Common tools of new distributed database systems

Iztok Savnik, FAMNIT, 2021-22.

Outline

- Time and clocks
- Paxos
- Gosip
- Partitioning
- LSM tree

Timestamps and Vector Clocks

- Determining a history of a row.
- Eventual consistency relies on deciding what value a row will eventually converge to.
  - In the case of multiple versions of data item, an ordering of events in a distributed system can help in deciding the “last” value assigned.
  - The ordering of the events in distributed systems should reflect the causality.
    - What happened before something else.
  - In the case of two simultaneous writers this is difficult.
    - Writing at “the same” time. possibly on different sites (concurrent events),

- Timestamps (absolute time) are one solution, but rely on synchronized clocks and don’t capture causality.

- Lamport timestamps and vector clocks are alternative methods of capturing order in a distributed system.

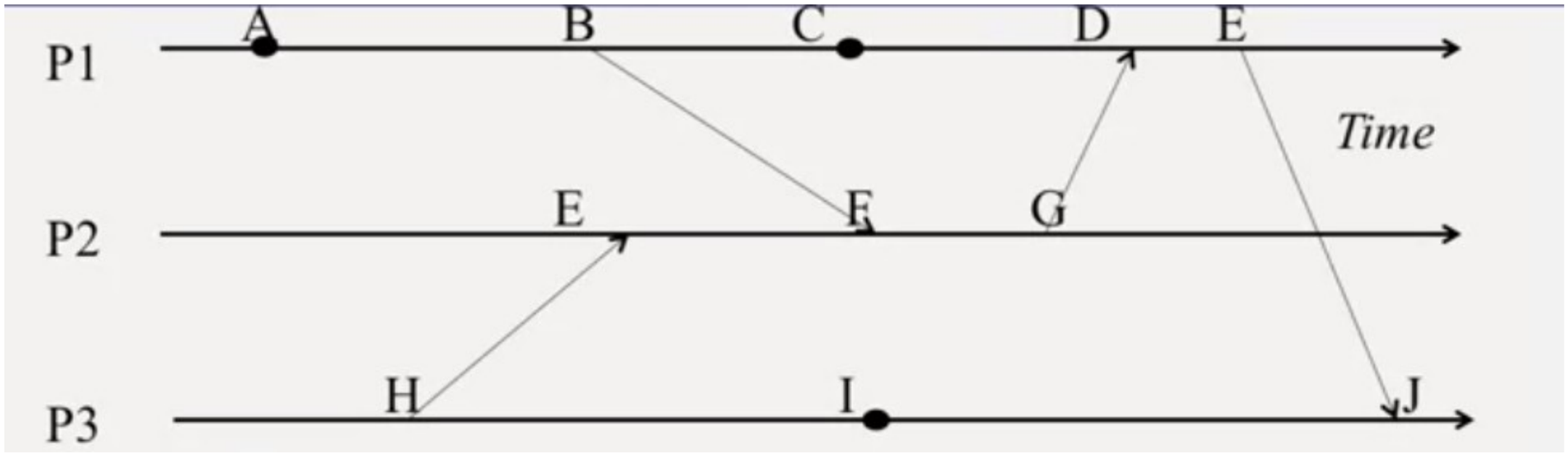
Lamport Timestamps

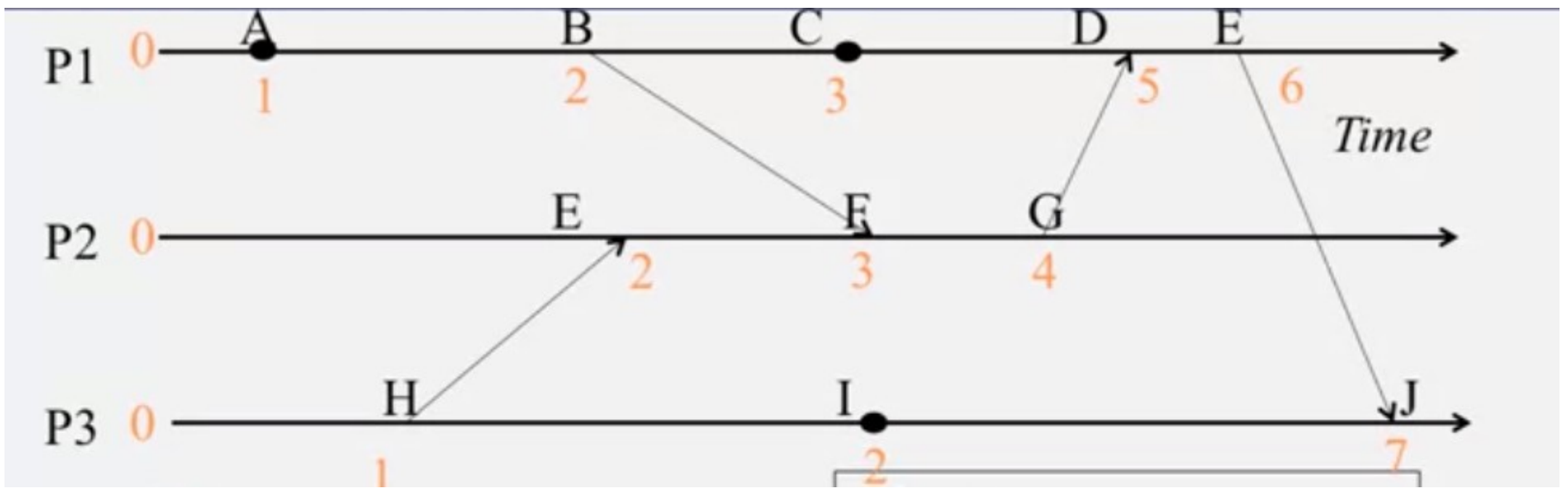
- Ordering events in a distributed system.
  - Trying to sync clocks is one approach.
- Timestamps that are not absolute time.
- Such timestamps should obey causality
  - If an event A happens causally before an event B then timestamp(A) < timestamp(B)
- Humans use causallity often
  - “I entered the house only after I unlocked the door”
  - “You receive the letter only after I send it”

Lamport Timestamps

- Define a logical relationship Happens-Before among pairs of events
  - Happens-Before is denoted →
- Three rules
  - On the same computer: A → B, if time(A) < time(B) (local clock)
  - Site s1 sends a message m to the site s2: send(m) → receive(m)
  - Transitivity: A → B and B → C implies A → C
- Relationship → is partially ordered.
  - Not all events are related by → !
  - Reflexivity: $a \leq a$
  - Antisymmetry: if $a \leq b$ and $b \leq a$ then $a = b$
  - Transitivity: if $a \leq b$ and $b \leq c$ then $a \leq c$

Lamport Timestamps





Lamport Timestamps

- Task: Assign logical timestamps to each event.
  - Timestamps that can capture causality among events.
- Rules:
  - Each site uses local counter (integer); initial value is 0.
  - Site increment its counter by one when
    - it sends a message to some other site or
    - an operation happens.
  - A message sent to some other site carries its timestamp.
  - For a receive message, timestamp is updated to
    - max(local clock,message timestamp)+1.

Lamport Timestamps

- A pair of concurrent events does not have a causal path from one event to another.
- For concurrent events Lamport timestamps are not ordered!
  - This is OK, since there is no relationship between the events.
- In some cases, there is no way to distinguish between the concurrent event and causal event.
  - Timestamps could not capture this.
  - Vector clocks can show that two events are concurrent!

- Here is what can be judged from the Lamport timestamps:
  - E1 → E2 ==> timestamp(E1) < timestamp(E2)
  - timestamp(E1) < timestamp(E2) ==> either E1,E2 are concurrent events, or E1 → E2.

