CAP Theorem & Weak Consistency

CS4405 – Analysis of Concurrent and Distributed Programs

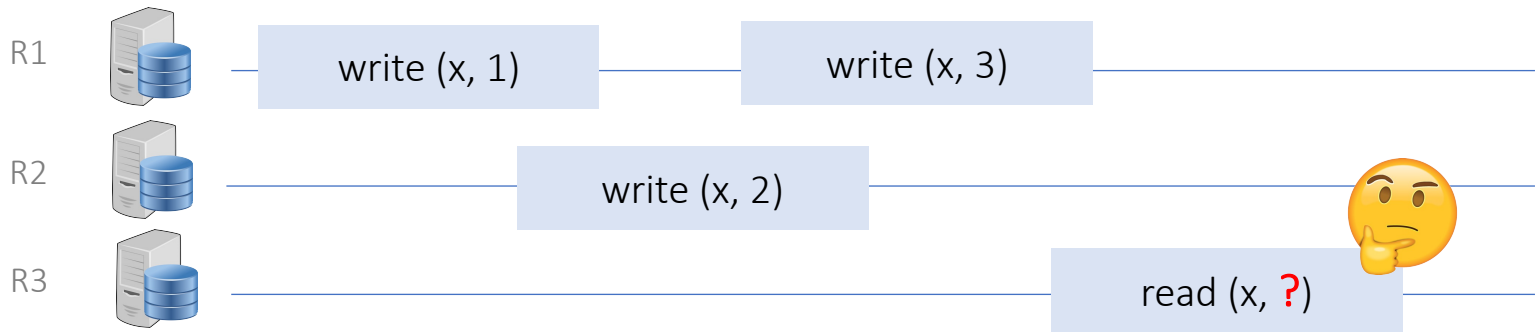
Burcu Kulahcioglu Ozkan

TU Delft

Revisit: Correctness for concurrent objects

Consistency model is a contract between programmer and replicated system, i.e., it specifies the consistency between replicas and what can be observed as possible results of operations.

- Concurrent accesses to multiple copies of the object
- Consistency condition defines which concurrent operation executions are considered correct



Spectrum of consistency models

- Strong consistency models (illusion of maintaining a single copy)
  - Linearizable consistency
  - Sequential consistency
- Weak consistency models
  - Causal consistency
  - Client-centric consistency models
    (Read your writes, monotonic reads, monotonic writes)
  - Eventual consistency

Last lecture

Weak consistency models trade off strong consistency for better performance.

How is a replicated data set accessed and updated?

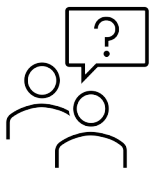
Replication architectures:

- **Single leader** or **master-worker**: A single leader accepts writes, which are distributed to followers
- **Multi-leader** or **master-master**: Multiple leaders accept writes, keep themselves in sync, then update followers
- **Leaderless replication** All nodes are peers in the replication network

Synchronous vs asynchronous replication

When a write occurs in node, this write is distributed to other nodes in either of the two following modes:

- **Synchronously:** Writes need to be confirmed by the followers
- **Asynchronously:** The leader reports success immediately after a write was committed to its own disk; followers apply the changes in their own pace

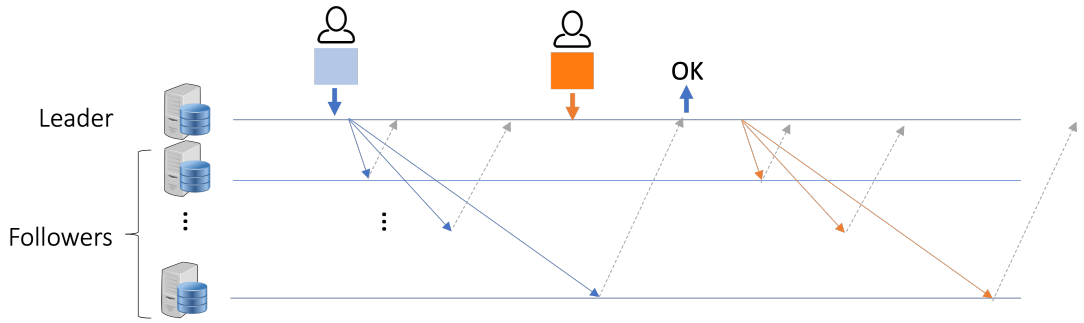


What are the advantages/disadvantages of each replication type?

Synchronous replication

The leader waits until the followers receive the update and before reporting success

- A follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader.
- If the leader suddenly fails, we can be sure that the data is still available on the follower.
- If the synchronous follower does not respond, the write cannot be processed.
- The leader must block all writes and wait until the synchronous replica is available again.

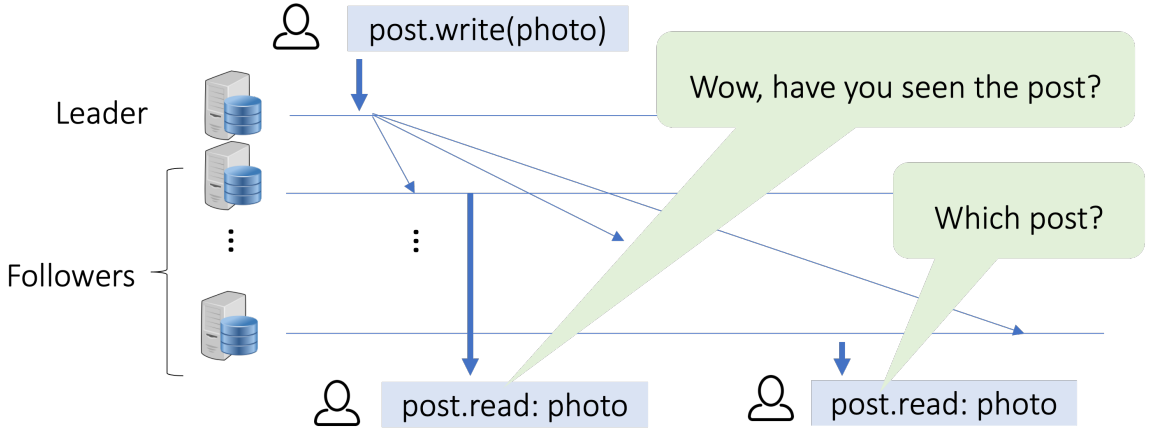


It’s impractical for all followers to be synchronous.

