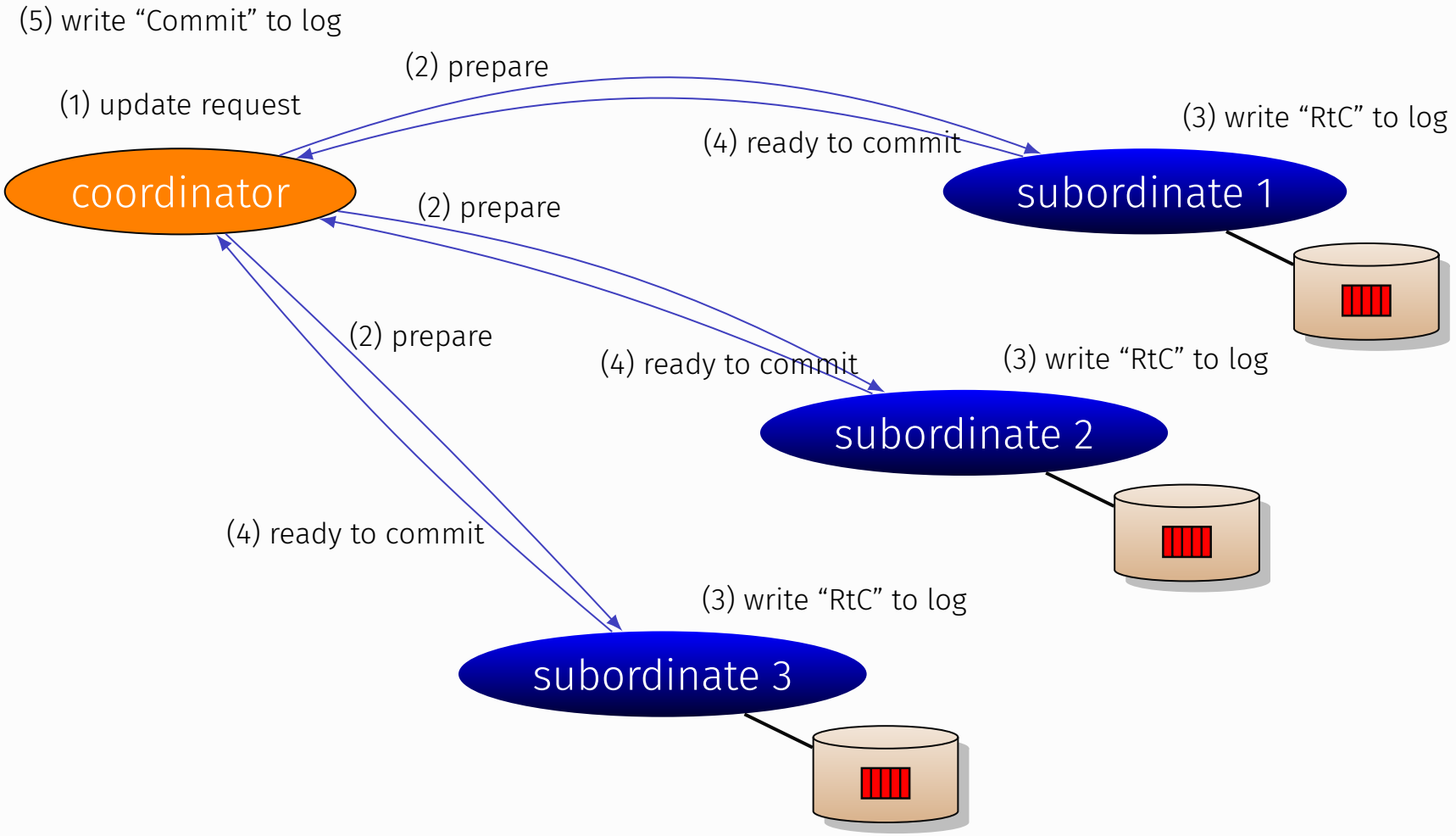
2PC EXAMPLE ONE



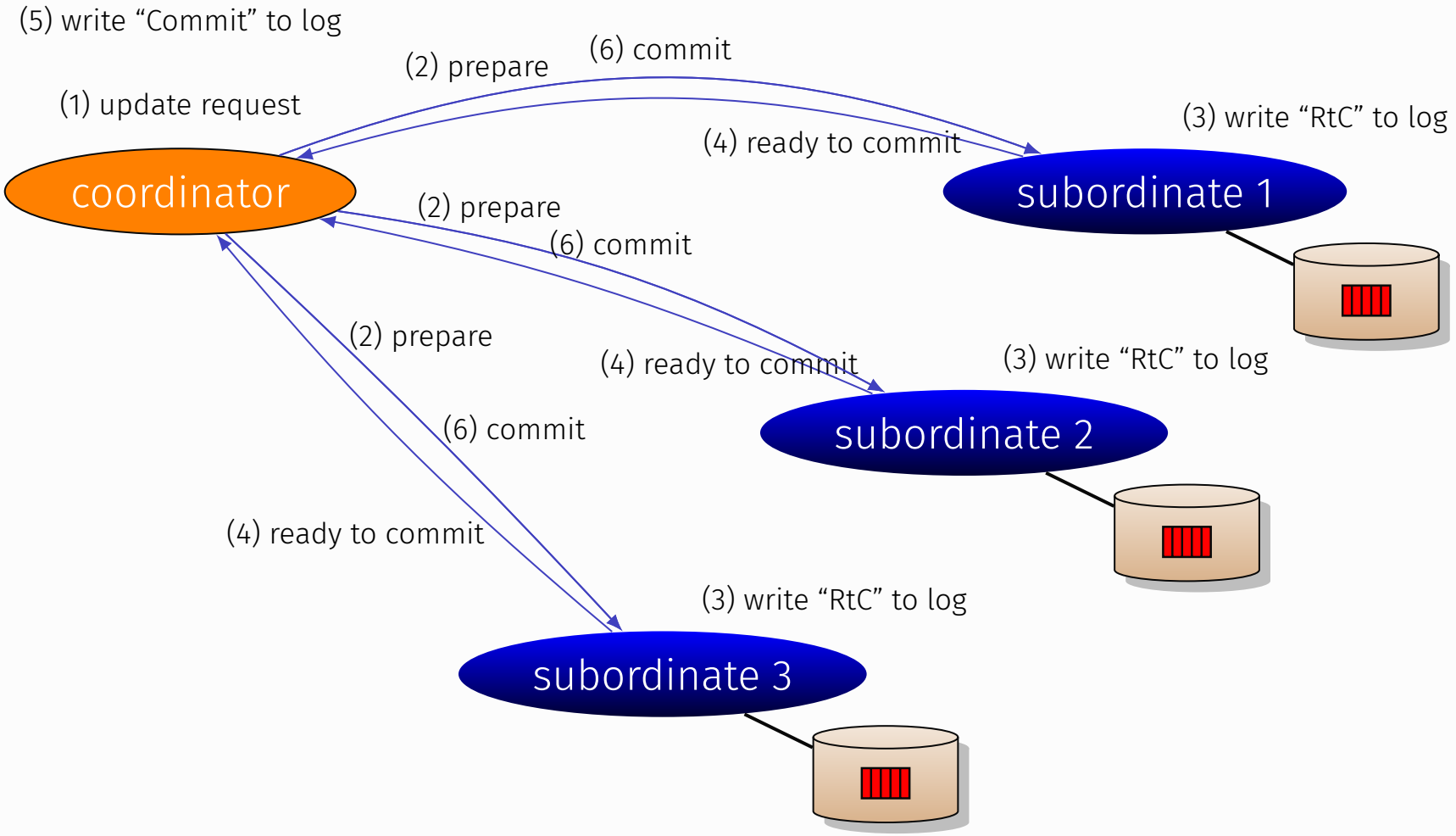
2PC EXAMPLE ONE



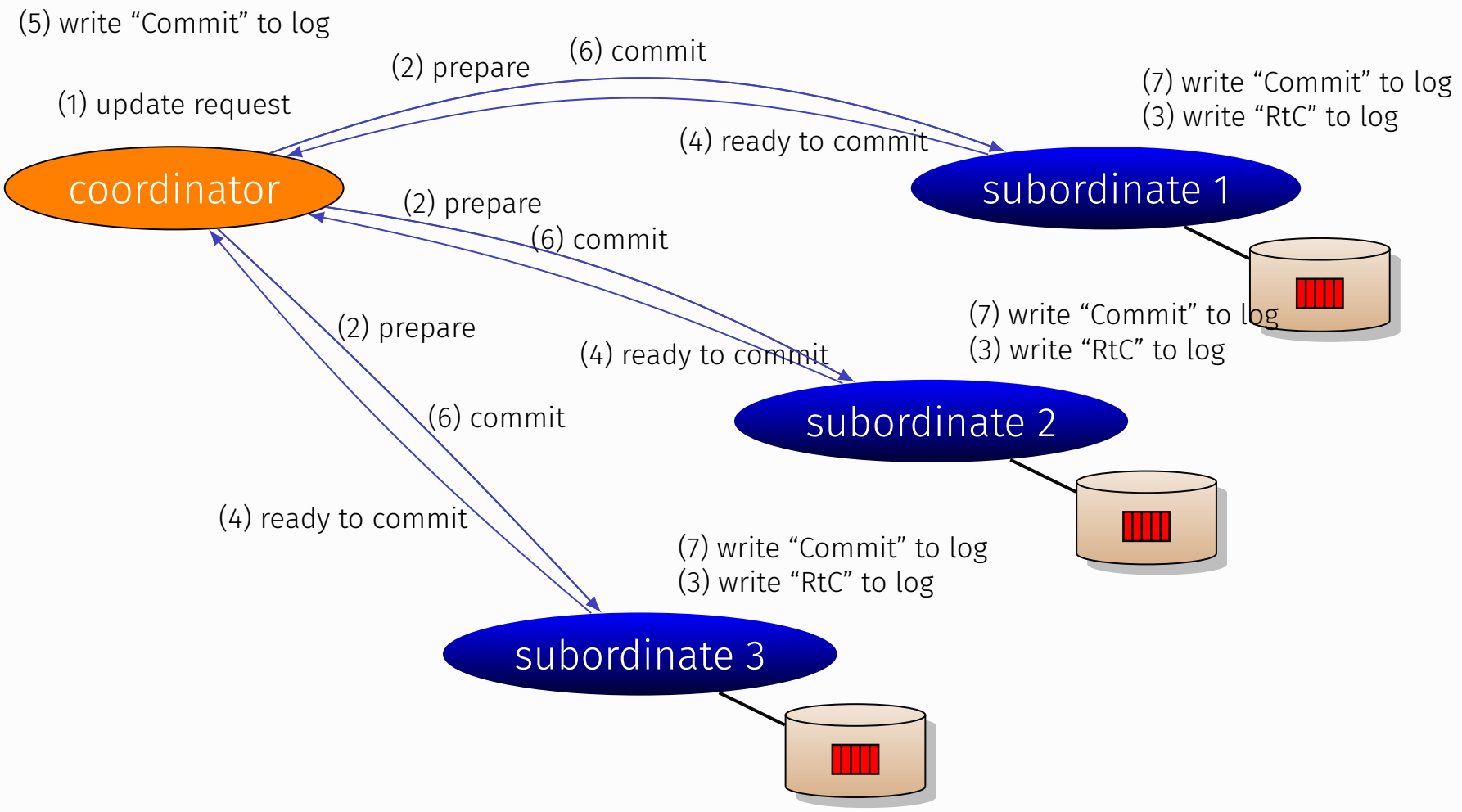
2PC EXAMPLE ONE



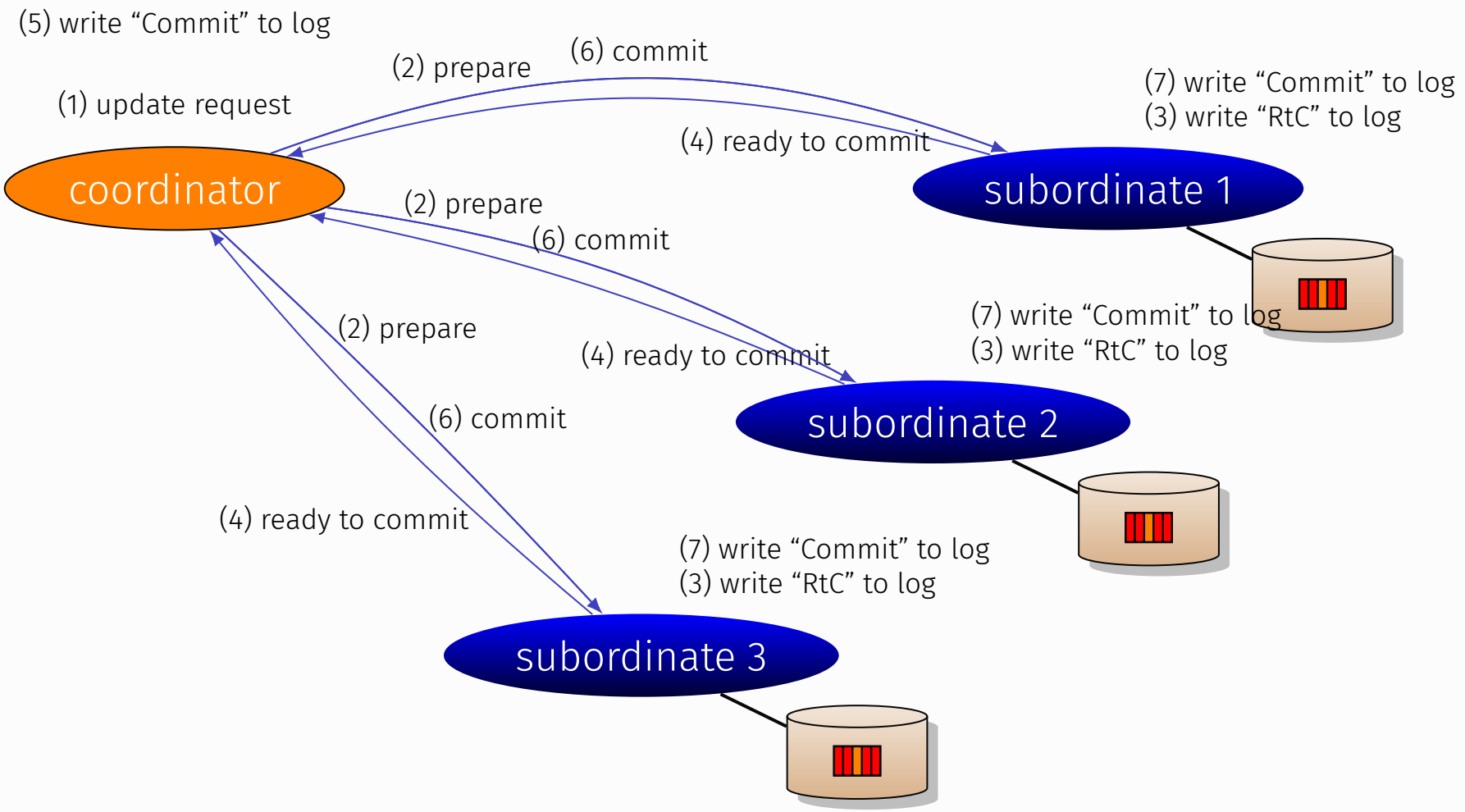
2PC EXAMPLE ONE



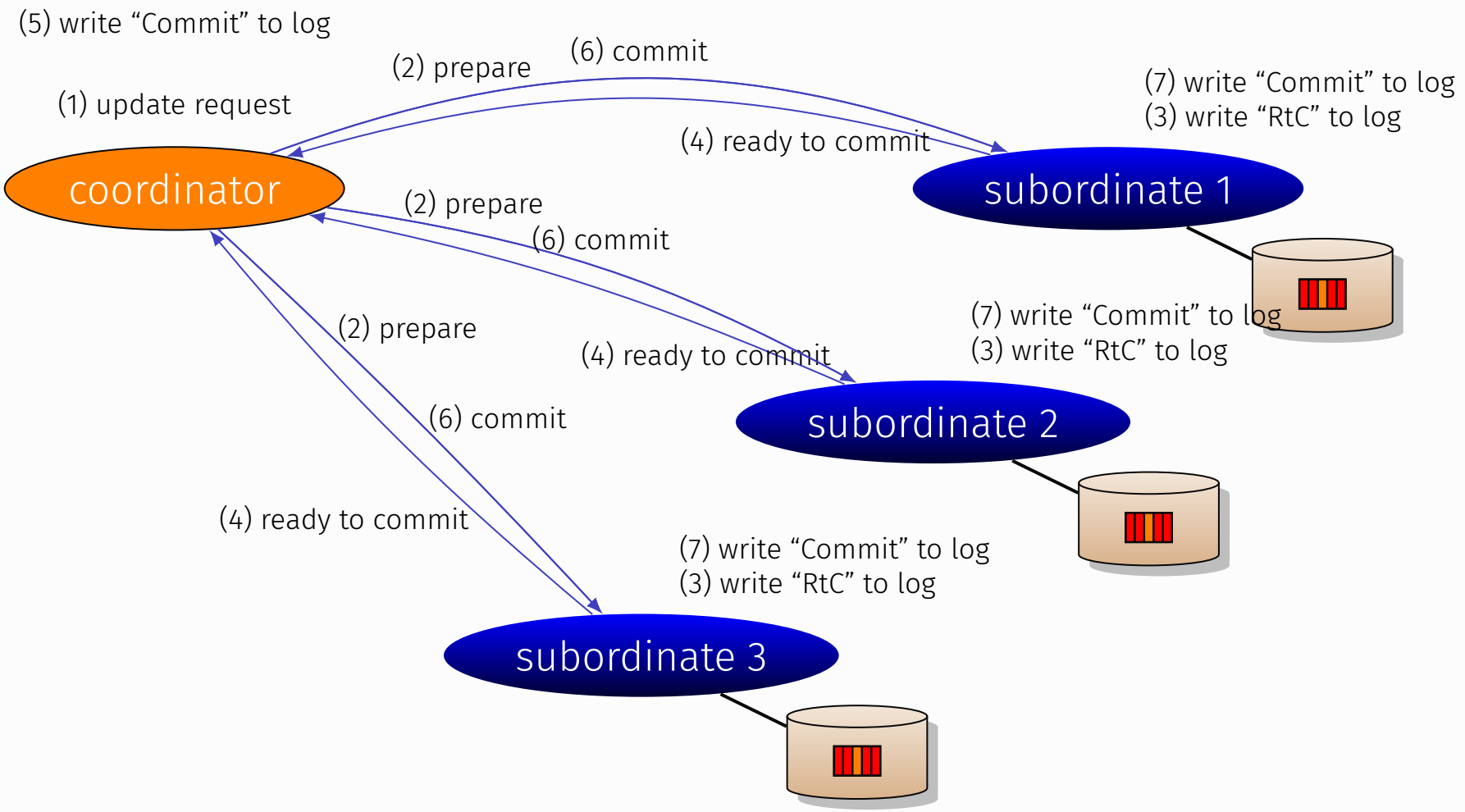
2PC EXAMPLE ONE



2PC EXAMPLE ONE

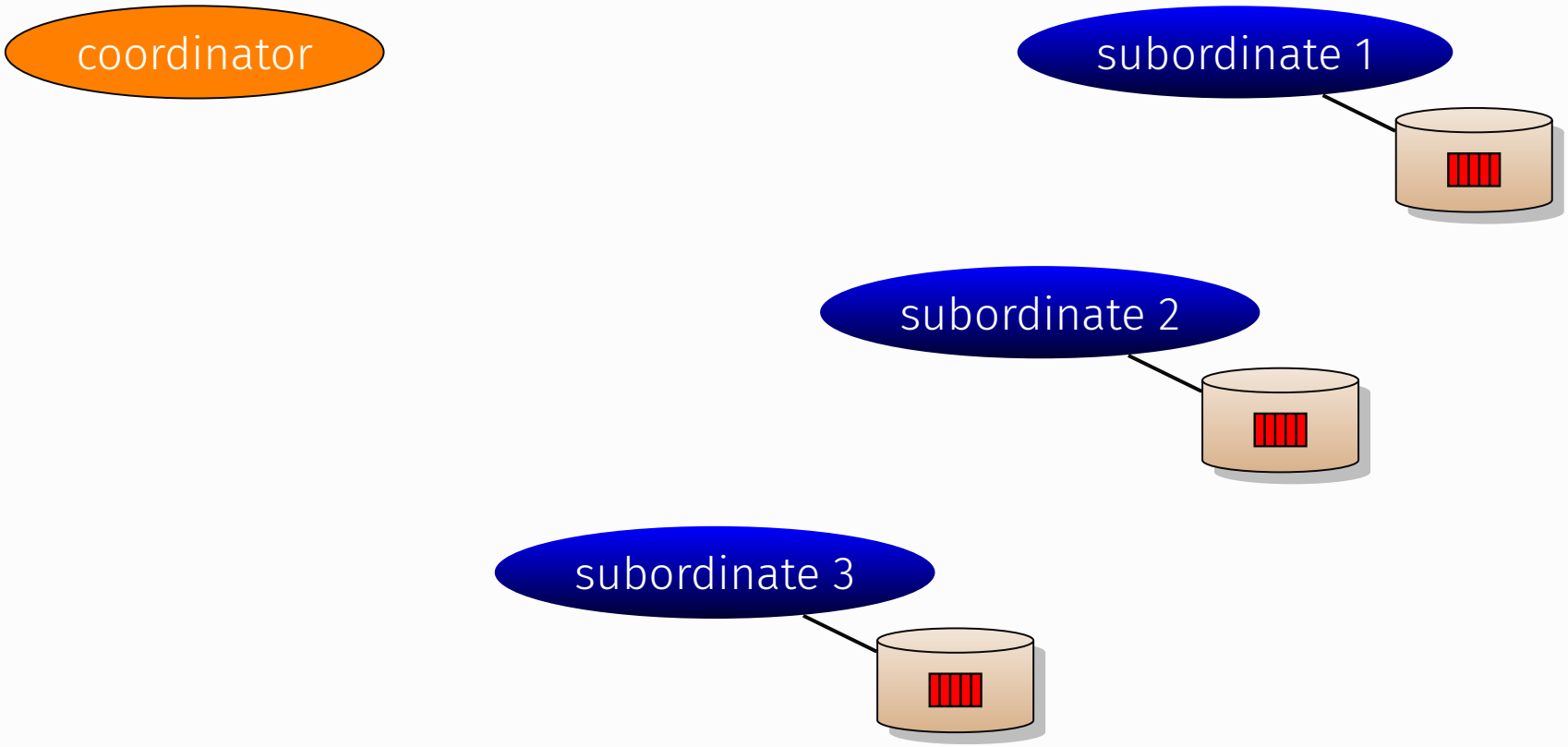


2PC EXAMPLE ONE



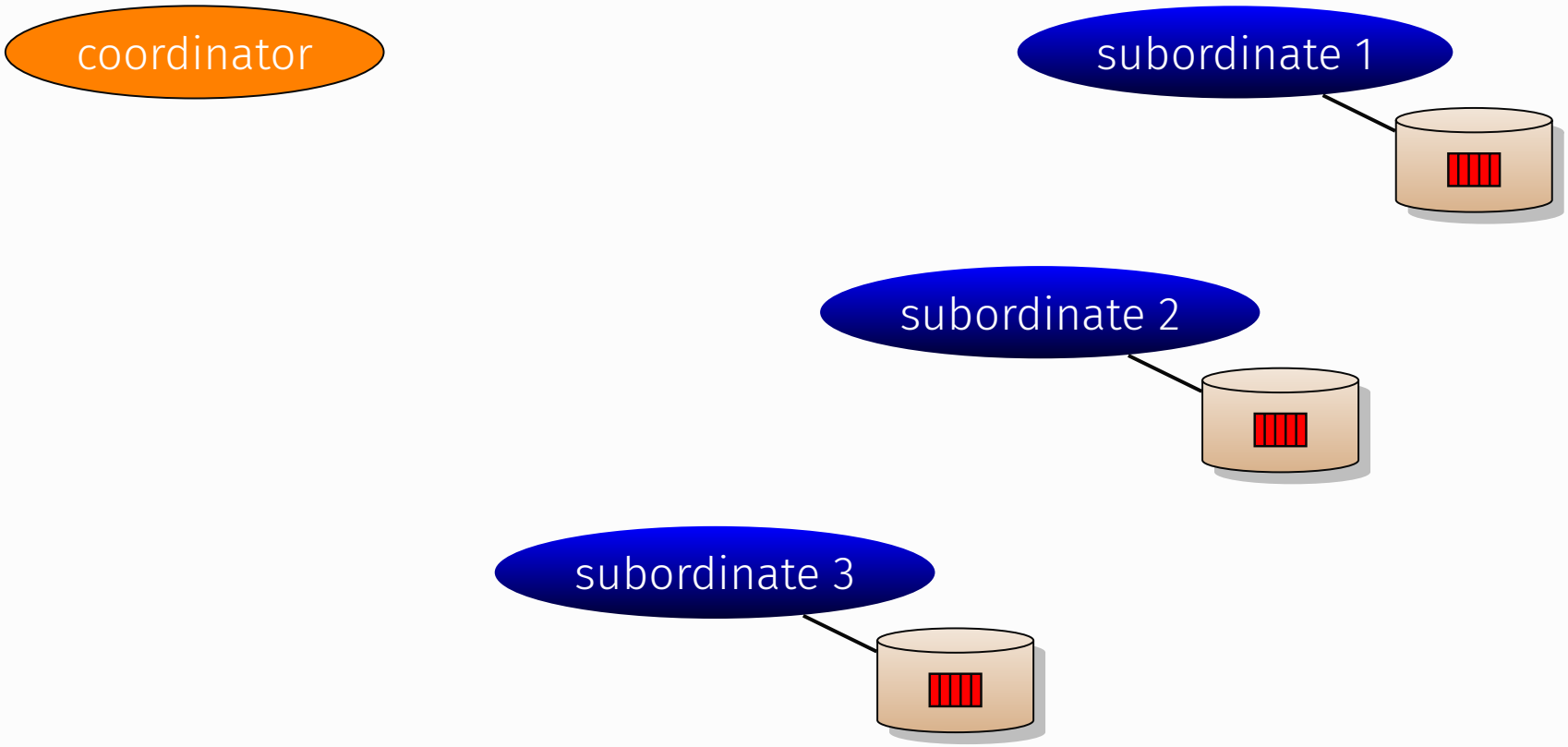
- Ends in a consistent state.