Vector Clocks

- Proposed by Barbara Liskov, Rivka Ladin (1986), as multipart timestamp.
  - "Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection". 5th Symp. on the Principles of Distributed Computing. ACM.
  - "vector clock" was first used independently by Colin Fidge and Friedemann Mattern in 1988
- A vector clock is defined as a tuple V[0], V[1], ..., V[n] of clock values from each node.
- In a distributed scenario node i maintains such a tuple of clock values, which represent the state of itself and the other (replica) nodes' state as it is aware about at a given time.
  - $V_i[0]$ for the clock value of the first node,
  - $V_i[1]$ for the clock value of the second node, . . .
  - $V_i[i]$ for itself, . . .
  - $V_i[n]$ for the clock value of the last node.
- Clock values may be real timestamps derived from a node's local clock, version/revision numbers or some other ordinal values.

Vector Clocks

- Vector clocks are updated in a way defined by the following rules.
  - If an internal operation happens at node i, this node will increment its clock $V_i[i]$.
    - This means that internal updates are seen immediately by the executing node.
  - If node i sends a message to node k, it first advances its own clock value $V_i[i]$ and attaches the vector clock V_i to the message to node k.
    - Thereby, he tells the receiving node about his internal state and his view of the other nodes at the time the message is sent.
  - If node i receives a message from node j, it first advances its vector clock $V_i[i]$ and then merges its own vector clock with the vector clock V message attached to the message from node j so that:

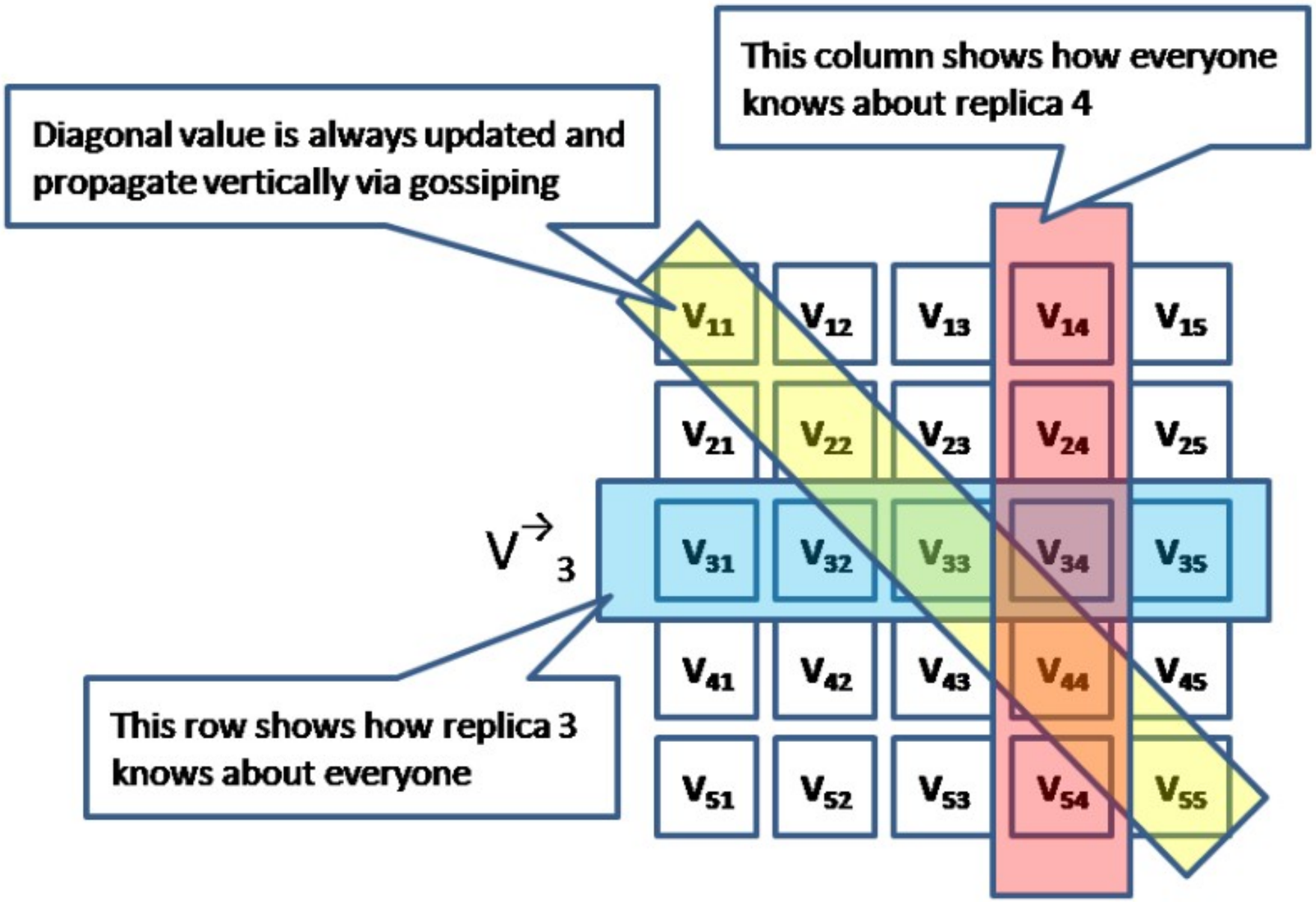
$$V_i = \max(V_i , V_{message})$$

- To compare two vector clocks V_i and Vj in order to derive a partial ordering, the following rule is applied:

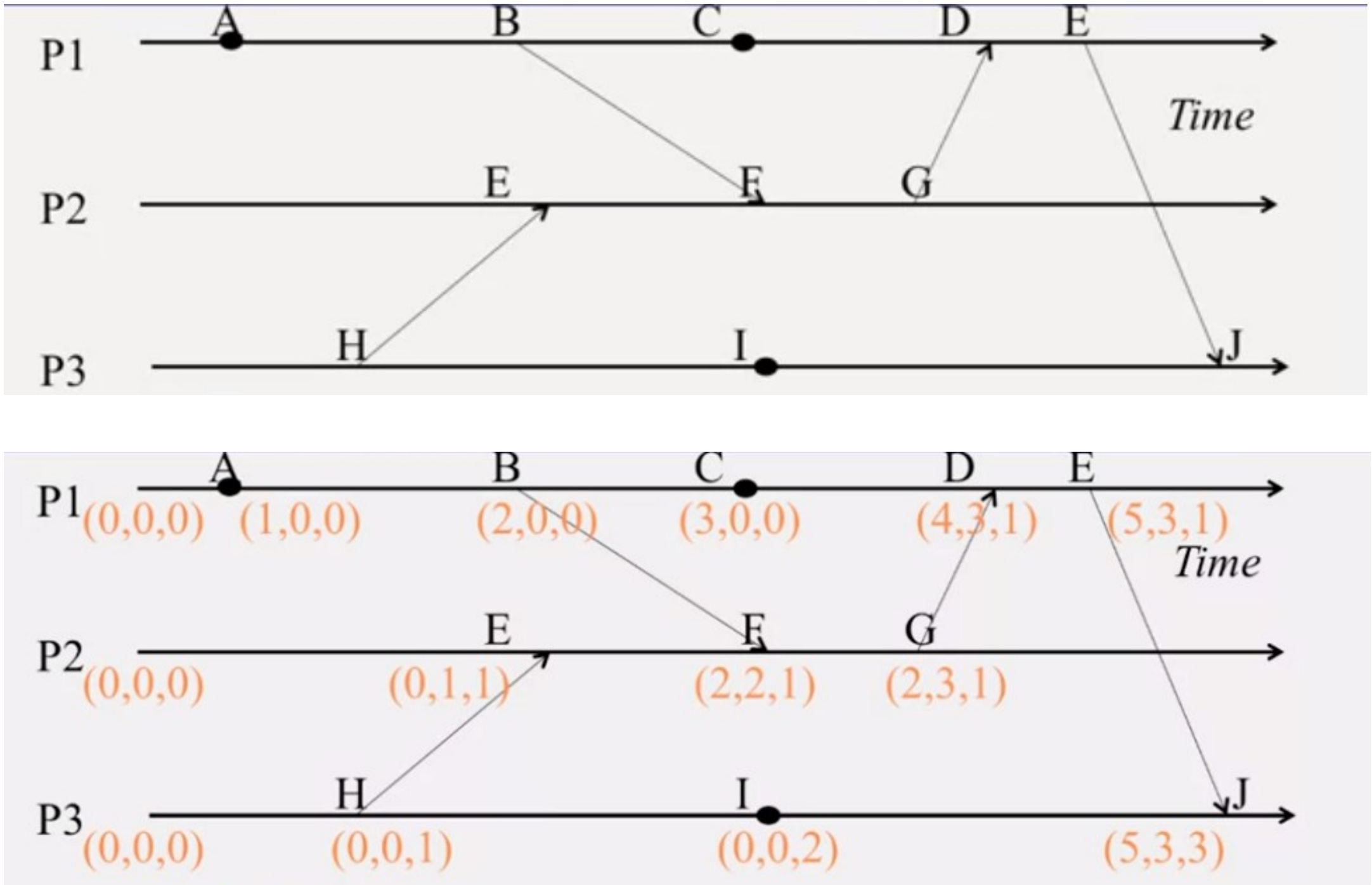
$$V_i > V_j \text{ , if } \forall k(V_i[k] >= V_j[k]) \text{ and } \exists l(V_i[l] > V_j[l])$$