Asynchronous replication

The leader reports success and asynchronously updates the followers.

- Higher availability: The leader does not need to block writes in case of inaccessible follower.
- A follower **is not guaranteed to have an up-to-date copy** of the data that is consistent with the leader.
- Writes are not guaranteed to be durable in case of leader failure.



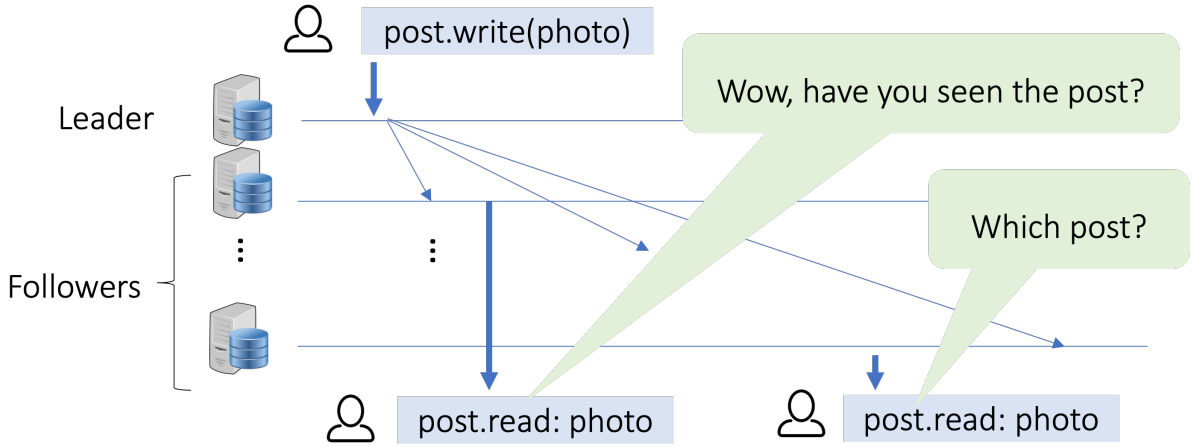
Compromise between two models, semi-synchronous:

Some followers are updated synchronously some are updated asynchronously

Asynchronous replication complications: Ordering problems

Asynchronous replication may cause clients to observe anomalous scenarios such as:

- **Read-after-write:** Clients may not see their own writes, i.e., when they connect to a replica which does not have the update.
- **(non-) Monotonic reads:** When reading from multiple replicas concurrently, a stale replica might not return records that the client read from an up to date one.
- **Causality violations:** Updates might be visible in different orders at different replicas. This allows for violations of the *happens-before* property.

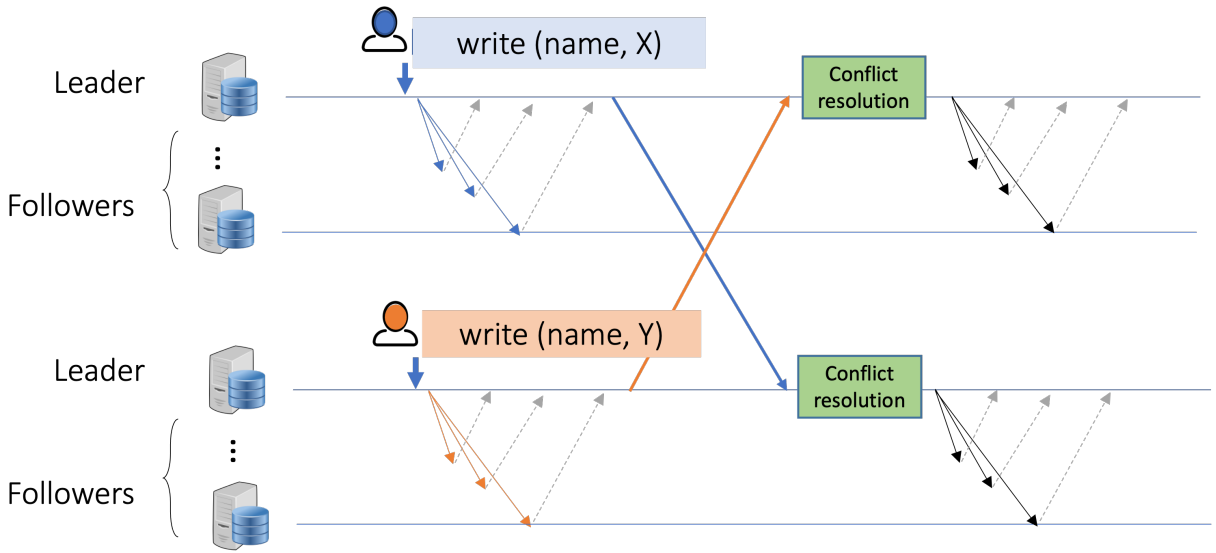


The client reads stale data

Multi-leader replication and its complications: Write conflicts

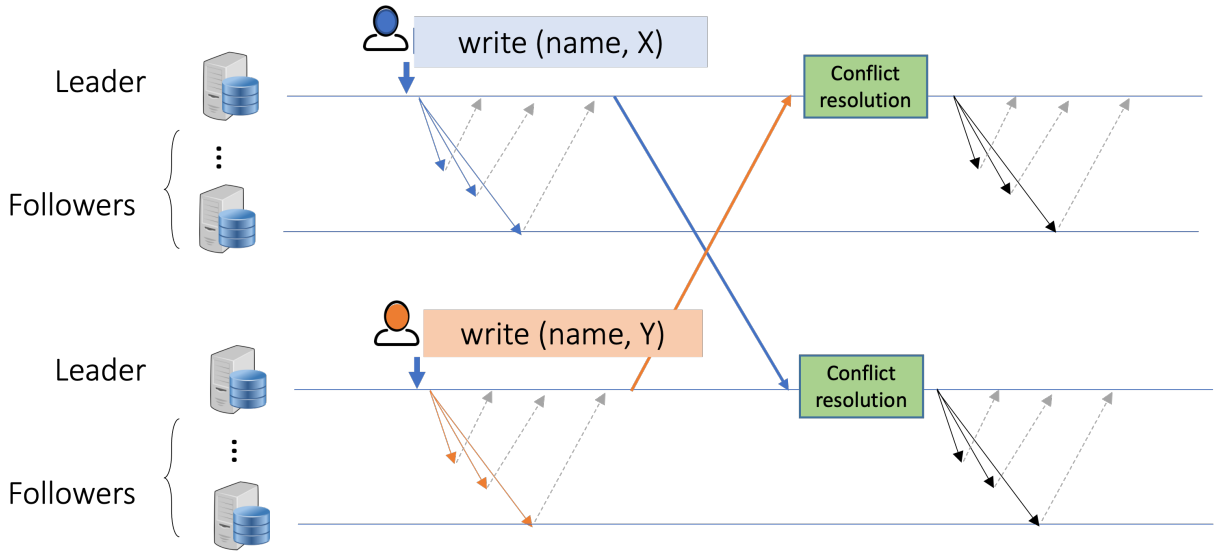
Multiple leader nodes to accept writes. Replication to followers in a similar way to single-leader case