2PC EXAMPLE TWO

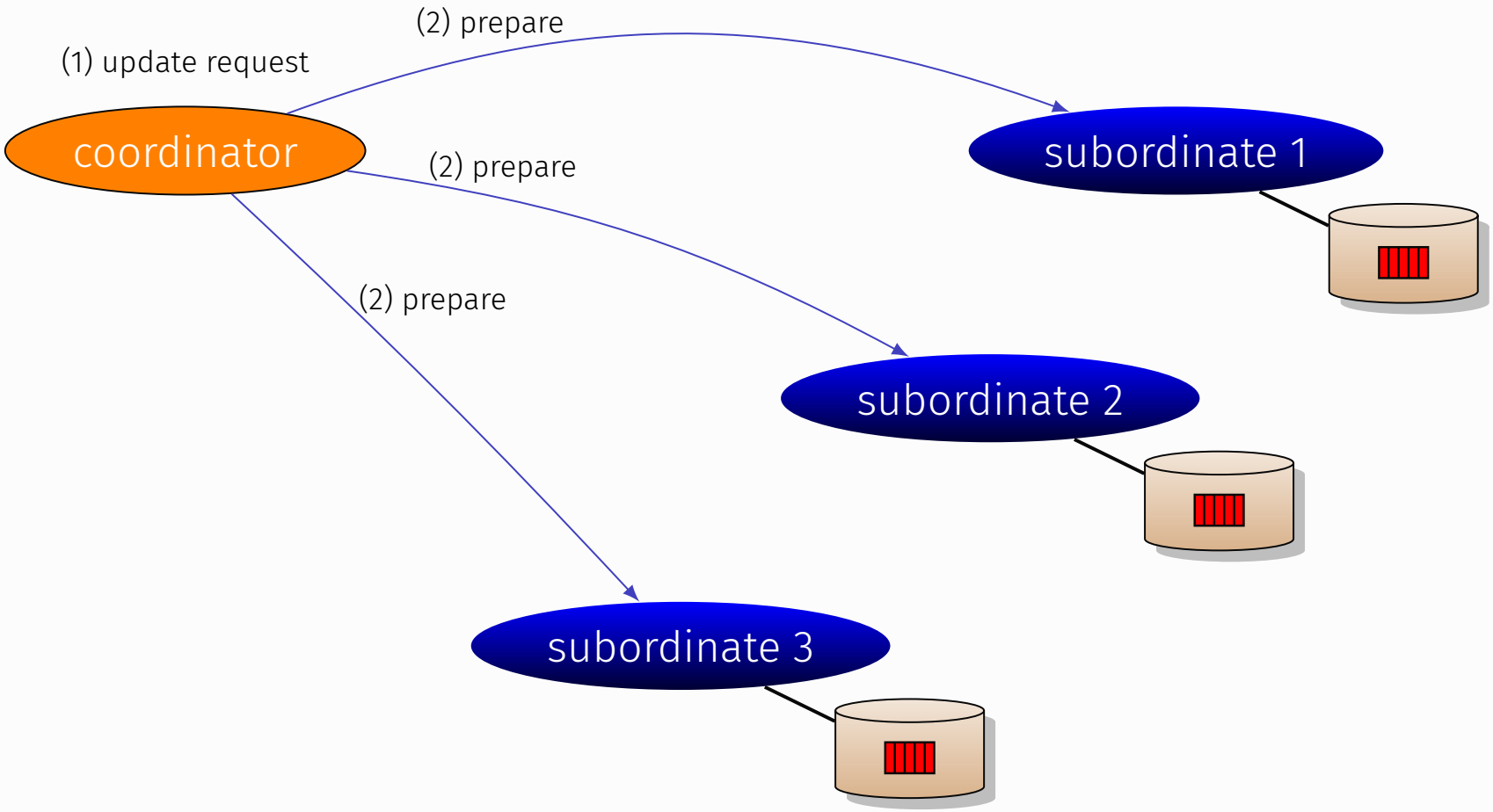


2PC EXAMPLE TWO

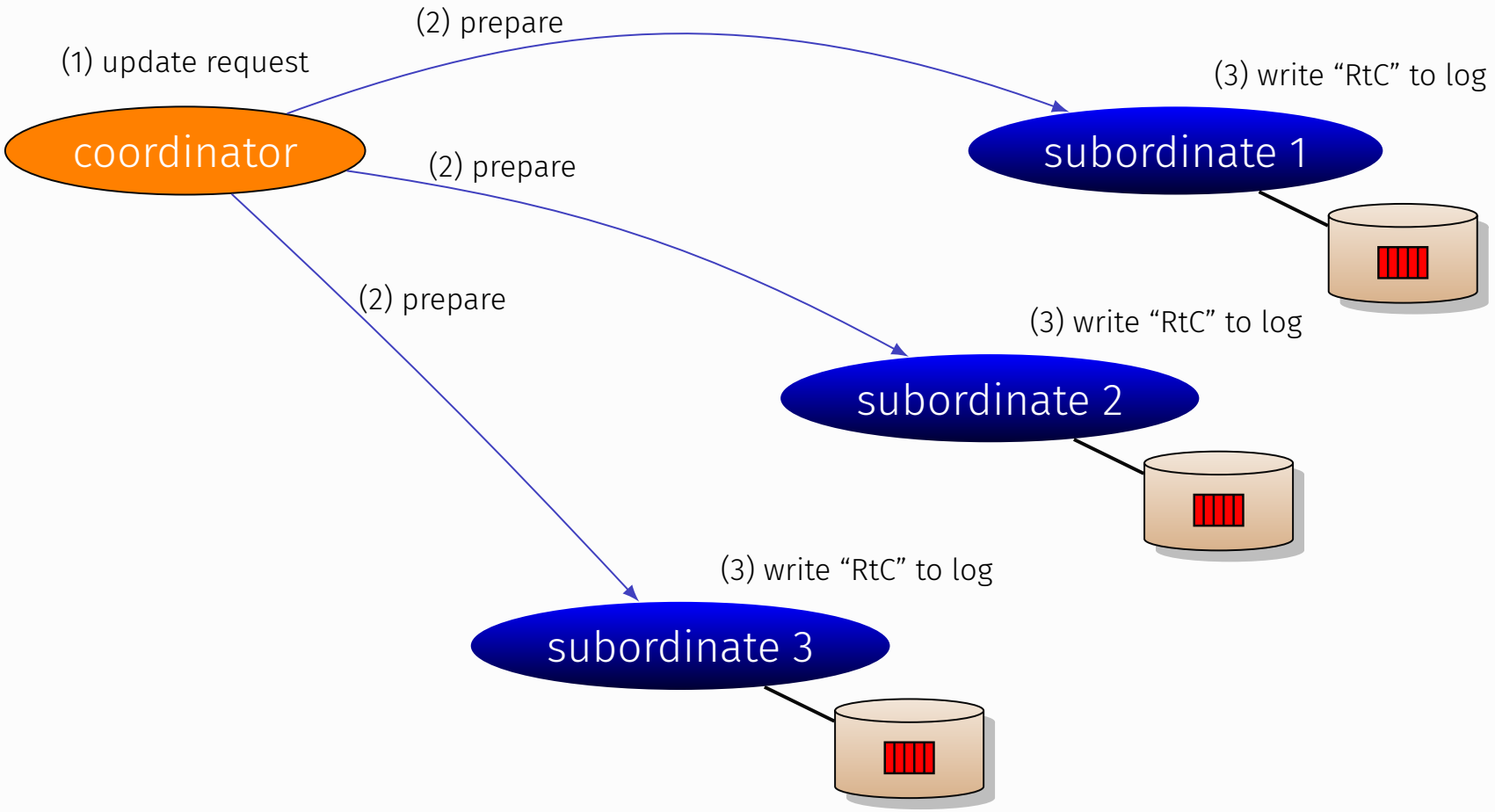
(1) update request



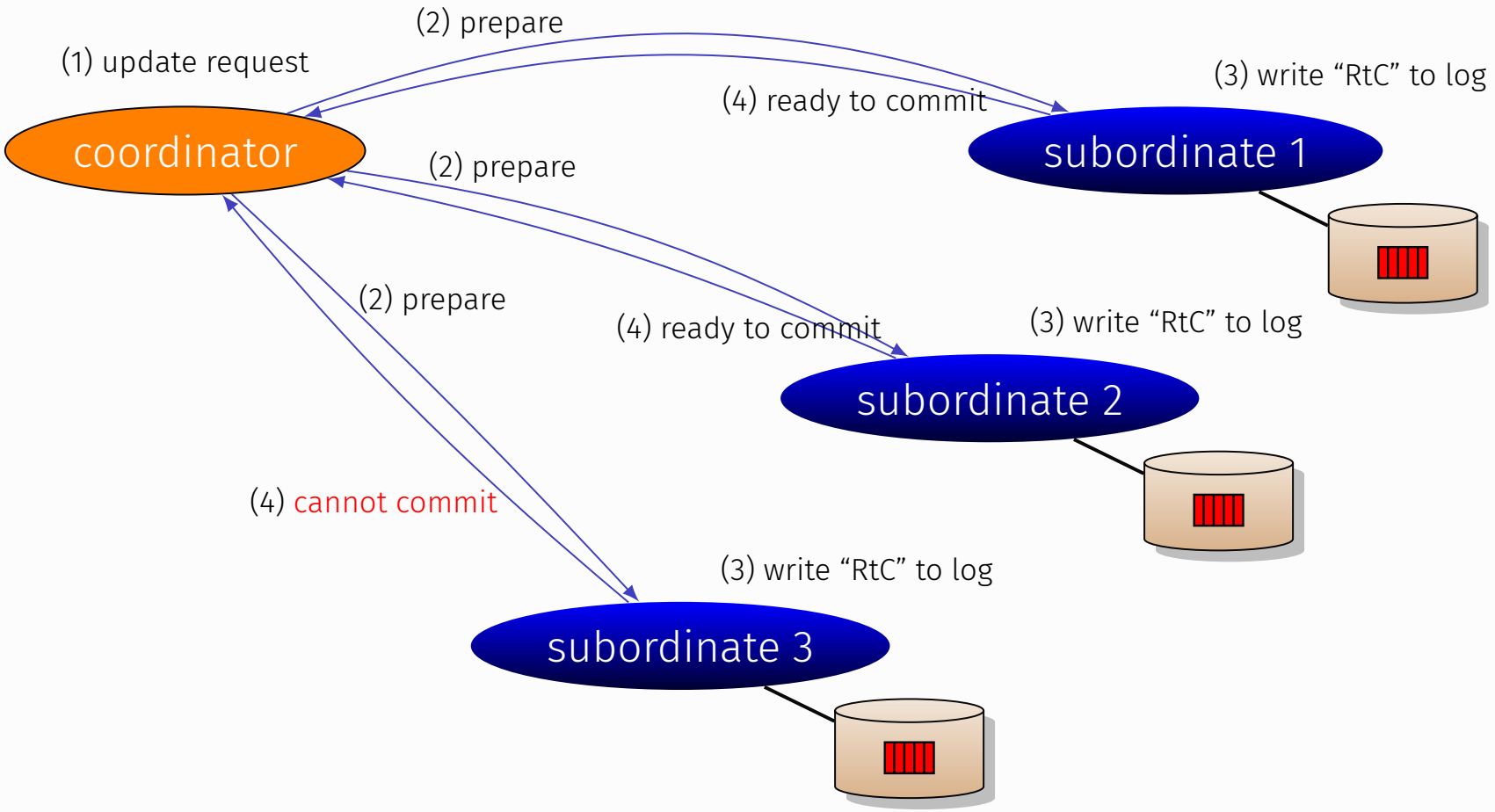
2PC EXAMPLE TWO



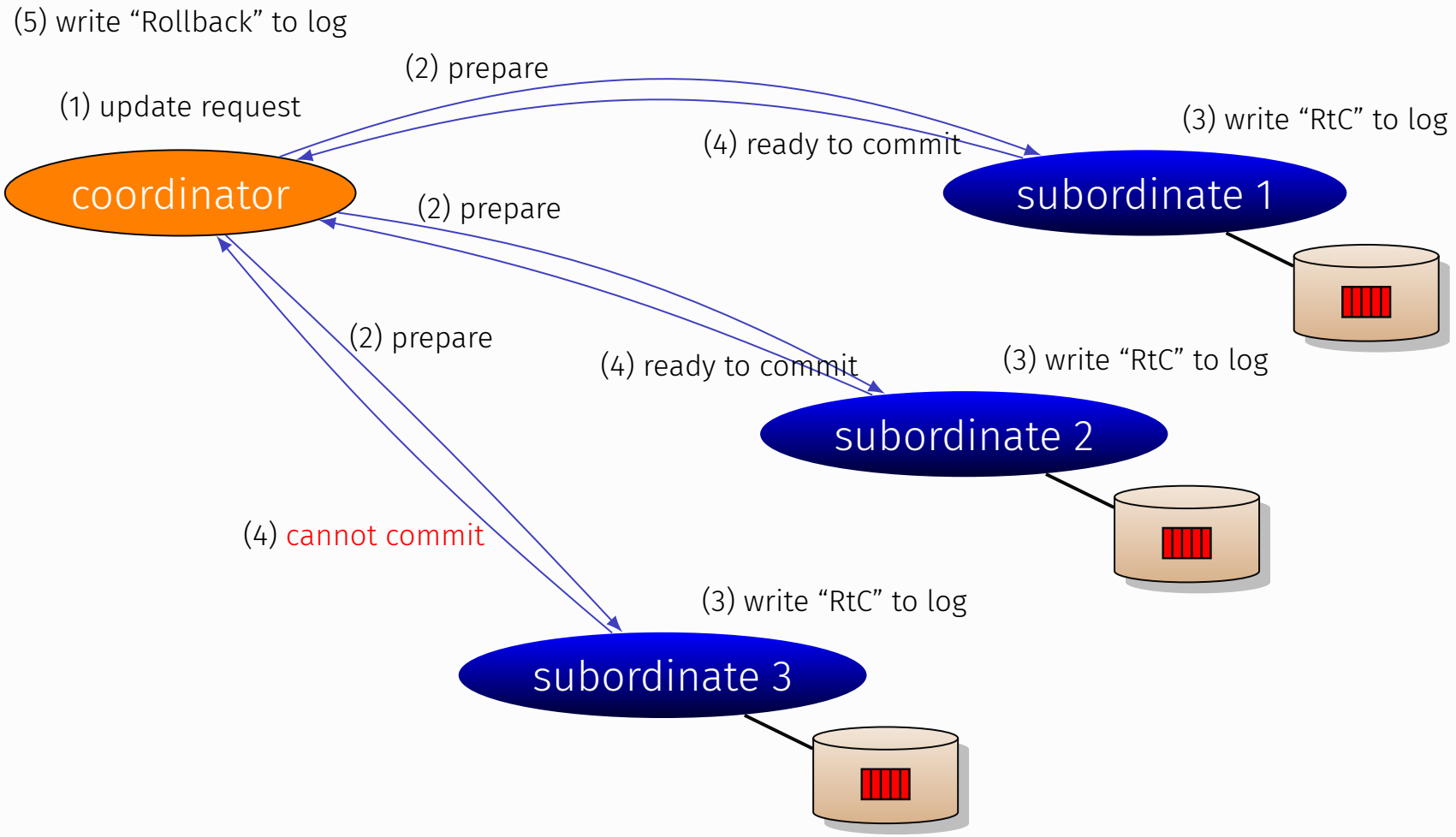
2PC EXAMPLE TWO



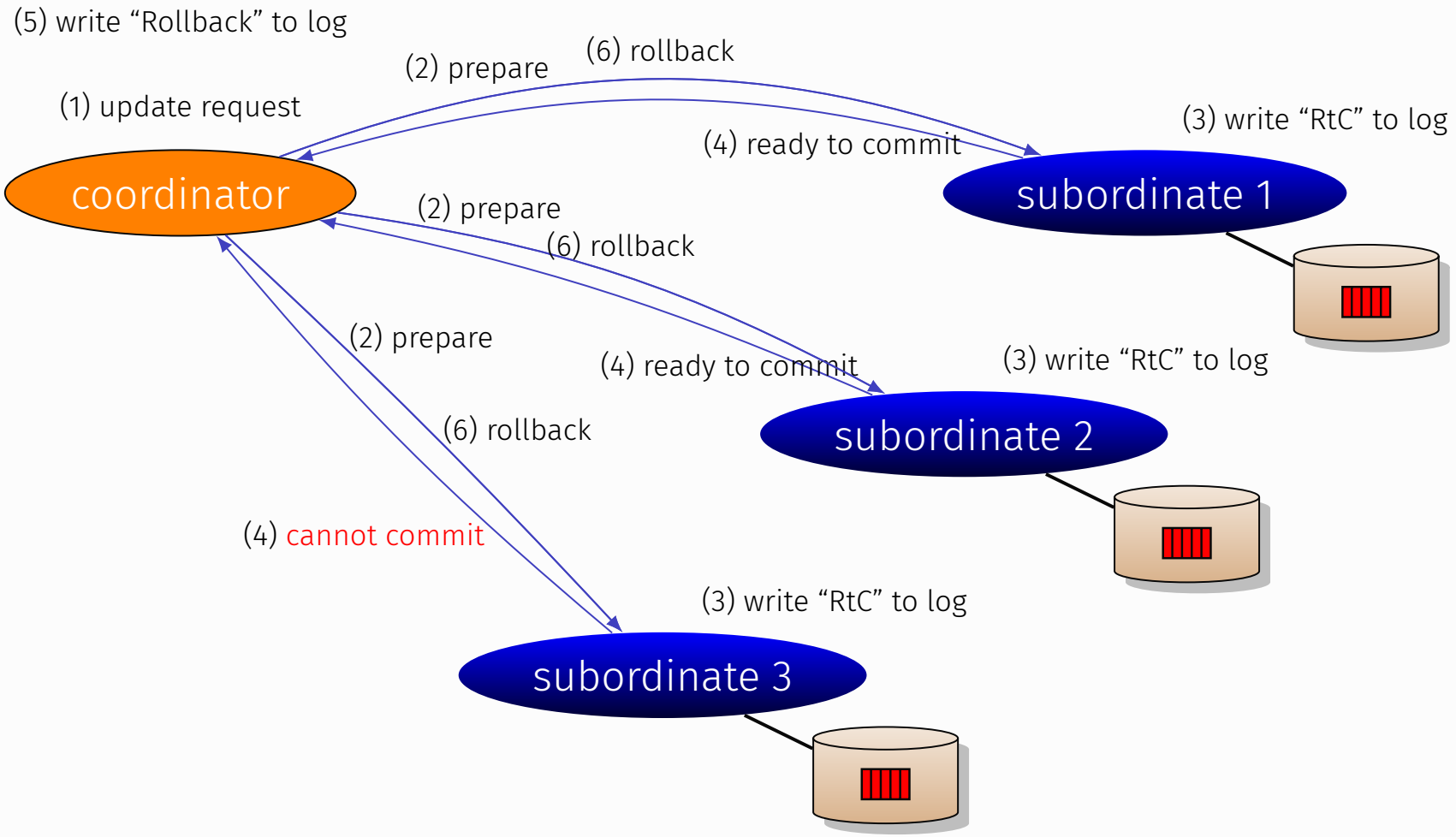
2PC EXAMPLE TWO



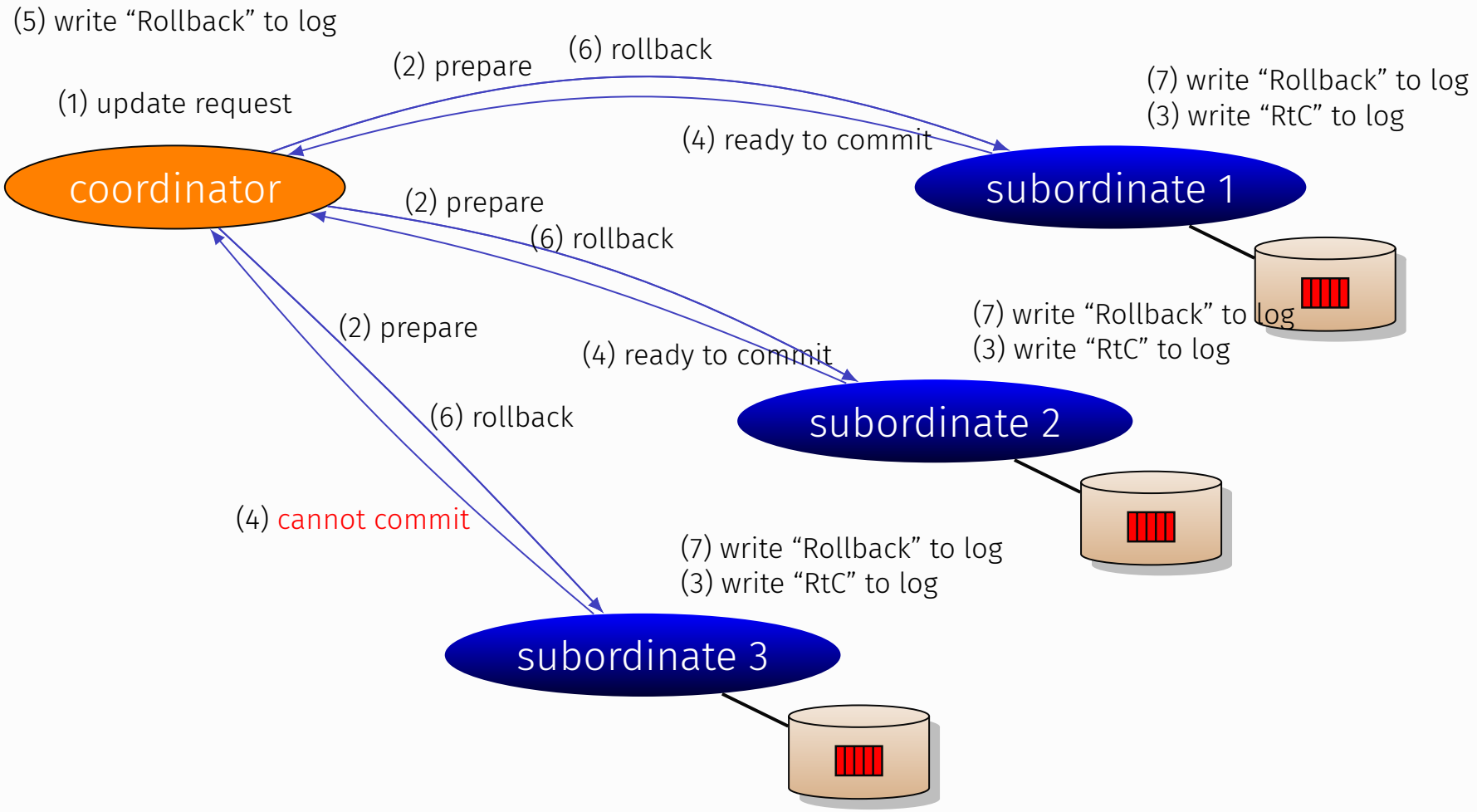
2PC EXAMPLE TWO



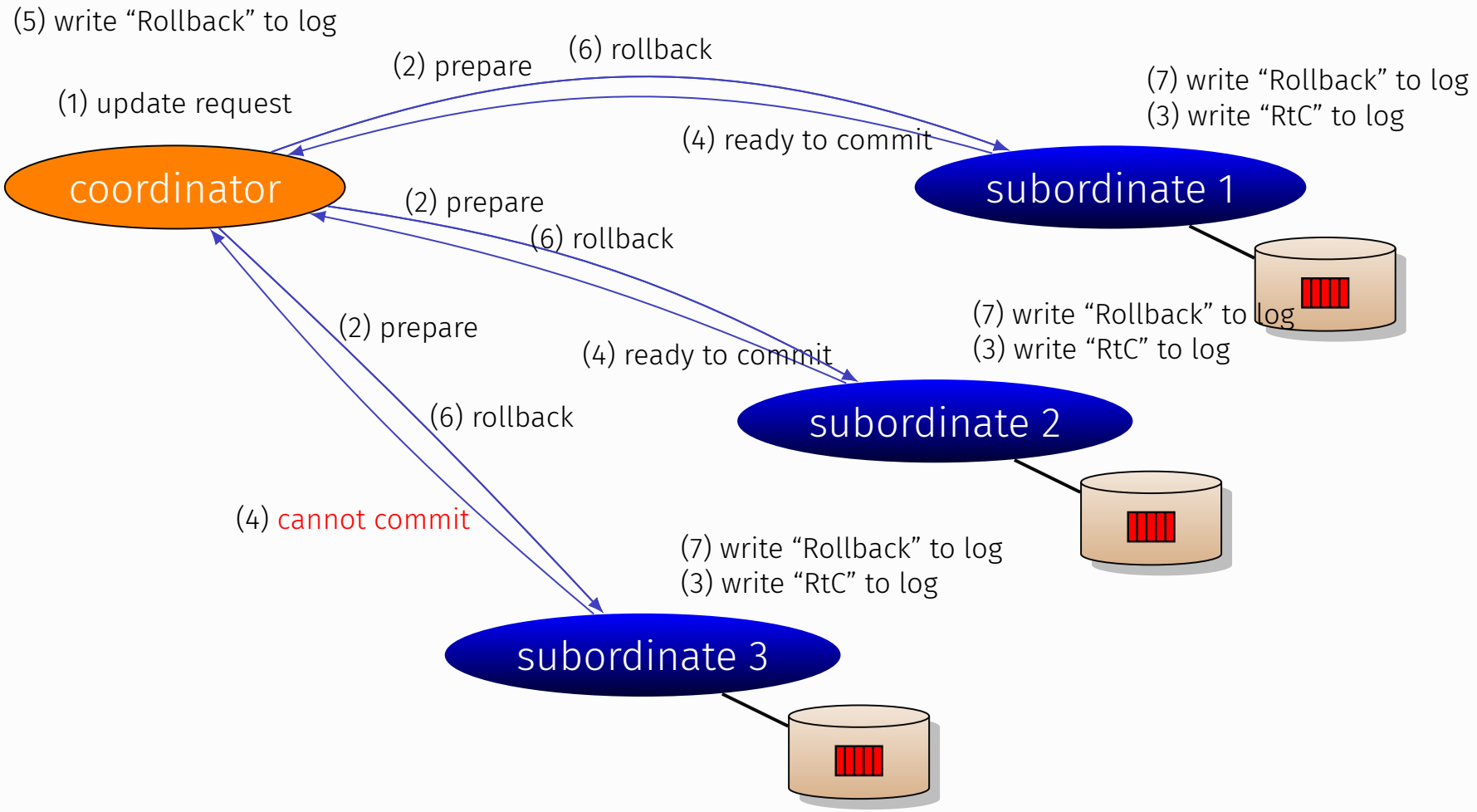
2PC EXAMPLE TWO



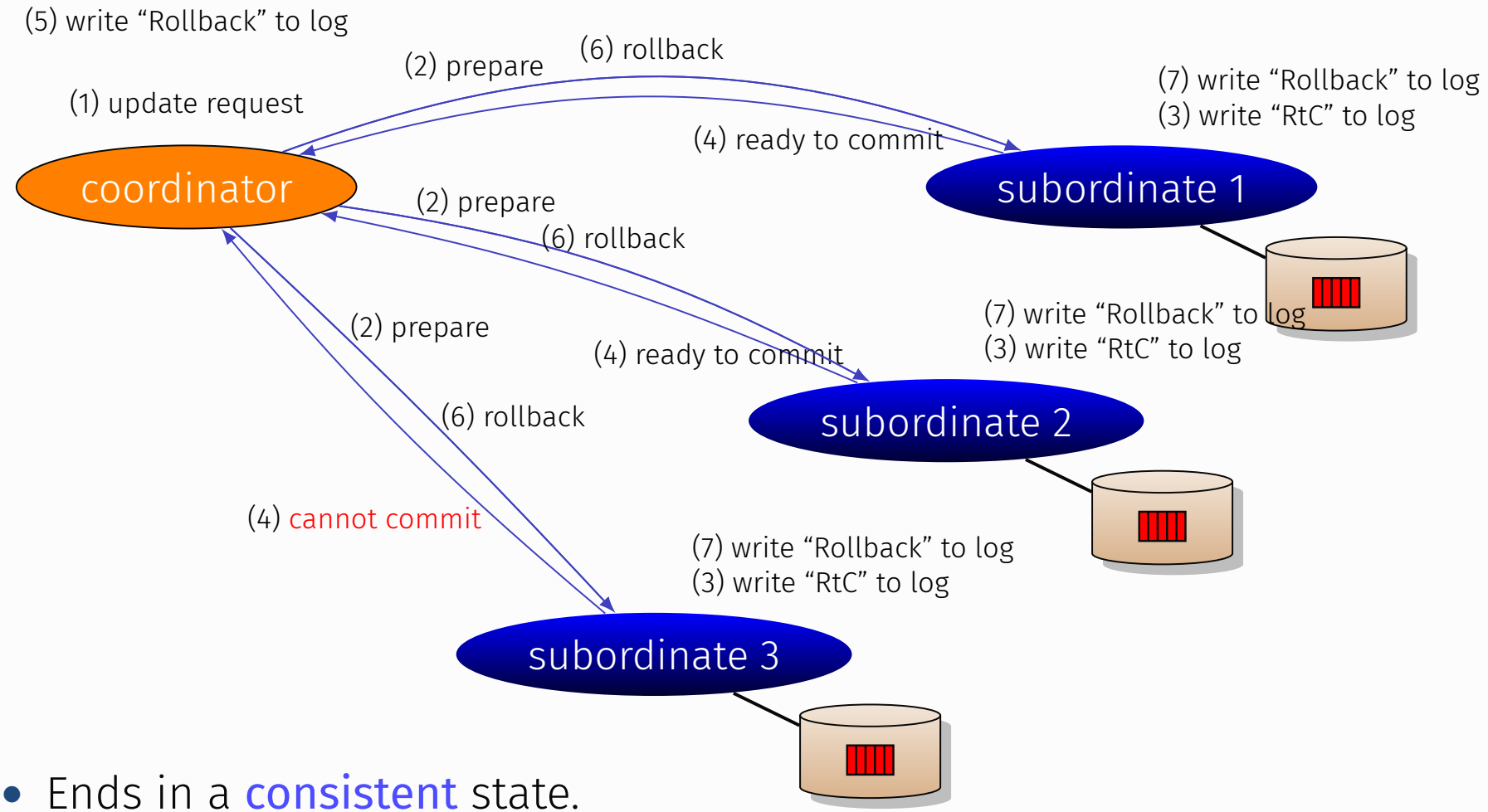
2PC EXAMPLE TWO



2PC EXAMPLE TWO

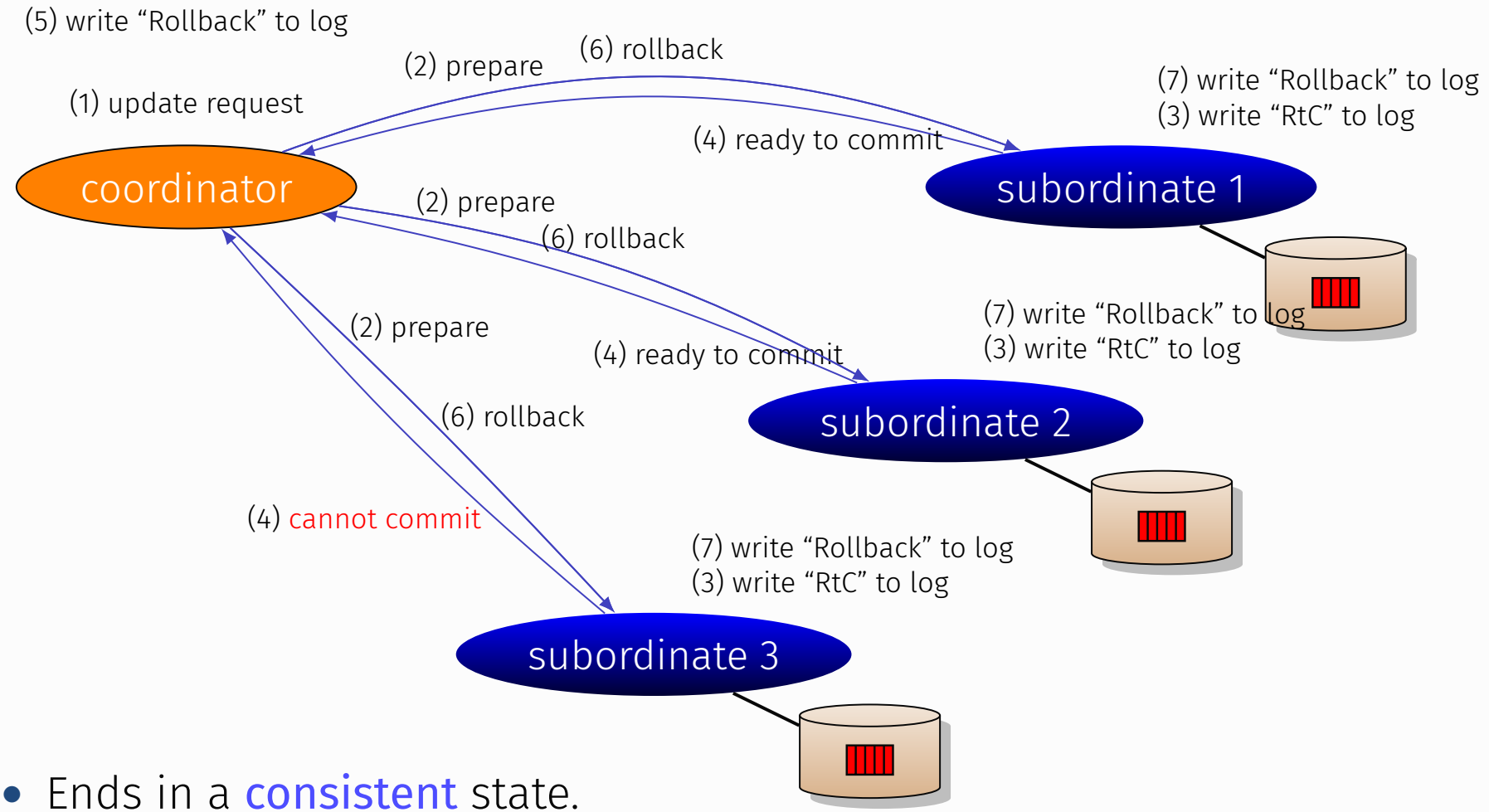


2PC EXAMPLE TWO



- Ends in a consistent state.

2PC EXAMPLE TWO



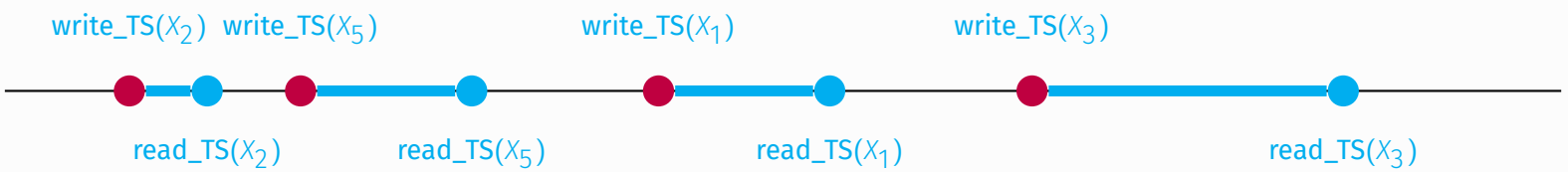
- Ends in a consistent state.
- 2PC can freeze parts of your database for a long time if the Coordinator goes down for a while after “ready to commit”.

MULTIVERSION CONCURRENCY CONTROL (MVCC)

- Implemented in quite a few SQL and NoSQL databases:
  - *CouchDB, IBM DB2, HBase, MariaDB, MySQL, Oracle, PostgreSQL, SAP Hana*
- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction.
- Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effect:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions.
  - To check unlimited growth of versions, a garbage collection is run when certain criteria are satisfied.

MVCC BASICS

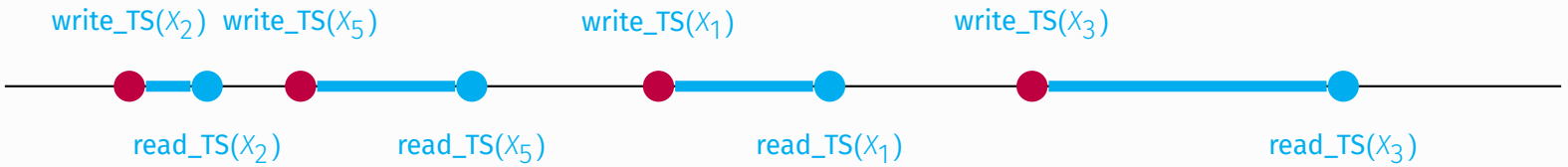
- Every transaction T is assumed to have a timestamp $\mathrm{TS}(T)$
  - We will pretend this is when the transaction was atomically executed.
- Assume $X_1, X_2, \ldots, X_n$ are the versions of a data item X created by a write operations of transactions.
- With each X_i we associate two timestamps:
  - $\mathrm{read\_TS}(X_i)$: the largest of the timestamps of transactions that have successfully read version X_i.
  - $\mathrm{write\_TS}(X_i)$: the timestamp of the transaction that wrote the value of version X_i.
- A new version of X_i is created only by a write operation.



MVCC WRITE-RULE

To ensure serializability, the following rule for writing is used:

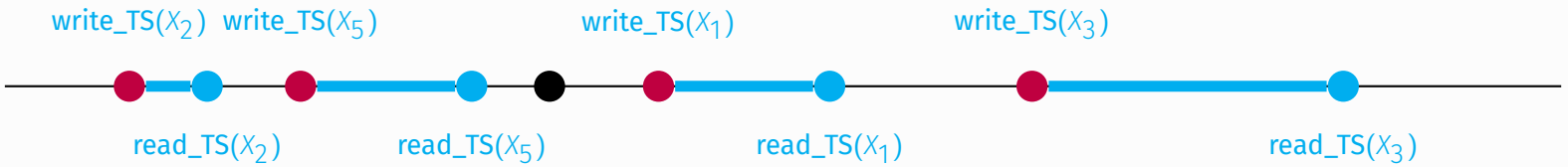
- Assume transaction T attempts to write X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- If read_TS(X_i) $>$ TS(T) then reject
- Else, create a new version X_j and read_TS(X_j) $=$ write_TS(X_j) $=$ TS(T).



