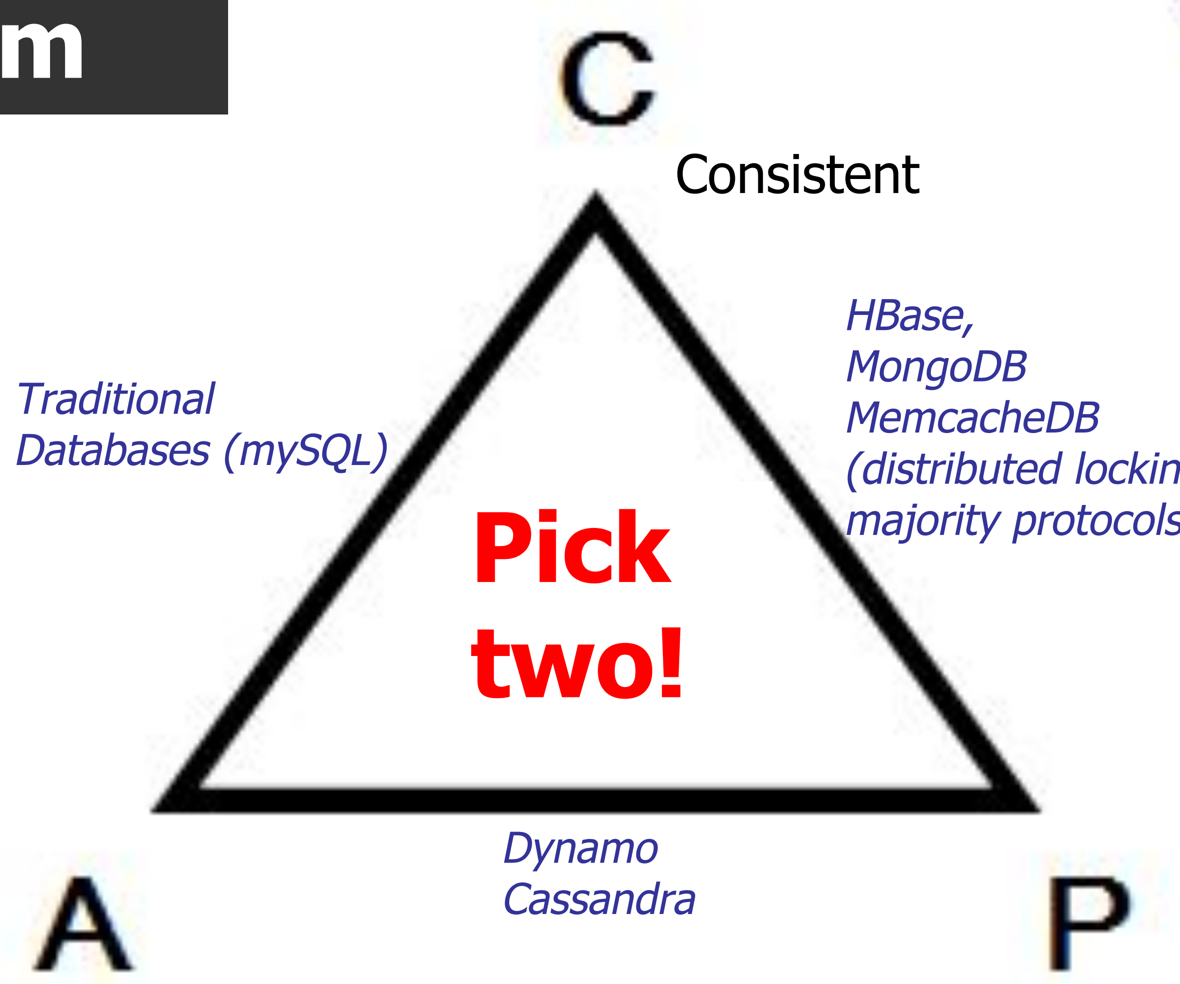
CAP Theorem



C

Consistent

HBase,
MongoDB
MemcacheDB
(distributed locking,
majority protocols)

Traditional
Databases (mySQL)

Pick two!

