CS-300: Data-Intensive Systems

Concurrency Control (Part II)

(Chapters 18)

Prof. Anastasia Ailamaki, ***Prof. Sanidhya Kashyap***

EPFL

Today's focus

- **Phantom and predicate locking for dynamic database**

  Readings: Chapter 17.10 and 18.4

- Eventual consistency

  Readings: Chapter 21.4 and 23.6

Today's world: Dynamic databases

- Until now, we only consider the scenario in which transactions perform read and update operations on existing objects in the databases

- However, data keeps evolving all the time
- That is transactions perform insertions, updates, and deletions of tuples ….

- This is a critical problem as data may change while transactions are executing

The Phantom problem with an example

- Let's consider that a DB is not fixed size, i.e., data is growing
- Then ensuring serializability is not possible with **Strict2PL** (on individual items)
- Consider T_1: "Find oldest sailor"
  - T_1 locks all records and finds **oldest** sailor (age = 71)
  - Next, T_2 inserts a new sailor, age = 96, and commits
  - T_1 (within the same txn) checks for oldest sailor again and finds sailor aged 96!
- The sailor with age 96 is a "**phantom tuple**" from T_1's point of view
  - First, it was not present and now it is there
- No serial execution exists where T_1's result can be produced

The Phantom problem with another example

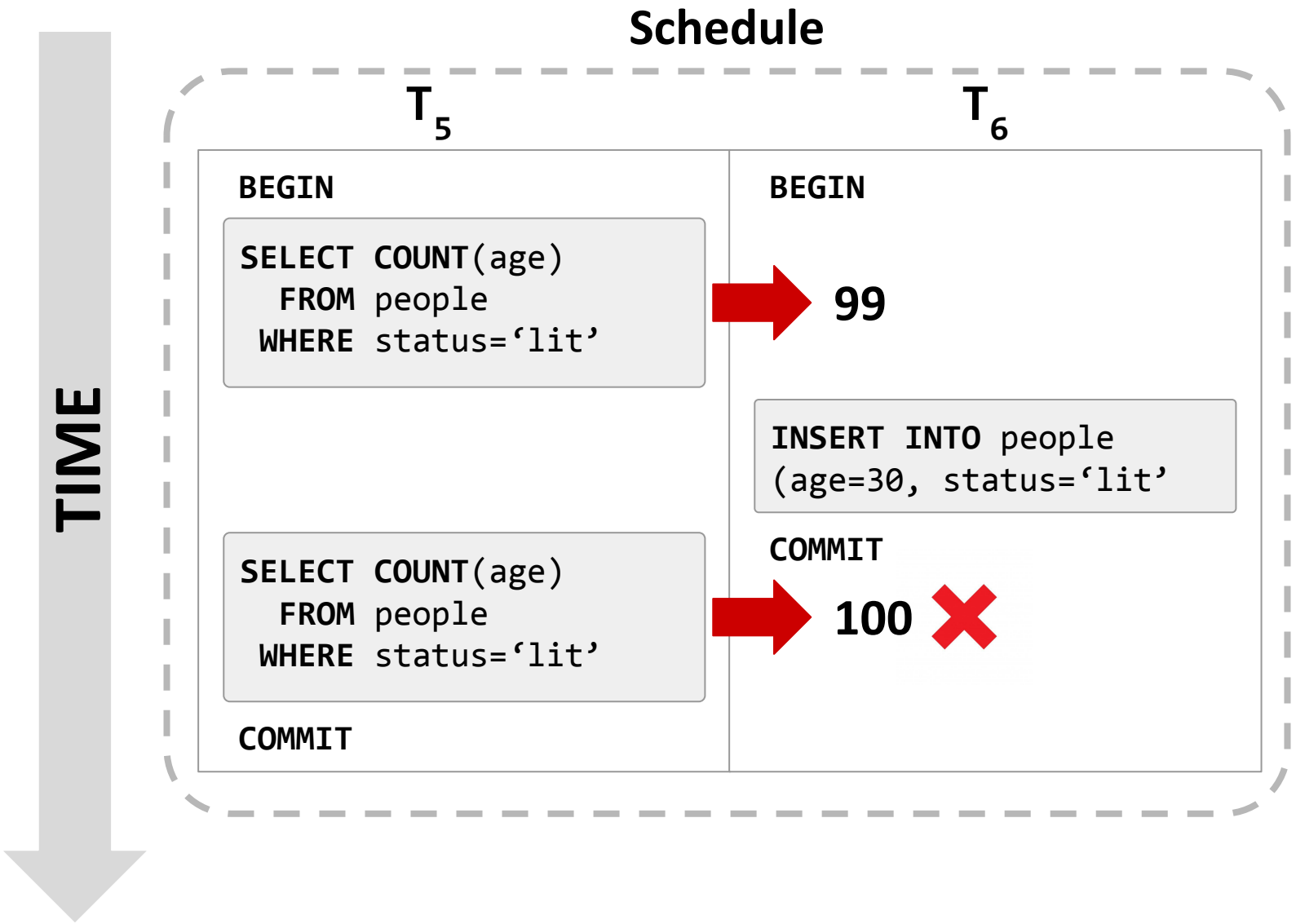
- Consider T_3: “Find the oldest sailor for each rating”
  - T_3 locks all pages containing sailors records with rating=1, and find oldest sailor (age=71)
  - Next, T_4 inserts a new sailor, rating=1 and age=96
  - T_4 also deletes oldest sailor with rating=2 and age=80 and commits
  - T_3 now locks all pages containing sailor records with rating=2 and finds oldest (age=63)
- T_3 only saw part of T_4’s effects
- No serial execution where T_3 execution could happen

Why this problem occurs?

- T_1 and T_3 implicitly assumes that they had locked the set of all sailor records satisfying a predicate
  - T_1 and T_3 locked only **existing records** and not the ones underway
  - Assumption only holds if no sailor records are added while they were executed

- Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the **set of objects is fixed**

The Phantom problem with visualization




```
CREATE TABLE people (
  id SERIAL,
  name VARCHAR,
  age INT,
  status VARCHAR
);
```

Approach to handling the phantom problem

Approach #1: Re-execute scans

- Run queries again at commit to see whether they produce a different result to identify missed changes

Approach #2: Predicate locking

- Logically determine the overlap of predicates before queries start running

Approach #3: Index locking

- Use keys in indexes to protect ranges

Re-execute scans

- The DBMS tracks the **WHERE** clause for all queries that the transaction executes
  - Retain the scan set for every range query in a transaction

- Upon commit, re-execute just the scan portion of each query and check whether it generates the same result
  - Example: Run the scan for an **UPDATE** query but do not modify matching tuples

Predicate locking

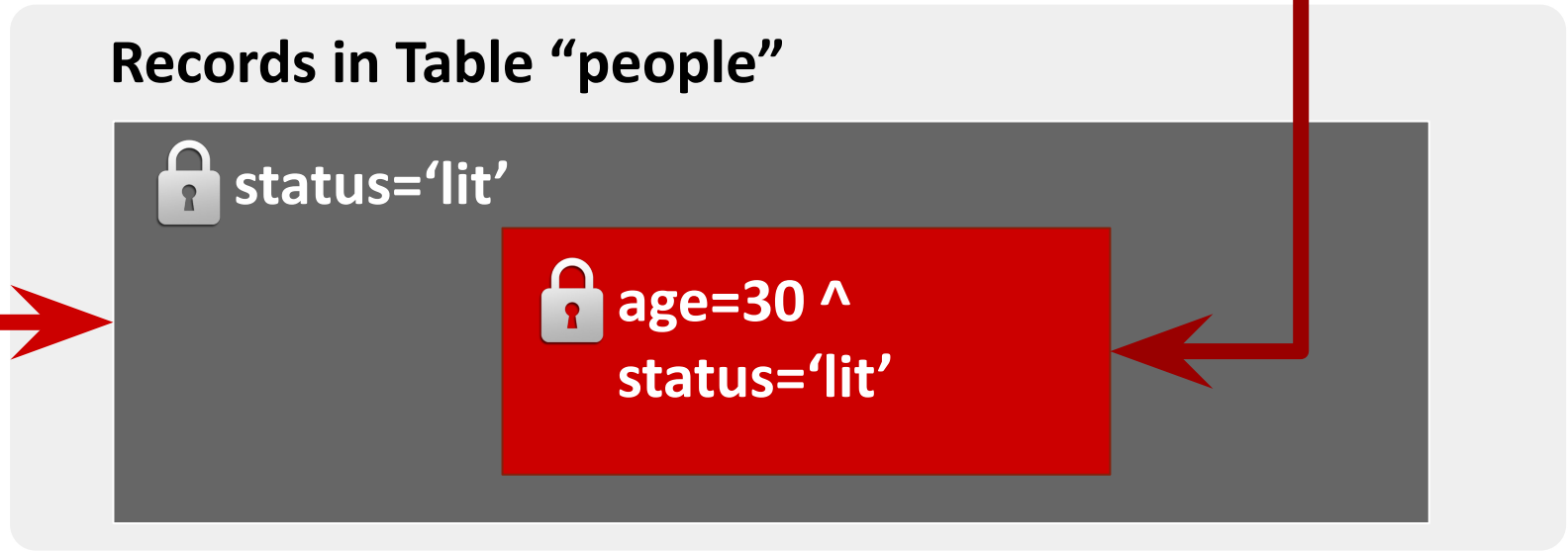
- Proposed locking scheme in System R
- Grants lock on all records that satisfy some logical predicates
  - Shared lock on the predicate **WHERE** clause of a **SELECT** query
  - Exclusive lock on the predicate in a **WHERE** clause of any **UPDATE**, **INSERT**, and **DELETE** query

- Rarely used in today's systems

Predicate locking

```
SELECT COUNT(age)
  FROM people
WHERE status='lit'
```

```
INSERT INTO people VALUES
(age=30, status='lit')
```

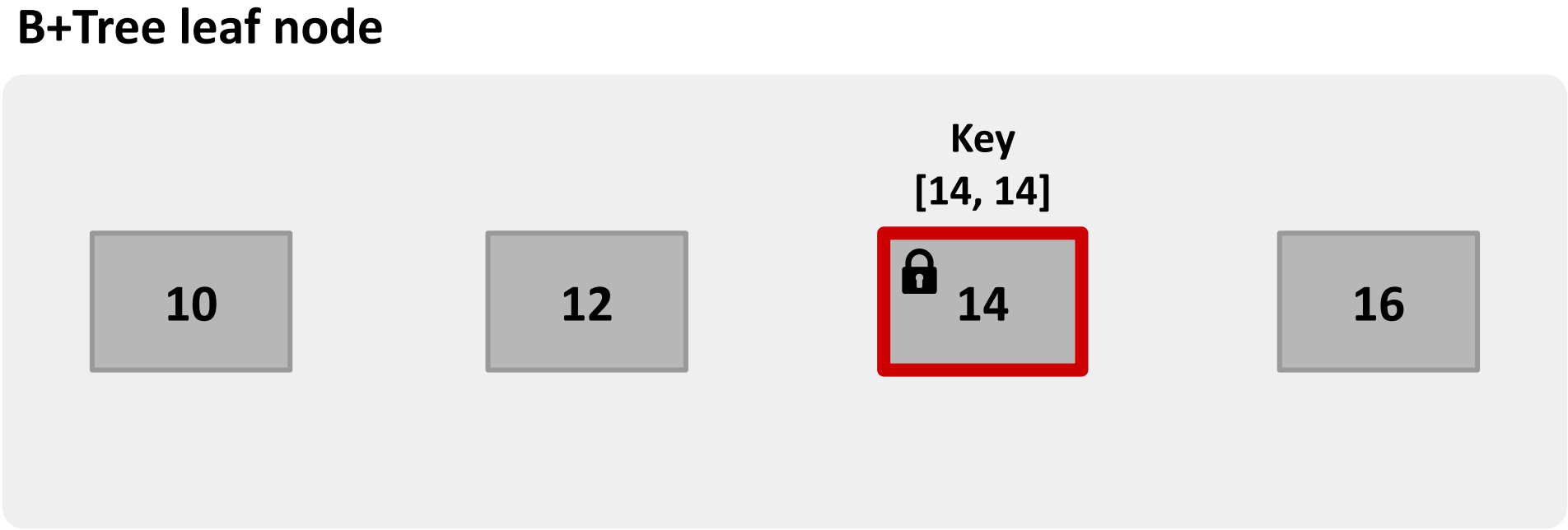


Index locking

- Use keys in indexes to protect ranges
- Types of schemes:
  - Key-value locks
  - Gap locks
  - Key-range locks
  - Hierarchical locks