The biggest problem with multi-leader replication are write conflicts.



How to avoid or resolve write conflicts?

- **Converge to consistent state:** Apply conflict resolution policies:
  - *last-write-wins* policy to order writes by timestamp (may lose data)
  - report conflict to the application and let it resolve it (same as git or Google docs)
  - use conflict-free replicated data types (CRDTs) with specific conflict resolution logics



"Conflict-free replicated data types", M. Shapiro, N. M. Preguiça, C. Baquero, M. Zawirski, SSS 2011

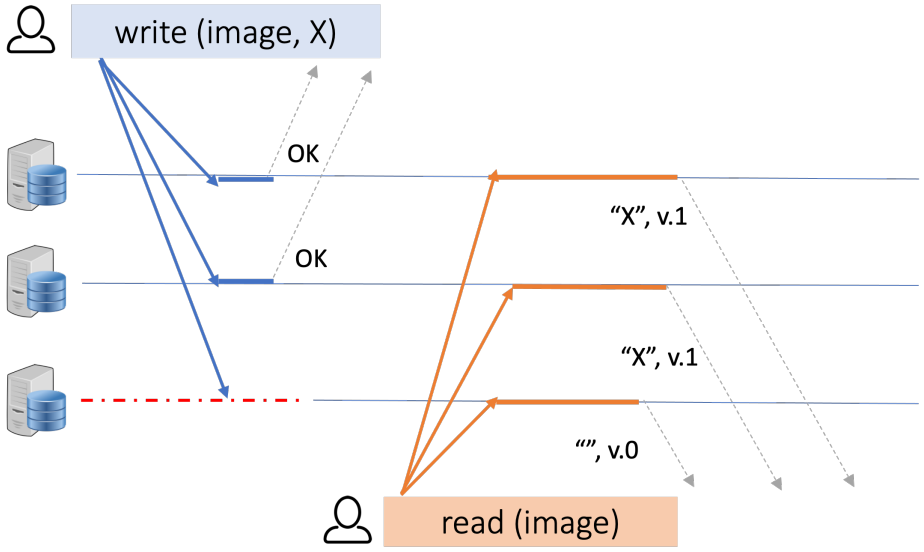
Leaderless/Quorum-based replication

Any replica can accept read/write queries from the clients.

Writes are successful if written to W replicas and reads are successful if written to R replicas:

- W + R > N : We expect to read up-to-date value
- W < N : We can process writes if a node is unavailable
- R < N : We can process reads if a node is unavailable

Example scenario with N=3, W=2, R=2:

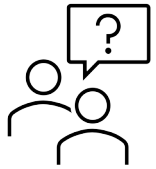


SMR and consensus Protocols

- State machine replication
  - FIFO Total order broadcast (All messages in the same order)
    - FIFO total order broadcast to every update
    - Replica applies to their local states
    - Deterministic updates
    - Replica is a state machine
  - Replicating transactions, blockchains, distributed ledgers
- Consensus for replication
- Paxos, VR, Raft
  - Provide strong consistency in the ordering of operations

Trade-off between consistency and availability

- Linearizability and SC provide strong consistency guarantees
  - They provide an illusion of a single copy of data



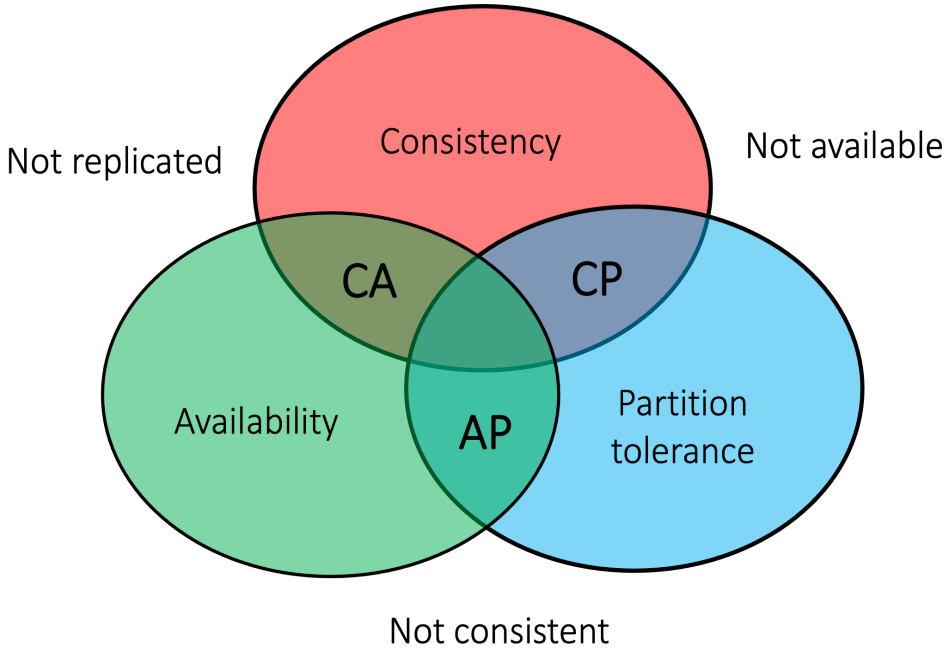
Can we design a system that is available even when some nodes in the system are not available or unreachable?