Dynamo
Cassandra

A

Availabile

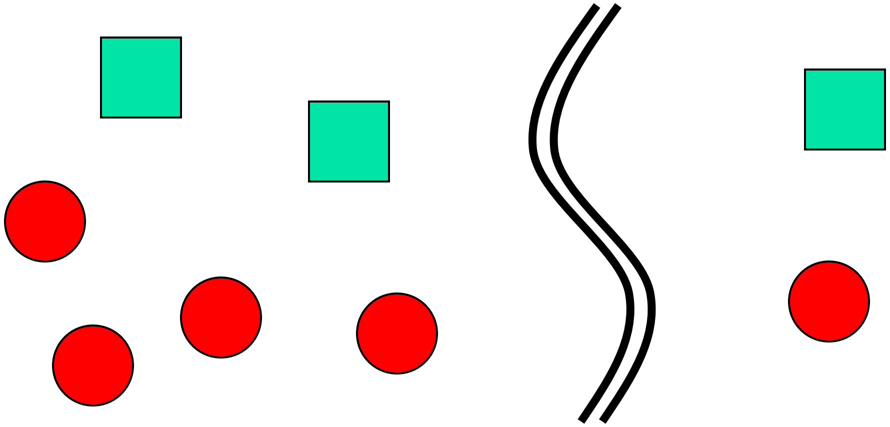
P

Partition tolerant

CAP Twelve Years After – How the "Rules" Have Changed, IEEE Spectrum 2012 [link]
Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33(2): 51-59 (2002)

🟩 : Replicas

● : Clients



C+PT (-A)

Option 1: reads

accept reads

accept

reject writes

reject

Option 2:

writes

accept reads

reject

reads

accept writes

reject

writes

A+PT (-C)

writes

accept reads + writes

accept reads +

‘inconsistent’ results

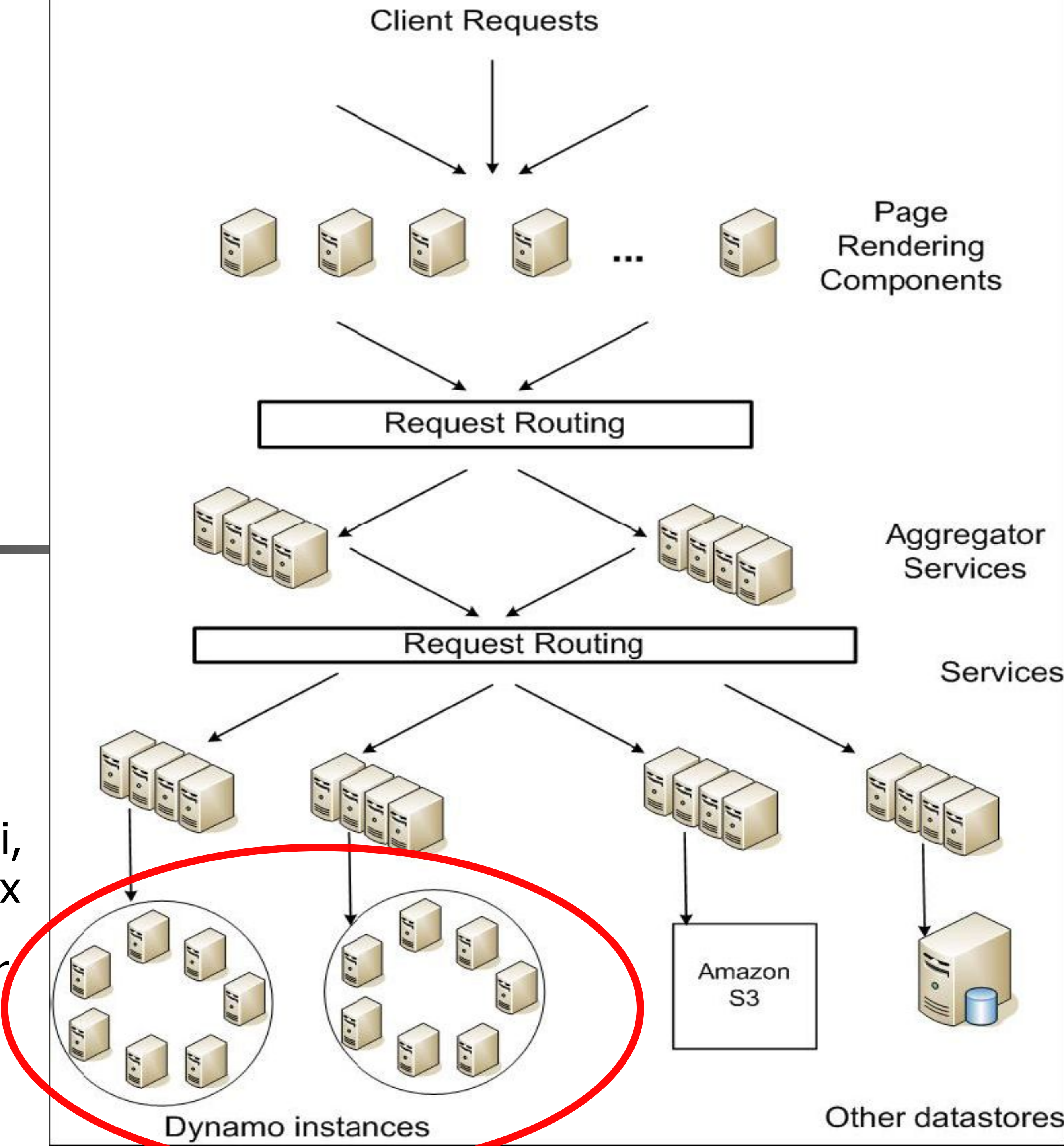
‘inconsistent’