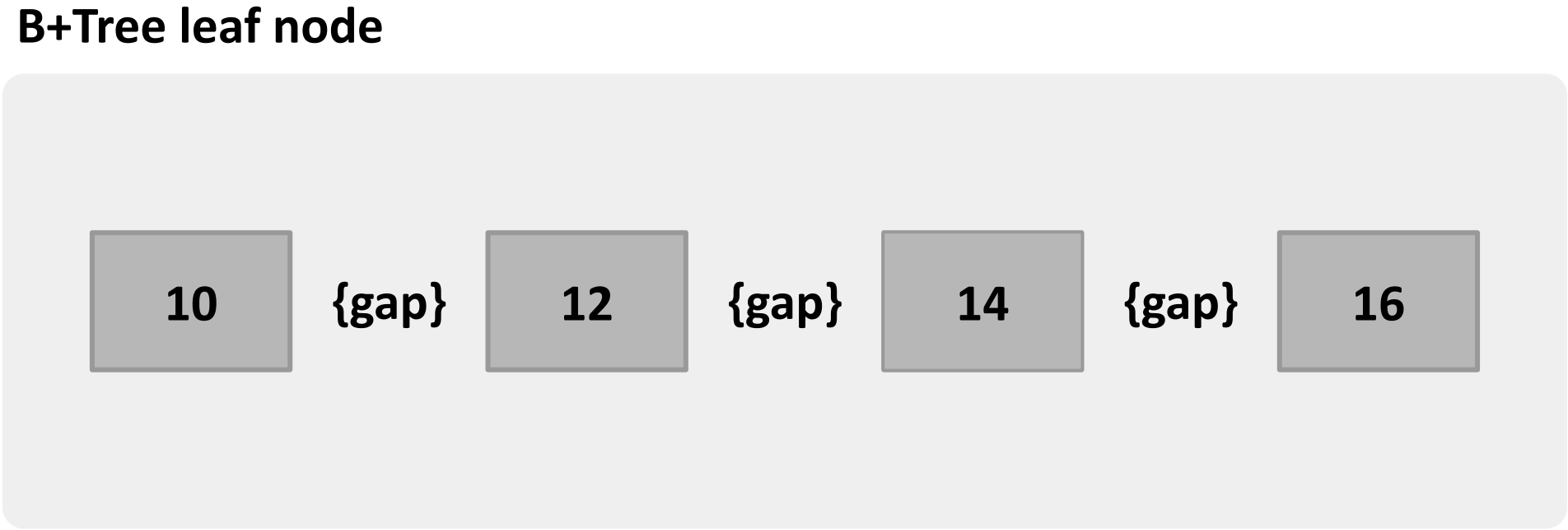
Key value locks

- Locks that cover a single key-value in an index
- Need “virtual keys” for non-existent values



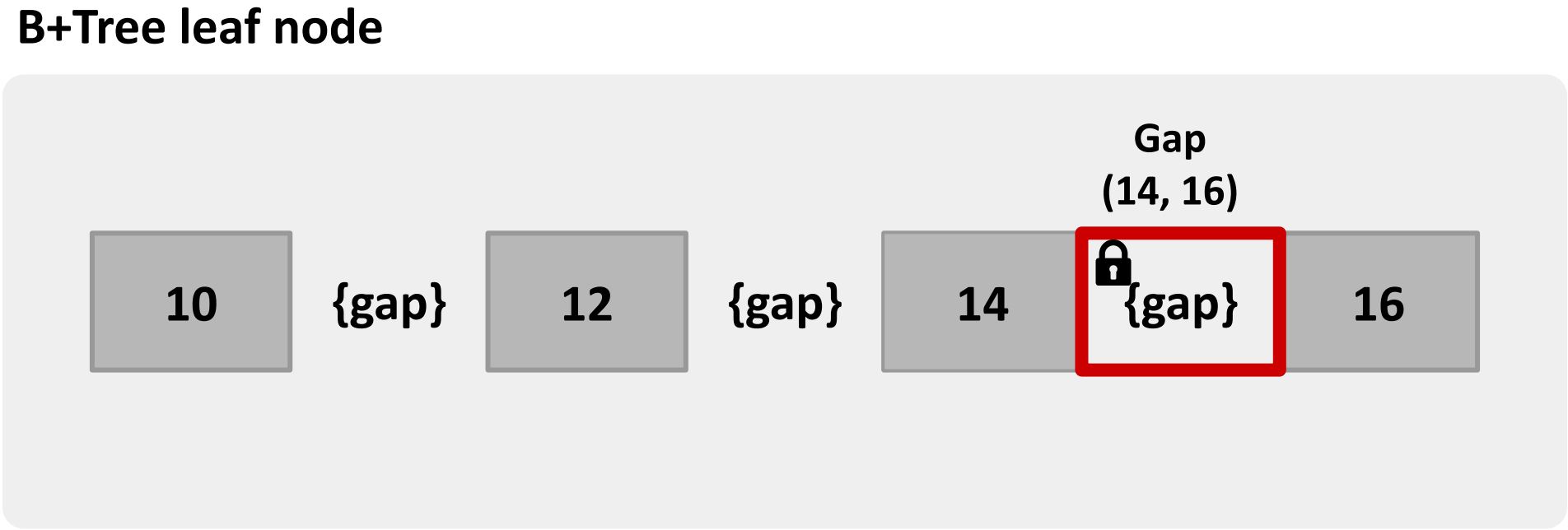
Gap locks

- Each transaction acquires a key-value lock on the single key that it wants to access
- Also gets a gap lock on the next key gap



Gap locks

- Each transaction acquires a key-value lock on the single key that it wants to access
- Also gets a gap lock on the next key gap

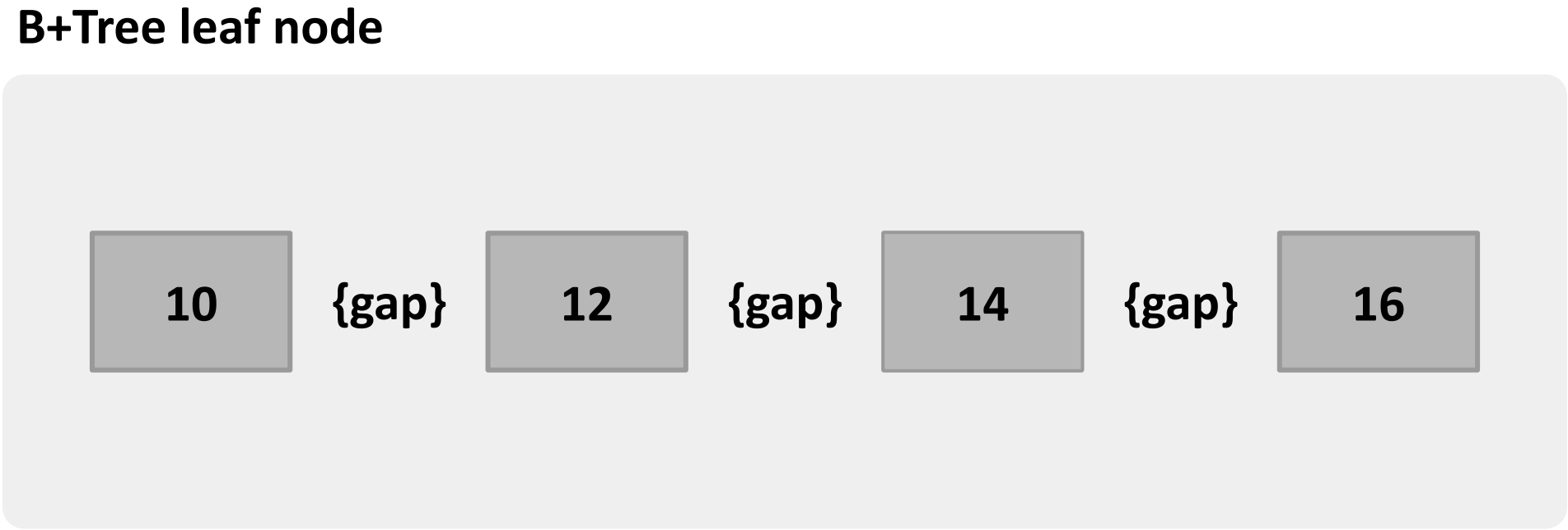


Key-range locks

- A transaction takes locks on ranges in the key space
  - Each range is from one key that appears in the relation, to the next that appears
  - Define lock modes so conflict table will capture commutativity of the operations available

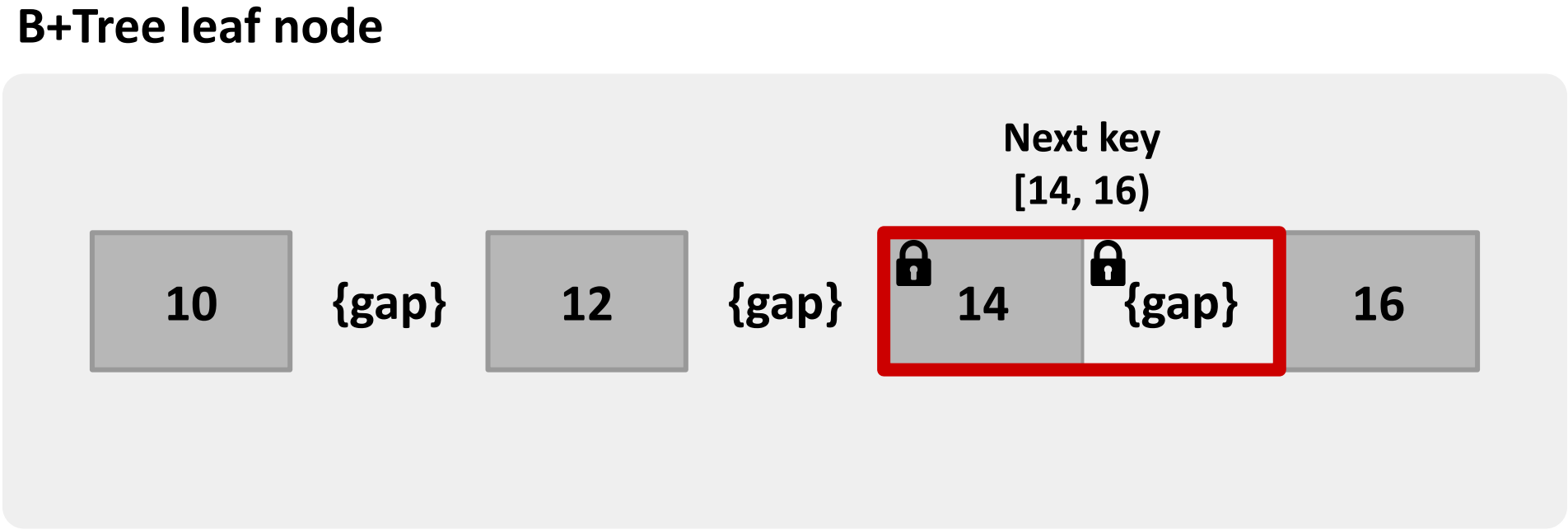
Key-range locks

- Locks that cover a key value and the gap to the next key value in a single index
  - Need “virtual keys” for artificial values (infinity)