MVCC WRITE-RULE

To ensure serializability, the following rule for writing is used:

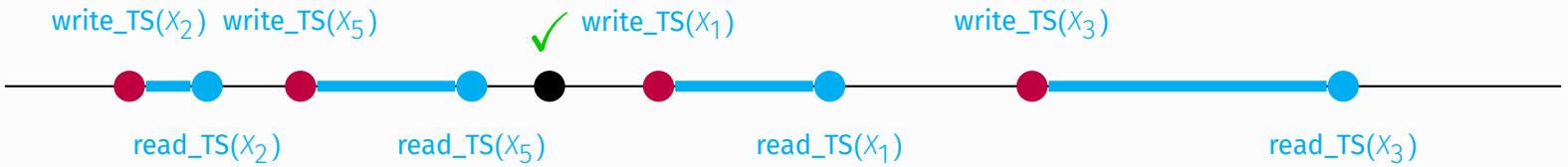
- Assume transaction T attempts to write X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- If read_TS(X_i) $>$ TS(T) then reject
- Else, create a new version X_j and read_TS(X_j) $=$ write_TS(X_j) $=$ TS(T).



MVCC WRITE-RULE

To ensure serializability, the following rule for writing is used:

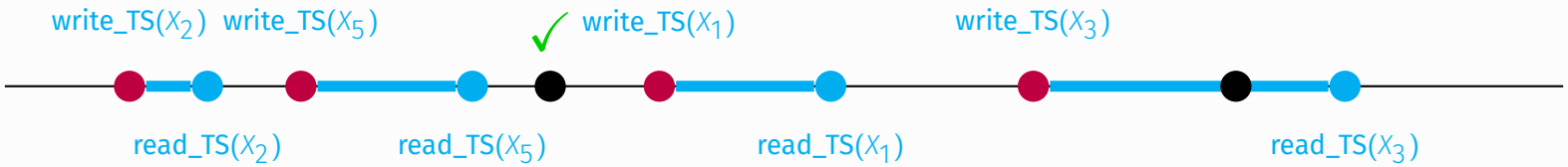
- Assume transaction T attempts to write X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- If read_TS(X_i) $>$ TS(T) then reject
- Else, create a new version X_j and read_TS(X_j) $=$ write_TS(X_j) $=$ TS(T).



MVCC WRITE-RULE

To ensure serializability, the following rule for writing is used:

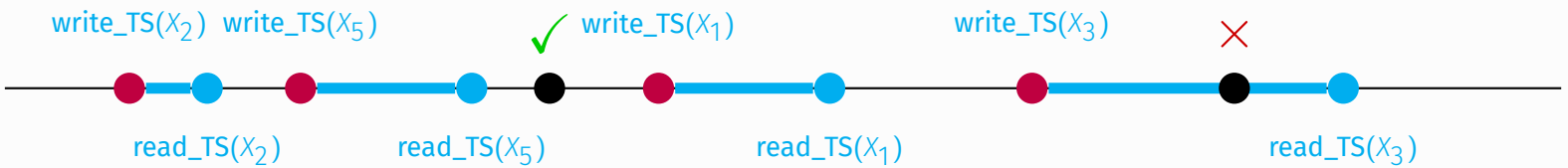
- Assume transaction T attempts to write X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- If read_TS(X_i) $>$ TS(T) then reject
- Else, create a new version X_j and read_TS(X_j) $=$ write_TS(X_j) $=$ TS(T).



MVCC WRITE-RULE

To ensure serializability, the following rule for writing is used:

- Assume transaction T attempts to write X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- If read_TS(X_i) $>$ TS(T) then reject
- Else, create a new version X_j and read_TS(X_j) $=$ write_TS(X_j) $=$ TS(T).

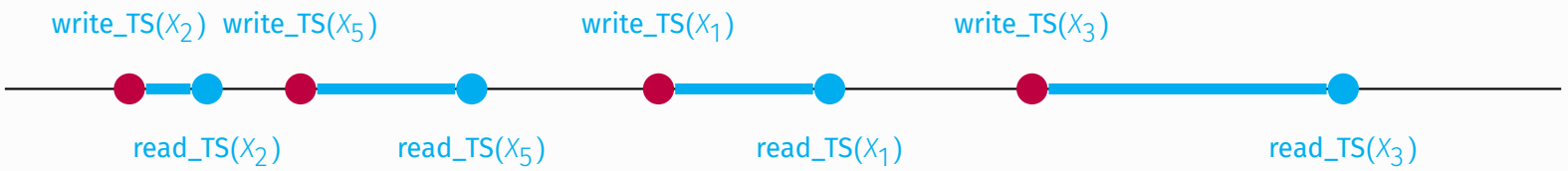


MVCC READ-RULE

To ensure serializability, the following rule for reading is used:

- Assume transaction T attempts to read X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- Return the value of X_i to T.
- Set read_TS(X_i) to the largest of TS(T) and read_TS(X_i).

Note: This rule never rejects a read operation.

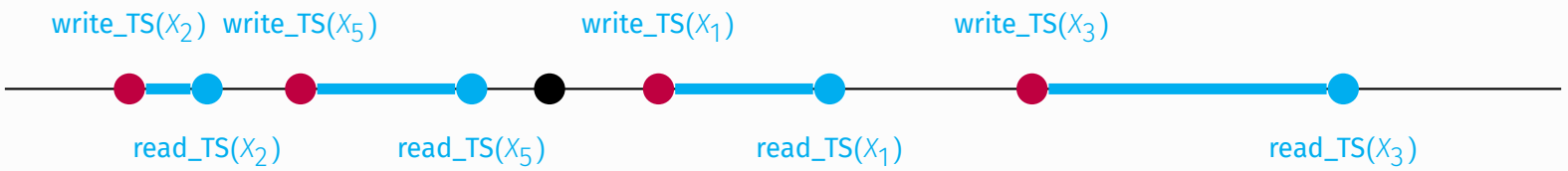


MVCC READ-RULE

To ensure serializability, the following rule for reading is used:

- Assume transaction T attempts to read X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- Return the value of X_i to T.
- Set read_TS(X_i) to the largest of TS(T) and read_TS(X_i).

Note: This rule never rejects a read operation.

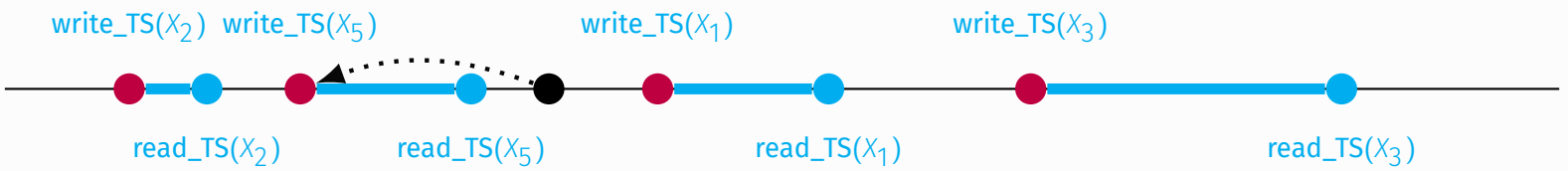


MVCC READ-RULE

To ensure serializability, the following rule for reading is used:

- Assume transaction T attempts to read X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- Return the value of X_i to T.
- Set read_TS(X_i) to the largest of TS(T) and read_TS(X_i).

Note: This rule never rejects a read operation.

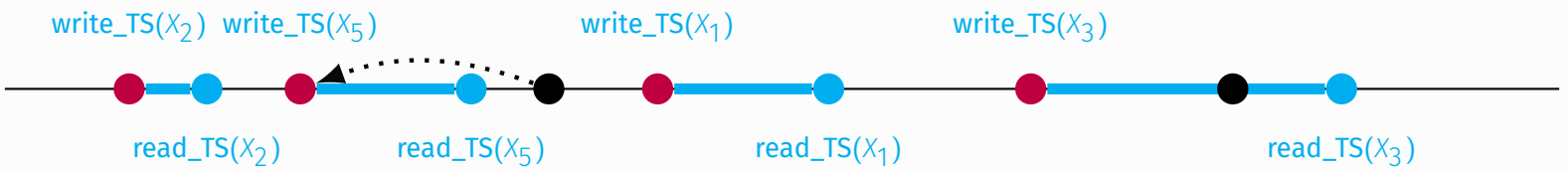


MVCC READ-RULE

To ensure serializability, the following rule for reading is used:

- Assume transaction T attempts to read X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- Return the value of X_i to T.
- Set read_TS(X_i) to the largest of TS(T) and read_TS(X_i).

Note: This rule never rejects a read operation.

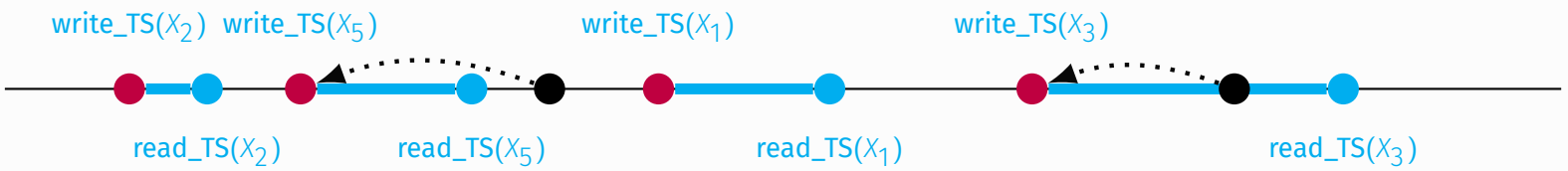


MVCC READ-RULE

To ensure serializability, the following rule for reading is used:

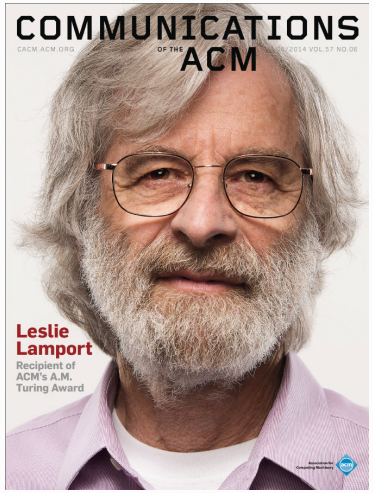
- Assume transaction T attempts to read X.
- Let X_i be the version of X with the highest write_TS(X_i) that is also less than or equal to TS(T).
- Return the value of X_i to T.
- Set read_TS(X_i) to the largest of TS(T) and read_TS(X_i).

Note: This rule never rejects a read operation.



THE PAXOS PROTOCOL

- One of the most well-known and widely used agreement protocols to let a group of agents collectively agree on something.
  - Even if some agents disappear temporarily or messages are delayed and/or lost.



Leslie Lamport. 1998. *The part-time parliament.* ACM Trans. Comput. Syst. 16, 2 (May 1998), 133-169. DOI=http://dx.doi.org/10.1145/279227.279229

- Used in:
  - Google: Chubby (Paxos-based distributed lock service)
    - Most Google services use Chubby directly or indirectly
  - Yahoo: Zookeeper (Paxos-based distributed lock service)
    - Zookeeper is open-source and integrates with Hadoop

PAXOS BASICS

The different roles that participants can play:

- Proposers
  - Suggests values for consideration by acceptors.
  - *E.g., nodes that want a lock to update the same record.*
- Acceptors
  - Considers the values proposed by proposers.
  - Renders an accept/reject decision.
  - *E.g., a set of nodes assigned for arbitration of update conflicts.*
- Learners
  - Learns the chosen value.
  - *E.g., nodes that store the record and will register the lock if there is agreement.*

A node can act in more than one role at the same time.