results

Dynamo: Amazon's Highly Available Key-value Store

(SOSP'07)

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels



Amazon eCommerce platform

Throughput & Scale:

- 80M checkout operations per day (peak season) [2017]

Problem: availability/reliability at massive scale:

- Slightest outage has significant financial consequences
  - $1M/minute (2021)
  - ….and impacts customer trust
- At this scale component-level outages are continuous!

Amazon eCommerce platform - Requirements

Main application requirements:

- Key issue: data/service availability.
  - Particularly for writes: “Always writeable” data-store
- Low latency – delivered to (almost) all clients/users
  - Example SLA: provide a 300ms response time, for 99.9% of requests, for a peak load of 500requests/sec.
  - Why not average/median?

Architectural requirements

- Incremental scalability
- Symmetry (no ‘special’ node)
- Ability to run on a heterogeneous platform

Data Access Model

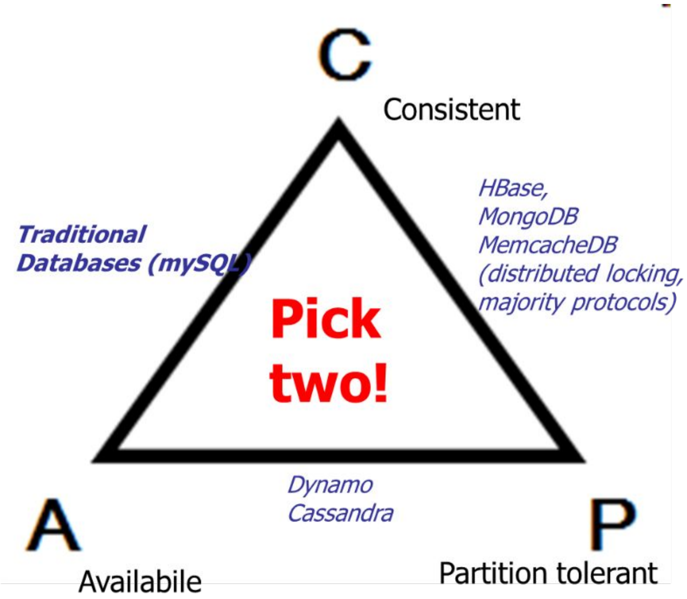
- Data stored as (key, object) pairs:
  - Interface *put(key, object), get(key)*
    - 'identifier' (key) generated as a hash for object
  - Objects: Opaque