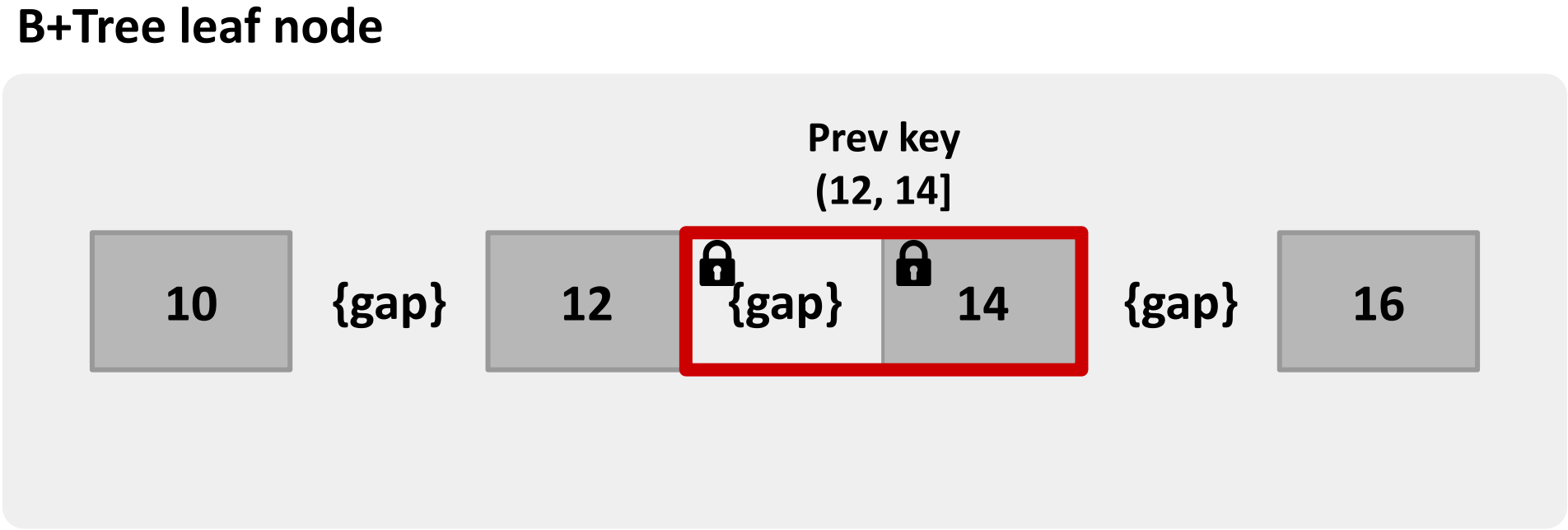
Key-range locks

- Locks that cover a key value and the gap to the next key value in a single index
  - Need “virtual keys” for artificial values (infinity)



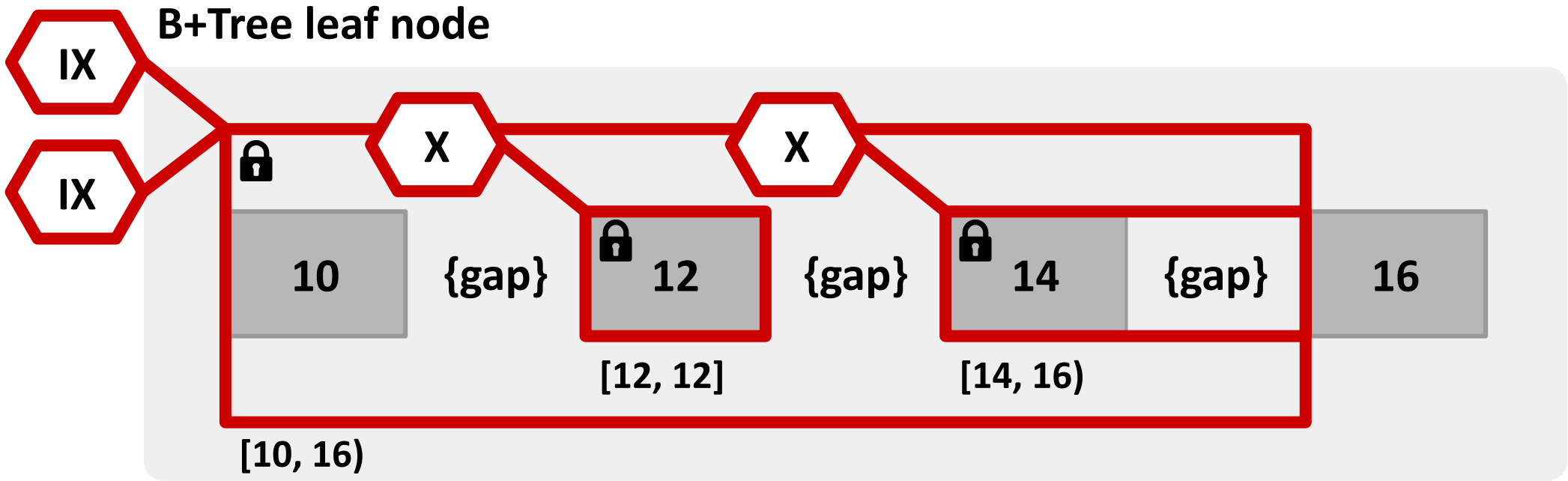
Key-range locks

- Locks that cover a key value and the gap to the next key value in a single index
  - Need “virtual keys” for artificial values (infinity)



Hierarchical locks

- Allow for a transaction to hold wider key-range locks with different locking modes
  - Reduces the number of visits to the lock manager



Locking without an index

- If there is no suitable index, then a transaction must obtain the following to avoid phantoms:
  - A lock on every page in the table to prevent a record's **`status='lit'`** from being changed to **`lit`**
  - The lock for the table itself to prevent records with **`status='lit'`** from being added or deleted

Weaker levels of isolation

- Serializability is useful because it allows programmers to ignore concurrency issues

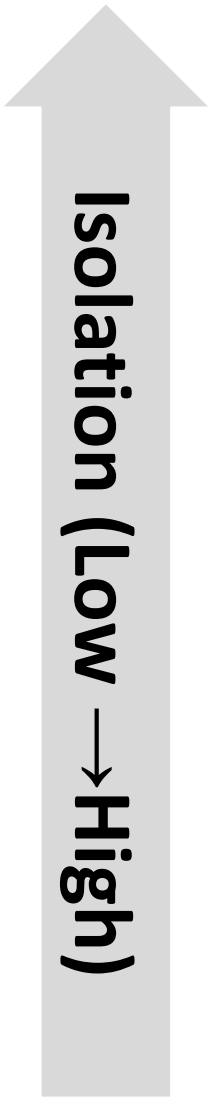
- But enforcing it may allow too little concurrency and limit performance
  - Ex: Suppose a DB wants to read all data to build statistics for the optimizer (such as building a histogram)
    - You do not want to hold S lock on that to do that.

- We may want to use a weaker level of consistency to improve scalability

Isolation levels

- Controls the extent that a transaction is exposed to the actions of other concurrent transactions
- Provides for greater concurrency at the expense of exposing transactions to uncommitted changes
  - **Dirty reads**: A transaction reads data that has not been committed yet
  - **Unrepeatable reads**: A transaction reads the same row twice and gets different data
  - **Phantom reads**: A row that matches a search criteria that was not initially seen

Isolation levels



SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads

REPEATABLE READS: Phantoms may happen

READ COMMITTED: Phantoms and repeatable reads may happen

READ UNCOMMITTED: All of them may happen

Isolation levels

SERIALIZABLE: Obtain all locks first; plus index locks, plus strong strict 2PL

REPEATABLE READS: Same as above, but no index locks

READ COMMITTED: Same as above, but **S** locks are released immediately

READ UNCOMMITTED: Same as above but allow dirty reads (no **S** locks)

More on isolation levels

- Transaction levels are set before executing a transaction

- Not all DBMS support all isolation levels in all execution scenarios
  - Replicated environment
- The default depends on the implementation
- Lot of well-known databases support **READ COMMITTED** isolation level
- Current description of isolation levels is based on phenomena (**DIRTY READS**, **NON-REPEATABLE READS**, and **PHANTOMS**)

Today's focus

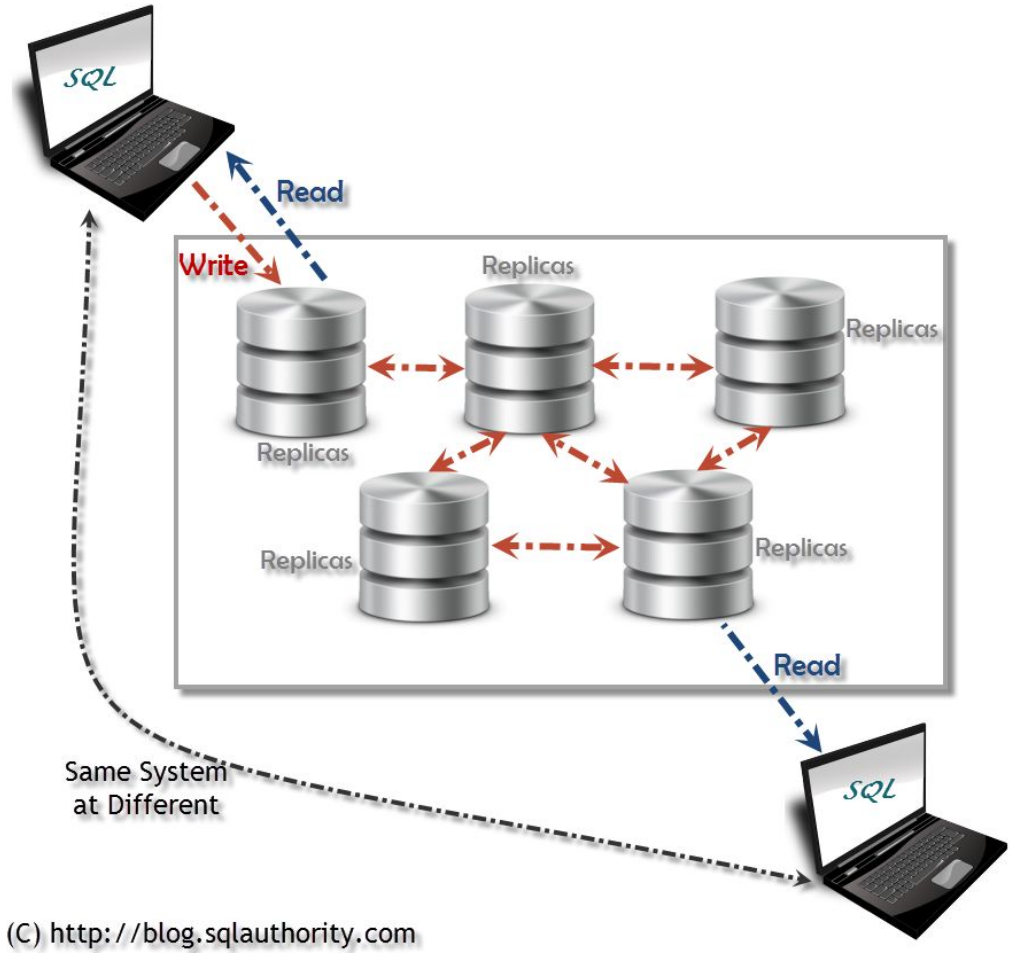
- **Phantom and predicate locking for dynamic database**