- If neither $V_i > V_j$ nor $V_i < V_j$ applies, a conflict caused by concurrent updates has occurred and needs to be resolved by e.g. a client application.

Vector Clocks



Vector clocks – example



Vector Clocks

- Vector clocks allow causal reasoning between updates.
- Clients participate in the vector clock scenario.
  - They keep a vector clock of the last replica node they have talked to and use this vector clock depending on the client consistency model.
  - For *monotonic read consistency* a client attaches this last vector clock it received to requests and the contacted replica node makes sure that the vector clock of its response is greater than the vector clock the client submitted.
  - Clients can be sure to see only newer versions of some data item.
- Advantages of vector clocks are:
  - No dependence on synchronized clocks.
  - No total ordering of *revision numbers* required for causal reasoning.
  - *Concurrent events* can be identified!

Outline

- Time and clocks
- Paxos
- Gossip
- Partitioning
- LSM tree

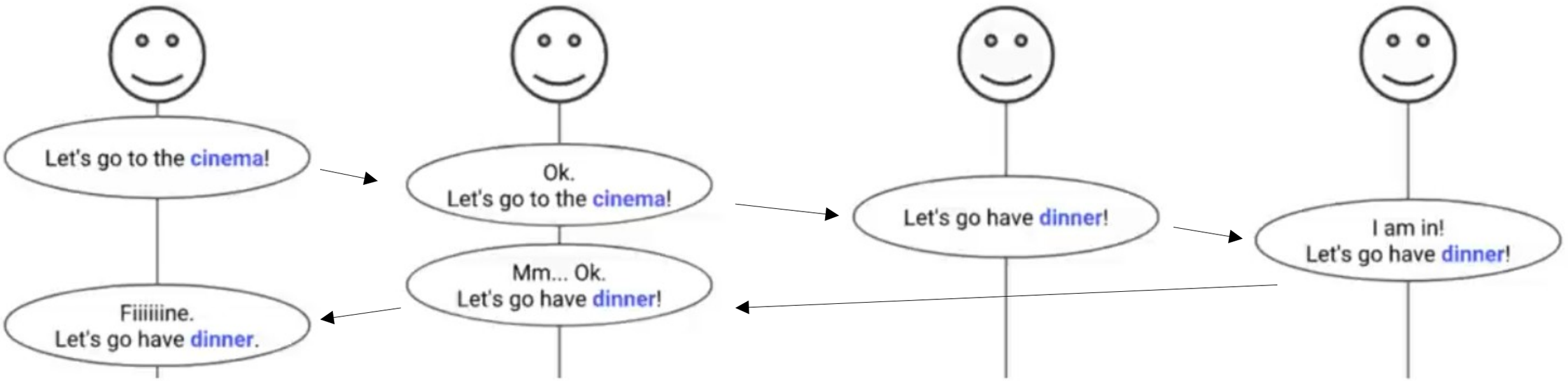
Paxos

- Paxos is a family of distributed algorithms used to reach consensus.
- There are many variants of the algorithm.
  - We present a variant called Vanilla Paxos algorithm.
- Paxos is a distributed algorithm.
  - It works in a set of computers.
  - Distributed sites running separate sequential programs to solve a common goal.
- What exactly is a consensus is presented in the next slide.

- Paxos was proposed by Leslie Lamport in the paper:
  Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.
- This presentation based on
  Luis Quesada Torres, The Paxos Algorithm, Google Techtalks, 2018.

Consensus

- What is to reach consensus with Paxos?



- Consensus is agreeing on one result.
- Once a majority agrees on a proposal, that is the consensus.
- The reached consensus can be eventually known by everyone.
- The involved parties want to agree on any result, not on their proposal.
- Communication channels may be faulty, i.e., messages can be lost.

Consensus

- Why do systems need to reach consensus?
- How to scale a system?
  - Centralized systems can not scale.
- Example1: Leader-replicas schema
  - Master is leading the selection of a new value.
  - Slaves can propose a new value to a master.
  - Master serializes the proposals.
  - Consensus needed to decide about the new master, if existing one crashes.
- Example2: Peer-to-peer schema
  - All nodes are the same.
  - When they want to propose a new value they send a proposal to each other.
  - The system must continuously reach consensus on a value.

Paxos Algorithm

- Paxos defines three roles: proposers, acceptors, and learners.
- Paxos nodes can have multiple roles, even all of them.
- Nodes must know *how many acceptors the majority* is.
  - Two majorities always have at least one node in common.
- Paxos nodes must persist.
  - They can not forget what they accepted.
- Paxos run aim at reaching a single consensus.
  - Once a consensus has been reached it can not progress to another consensus.
  - In order to reach another consensus, a different Paxos run must start (in Vanilla Paxos).

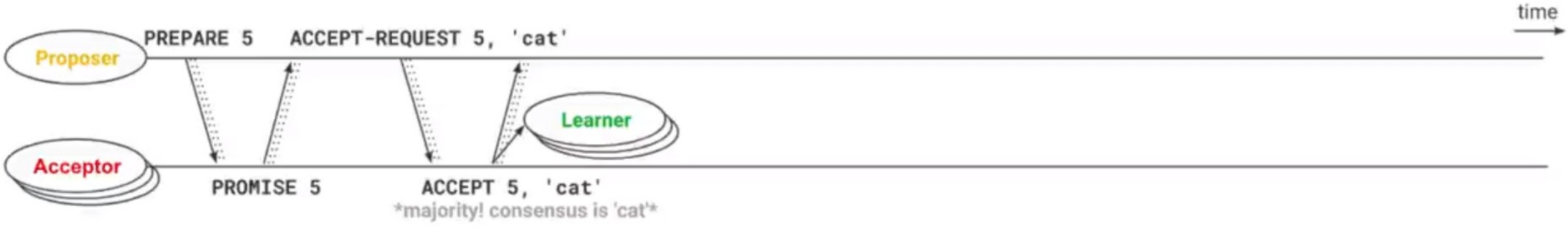
Paxos Algorithm

- Proposer wants to propose a certain value:
  - It sends PREPARE IDp to a majority (or all) of Acceptors
  - IDp must be unique, e.g., timestamp in nanoseconds
  - Timeout? Retry with a new (higher) IDp.