- Synchronous replication: Up-to-date replicas (consistent), but not available in case a failure of any follower
- Asynchronous replication: Replicas may not have up to date data (inconsistent), but the system is available in the existence of failures

Brewer's CAP Theorem:

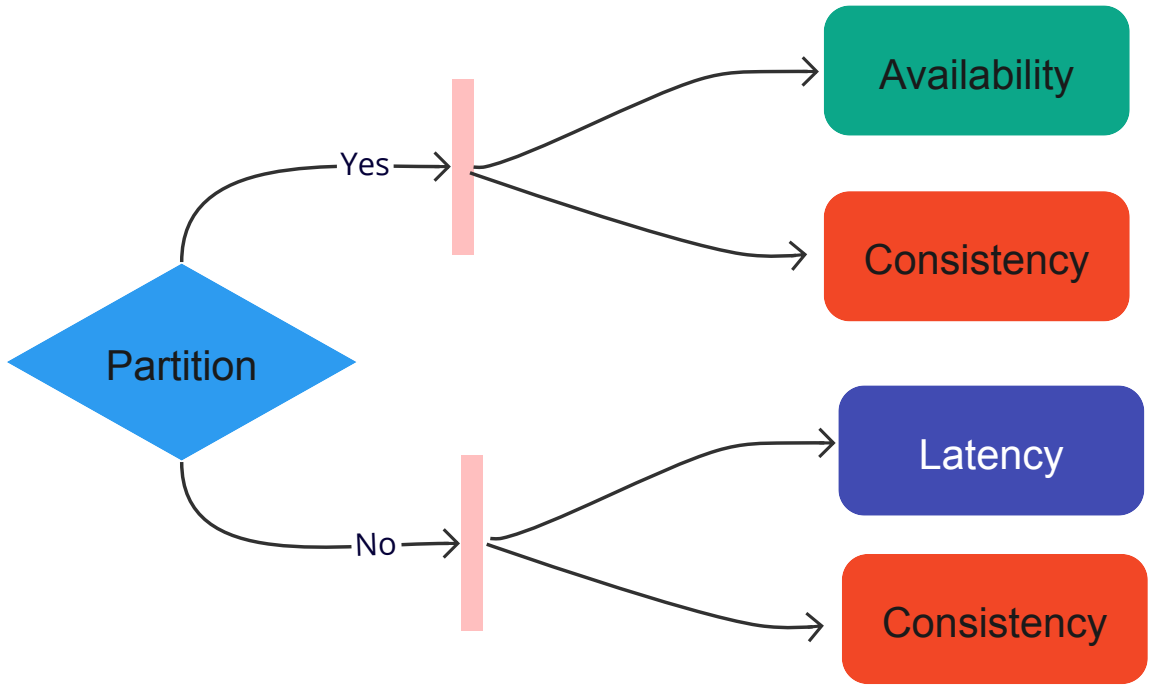
CAP Theorem: In a replicated system, it is impossible to get all three of:

- (Strong) **Consistency:** – All nodes in the network have the same (most recent) value
- **Availability:** – Every request to a non-failing replica receives a response
- **Partition tolerance:** The system continues to operate in the existence of component or network faults



The trade-off is not
"Availability vs Consistency"
but
"Availability vs *Strong* Consistency"