  Readings: Chapter 17.10 and 18.4

- Eventual consistency

  Readings: Chapter 21.4 and 23.6

Eventual consistency



The problem: server failures

- What to do when a server crashes?
  - Servers continuously fail in large-scale systems
  - Wait for recovery can be too long …
- Use replication
  - Keep several copies of data, i.e., ***replica***, in other servers
  - If a server fails, another server takes over
  - Also applicable in the case of load balancing
- Until now: **strong consistency**
  - All copies have to have the same value

CAP theorem

- Consistency: Possible answers to read requests: newest value or error
- Availability: Possible answers to read requests: a value, not necessarily the newest
- Partition tolerance: System still works if some of the nodes are unreachable
- **CAP theorem**: Can't have all three at once in a distributed key-value store
  - In a real network that can fail, we can't have both consistency and availability
- Can have a combination of two of C, A, and P
  - Choose C+P: strongly consistent systems (e.g., ACID DBMS)
  - Choose A+P: eventually consistent systems (e.g., NOSQL systems, DynamoDB)
  - Choose C+A: single node systems

Consistency vs latency (PACELC) theorem

- Latency is similar to a partition
- Follows from the CAP theorem if you consider every network partition temporary
  - A request may block if a node cannot be reached
- Even in case of no partitions, contacting other machines for consensus takes time, like a network partition
- Consistency-latency tradeoff

Consensus does not scale

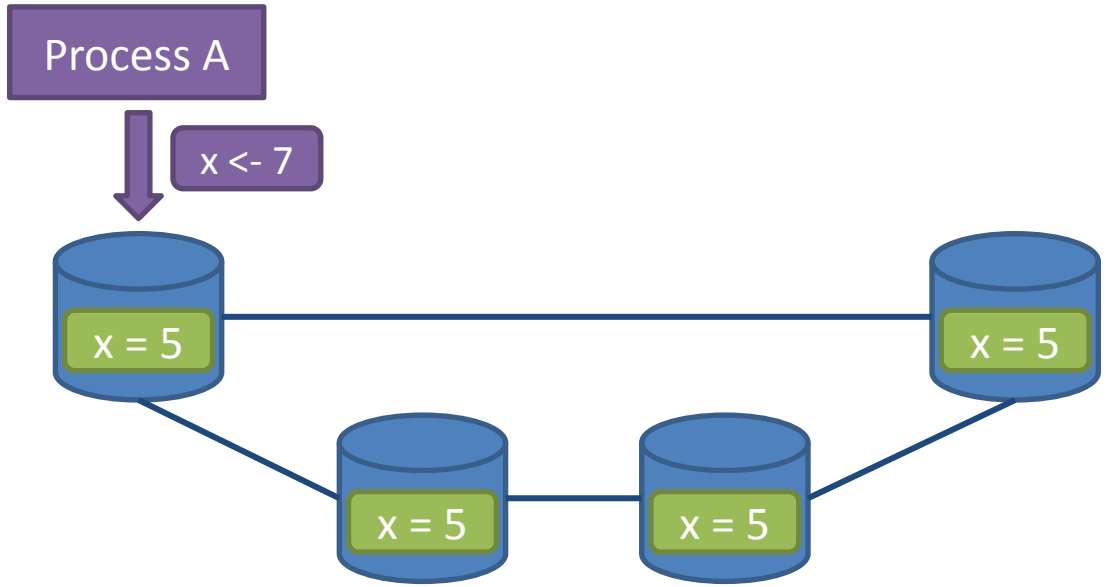
- No consensus without at least a communication of round-trip to each node
- Round-trip latency can’t be zero
- Transaction throughput upper-bounded by 1/latency
- Example: If consensus takes 1 ms, we cannot do more than 1000 X actions/sec
- Idea: Some applications are fine with weaker consistency models

Eventual consistency

- **Updates and replication**
- Eventual consistency protocol
- Dynamo

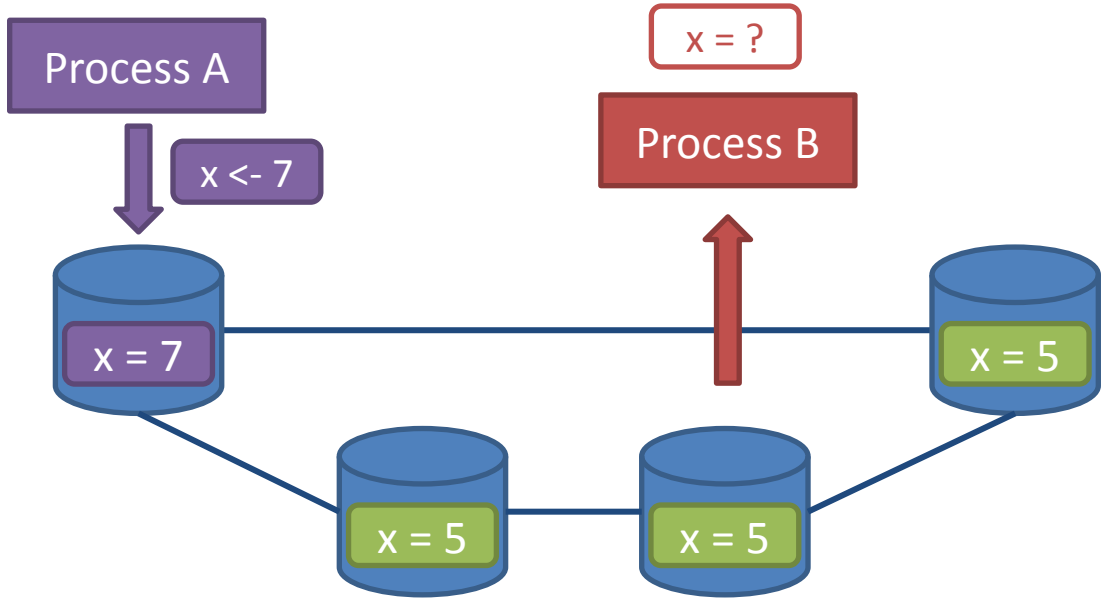
How to deal with updates?

- How to keep different copies of the data consistent?



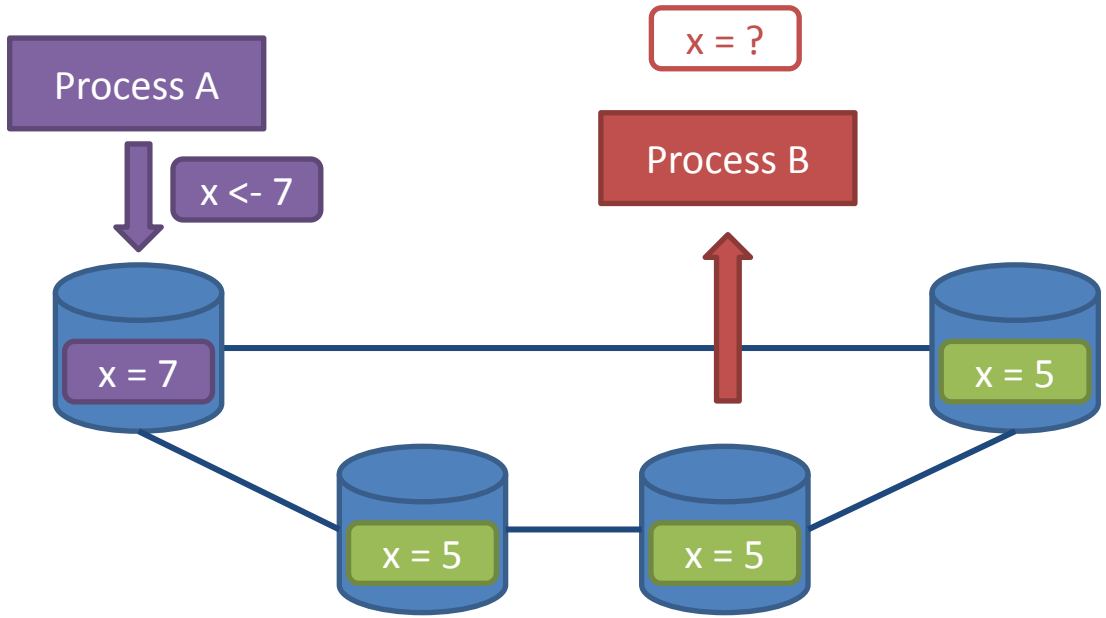
How to deal with updates?

- How to keep different copies of the data consistent?