- Acceptor receives a PREPARE message for IDp:
- Did it promise to ignore requests with this IDp?
  - Yes → Ignore
  - No → Promise to ignore any request lower than IDp.
    (?) Reply with PROMISE Idp.

If a majority of acceptors promise, no ID < IDp can make it through. 1

Paxos Algorithm



- Proposer gets majority of PROMISE messages for a specific IDp.
- It sends ACCEPT_REQUEST IDp, VALUE to a majority (or all) of Acceptors.
  - (?) It picks any value it wants.

- Acceptor receives an ACCEPT-REQUEST message for IDp, value:
- Did it promise to ignore requests with this IDp?
  - Yes → Ignore
  - No → Reply with ACCEPT IDp, value. Also send it to Learners.

If majority of Acceptors accept IDp, value, consensus is reached.
Consensus is always on value (not necessary Idp).

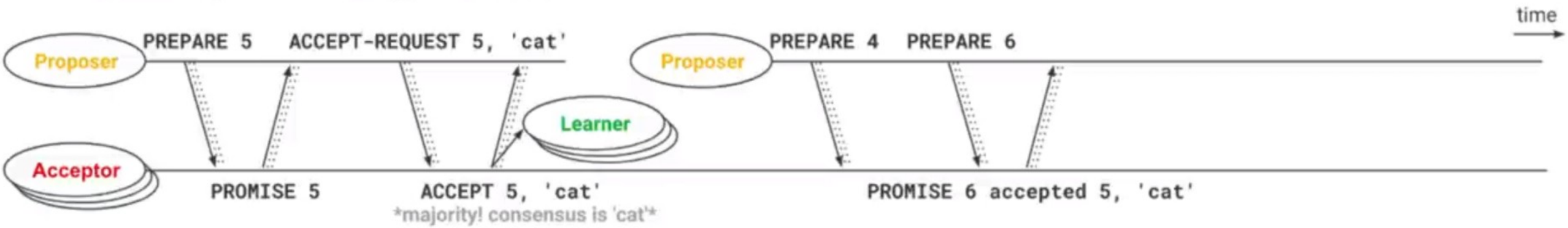
2

- Proposer or Learner get ACCEPT messages for IDp, value:

If a proposer/learner gets majority of accept for a specific Idp,
They know that consensus has been reached on value.

3

Paxos Algorithm



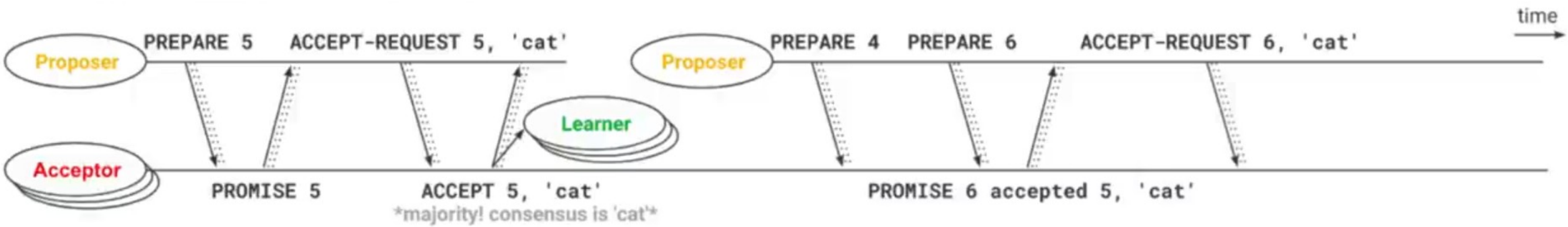
- Proposer wants to propose a certain value:
  - It sends PREPARE IDp to a majority (or all) of Acceptors
  - IDp must be unique, e.g., timestamp in nanoseconds
  - Timeout? Retry with a new (higher) IDp.

- Acceptor receives a PREPARE message for IDp:
- Did it promise to ignore requests with this IDp?
  - Yes → Ignore
  - No → Promise to ignore any request lower than IDp.
    Has it ever accepted any value? (assume accepted ID=IDa)
    Yes → Reply with PROMISE IDp accepted IDa, value.
    No → Reply with PROMISE Idp.

If a majority of acceptors promise, no ID < IDp can make it through.

1

Paxos Algorithm



- Proposer gets majority of PROMISE messages for a specific IDp.
- It sends ACCEPT_REQUEST IDp, VALUE to a majority (or all) of Acceptors.
  Has it got any already accepted value from promises?
  - Yes → It picks the value with the highest IDa that it got
  - No → It picks any value it wants.

- Acceptor receives an ACCEPT-REQUEST message for IDp, value:
- Did it promise to ignore requests with this IDp?
  - Yes → Ignore
  - No → Reply with ACCEPT IDp, value. Also send it to Learners.

If majority of Acceptors accept IDp, value, consensus is reached.
Consensus is always on value (not necessary Idp).

2

- Proposer or Learner get ACCEPT messages for IDp, value:

If a proposer/learner gets majority of accept for a specific Idp,
They know that consensus has been reached on value.