THE PAXOS PROTOCOL, PHASE 1

- **Phase 1a, Prepare:** A Proposer sends a Prepare message with a proposal number n to a majority of the Acceptors.
  - The number n must increase with each round.
- **Phase 1b, Promise:** The Acceptor checks if n is higher than any proposal number received from any Proposer. If so, the Acceptor returns a Promise message with m and u, where m and u are the proposal number and value of the highest-numbered proposal accepted by it so far.
  - This implies a promise to ignore any proposal with a number $< n$

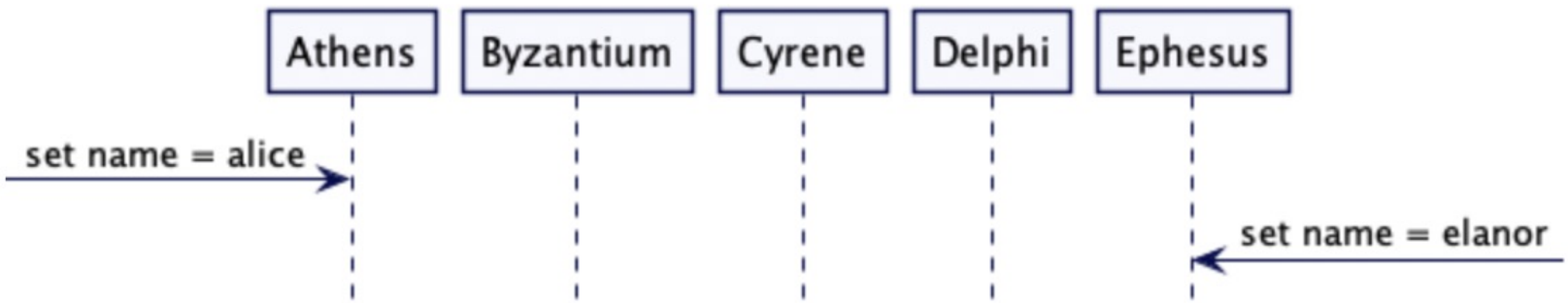
THE PAXOS PROTOCOL, PHASE 2

- **Phase 2a, Accept Request:** The Proposer waits until it has received promises from a majority of the Acceptors, and then checks if any of them already accepted a value earlier. If so, then v is set to the u of the Promise message with the highest m. Then, an Accept request message with n and v is sent to the Acceptors that responded.
- **Phase 2b, Accept:** If an Acceptor receives an Accept Request message with n and v, it checks if it promised to ignore all proposals with proposal number n. If not, it accepts the proposal and sends an Accept message to the Proposer and all the Learners.

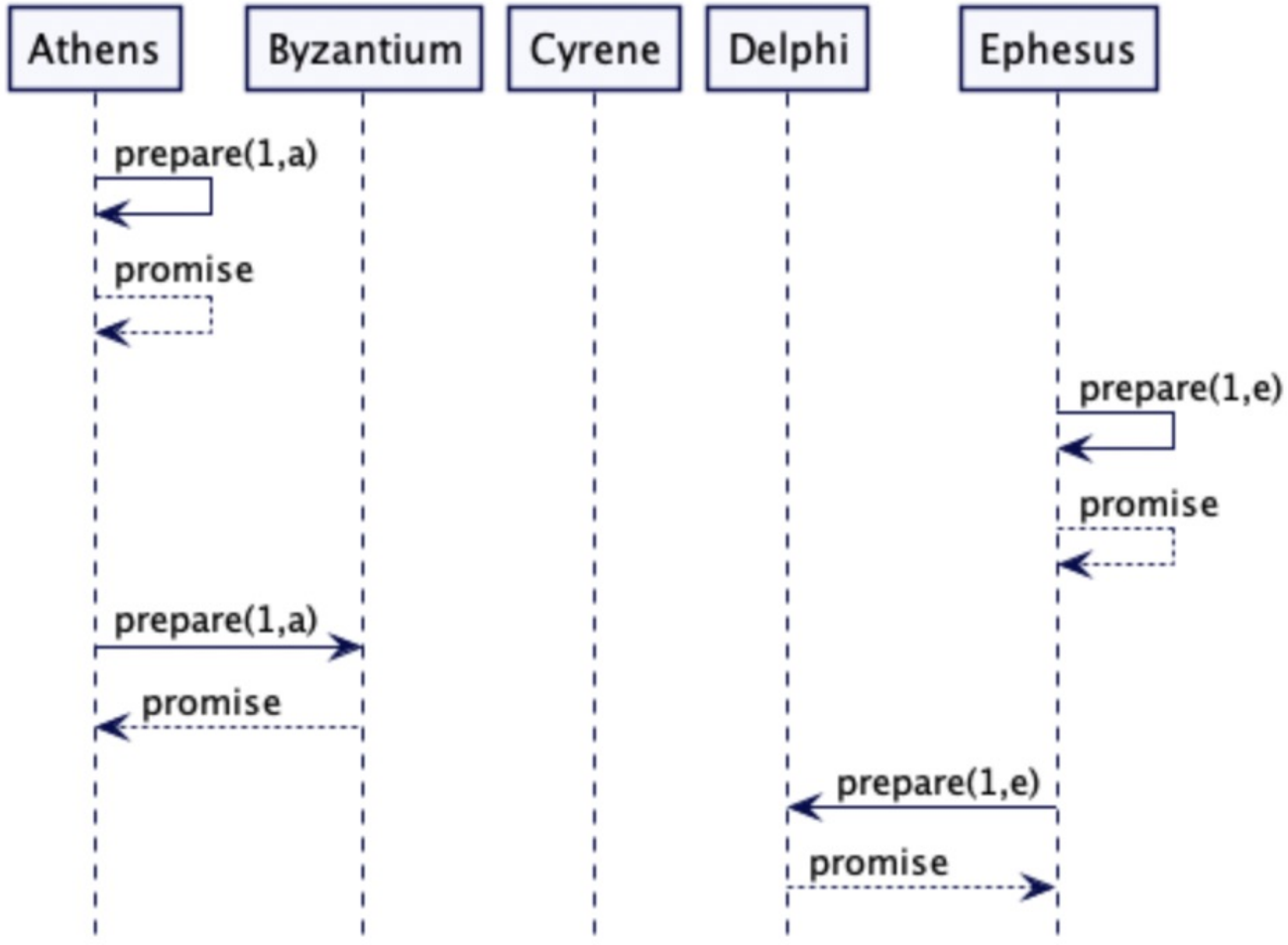
Conclude or Retry: If a Learner gets Accept messages from a majority of the Acceptors, a decision has been reached. If a Proposer notices that this is not happening, it may initiate a new round with higher n.

PAXOS EXAMPLE

there are five nodes
Athens and **Ephesus** receive an update, so they will act as a proposer
all nodes act as acceptors
Which update will reach consensus?

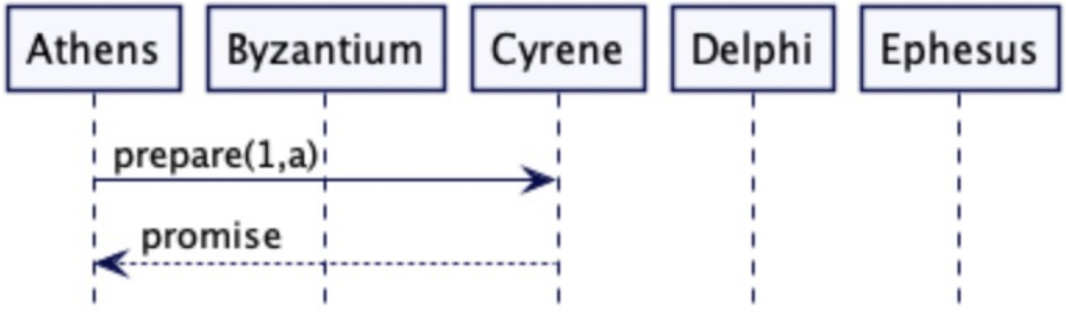


PAXOS EXAMPLE



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	0	1,e	1,e
accepted value	none	none	none	none	none

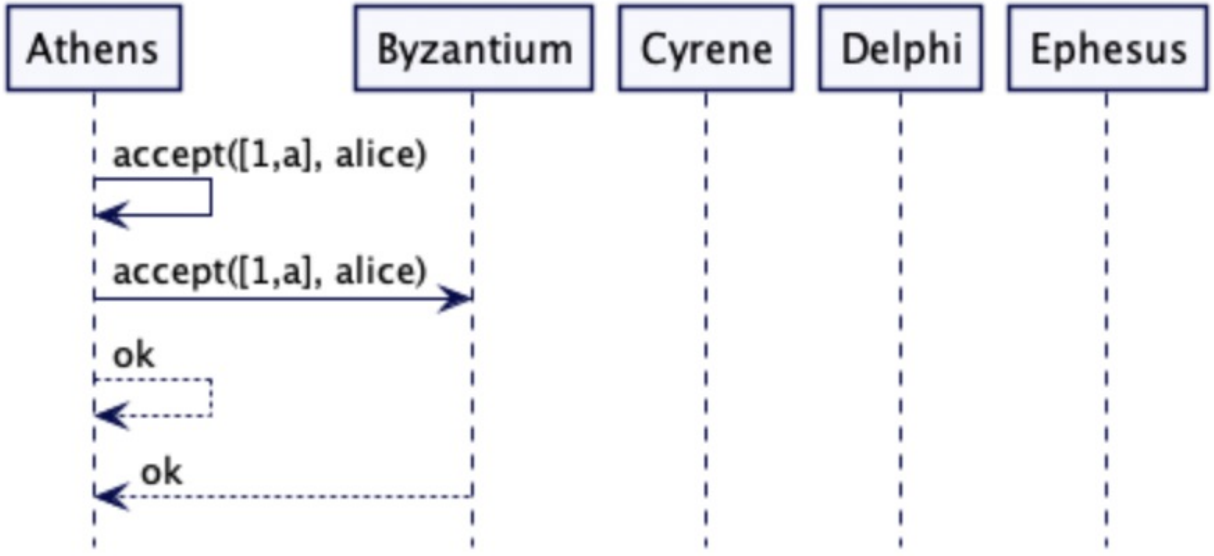
PAXOS EXAMPLE



Athens has reached a quorum of 3 nodes, so moves to Phase 2

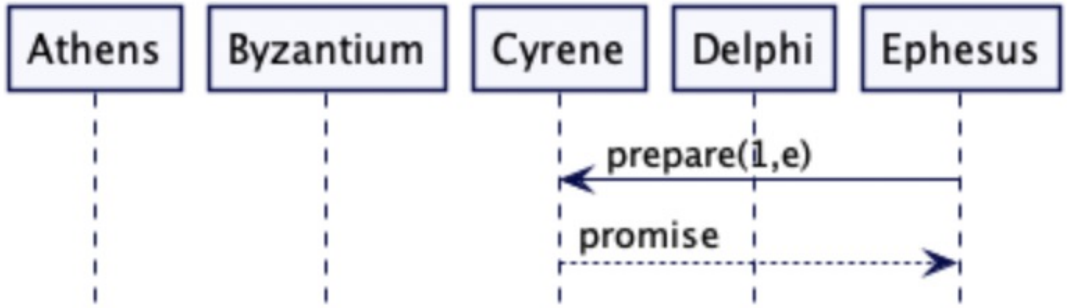
Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,a	1,e	1,e
accepted value	none	none	none	none	none

PAXOS EXAMPLE



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,a	1,e	1,e
accepted value	alice	alice	none	none	none

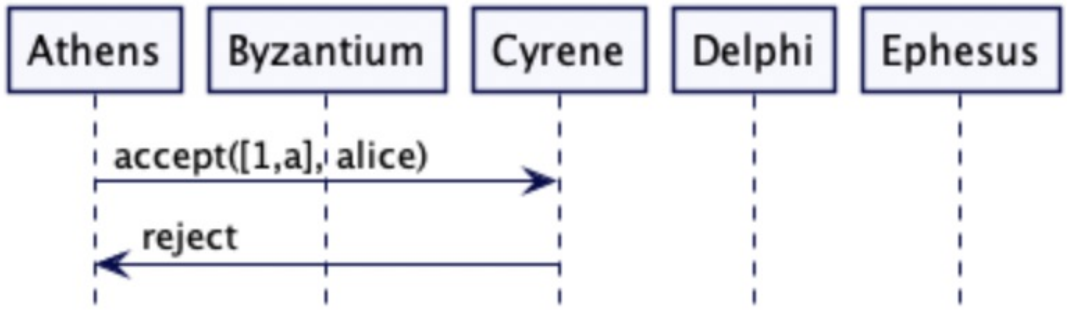
PAXOS EXAMPLE



Ephesus has also reached a quorum of 3 nodes, so moves to Phase 2

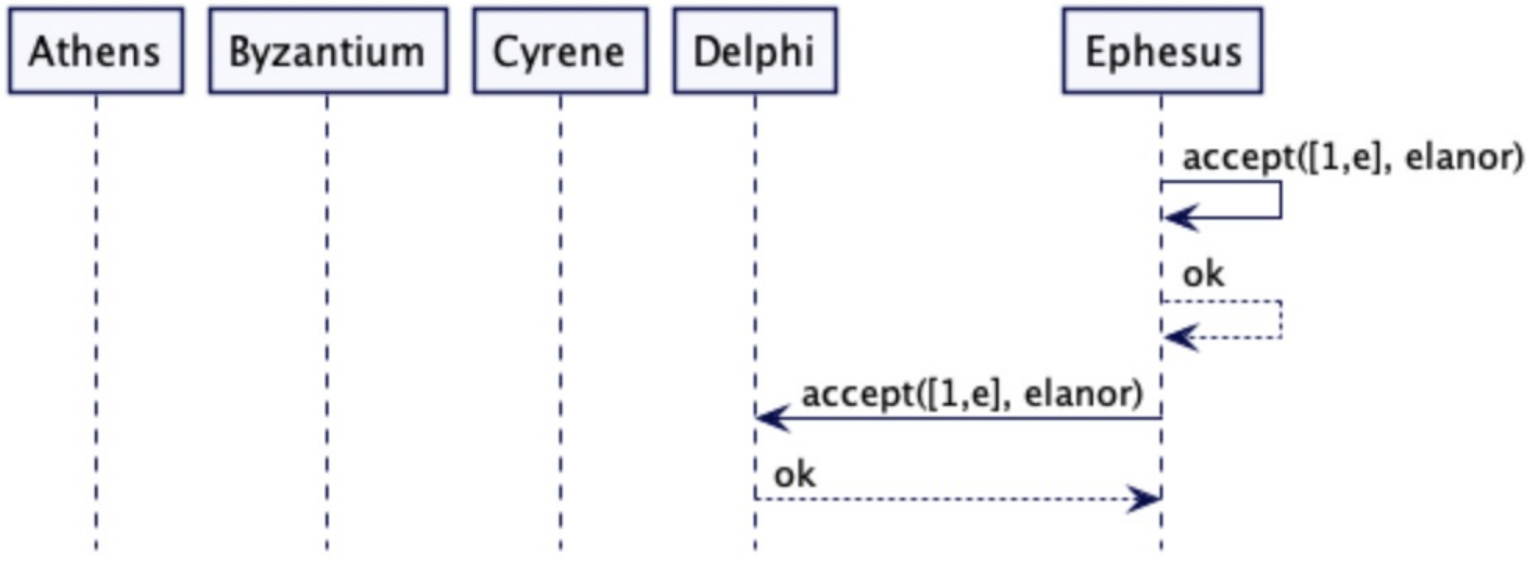
Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,e	1,e	1,e
accepted value	alice	alice	none	none	none

PAXOS EXAMPLE



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,e	1,e	1,e
accepted value	alice	alice	none	none	none

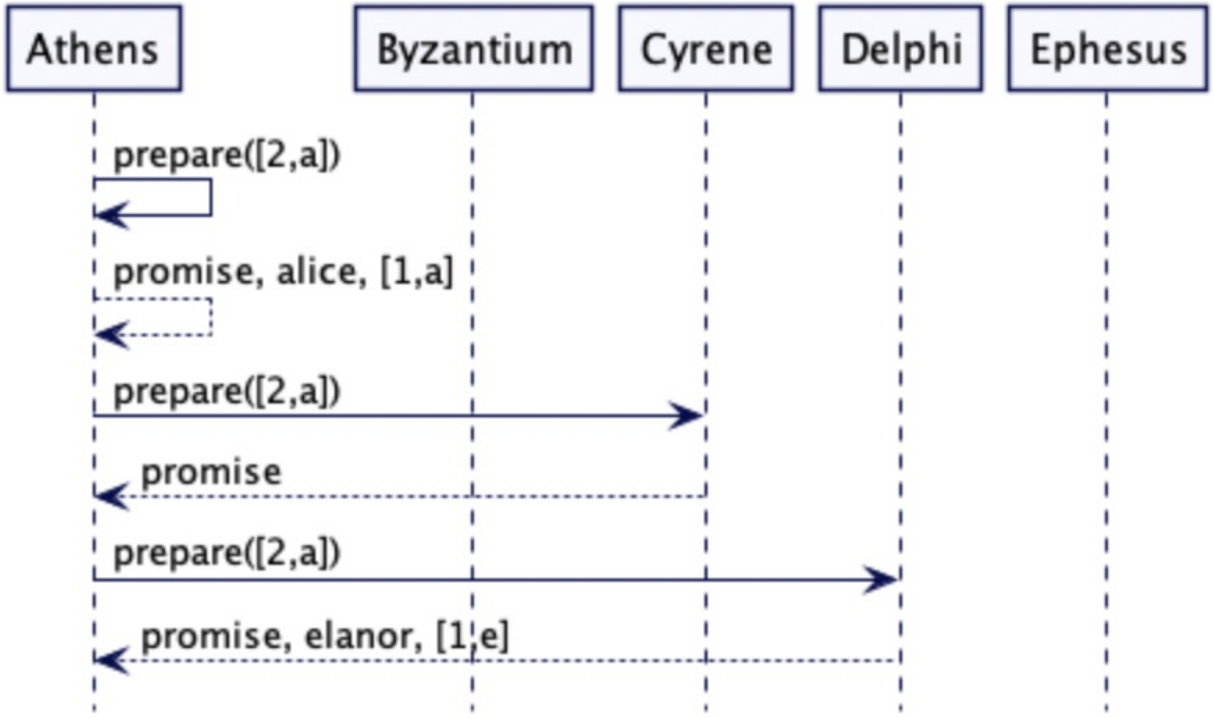
PAXOS EXAMPLE



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,e	1,e	1,e
accepted value	alice	alice	none	elanor	elanor

PAXOS EXAMPLE

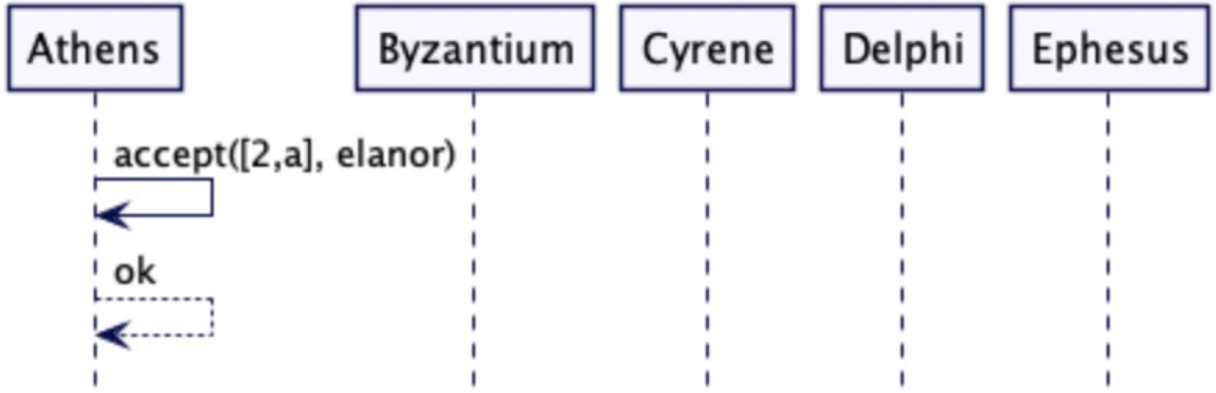
Ephesus had reached a quorum, but has now **crashed**
Athens begins another round of Phase 1, with a higher generation number [2,a]