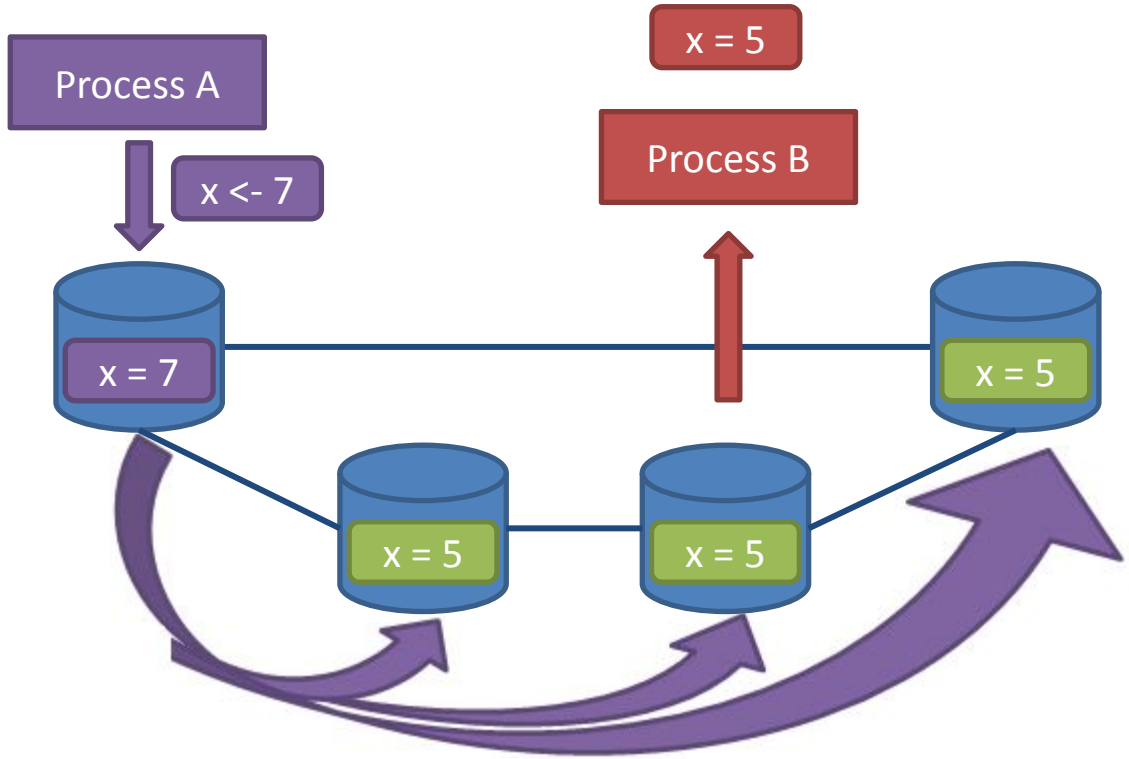
Client view

- Definition: if there are no further changes, eventually all the copies will have the same value



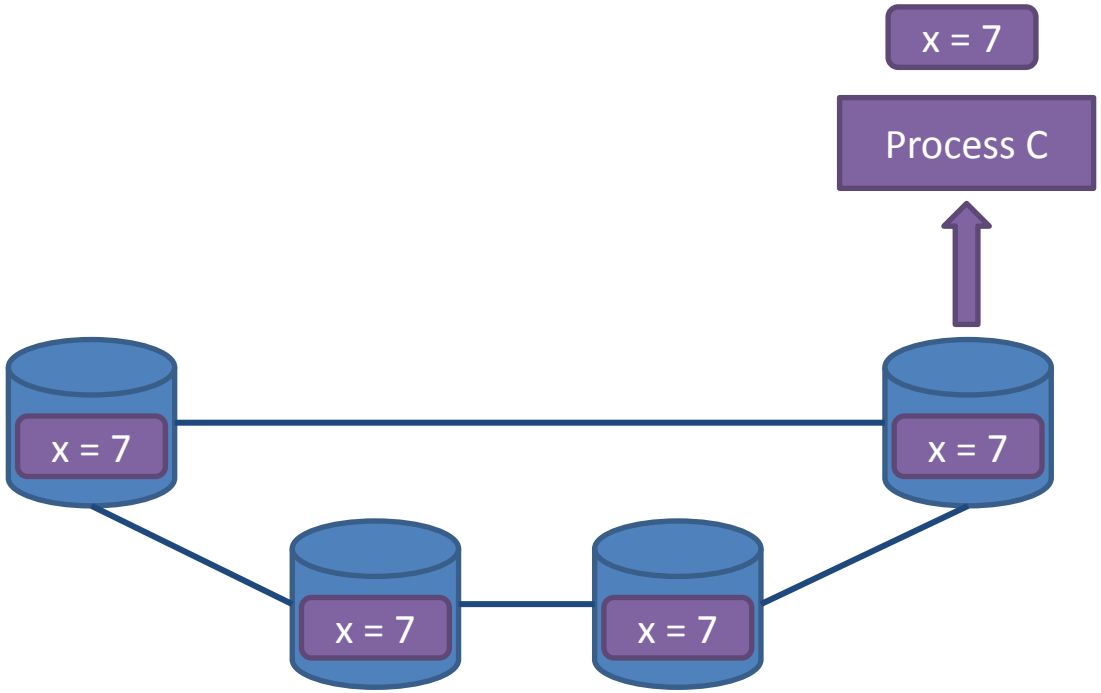
Client view

- Definition: if there are no further changes, eventually all the copies will have the same value



Client view

- Definition: if there are no further changes, eventually all the copies will have the same value



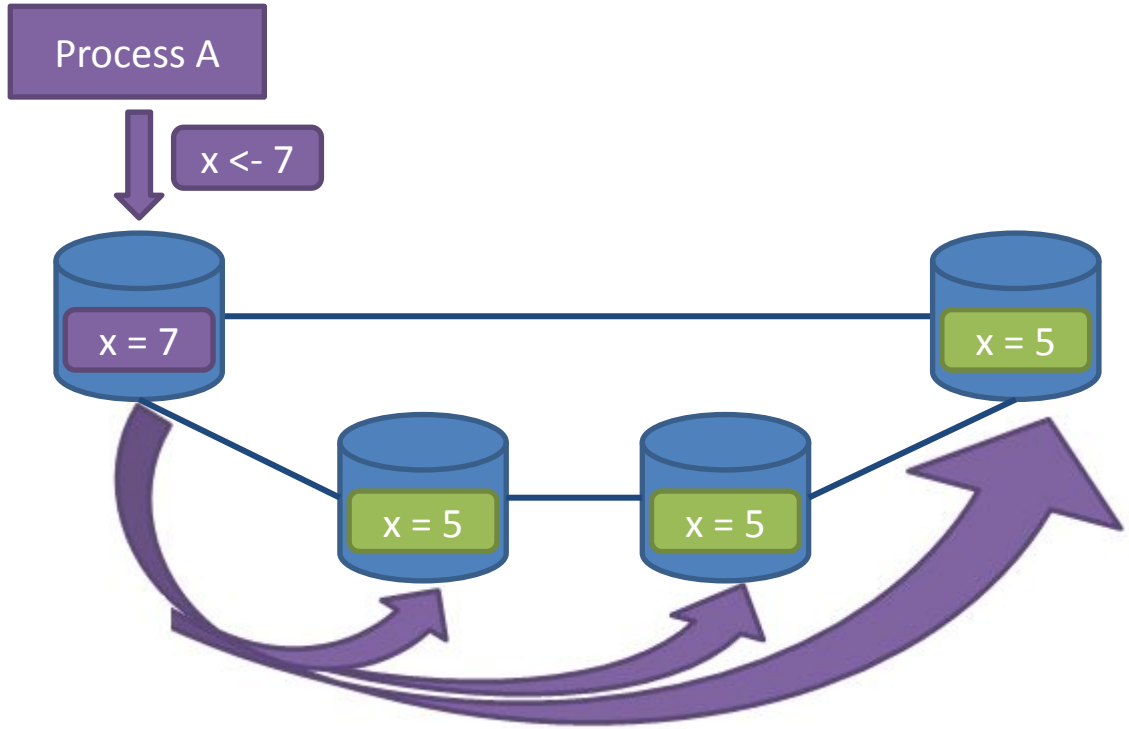
Data is *eventually* consistent

Eventual consistency

- Updates and replication
- **Eventual consistency protocol**
- Dynamo

Server view

- How do updates flow through the system?



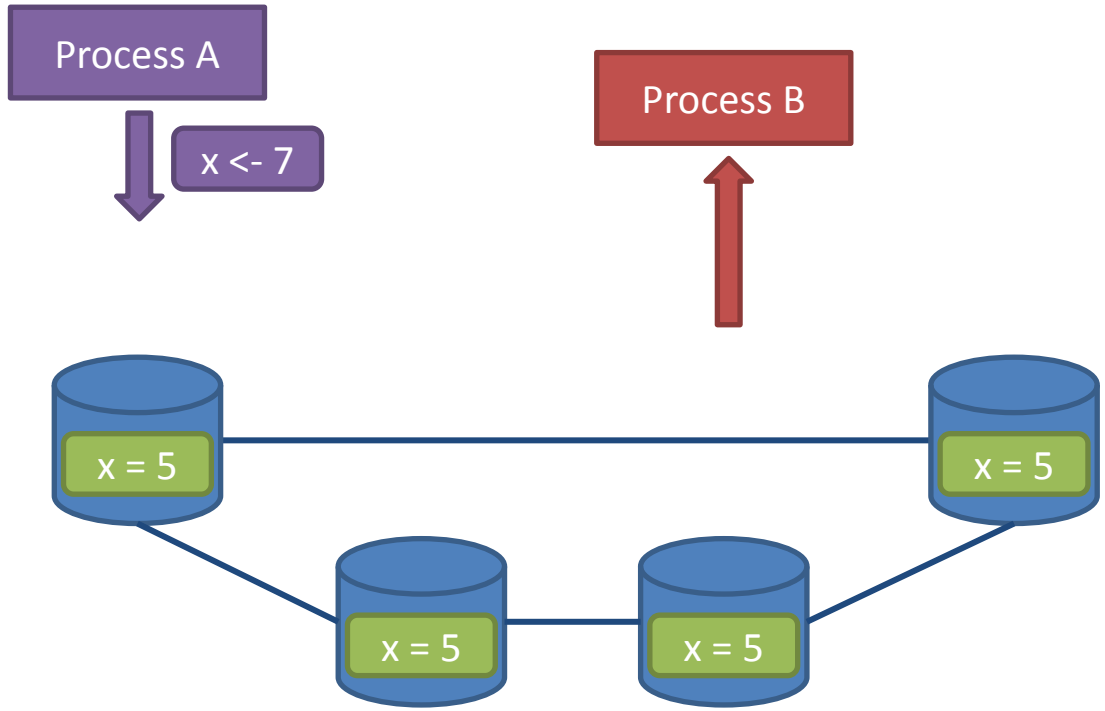
When to return the write as successful?

Server view

- Return the read(write) as successful only if a predetermined number of servers agree on the same value
- Assume:
  - **N** = # of servers storing the copies of data
  - **R** = # of servers that needs to agree on a value for a read operation
  - **W** = # of servers that needs to agree on a value for a write operation
- If R(W)-many servers agree on the same value, return the read(write) as successful