3

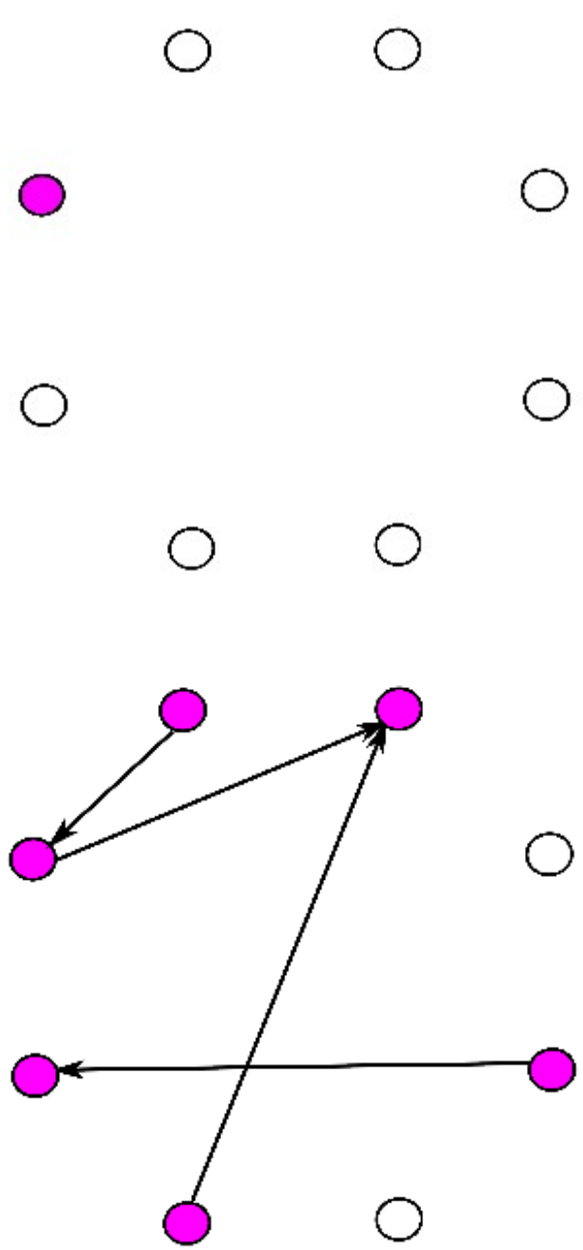
Outline

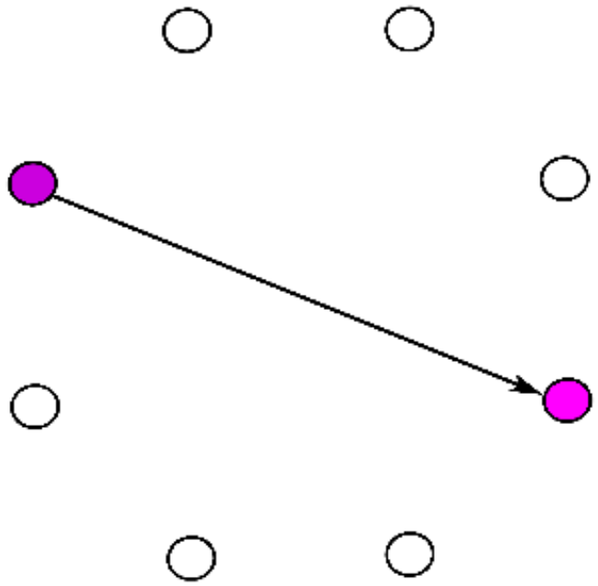
- Time and clocks
- Paxos
- Gossip
- Partitioning
- LSM tree

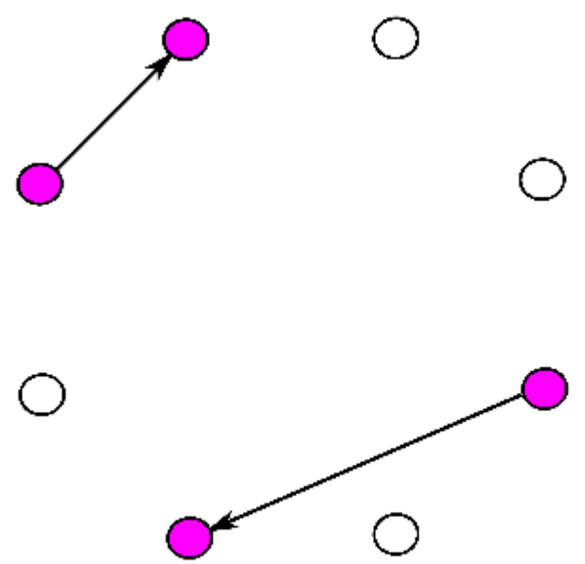
Gossip

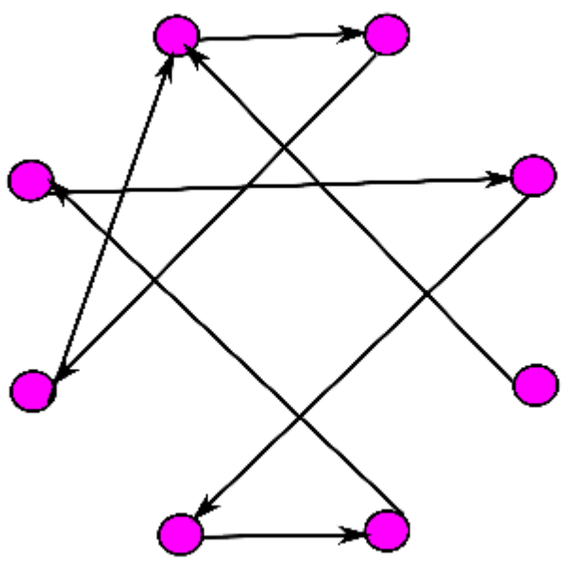
- Gossip is one method to propagate a view of cluster status.
- Every t seconds, on each node:
  - The node selects some other node to chat with.
  - The node reconciles its view of the cluster with its gossip buddy.
  - Each node maintains a “timestamp” for itself and for the most recent information it has from every other node.
- Information about cluster state spreads in O(log n) rounds (eventual consistency)
- Scalable and no SPOF, but state is only eventually consistent

Gossip









4 rounds of
the protocol

Propagate State via Gossip

- Gossip protocol operates in either
  - a state or an operation transfer model to handle read and update operations as well as replication among database nodes
- State Transfer Model
  - Data or deltas of data are exchanged between clients and servers as well as among servers
  - Database server nodes maintain vector clocks for their data and also state version trees for conflicting versions
    - Versions where corresponding vector clocks cannot be brought into a $V_A < V_B$ or $V_A > V_B$ relation
    - Clients maintain vector clocks for pieces of data they have already requested or updated
- Vector clocks are exchanged and processed in the following manner
  - Query Processing
    - When a client queries for data it sends its vector clock of the requested data along with the request
    - Server node responds with part of his state tree for the piece of data that precedes the vector clock attached to the client request and the server's vector clock
    - Client has to resolve potential version conflicts
  - Update Processing
  - Internode Gossiping

Propagate State via Gossip

- Operation Transfer Model
  - Operations applicable to locally maintained data are communicated among nodes in the operation transfer mode
    - Lesser bandwidth is consumed to interchange operations in contrast to actual data or deltas
  - Importance to apply the operations in correct order on each node
    - A replica node first has to determine a casual relationship between operations (by vector clock comparison) before applying them to its data
- In this model a replica node maintains the following vector clocks
  - V_{state} : The vector clock corresponding to the last updated state of its data.
  - V_i : A vector clock for itself where—compared to V_{state} —merges with received vector clocks may already have happened
  - V_j : A vector clock received by the last gossip message of replica node j (for each of those replica nodes).
- Exchange and processing vector clocks in
  - read-, update- and internode-messages
- Complex protocols for handling queue of (deferred) operations
  - correct (casual) order reasoned by the vector clocks attached to these operations

Outline

- Time and clocks
- Paxos
- Gossip
- Partitioning
- LSM tree

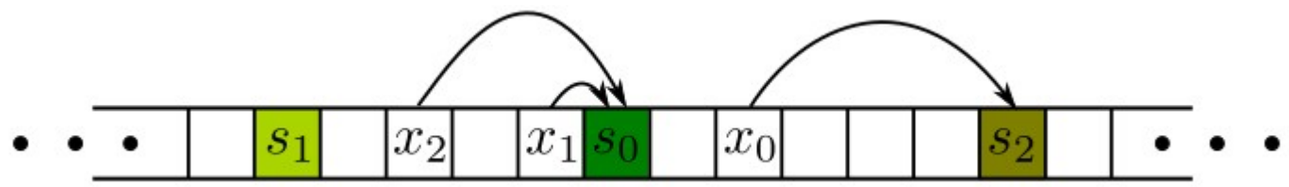
Consistent Hashing

- David Karger, Et.al., MIT, 1997
  - Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, STOC'97
- The topic is "modern"
  - It is motivated by issues in present-day systems
    - Web caching, Peer-to-peer systems, Database partitioning
  - It's general and flexible enough to be useful for other problems
- Web caching was original motivation for consistent hashing, 1997
  - Store requested pages in the local cache
    - Next time there is no need to retrieve the page the from remote source
  - Obvious benefits: less Internet traffic, faster access
  - The amount of the data used to store pages cached for 100 machines can be substantial
  - Where to store them? One machine? Cluster?
- Use hash function to map URLs to caches
  - First intuition of the computer scientist …
  - Nice properties of hash functions
    - Behaves like a random function, spreading data out evenly and without noticeable correlation across the possible buckets
  - Designing good hash functions is not easy