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	1,a	2,a	2,a	1,e
accepted value	alice	alice	none	elanor	elanor

PAXOS EXAMPLE

Athens has reached quorum, but must accept the already accepted value ‘elanor’ with the highest generation

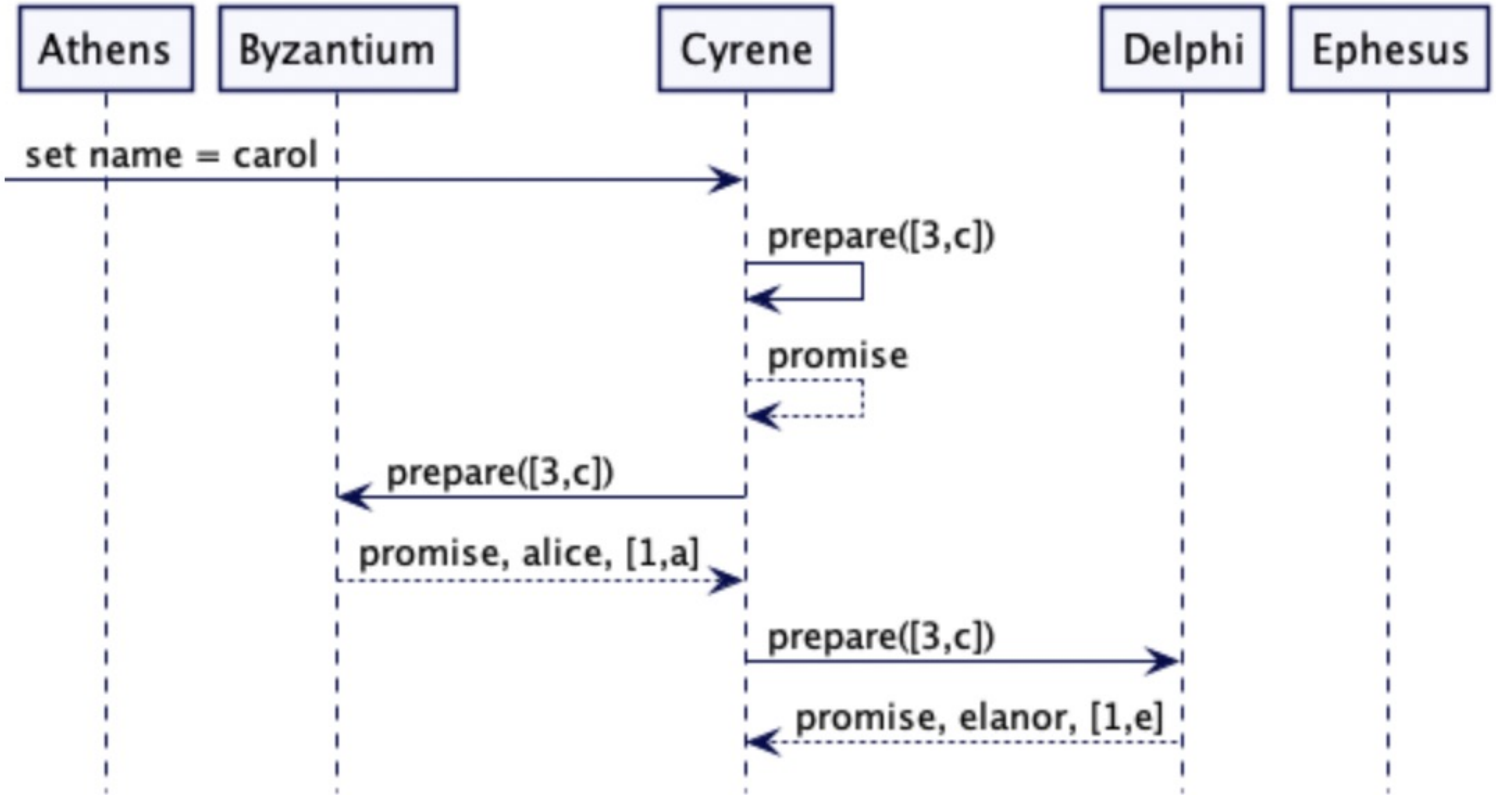


‘elanor’ is now the **chosen value** as there is a quorum that has accepted it however, the nodes do not know that yet!

Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	1,a	2,a	2,a	1,e
accepted value	elanor	alice	none	elanor	elanor

PAXOS EXAMPLE

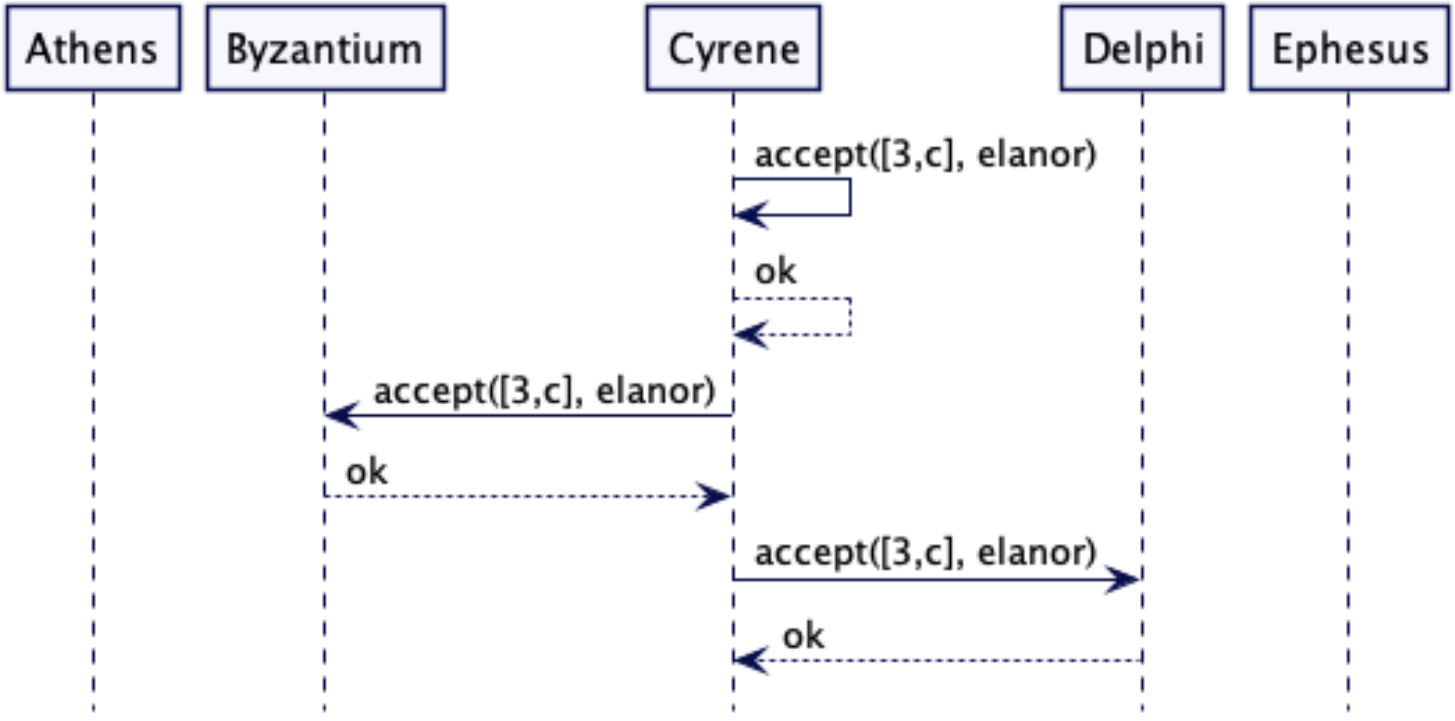
Athens has now **crashed**
Cyrene has received a value and will act as a proposer



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	3,c	3,c	3,c	1,e
accepted value	elanor	alice	none	elanor	elanor

PAXOS EXAMPLE

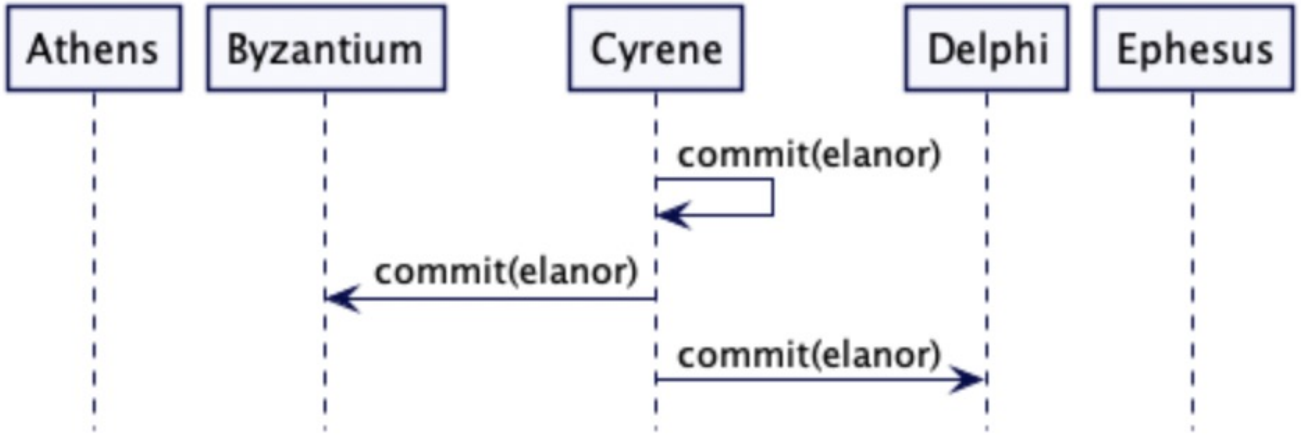
Cyrene sees that ‘elanor’ has been an accepted value with the highest generation number, so it moves to commit this value to all alive nodes (which act as learners)



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	3,c	3,c	3,c	1,e
accepted value	elanor	alice	elanor	elanor	elanor

PAXOS EXAMPLE

Consensus is reached and ‘elanor’ is the accepted value



KEY-VALUE STORES

STORAGE MODEL

- Simple interface
  - Insert/write “value” associated with “key”

    put(key, value);

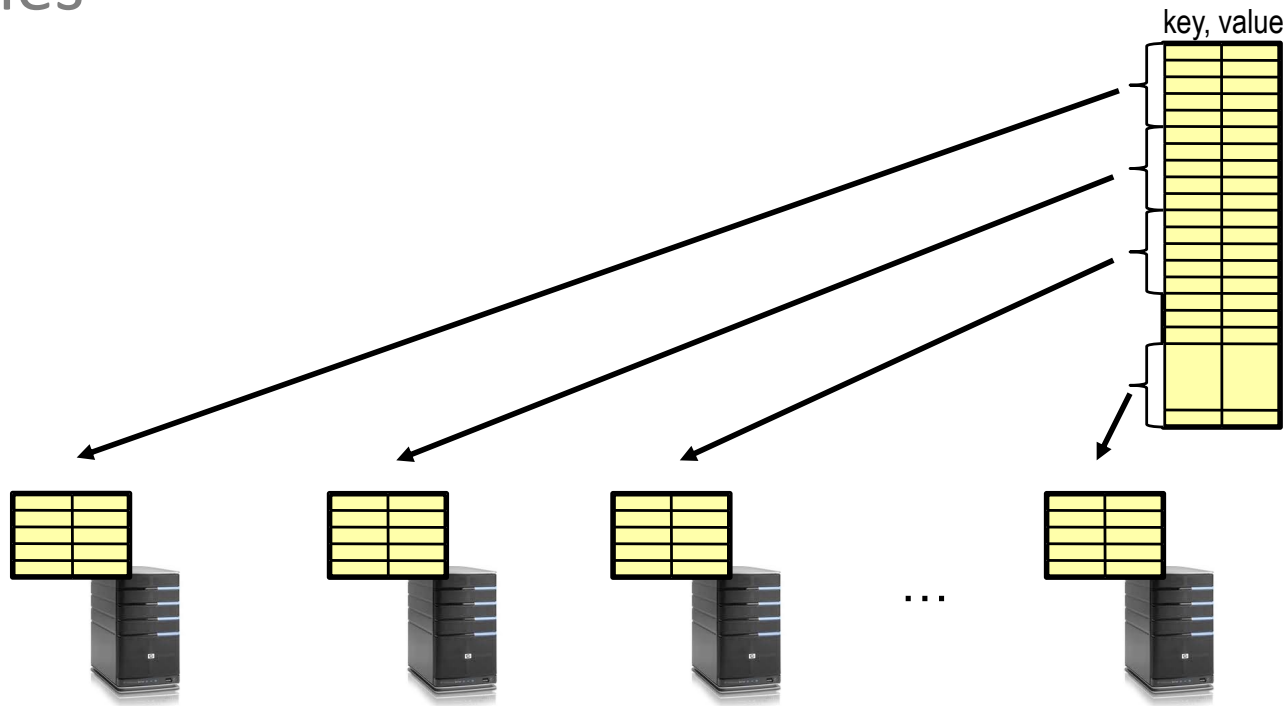
  - Get/read data associated with “key”

    value = get(key);

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

KEY-VALUE STORES

- Similar to Distributed Hash Tables (DHT)
- Main idea
  - Partition set of key-values across multiple machines



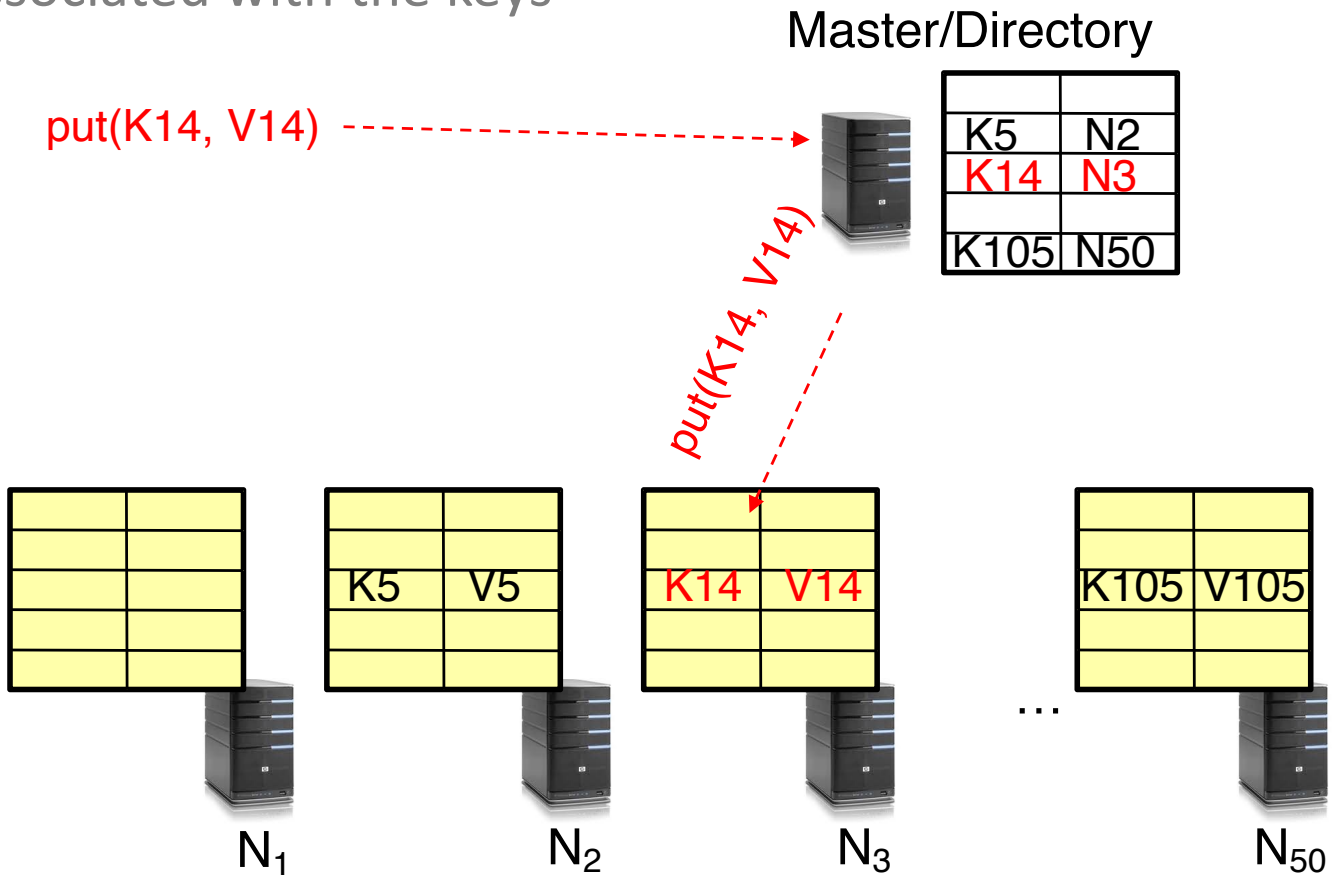
KEY QUESTIONS

- put(key, value)
  - Where do you store a new (key, value) tuple?
- get(key)
  - Where is the value associated with a given "key" stored?

- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

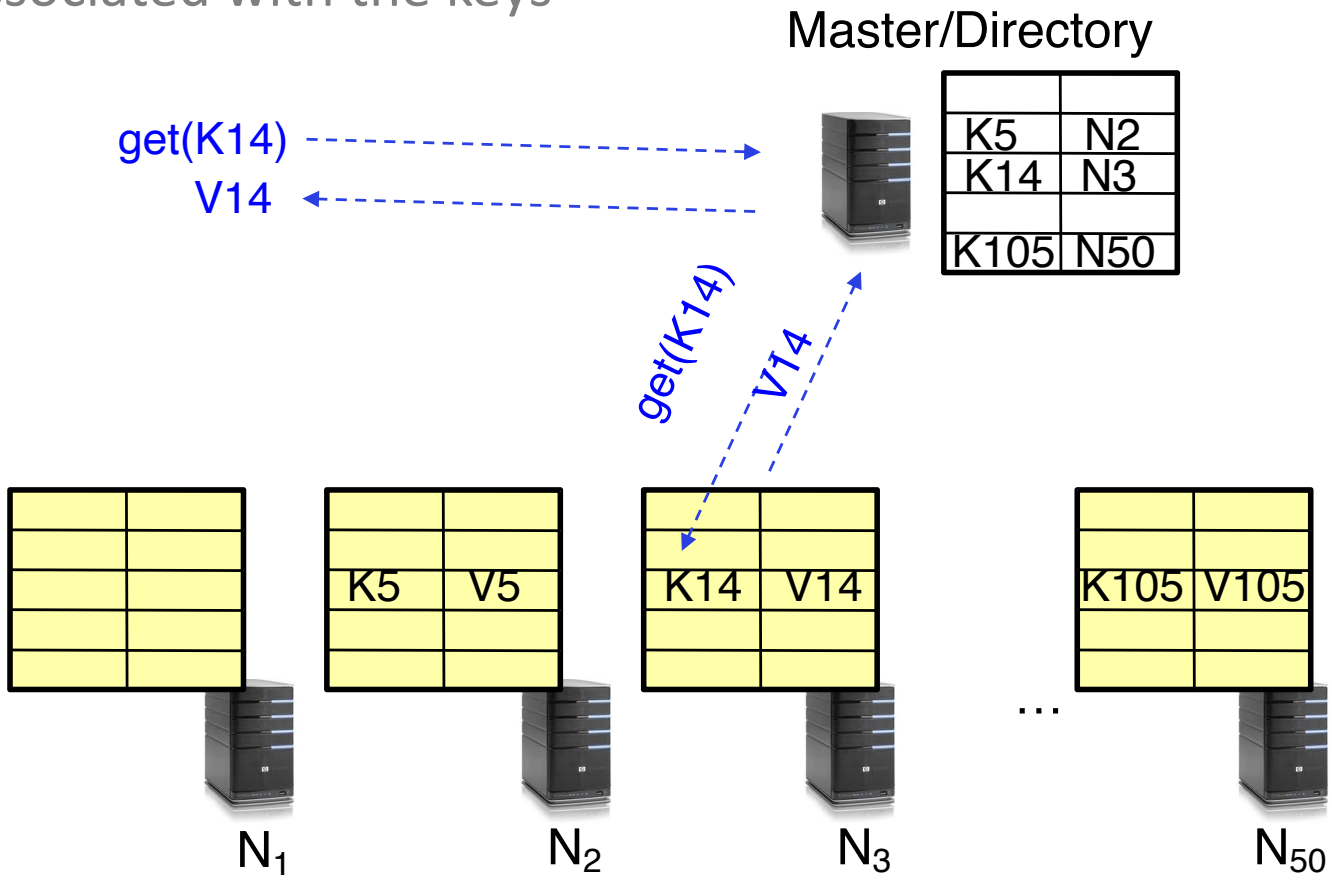
DIRECTORY-BASED ARCHITECTURE

- Designated node called the Master
  - Mapping between keys and the machines (nodes) that store the values associated with the keys



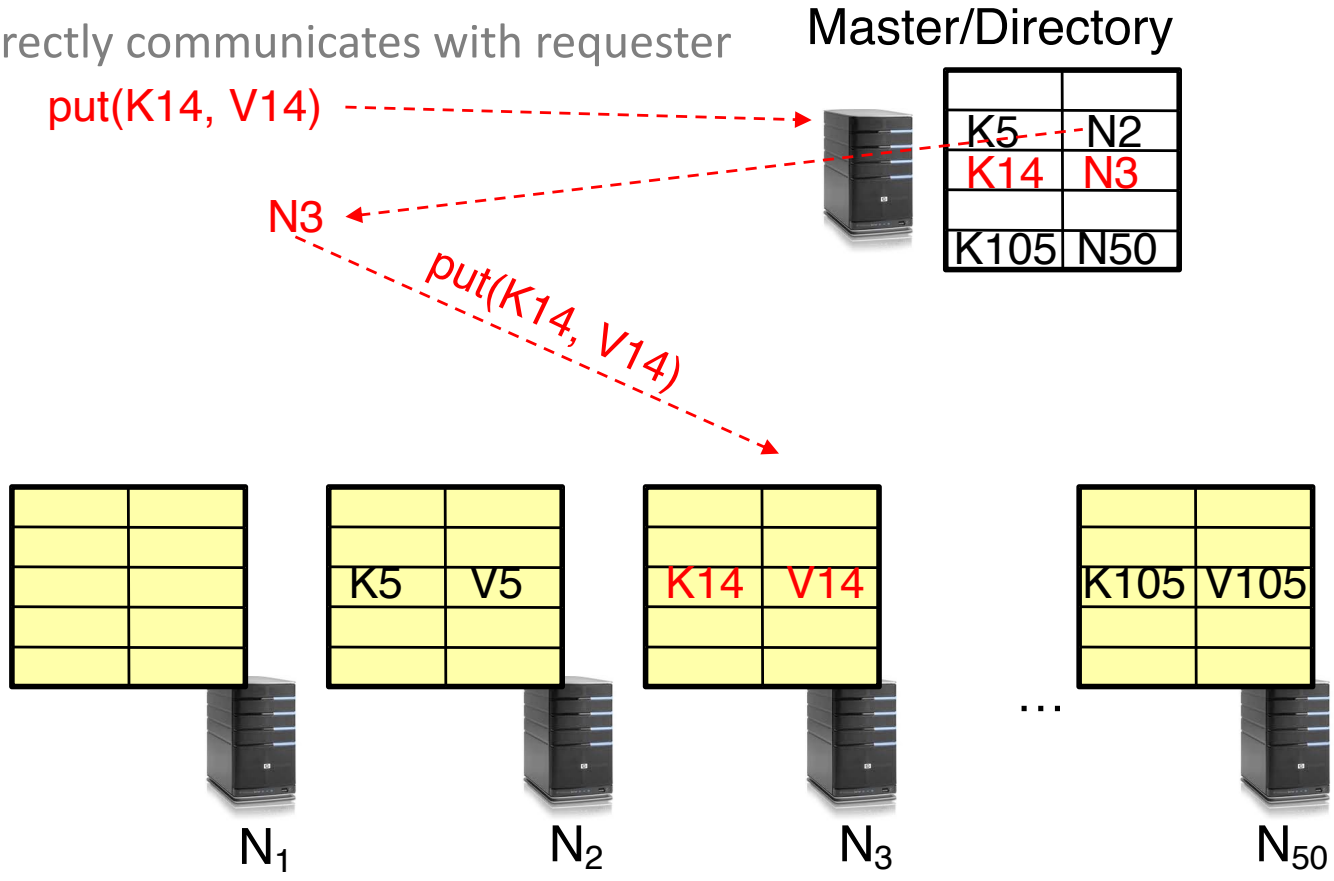
DIRECTORY-BASED ARCHITECTURE

- Designated node called the Master
  - Mapping between keys and the machines (nodes) that store the values associated with the keys



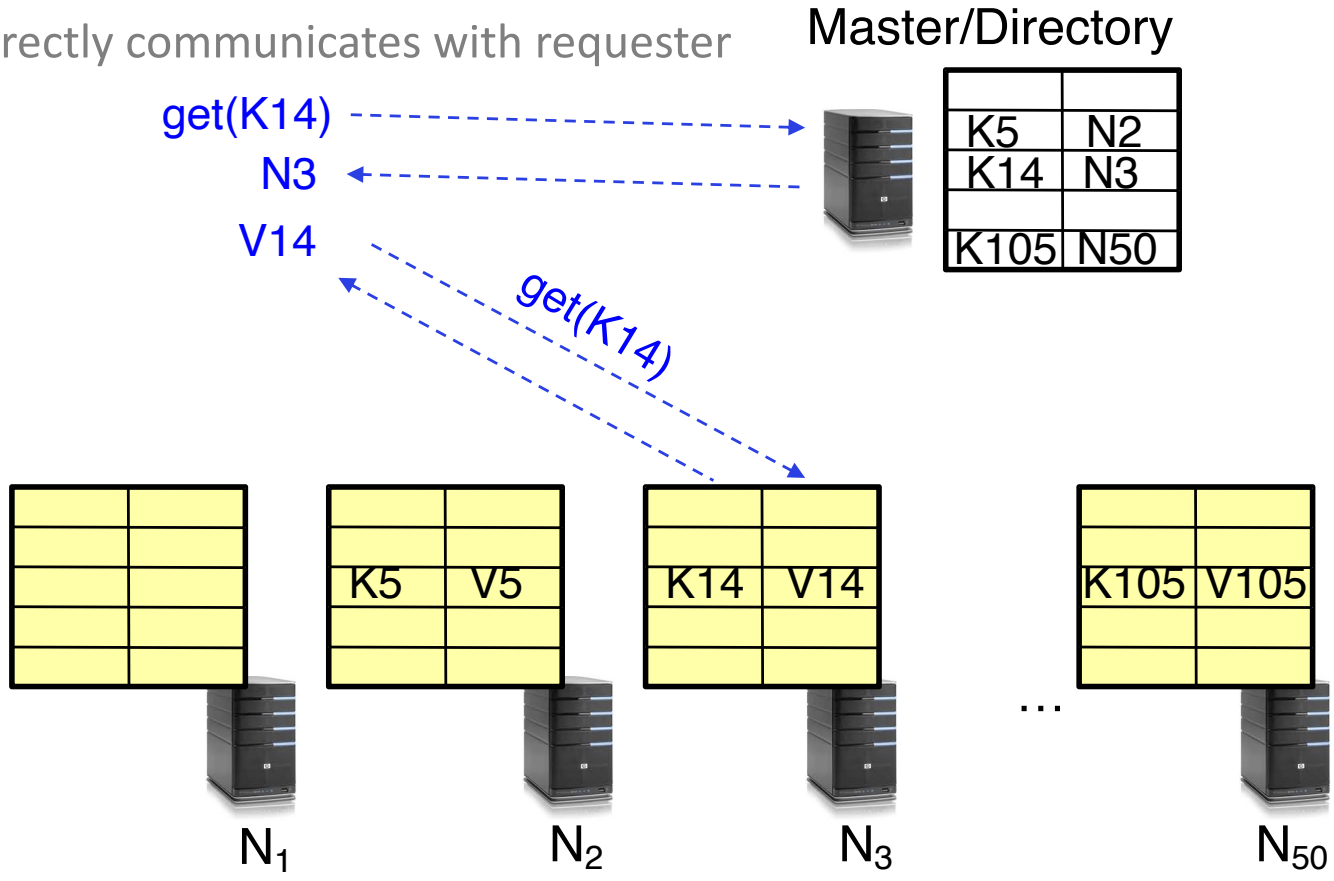
RECURSIVE VS ITERATIVE

- Recursive access (previous slides)
  - Master handles all requests
- Iterative access
  - Data node directly communicates with requester



RECURSIVE VS ITERATIVE

- Recursive access (previous slides)
  - Master handles all requests
- Iterative access
  - Data node directly communicates with requester



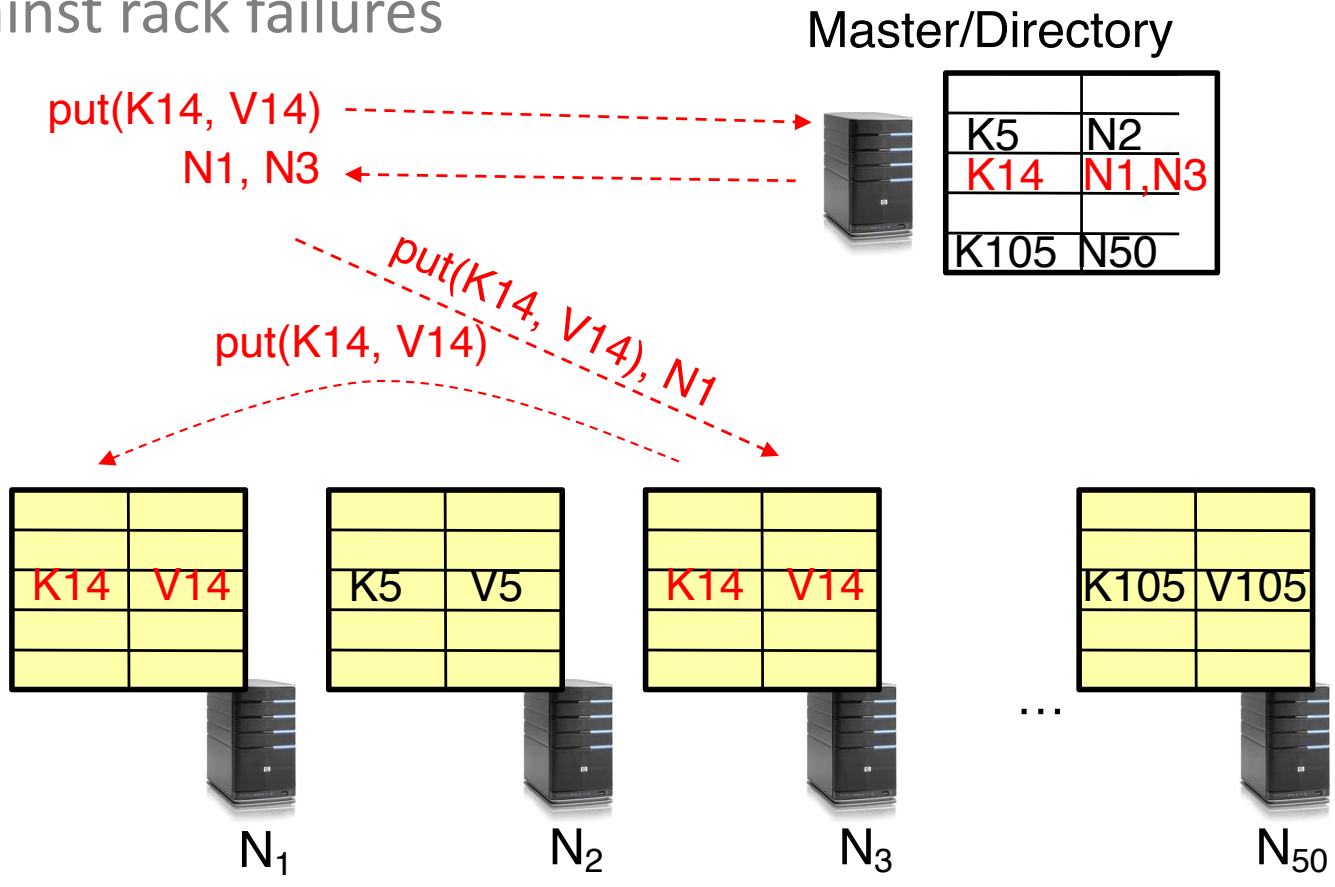
RECURSIVE VS ITERATIVE

- Recursive access
  - Pros
    - Faster, as typically master/directory closer to nodes
    - Easier to maintain consistency, as master/directory can serialize puts()/gets()
  - Cons
    - Scalability bottleneck, as all “Values” go through master/directory

- Iterative access
  - Pros
    - More scalable
  - Cons
    - Slower, harder to enforce data consistency

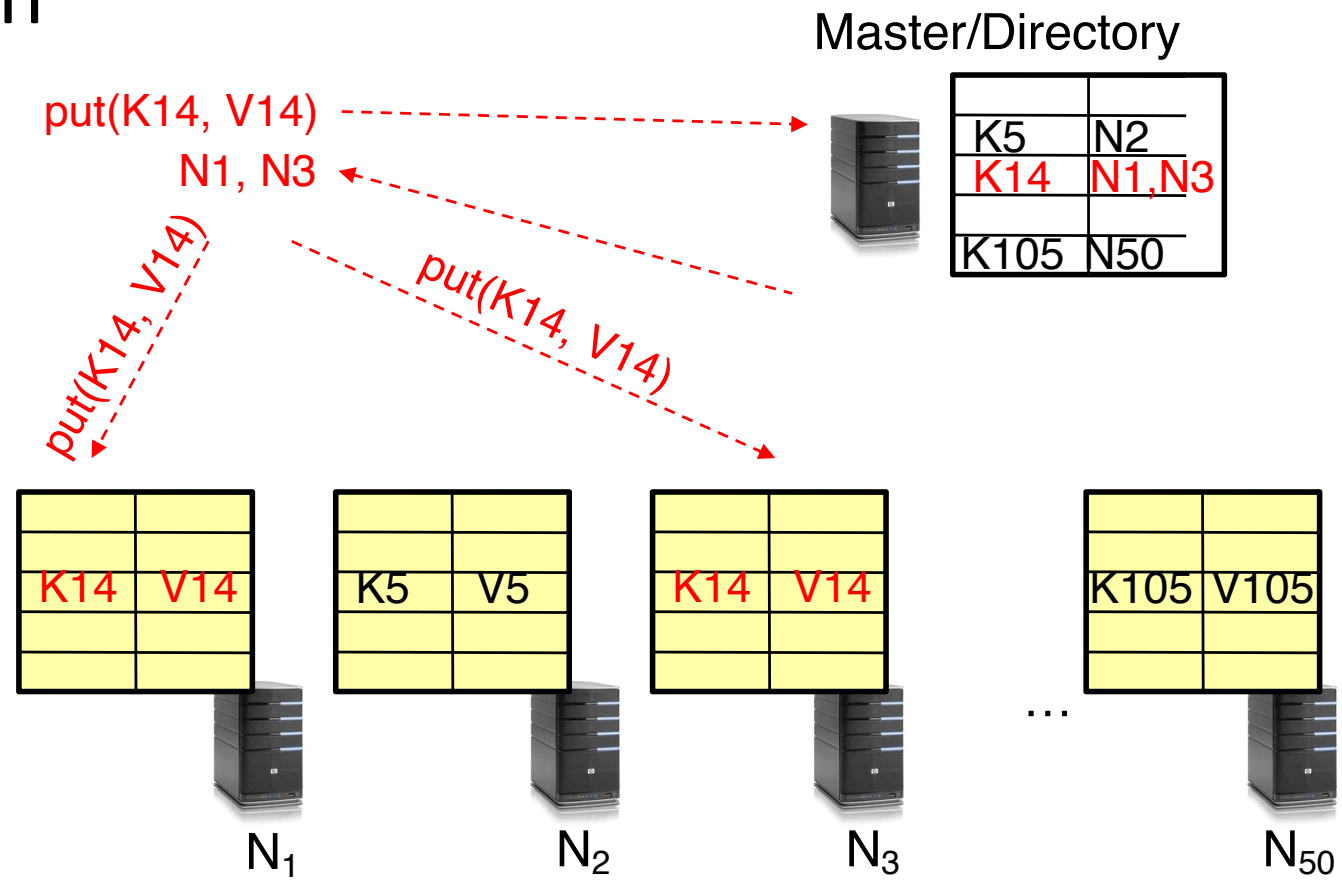
FAULT TOLERANCE

- Replicate value on several nodes
  - Usually, place replicas on different racks in a datacenter to guard against rack failures