CAP extended to PACELC



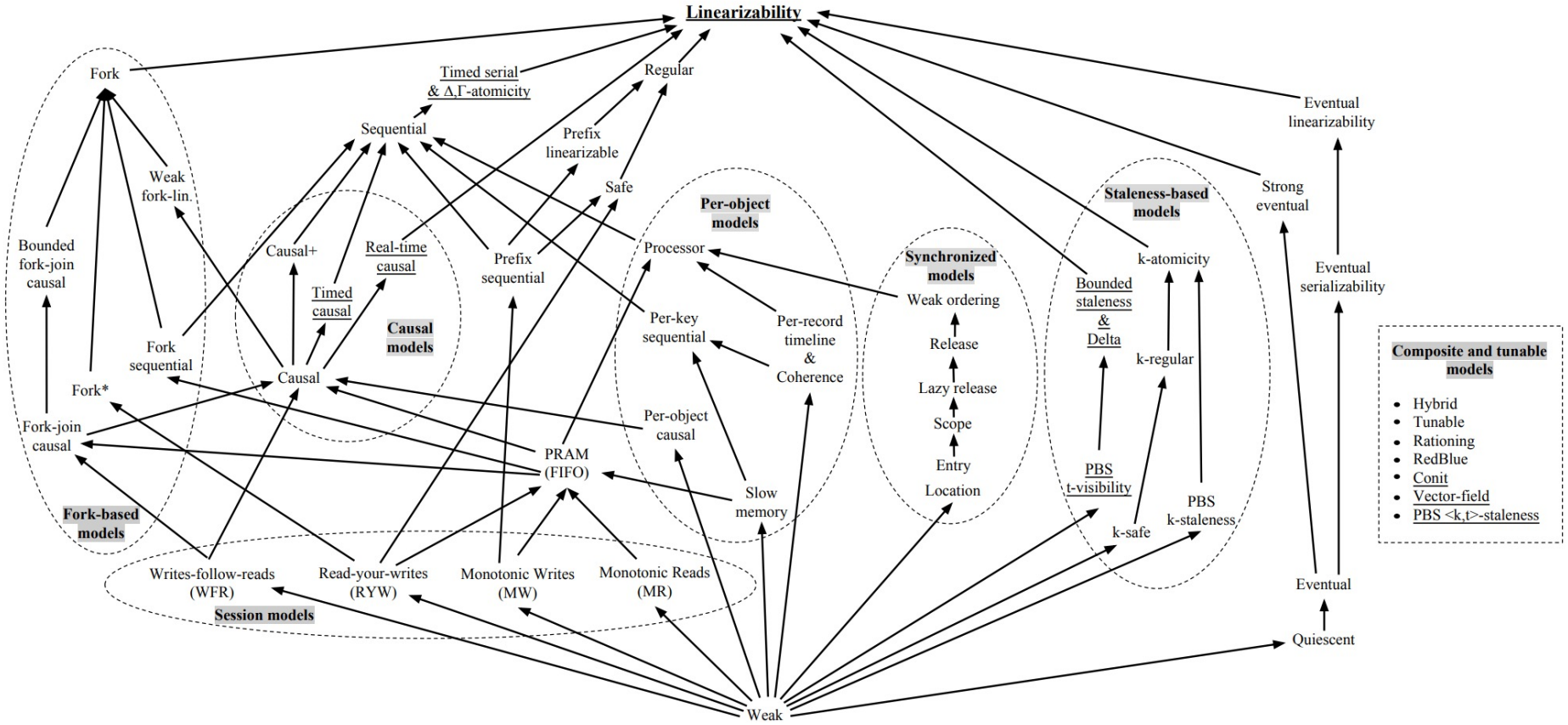
Spectrum of consistency models

- Strong consistency models (illusion of maintaining a single copy)
  - Linearizable consistency
  - Sequential consistency
- Weak consistency models
  - Causal consistency
  - Client-centric consistency models
    (Read your writes, monotonic reads, monotonic writes)
  - Eventual consistency

Weak consistency models trade off strong consistency for better performance.

Non-transactional consistency models

A wide variety of consistency models has been proposed and implemented.



Hierarchy of non-transactional consistency models (P. Viotti, M. Vikolic, ACM Computing Surveys, 2016)

Eventual consistency

All updates are *eventually* delivered to all replicas.

All replicas reach a consistent state if no more user updates arrive

Examples:

- Search engines: Search results are not always consistent with the current state of the web
- Cloud file systems: File contents may be out-of-sync with their latest versions
- Social media applications: Number of likes for a video may not be up to date

Client-centric consistency models

Provide guarantees about ordering of operations only for a single client process:

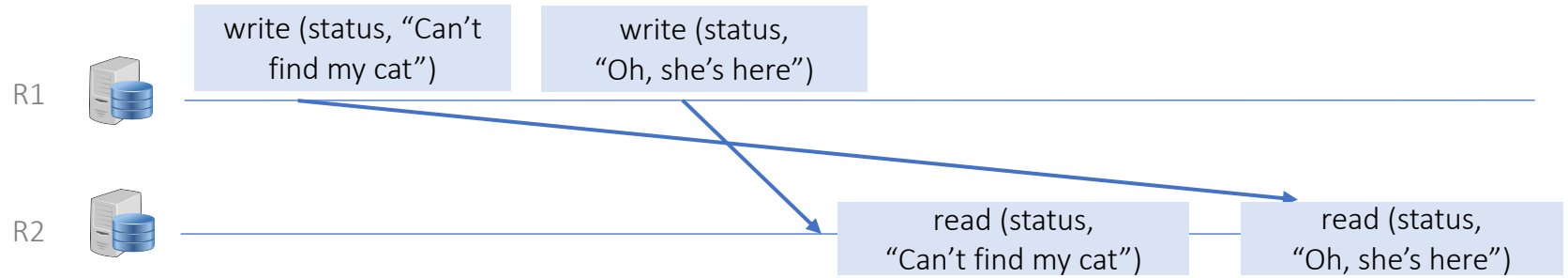
- Monotonic reads: If a process reads the value of x, any successive read operation on x by that process will always return that same value or a more recent value.
- Monotonic writes: If a process writes a value to x, the replica on which a successive operation is performed reflects the effect of a previous write operation by the same process
- Read your writes: The effect of a write operation by a process on x will always be seen by a successive read operation on x by the same process
- Writes follow reads: If a process writes a value to x following a previous read operation on x by the same process, it is guaranteed to take place on the same or more recent values of x that was read.

Causal consistency

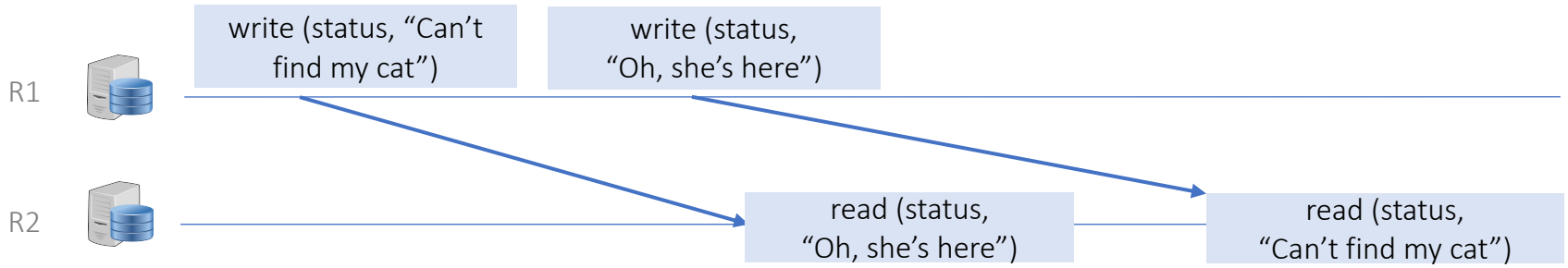
Causally related writes must be seen by all the replicas in the same order

- Maintains a partial order of events based on causality

Disallowed by causal consistency:

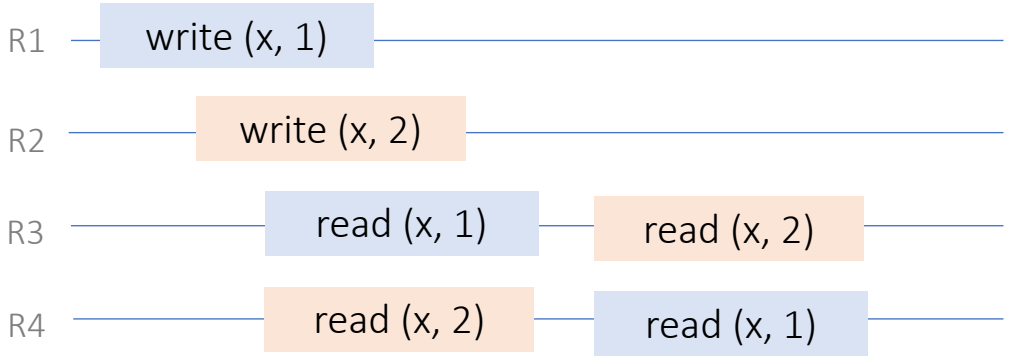


Allowed by causal consistency:

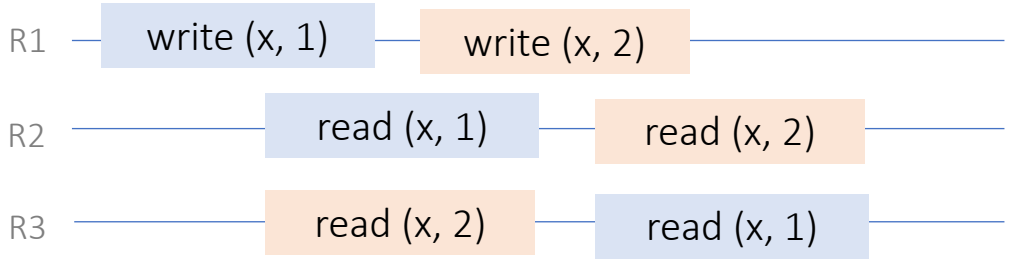


Causal consistency: Examples

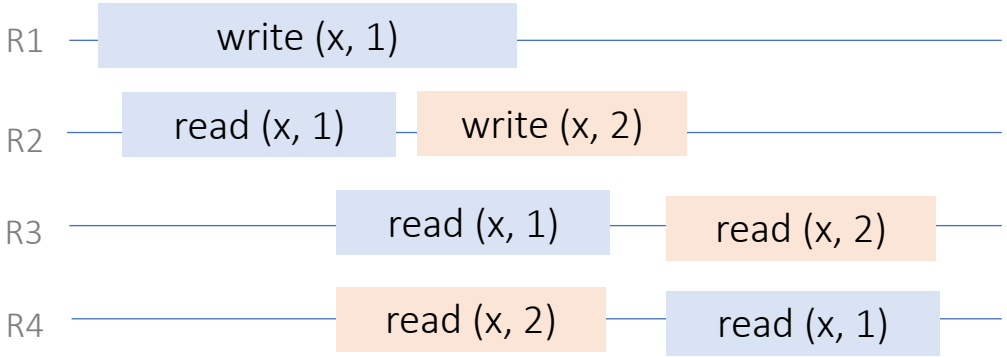
CC



Not CC

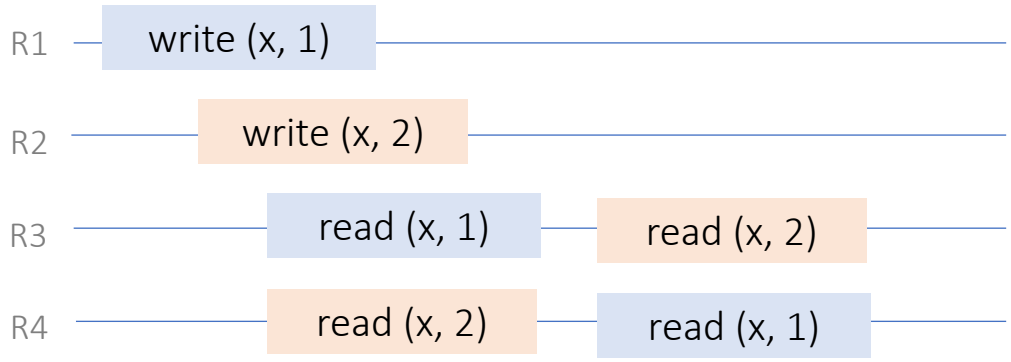


Not CC



Checking causal consistency

- Is the execution history causally consistent?
  - There exists a causality order CO such that the causal past of every read can explain its value
  - CO includes the site order



- Checking whether an execution is causally consistent is NP-complete in general

On Verifying Causal Consistency. A. Bouajjani, C. Enea, R. Guerraoui, J. Hamza. POPL 2017.

Transactions and Isolation Models

Transactions in relational databases

The concept of the transaction started with the first SQL database, System R; while the mechanics changed, the semantics are stable for the last 40 years.

Transactions provide the following guarantees:

- Atomicity: The transaction either succeeds or fails; in case of failure, outstanding writes are ignored (or all nothing).
- Consistency: Any transaction will bring the database from one valid state to another.
- Isolation: Concurrent execution of transactions do not interfere and effect each other.
- Durability: Once a transaction has been committed, it will remain so.

Note that the notion of consistency is different from the “consistency” of replication.

In the context of ACID, consistency refers to the good state of the database where the data integrity constraints hold.

Isolation

Databases try to hide concurrency issues from applications by isolating transactions from each other.

Isolation ensures that transactions are processed without interference: The goal is to prevent reads and writes of data written by incomplete or aborting transactions.

Isolation guarantees restrict how and when writes become visible.

There are various degrees of isolation (degree of how isolated a transaction is):

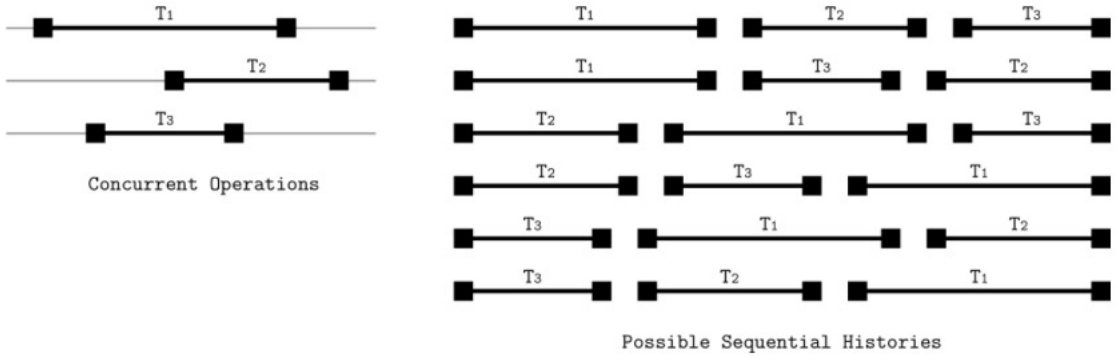
- High isolation: Discourages concurrency (it uses more locks) and availability, favors database consistency (valid state)
- Low isolation: Encourages concurrency and availability, allows some inconsistent data

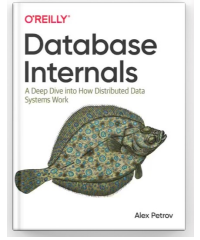
The highest level of isolation provides serializable transactions.