Consistent Hashing

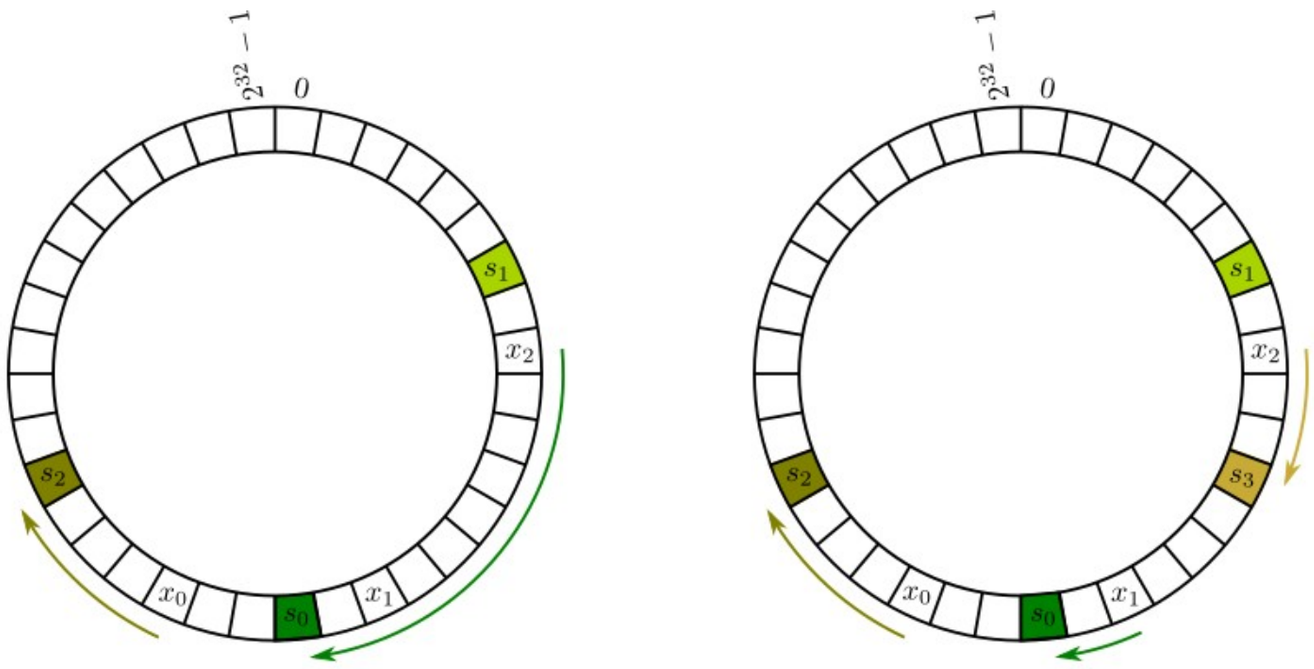
- Say there are n caches, named {0, 1, 2, ... , n−1}.
- We store the Web page with URL x at the cache server h(x) mod n
  - h(x) way bigger than n
- Solution works well if you do not change n
  - All elements would have to be re-hashed if n changes
- n changes if a cache is added or a cache fails

- Consistent hashing
- What we want?
  - hash table-type functionality and
  - changing n does not affect hashing and
  - almost all objects stay assigned to the same cache when n changes
- Leading idea: hash server names and URLs with the same hash function
- Which objects are assigned to which caches



Consistent Hashing

- Inserting a new object x to the database
  - Given an object x that hashes to the bucket h(x)
  - Imagine we have already hashed all the cache server names
  - We scan buckets to the right of h(x) until we find a bucket h(s) to which the name of some cache s hashes.
- Expected load on each of the n cache servers is exactly 1/n fraction of the objects
- Suppose we add a new cache server s — which objects have to move?
  - Only the objects stored at s.
  - Adding the n-th cache causes only a 1/n fraction of the objects to relocate
- This approach to consistent hashing can also be visualized on a circle

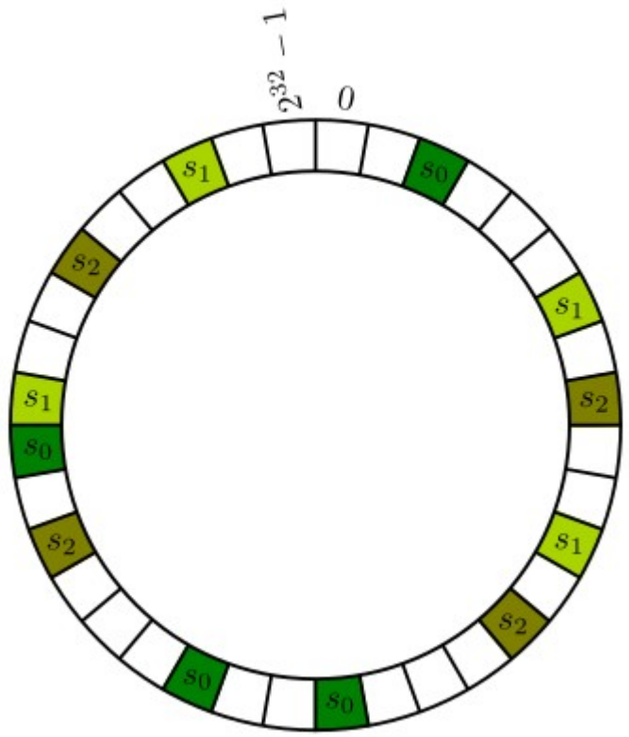


Consistent Hashing

- Standard hash table operations Lookup and Insert?
  - Efficiently implementation of the rightward/clockwise scan for the cache server s that minimizes h(s) subject to h(s) ≥ h(x)
  - We want a data structure for storing the cache names, with the corresponding hash values as keys, that supports a fast Successor operation
  - Which data structure? Given a key we need the next key and its value.
    - Not hash table, heap? How about binary search tree?
  - Complexity is O(log n)
- Reducing the variance
  - Expected load of each cache server is a 1/n fraction of the objects
  - The realized load of each cache will vary
  - An easy way to decrease this variance is to make k “virtual copies” of each cache s
    - hashing with k different hash functions to get h1(s), ... , hk(s)
    - Each server now has k copies—all of them are labeled s
    - Objects are assigned as before — from h(x), we scan clockwise until we encounter one of the hash values of some cache s
    - Choosing k ≈ log 2 n is large enough to obtain reasonably balanced loads

Consistent Hashing

- Virtual copies are also useful for dealing with heterogeneous caches that have different capacities.
  - Use the number of virtual copies of a cache server proportional to the server's capacity;