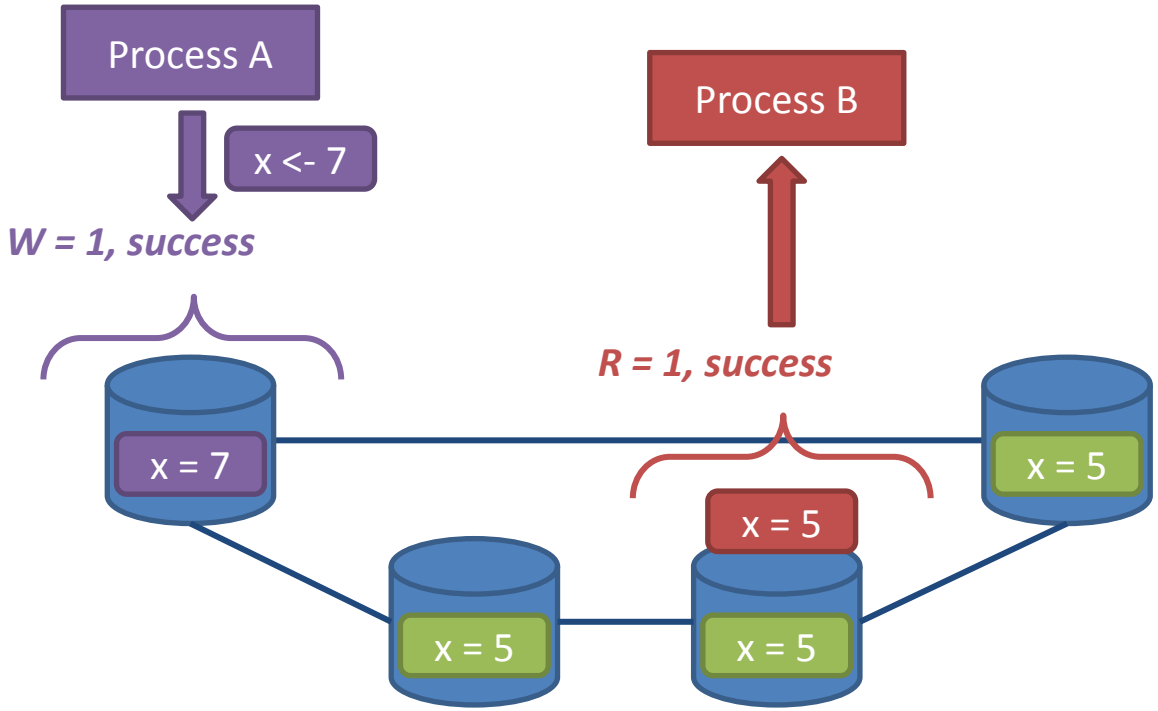
Server view

- If W = 1 and R = 1



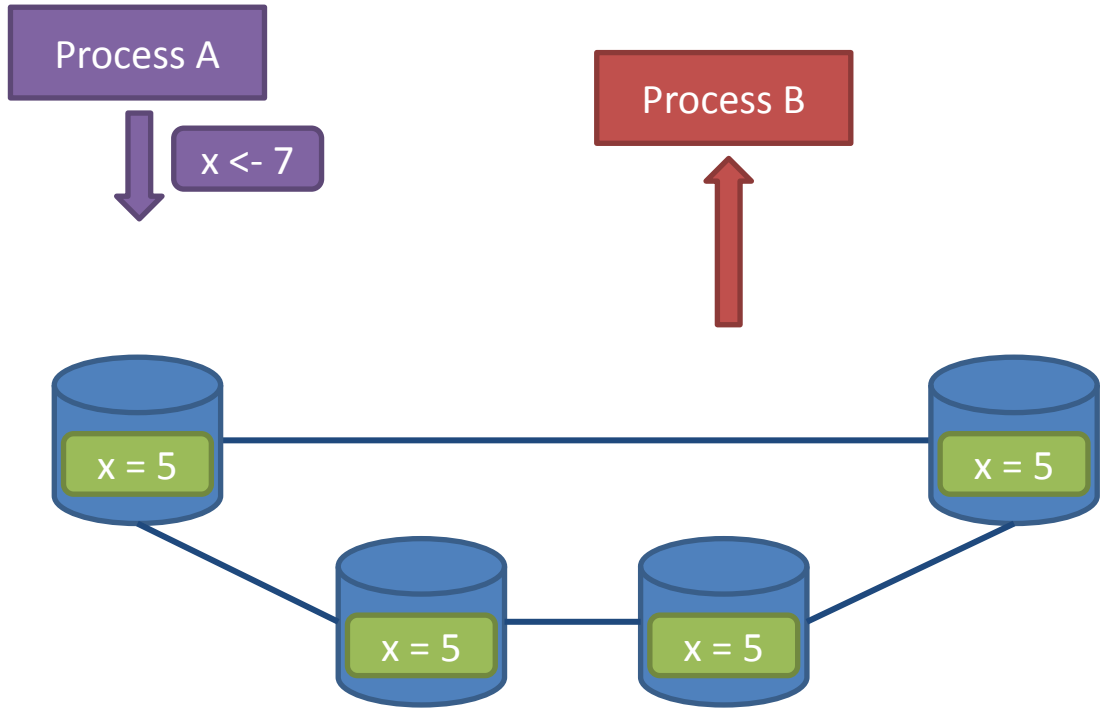
Server view

- If W = 1 and R = 1



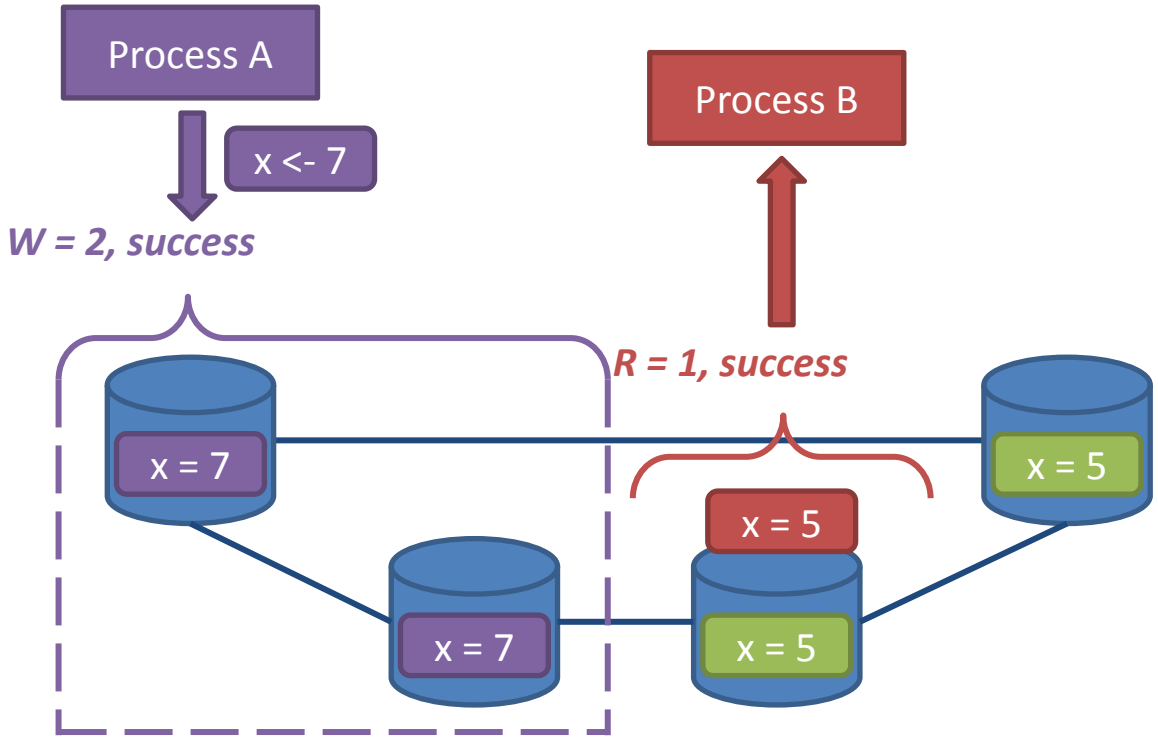
Server view

- If W = 2 and R = 1



Server view

- If W = 2 and R = 1



Write needs to wait for two servers

How to set R and W?

- R = 1 or W = 1 provides the fastest response
- R = N or W = N provides the slowest response
- Sacrifice consistency for faster response-time by tuning R and W parameters
  - If read(write)-optimized, set R(W) = 1
- R + W > N
  - Strong consistency: at least one server overlaps

Eventual consistency

- Updates and replication
- Eventual consistency protocol
- **Dynamo**

Example system: Dynamo

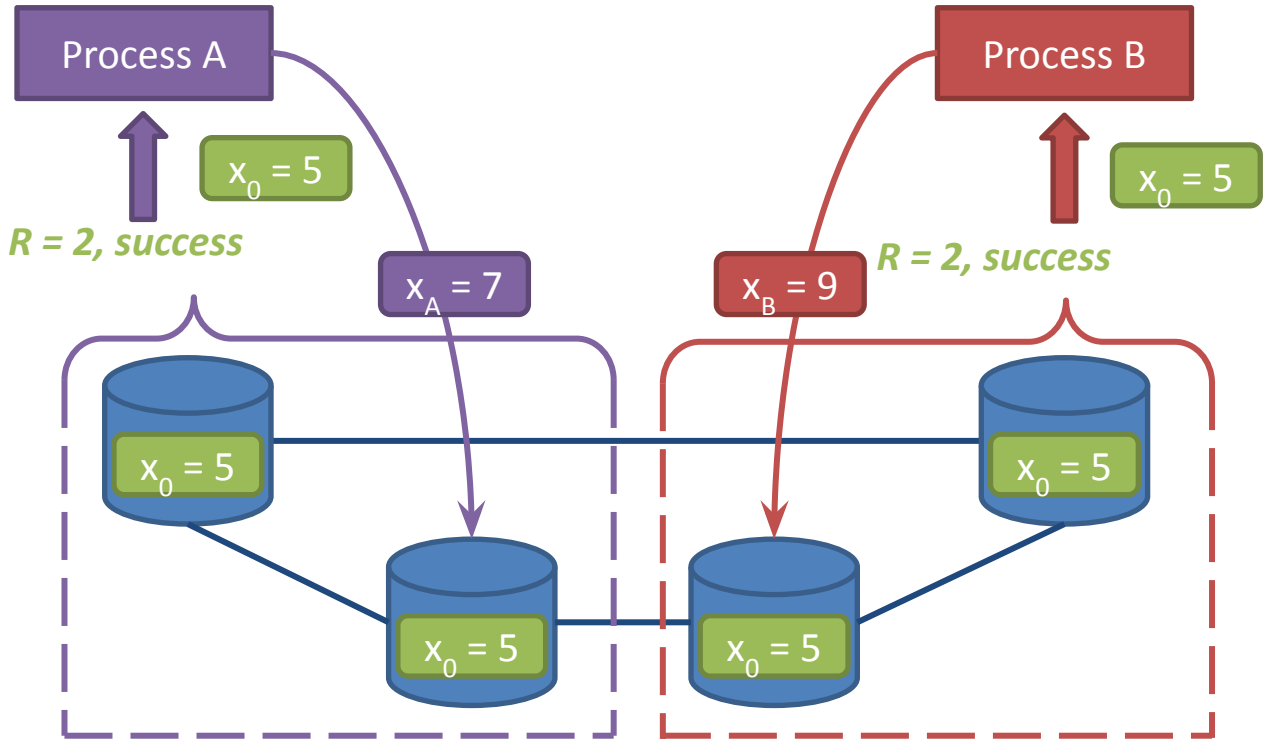
- Amazon's e-commerce platform
  - Simple key-value data model
  - Simple geet()/put() interface
  - Eventually consistent

- Serves tens of millions of customers using tens of thousands of servers located in many data centers around the world

Dynamo's eventual consistency algorithm

- Keep updates as separate versions of the data
- Return all the versions in case of conflict