Serializability

Highest level of isolation entails executing transactions *as if* they were serially executed

- A schedule is a list of operations (that interact with the database state, such as read, write, commit, or abort) required to execute a set of transactions
- A schedule is said to be serial when transactions in it are executed completely independently and without any interleaving
- The transaction operations are run concurrently, while maintaining the correctness and simplicity of a serial schedule.





A serializable schedule is the one that always leaves the database in consistent* state.

Checking serializability

- Serializability violations in an execution can be detected by constructing a precedence/dependency graph and by checking if it contains cycles
- (Checking for conflict-serializability) Given a schedule S, construct G = (V, E):
  - The transactions are represented by the vertices
  - The relations between the transactions are represented with the edges:
    - $T_i \rightarrow T_j$ if T_i executes **`write(Q)`** before T_j executes r**`ead(Q)`**
    - $T_i \rightarrow T_j$ if T_i executes r**`ead(Q)`** before T_j executes **`write(Q)`**
    - $T_i \rightarrow T_j$ if T_i executes **`write(Q)`** before T_j executes **`write(Q)`**
  - If G contains a cycle, then S is not serializable.
  - If G contains no cycles, a serializability order can be obtained by topological sorting

Example schedules

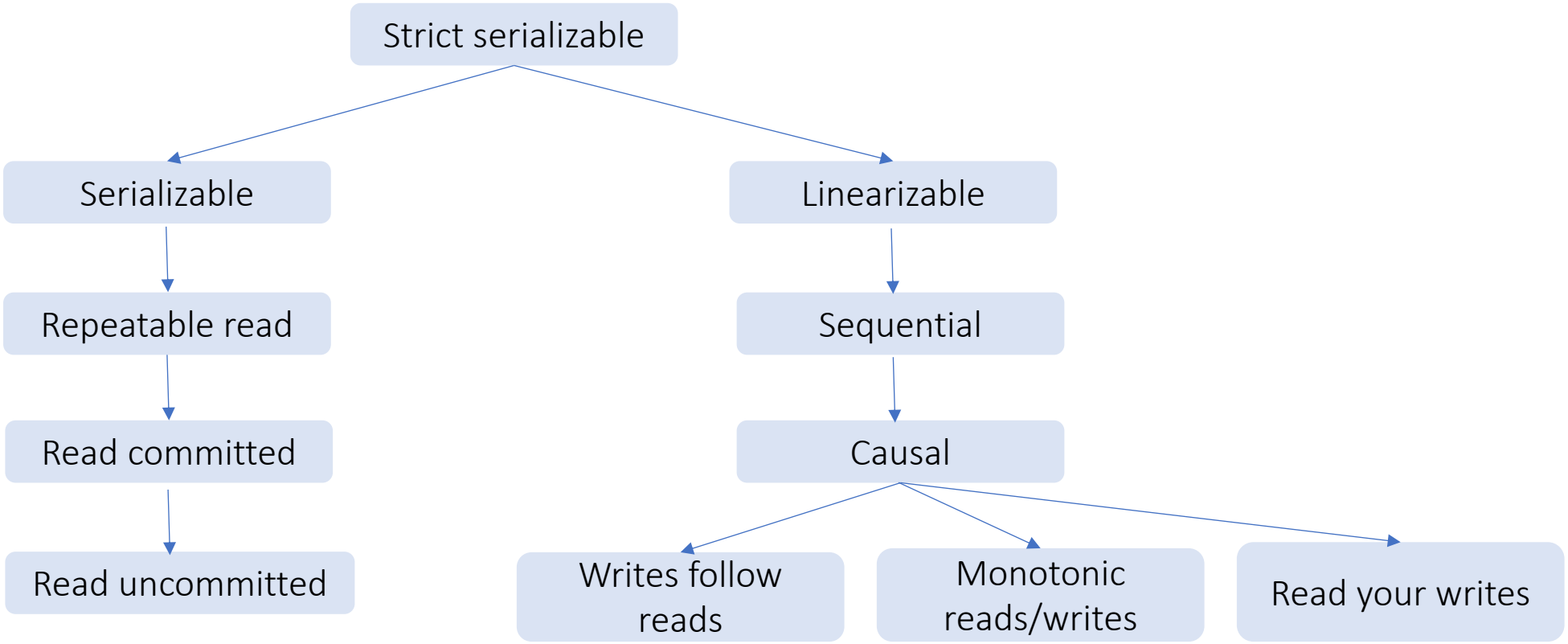
Weaker isolation levels - ANSI

While serializable isolation works it may be slow. Consequently, historically, databases made compromises in how they implement isolation.

- *Dirty reads*: A transaction reads data written by a concurrent uncommitted transaction
- *Non Repeatable Reads*: A transaction queries the same row twice and gets different results
- *Phantom reads*: A transaction queries the *same set of rows* twice and receives different results. (for range queries)

Isolation	DR	NRR	PR
Read uncommitted	Allowed	Allowed	Allowed
Read committed		Allowed	Allowed
Repeatable read			Allowed
Serializable			