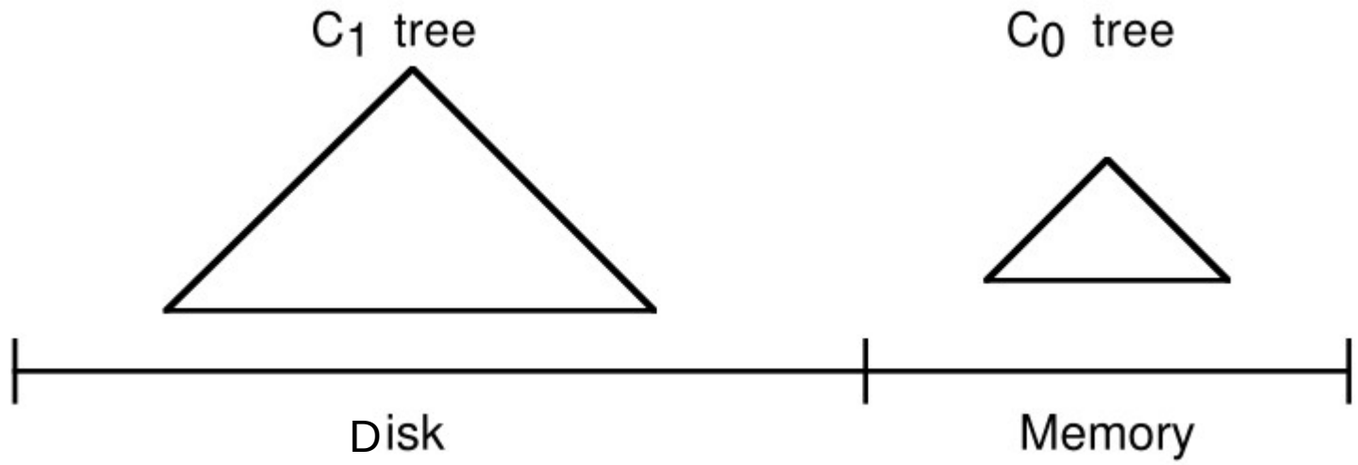
- Systems where consistent hashing is used.
  - Couchbase, Dynamo, Apache Cassandra, Voldemort, Riak, Gluster, Azure Cosmos DB, ...

Outline

- Time and clocks
- Paxos
- Gossip
- Partitioning
- LSM tree

Log structured merge-tree

- Patrick O'Neil, Et.al.: (University of Boston, DEC, Oracle)
  - The Log-Structured Merge-Tree (LSM-Tree), Acta Informatica 33, 1996.
- LSM-tree is composed of two or more tree-like component data structures.
  - Using large blocks to increase the speed of disk transfer.
- Two component LSM-tree
  - Smaller memory resident component C0 (C0 tree).
  - Larger component which is resident on disk C1 (C1 tree) with large buffer pool.
    Frequently visited pages (index-level) stay in buffer.

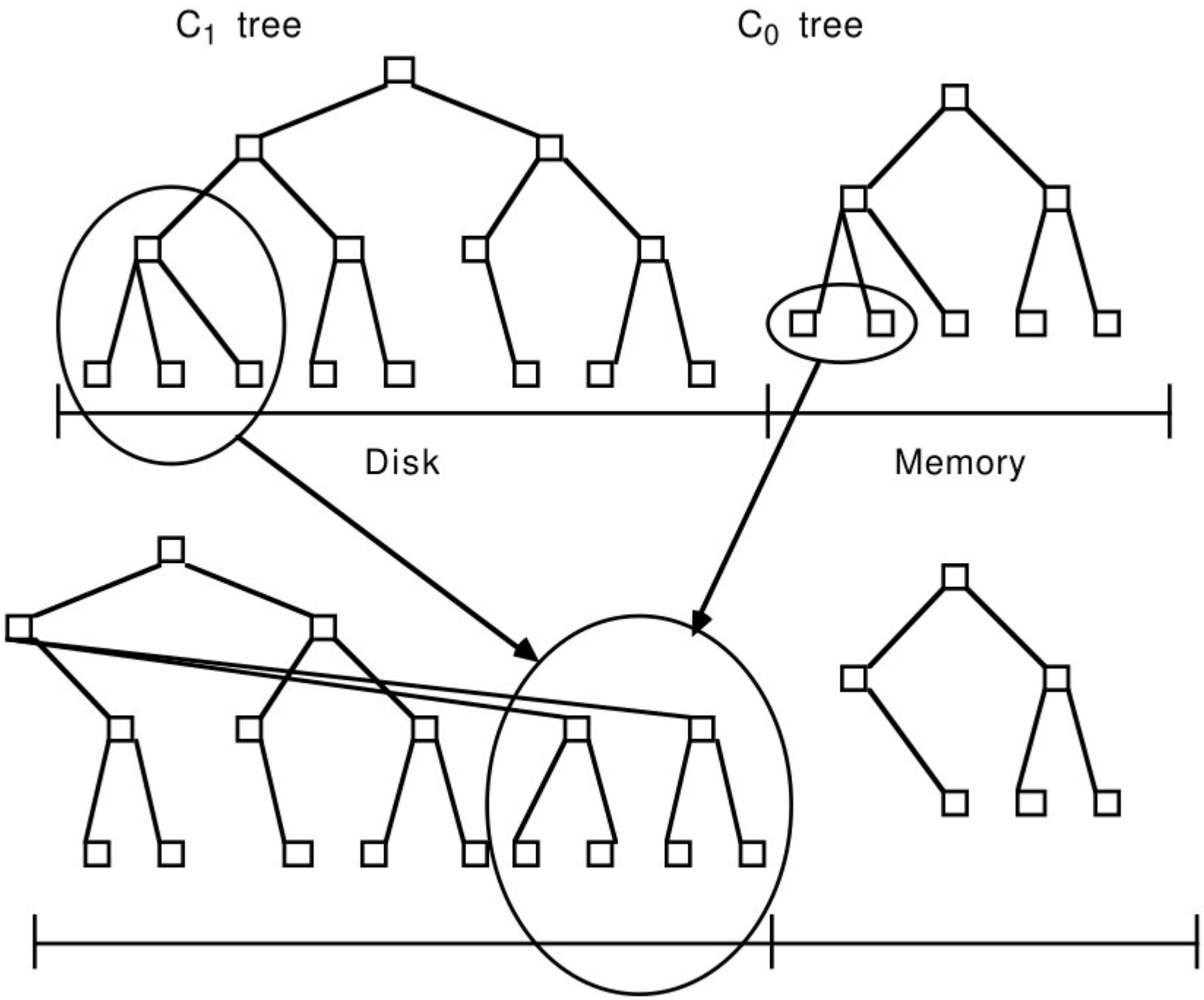


- Multi-component LSM tree
  - Limiting the cost of disk arms while reducing the size of the C0 component.
  - Separate processes doing rolling merge of trees of the increasing size.
  - High-speed rolling merge process in train between pairs of increasing trees.

Log Structured Merge-tree (LSM)

- Inserting a new index entry
  - Log record to recover this insert is first written to the sequential log file.
  - The index entry is then inserted into the C0 tree.
    - Always inserted in C0 => LSM tree has very fast insert!
    - Recall memtable in Bigtable, write buffer Dynamo (a bit different)
  - In time, records migrate out of C0 to C1 tree on disk.
- Search for an index entry will look first in C0 and then in C1.
- The C1 tree is similar to B-tree
  - Optimized for sequential disk access; nodes 100% full.
  - Seq. of pages packed together in contiguous multi-page disk blocks (for speed).
  - Multi-page block I/O (256KB) used in:
    1) rolling merge processes and 2) for long range retrievals.
  - Single-page nodes are used for matching indexed finds to minimize buffering.
- Rolling merge process
  - When the C0 tree reaches a threshold size, delete some contiguous segment of entries from the C0 tree and merge it into the C1.
  - The rolling merge acts in a series of merge steps.