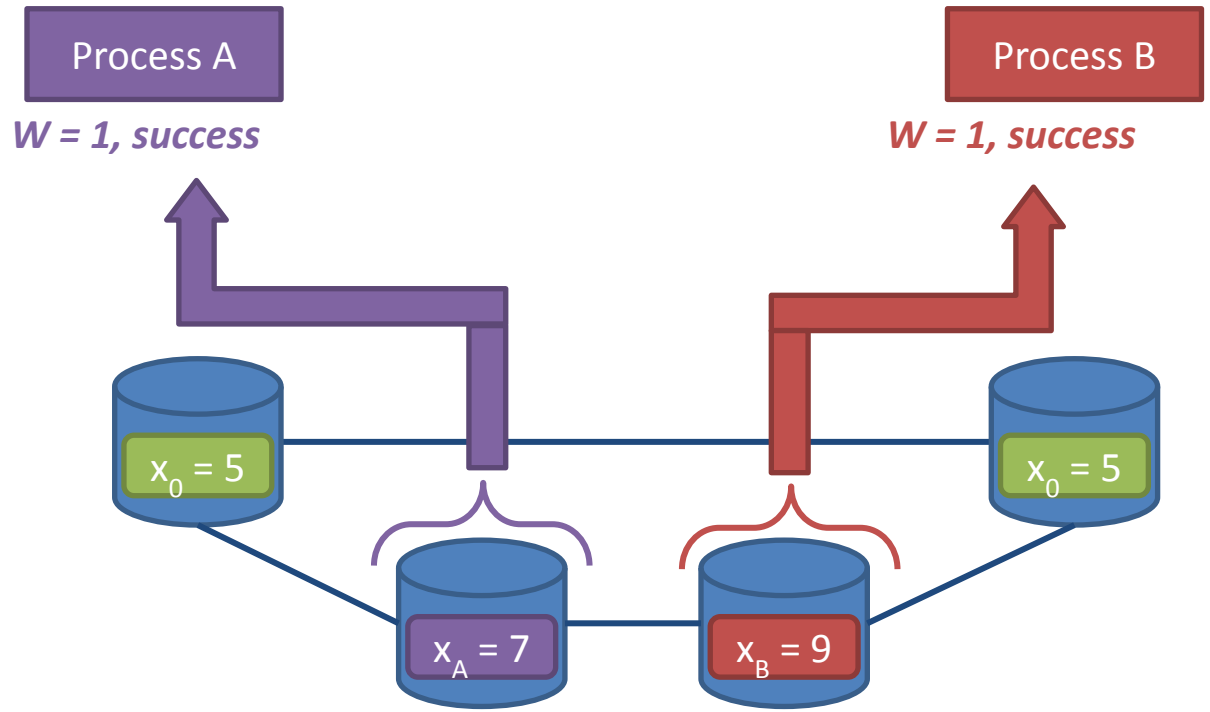
R = 2
W = 1



Dynamo's eventual consistency algorithm

- Keep updates as separate versions of the data
- Return all the versions in case of conflict

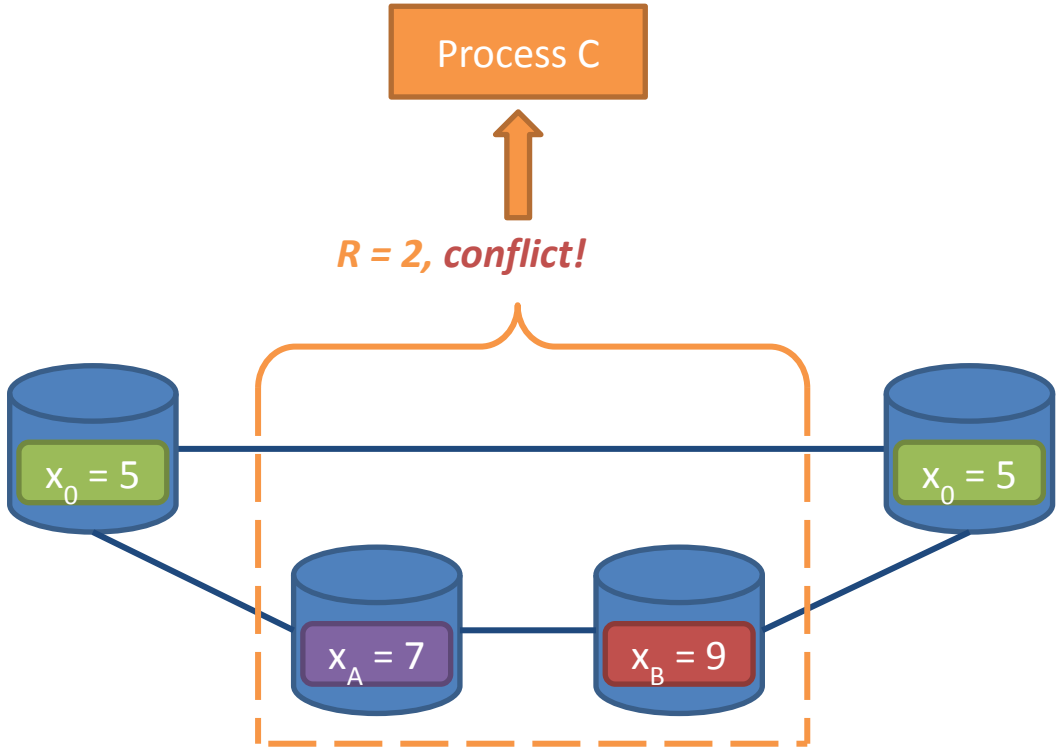
R = 2
W = 1



Dynamo's eventual consistency algorithm

- Keep updates as separate versions of the data
- Return all the versions in case of conflict

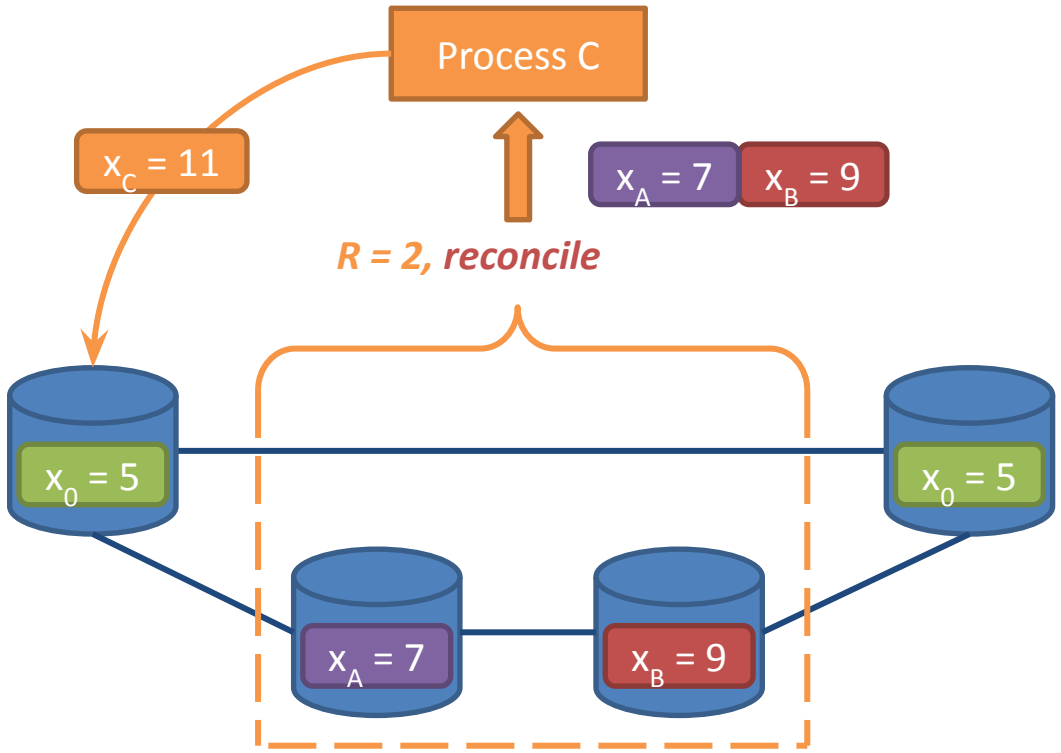
R = 2
W = 1



Dynamo's eventual consistency algorithm

- Keep updates as separate versions of the data
- Return all the versions in case of conflict

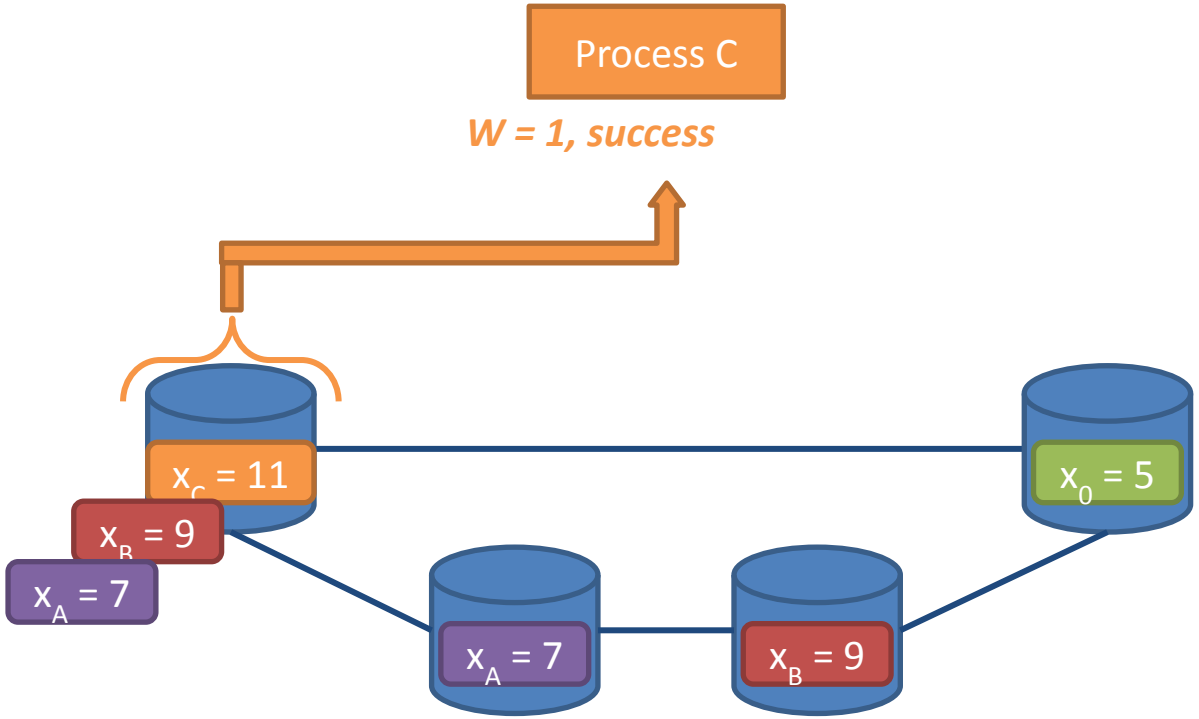
R = 2
W = 1



Dynamo's eventual consistency algorithm

- Keep updates as separate versions of the data
- Return all the versions in case of conflict