Isolation levels and consistency models

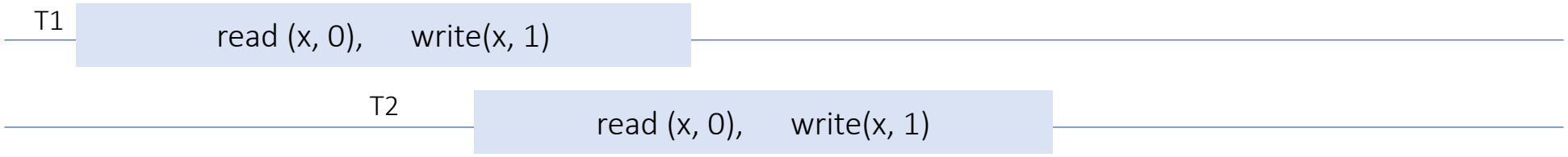


ANSI/ISO SQL isolation level specifications

basic consistency specifications

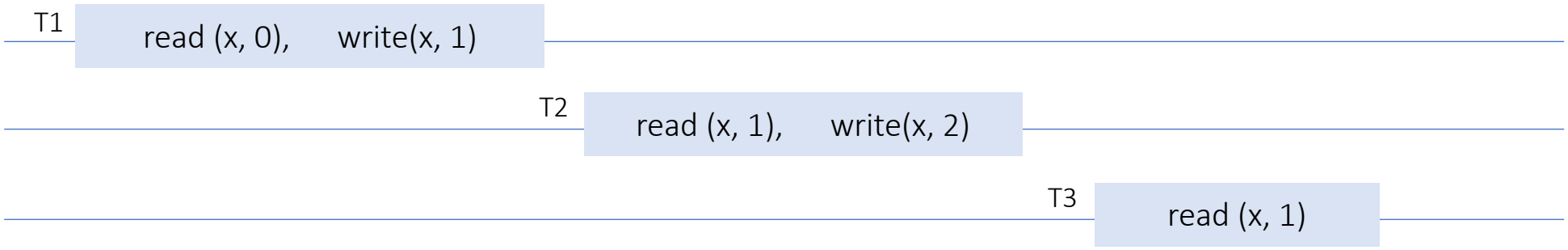
Isolation vs Consistency

Consistency without isolation:



Two concurrent transactions incrementing the value of “x”. The transactions are not serialized, the final value of x is incorrect

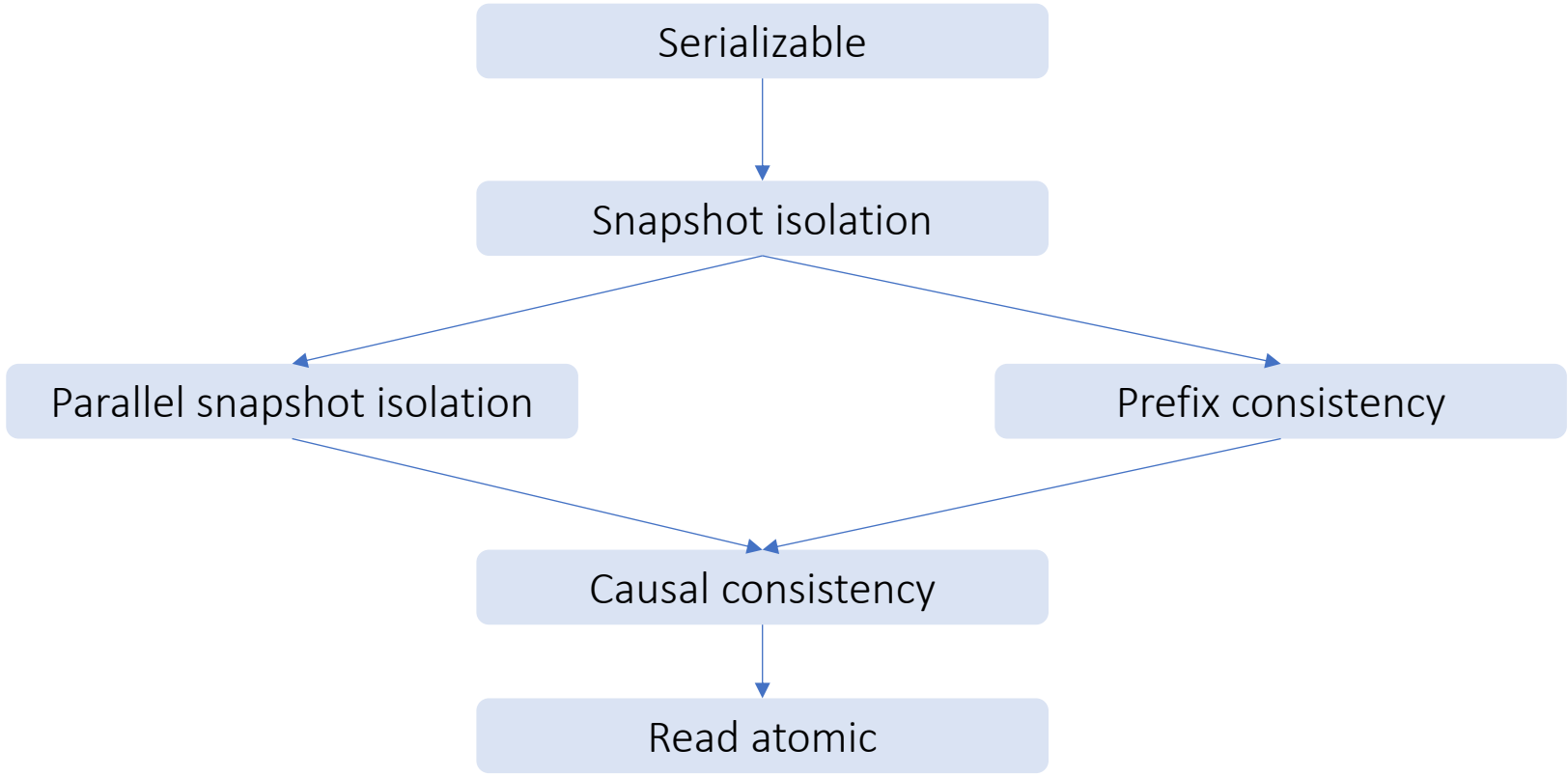
Isolation without consistency:



Transactions run in isolation, but T3 reads and operates on stale/incorrect data

Examples adapted from: Demystifying Database Systems, Part 4: Isolation levels vs. Consistency levels

Transactional consistency models



A Framework for Transactional Consistency Models with Atomic Visibility. A. Cerone, G. Bernardi, A Gotsman. CONCUR 2015.

Take aways

- Distributed systems are hard
- Know the guarantees of the systems you use:
  - Trade-off between Consistency and Availability/Latency
  - Weaker models of consistency
  - Weak models of isolation
- Beware of possible violations to guarantees
  - We need to test distributed systems
  - More at the next lecture on active research directions ☺