Log structured merge-tree



Log structured merge-tree

- Emptying block: a multi-page block (leaf nodes) of the C1 tree read into the buffer pool.
- Filling block: merge pages from emptying block with the leaf pages of C0.
- When filling block is full, write it to new free area on the disk.
- Then the next block of pages is read and merged with next block of C0.
- When we reach the maximum values then the rolling merge starts again from the smallest values.
- Newly merged blocks are written to new disk positions.
  - Old blocks are not over-written and are available for recovery in case of a crash.
  - Parent index (directory) nodes in C1 (buffered) are updated to reflect new leaf structure (usually remain in the buff for long time).
  - Old leaf nodes from the C1 component are invalidated after the merge step is complete and are deleted from the C1 directory.

Metamorphosis of an LSM-tree

1) First insertions to the C0 tree component in memory.
   - C0 does not have B-tree structure. Can be AVL or 2-3 tree. Nodes can be of any size.
   - When the C0 size threshold reached then leftmost sequence of entries is cut (efficient batch) and reorganized in a C1 tree leaf node (100% full).
   - Successive leaf nodes are stored (from left-to-right) in a multi-page block.

2) C1 tree management
   - … then the multi-page block is written to disk as the leaf level of C1.
   - A directory structure is created (in buffer) for each leaf node entered.
     - Recall the structure of the Google's SSTable
     - Index of a SSTable tablet is stored at the end of the tablet.
   - C1 tree directory nodes are maintained in separate multi-page block buffers, or in single page buffers (whichever is better for disk arm cost).
     - Single page index nodes provide efficient exact-match access.
     - Multi-page blocks used for the rolling merge or for long range retrievals.
   - After the rightmost leaf entry of C0 is written out to C1 (for the first time) the process starts over on the left end of the two trees.
   - Now (and in further cycles) multi-page leaf-level blocks of C1 are merged with the entries in C0!

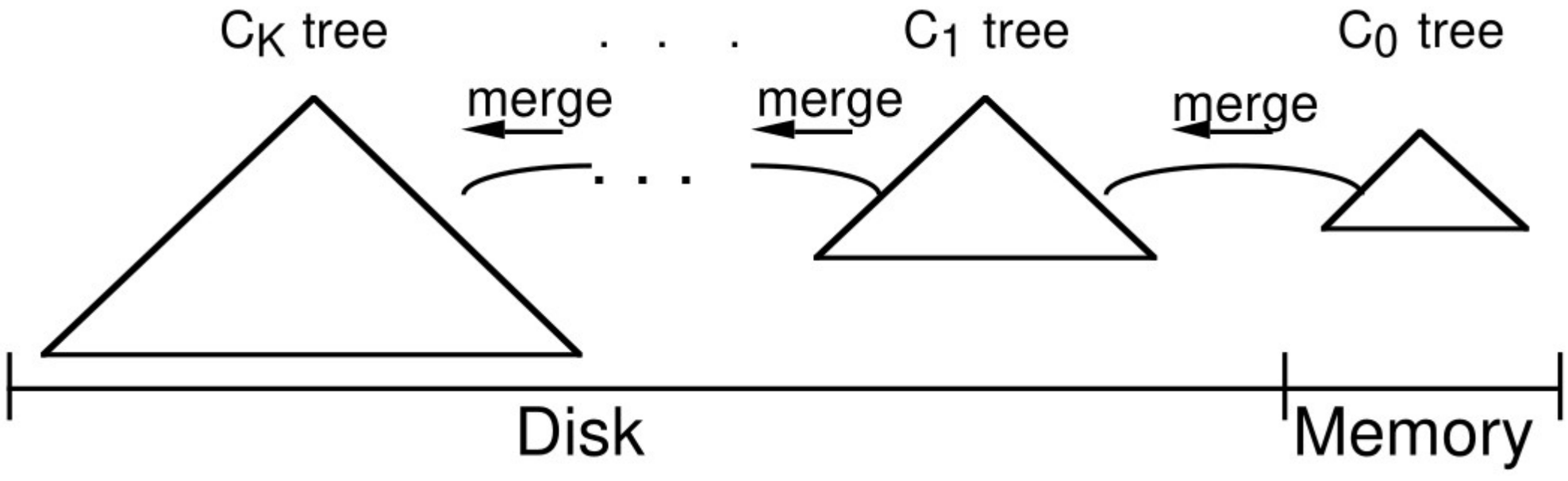
Metamorphosis of an LSM-tree

3) Rolling merge process

- Cursor slowly circulates in quantized steps through equal key values of C0 and C1 drawing data entries out from C0 to C1.
- Rolling process takes place at leaf level and on the higher directory levels.
- At each level, multi-page blocks of C1 are split into two blocks: emptying block and the filling block.
  - Process is reading from emptying block and from C0, and writing to filling block.
  - During the merge step concurrent access to the entries on those nodes is blocked.
- When the filling block in buffer on some level of C1 is full, it is flushed to disk (to a new position!).
  - Old information might be needed during recovery. It is invalidated (new writes include more up-to-date information).
- Cyclically: complete flush of all buffered nodes to disk:
  - All buffered information at each level written to new positions on disk (+ a sequential log entry).

Multi component LSM-tree

- When the size of LSM tree increases the size of C0 has to be increased to retain the speed of processing.
  - This works up to a point when the size of C0 is too large.
  - Blocks read from C1 do not intersect well (increasingly) with C0.
- Multi component LSM-tree has components C0, C1, C2, ... and CK .
  - LSM trees are of increasing size; C1-CK are disk resident.
  - Under pressure from inserts, there are asynchronous rolling merge processes in train between all component pairs (Ci-1,Ci).
- Moving data from Ci-1 to Ci each time Ci-1 exceeds its threshold size.



Exact match finds

- First search C0 tree and then C1,C2, …, CK.
  - 2 component LSM tree has slight CPU overhead compared to the B-tree.
- Worst case: exact-match find, or range find have to access all Ci-s.
- There are a number of possible optimizations.
  - (Note) unique index values are guaranteed; timestamp is part of a key!
  - *Optim. 1*: (timestamps guaranteed distinct; no duplicates) search complete if key found in early Ci.
  - *Optim. 2*: when the find criterion uses recent timestamp values; entries have not migrated yet out to the largest components.
    - Recent entries are retained in C0 (Ci) during a rolling merge; older entries go out, eventually.
    - The same happens in rolling merge processes (recent stay, older go).
  - If the most frequent searches are to recently inserted values, many finds can be completed in C0.

- Systems that use LSM tree
  - Apache AsterixDB, Bigtable, HBase, LevelDB, SQLite4, Tarantool, RocksDB, WiredTiger, Apache Cassandra, InfluxDB and ScyllaDB.

Literature

- Christof Strauch, NoSQL Databases. www.christof-strauch.de/nosqldbs.pdf
- Ho, Ricky: NOSQL Patterns. November 2009. horicky.blogspot.com/2009/11/nosql-patterns.html
- Lipcon, Todd: Design Patterns for Distributed Non-Relational Databases, Presentation, 2009. static.last.fm/johan/nosql-20090611/intro_nosql.pdf
- Karger, David, Et.Al., Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, ACM symposium on Theory of computing, 1997
- Microsoft Azure, Sharding pattern, Jun 2017. docs.microsoft.com/en-us/azure/architecture/patterns/sharding
- Tim Roughgarden, Gregory Valiant, CS168: The Modern Algorithmic Toolbox, Lecture #1: Introduction and Consistent Hashing, Stanford 2017.

Literature

- Patrick O'Neil, Et.al. (University of Boston, DEC, Oracle), The Log-Structured Merge-Tree (LSM-Tree), Acta Informatica, 1997.
- Lamport, L., Time Clocks Ordering, CACM, 1978.
- Cauchbase, Optimistic or pessimistic locking – Which one should you pick?. https://blog.couchbase.com/optimistic-or-pessimistic-locking-which-one-should-you-pick/