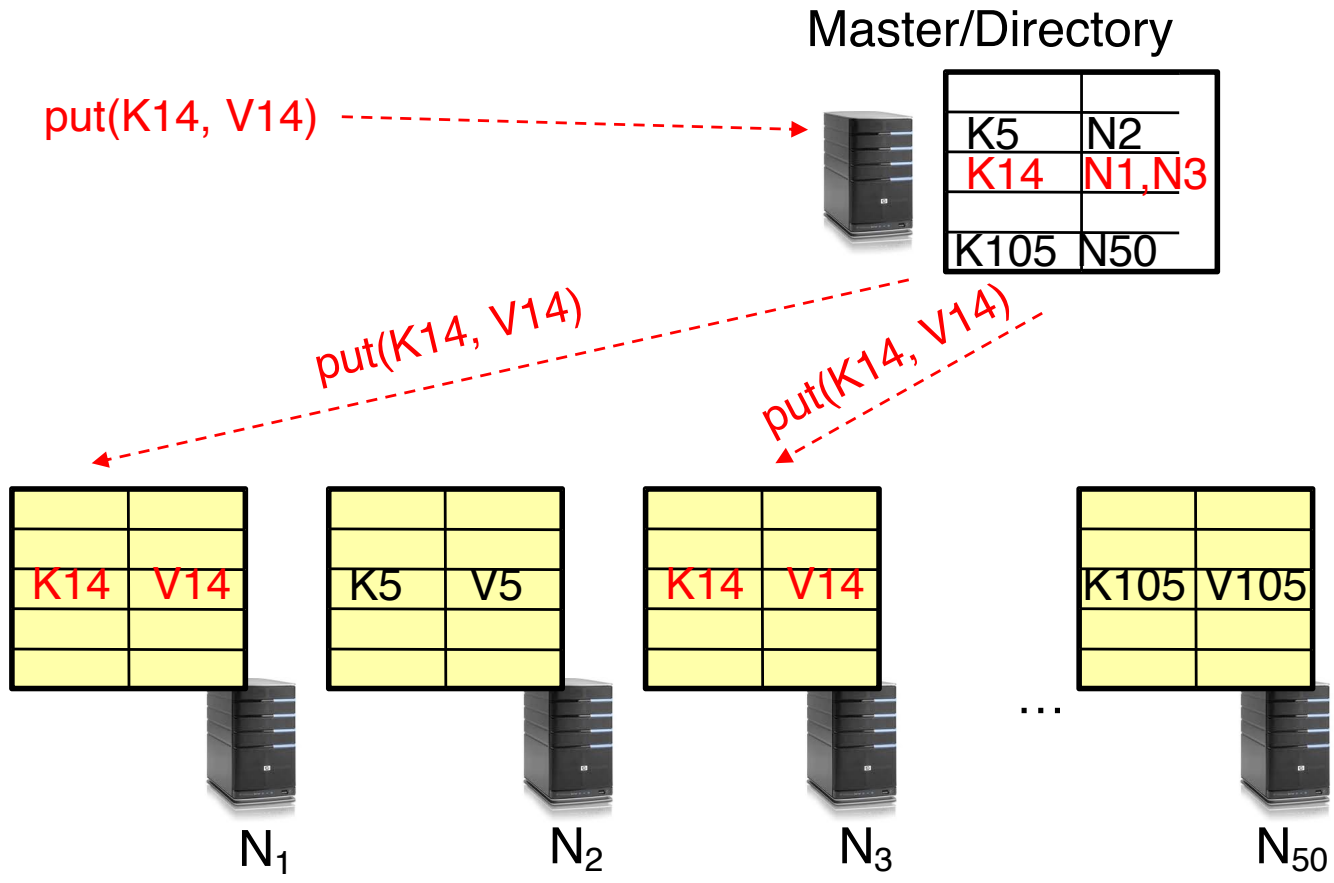
FAULT TOLERANCE

- Recursive (previous slide) Vs Iterative replication



FAULT TOLERANCE

- Recursive access and iterative replication



SCALABILITY

- Storage
  - Use more nodes
- Number of requests
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular value on more nodes
- Master/directory scalability
  - Replicate directory
  - Partition directory, so different keys are served by different masters/directories

SCALABILITY – LOAD BALANCING

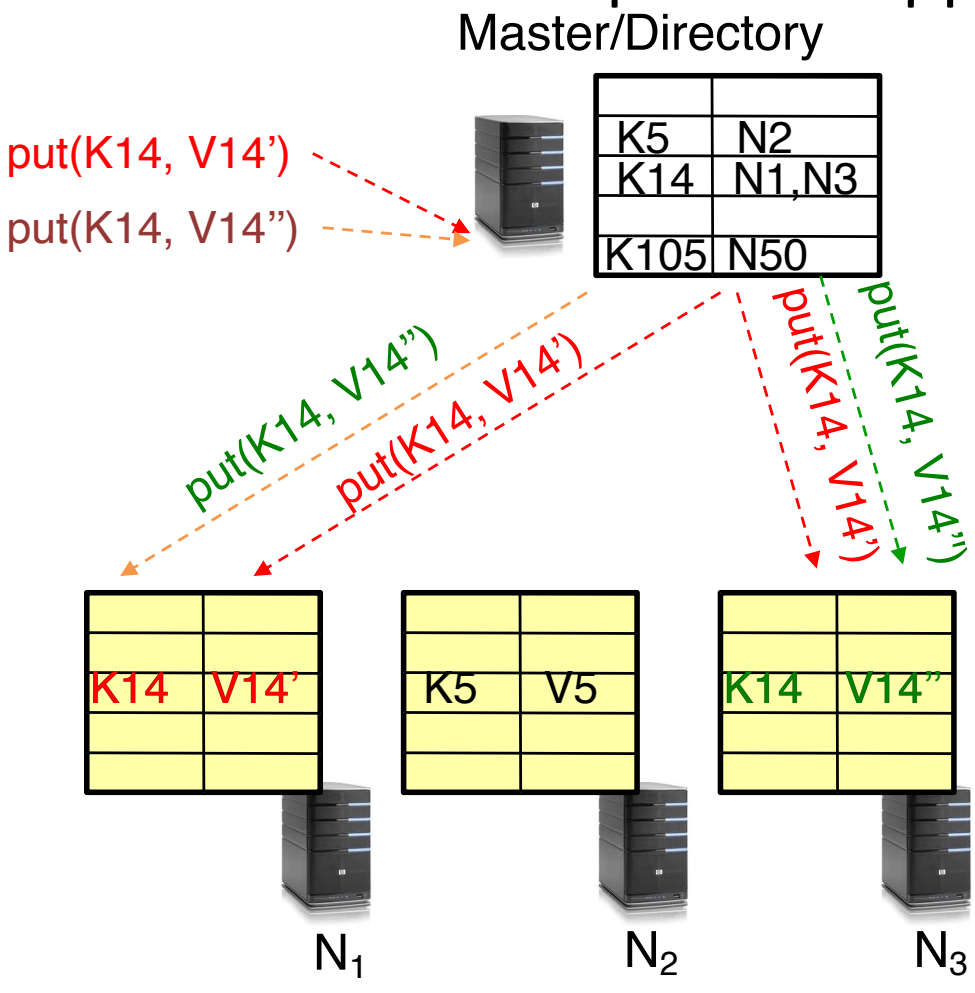
- Metadata in directory
  - Keep track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values on new node
  - Move values from the heavy loaded nodes to the new node
- What happens when a node fails?
  - Need to replicate values from fail node to other nodes

CONSISTENCY

- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  - Slow puts and fast gets

CONSISTENCY

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14") reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

CONSISTENCY MODELS

- Atomic consistency (linearizability)
  - Reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  - Think “one updated at a time”
  - Transactions
- Eventual consistency
  - Given enough time all updates will propagate through the system
  - One of the weakest form of consistency; used by many systems in practice
- And many others:
  - Causal consistency: if an operation or event A causes another operation B, then each thread observes A before B
  - Sequential consistency: writes to variables by different threads have to be seen in the same order by all threads
  - Strong consistency: all accesses are seen by all nodes in the same order (sequentially)

SCALING UP THE DIRECTORY

- Challenge
  - Directory contains a number of entries equal to number of (key, value) tuples in the system
  - Can be tens or hundreds of billions of entries in the system!
- Consistent hashing
  - Associate to each node a unique id in an uni-dimensional space $0..2^{m-1}$
  - Partition this space across m machines
  - Assume keys are in same uni-dimensional space
  - Each (key, value) is stored at the node with the smallest ID larger than key

CONSISTENT HASHING

- Suppose we use the following method to partition a set of key-value pairs over N servers. We use the function

$$hash(key) \rightarrow modulo(key, N) = i$$

  to determine that the pair $(key, value)$ is stored on server S_i.
- **Observation**: if N changes we need to repartition all the pairs.
- With Consistent hashing we solve this as follows:
  - a fixed hash function h maps both the keys and the servers IPs to a large address space A (e.g., $[0, 2^{64} - 1]$);
  - A is organized as a ring, scanned in clockwise order;
  - if server S is followed directly by S' on the ring, then all the keys in range $]h(S), h(S')]$ are mapped to S'.

ILLUSTRATION OF FINDING AN OBJECT

Example: key k_1 is mapped to server S_3; key k_2 is mapped to server S_1; etc.

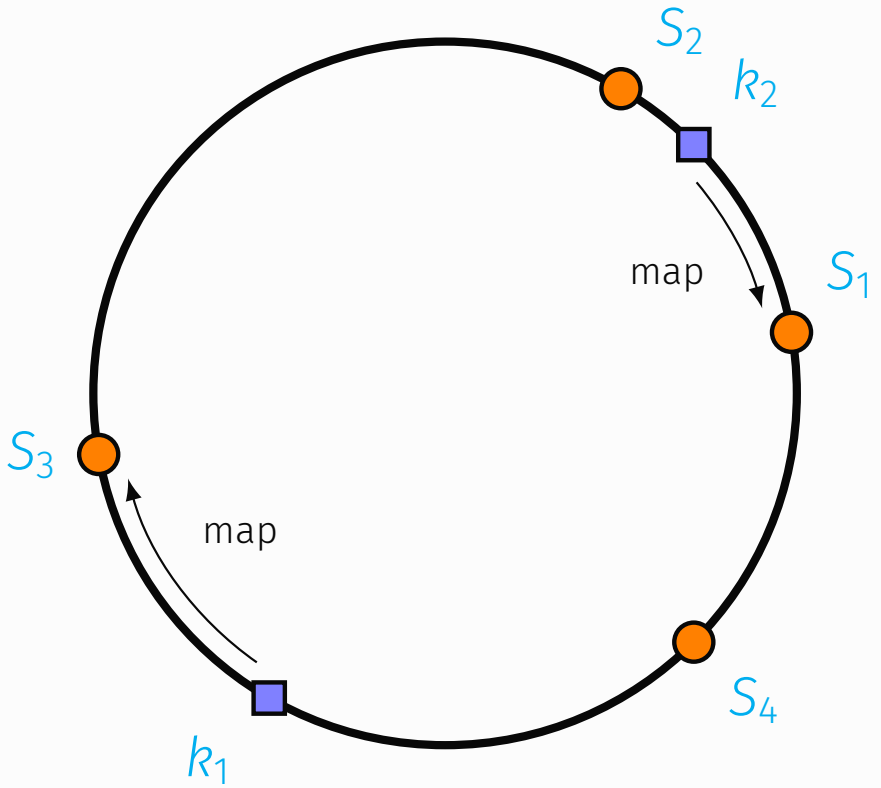
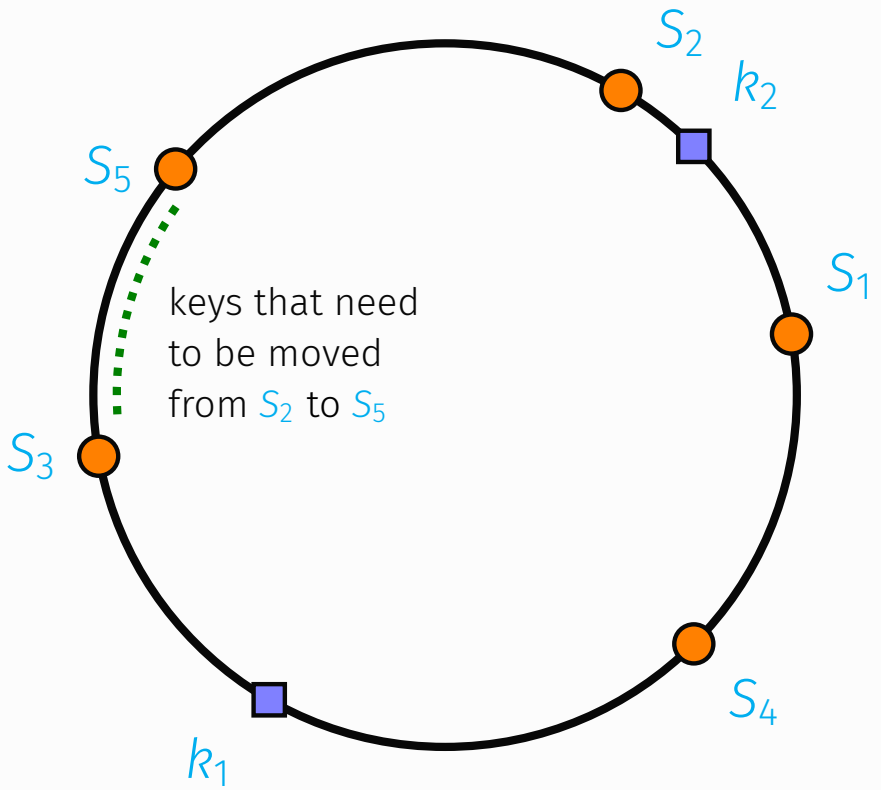


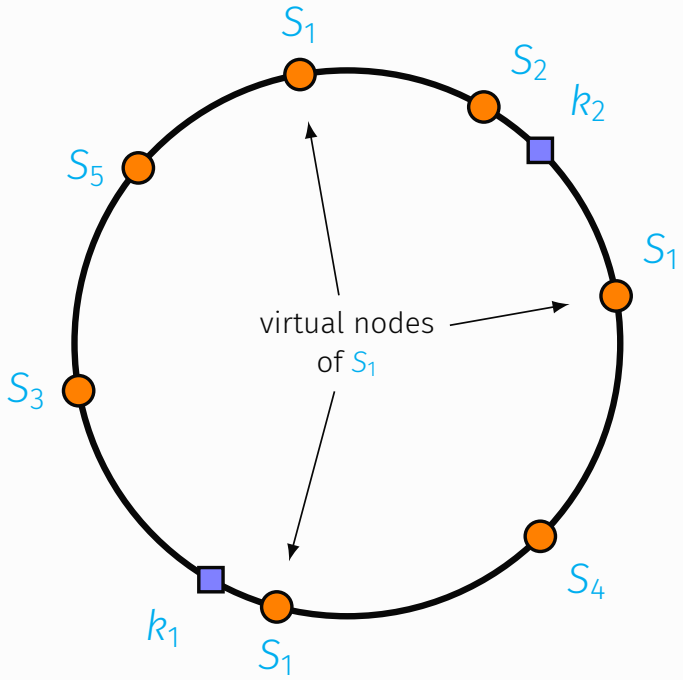
ILLUSTRATION OF ADDING A SERVER

A server is added? A local re-hashing is sufficient.



SOME REFINEMENTS

- **To deal with server failure** use replication; e.g., put a copy on the next 3 servers on the ring.
- **To balance the load** map a server to several points on the ring, called virtual nodes
  - more virtual points means more load
  - also useful if the server fails since its data will be spread over several servers
  - also useful if servers differ in capabilities, which is the rule in large-scale systems



DISTRIBUTED INDEXING BASED ON CONSISTENT HASHING

Main question: where is the hash directory with server locations?
Several possible answers:

- **On a specific ("Master") node**, acting as a load balancer. Example: caching systems. $\Rightarrow$ raises scalability issues.
- **Each node records its successor on the ring**. $\Rightarrow$ may require $O(N)$ messages for routing queries – not resilient to failures.
- **Each node records position of 1st, 2nd, 4th, 8th, 16th, .. follower in ring.** $\Rightarrow$ ensures $O(\log N)$ messages for routing queries – convenient trade-off for highly dynamic networks (e.g., P2P)
- **Full duplication of the hash directory at each node.** $\Rightarrow$ ensures 1 message for routing – heavy maintenance protocol which can be achieved through gossiping (broadcast of any event affecting the network topology).

THE HISTORY OF CONSISTENT HASHING

Paper by CS theoreticians from MIT that introduced CH

Karger, Lehman, Leighton, Panigrahy, Levine and Lewin, *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web'*, STOC 1997

- Consistent hashing gave birth to Akamai Technologies
  - Founded by Danny Lewin and Tom Leighton in 1998
  - Akamai's content delivery network is one of the largest distributed computing platforms
  - Now market cap is $12B and it has 6200 employees
  - Managing web-presence of many major companies
- **2001:** The concept of Distributed Hash Table (DHT) is proposed for locating files and CH was re-purposed
- Now used in Dynamo, Couchbase, Cassandra, Voldemort, Riak, ...

DYNAMODB

DynamoDB

amazon web services

- P2P key-value store at Amazon, ≈ 2007 Context and requirements at Amazon
- Infrastructure: tens of thousands of servers and network components located in many data centers around the world
  - Commodity hardware is used, where component failure is the "standard mode of operation"
  - Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services
  - Low latency and high throughput
  - Simple query model: unique keys, blobs, no schema, no multi-access
  - Scale out (elasticity)
- Simple API
  - get(key): returning a list of objects and a context
  - put(key, context, object): no return value
- Key and object values are not interpreted but handled as "an opaque array of bytes"

VOLDEMORT

- Key-value store initially developed for and still used at LinkedIn
  - Inspired by Amazon's Dynamo
- Features
  - Written in Java
  - Simple data model and only simple and efficient queries
    - no joins or complex queries, no constraints on foreign keys etc.
  - Performance of queries can be predicted well
  - P2P
  - Scale-out / elastic
  - Consistent hashing of keyspace
  - Eventual consistency / high availability
  - Pluggable storage, e.g., BerkeleyDB, In Memory, MySQL
- API
  - get(key): returning a value object
  - put(key, value): writing an object/value
  - delete(key): deleting an object
  - Keys and values can be complex, compound objects as well consisting of lists and maps

NOSQL REBUTTAL

WHY JOINS MAY OR MAY NOT MATTER

- Why do we not need support for joins?
  - We can precompute many joins by nesting the data, e.g., clustering all comments on a blog post with that blog post.
- Why do we need support for joins?
  - Dictates access-path, and makes other access hard and/or inefficient.
  - Asks programmer to pick join algorithm, where this might change in time.
  - E.g., “how to find all comments by Sue on blog posts by Jim”
    1. Find al comments by Sue, and for each retrieve the blog post and check which are by Jim.
    2. Find al blog posts by Jim, and for each retrieve the comments and check which are by Sue.
    3. Select the blog posts by Jim. Select the comments by Sue. Sort each on blog-id, and merge them while selecting the matches. (Aka. sort-merge join)
  - Which is best?

NOSQL CRITICISM

Analysis by Michael Stonebraker

Michael Stonebraker. 2011. Stonebraker on NoSQL and enterprises. Commun. ACM 54, 8 (August 2011), 10-11. DOI: https://doi.org/10.1145/1978542.1978546

Two value-propositions for using NoSQL

- Performance: *"I started with MySQL, but had a hard time scaling it out in a distributed environment"*
- Flexibility: *"My data doesn't conform to a rigid schema"*

NoSQL Criticism: Flexibility Argument

- Who are the customers of NoSQL?
  - Lots of startups
- Why very few enterprises?
  - Most applications are traditional OLTP on structured data; a few other application at the fringes, but considered less important.

NoSQL Criticism: Flexibility Argument

- No ACID equals No Interest
  - Screwing up mission-critical data is a no-no-no
- Low-level Query Language is Death
  - Remember CODASYL?
- NoSQL means NoStandards
  - One (typical) large enterprise has 10,000 databases. These need accepted standards.

Questions?

ACKNOWLEDGEMENTS

Based on content created by Stijn Vansummeren, Nikos Mamoulis, Johann Gamper, Panagiotis Bouros, and Unmesh Joshi.