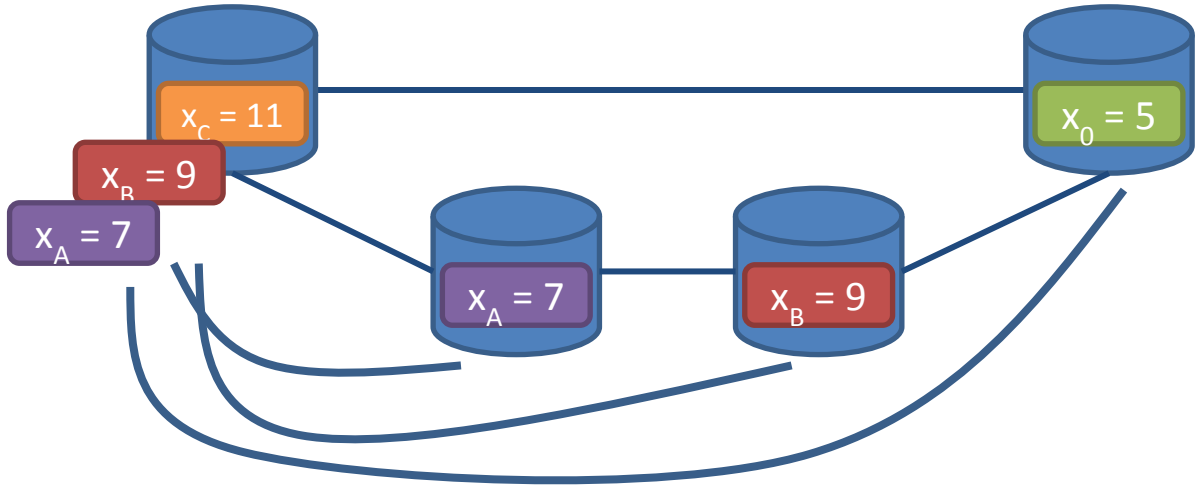
R = 2
W = 1



Dynamo's eventual consistency algorithm

- Keep updates as separate versions of the data
- Return all the versions in case of conflict

R = 2
W = 1



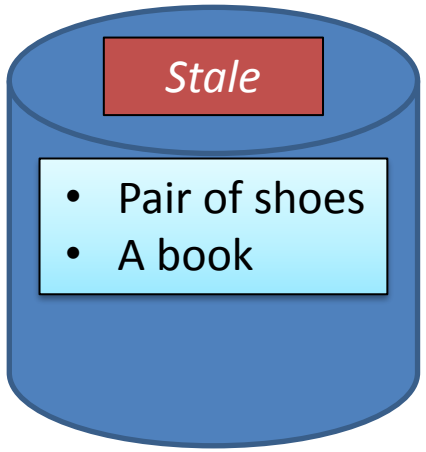
Updates propagate later on internally
Eventually, data become consistent

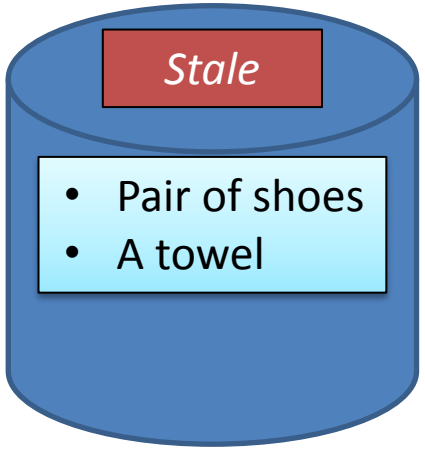
An example use-case: Shopping cart

Shopping cart

- Pair of shoes
- Sunglasses
- A book
- A towel

- Closed the web-browser and re-opened it





An example use-case: Shopping cart

Shopping cart

- Pair of shoes
- A book
- A towel

Reconcile

- Closed the web-browser and re-opened it
- Re-accessed shopping cart (assume R = 2)

Result is still meaningful

Application category can tolerate inconsistency

Can all application categories tolerate inconsistency?

What about your bank account?

Stronger than eventual: Causal consistency

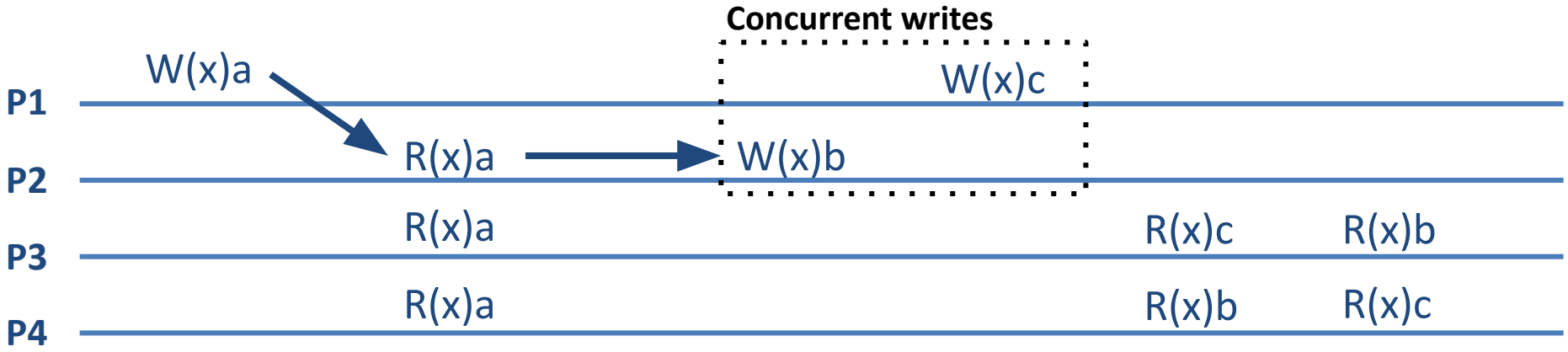
- Enforces the logical order of events that modify (write) in a distributed system
- The strongest achievable consistency model in the presence of partitions
- A system provides causal consistency if (potentially causally related) read/write memory operations are seen by every node of the system in the same order that reflects their causality
  - Writes that are potentially causally related must be read by all nodes in the same order
  - Concurrent writes may be seen in a different order on different nodes
  - Reads are fresh wrt the writes they causally depend on

Stronger than eventual: Causal consistency

- Why it could be important? Imagine the following consequence:
  - User A posts on Moodle message **M**: “I think value of pi is 3”
  - User A then quickly corrects the post to **M***: “I think value of pi is roughly approximated with 3.14”
  - User B then sees the post **M*** and responds to the post with message **R**: “Yes, we can use this approximation”
    - If causality was not respected, user C could perceive effects before their causes: If user C observes **M** and **R**, but not **M***, she could be misled to think pi=3 is acceptable approximation

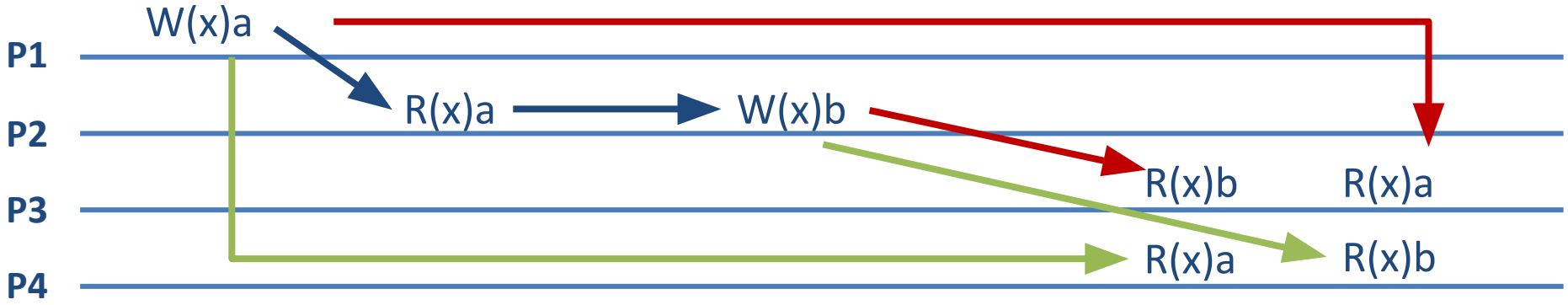
Causal consistency example

- P1, … P4: concurrent processes/servers/nodes
- W(x)a: write object's x'a attribute a; R(x)a: read object x's attribute a

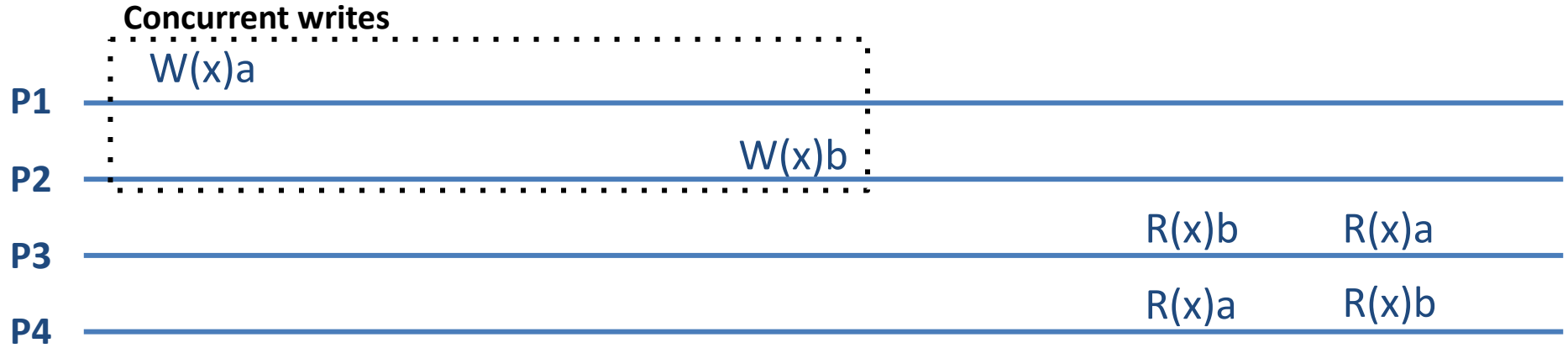


- W(x)b and W(x)c are concurrent (no causality): nodes don't need to see them in the same order
- P2, P3, and P4 read **a** and **b** in the order of writing as they are potentially causally related (through P2 W(x)b
- Execution order value under causal consistency rules: all nodes first read **a** and **b**

Causal consistency example



- Violation of causal consistency: P2 write W(x)b is potentially causally related to preceding write W(x)a
- Therefore, **a** and **b** must be read in the same order in nodes P3 and P4 as they are causally related (**P3 error**)



- Causally consistent: W(x)a and W(x)b are concurrent (non-causal), therefore there is no limitation in read order

Eventual consistency: Summary

- Data will eventually be consistent if no further updates are made
- Reads and writes need to agree on a value over a set of servers
- Example eventual consistency system: Dynamo
  - Keep every update as a separate version
  - Merge all versions in the case of conflicts
- Consistency models: design spectrum between strong and eventual
  - Strong > Causal > Eventual
- Causal consistency keeps track of which nodes have seen which writes
  - Stronger guarantees than eventual consistency, better performance than strong
  - Less popular in industry due to more complex communication, state maintenance

Who uses eventual consistency?

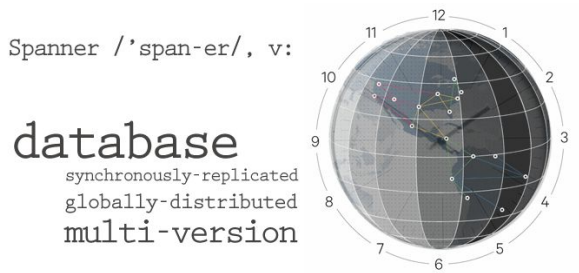
- Large-scale systems
  - Facebook, Google, Amazon, etc.
- Simple key-value data model
- Simple get()/put() interface

What is the downside?

- Programming eventually consistent systems is very hard and error-prone
  - Programmers need to reconcile all the corner cases in the application to ensure correctness
- Many companies go back to strong consistency

Eventually consistent

Consistency within entity groups



Strong consistency

SQL RDBMS