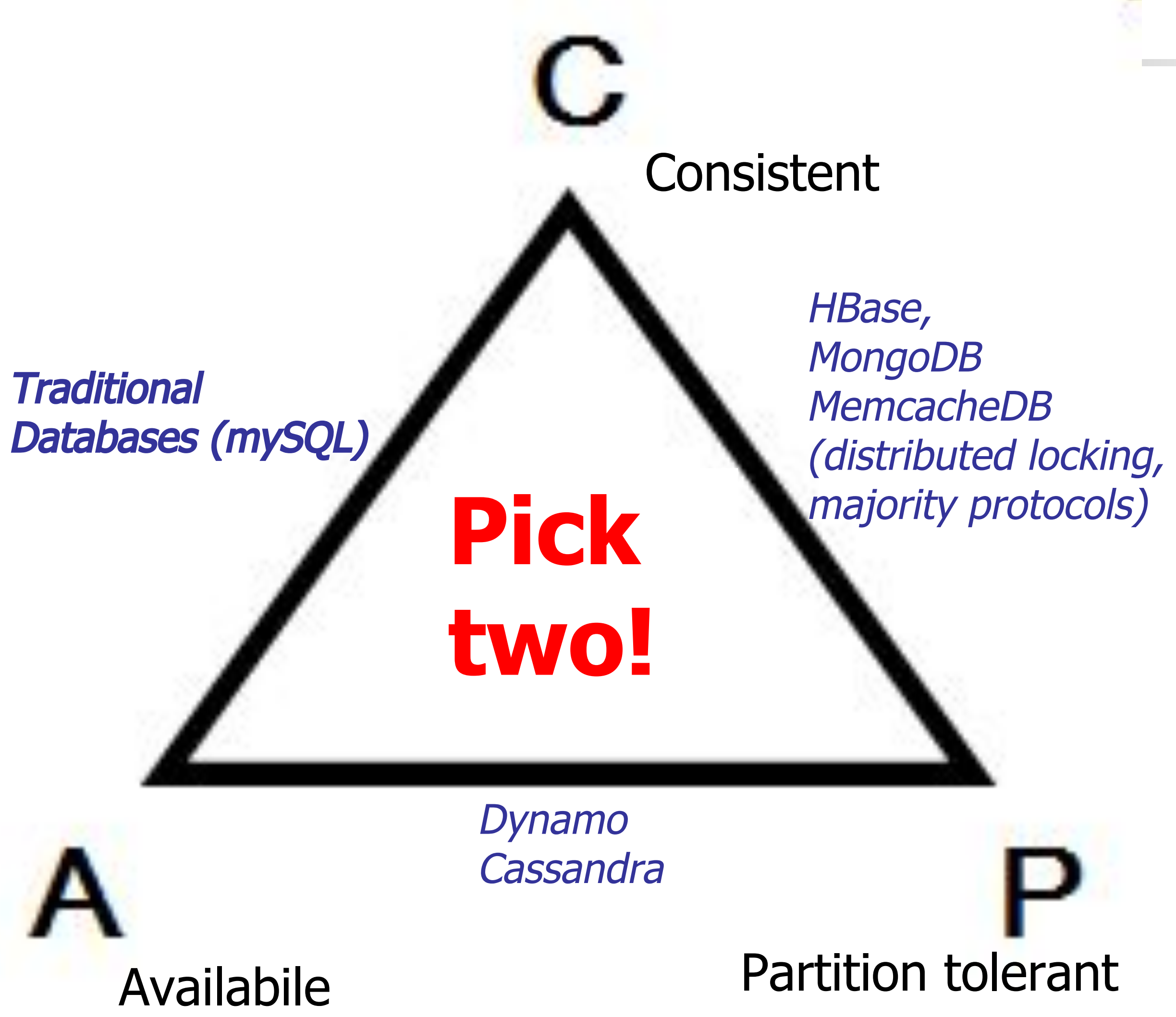
Application examples: shopping carts, customer preferences, session management, sales rank, product catalog, S3

Further assumptions:

- Relatively small objects (<1MB)
- Query by objectID only
- Operations do not span multiple objects
- Friendly (cooperative) environment
- One Dynamo instance per service □ 100s to 1,000s hosts/service

Why not a database?





Consistent

Traditional Databases (mySQL)

HBase,
MongoDB
MemcacheDB
(distributed locking, majority protocols)

Pick two!

Dynamo
Cassandra

Availabile

Partition tolerant

“Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”. SIGACT News 33(2): 51-59 (2002)

Requirements for Dynamo:

- (1) High data *availability*; (2) always writeable data store

.

Favor availability
(vs. consistency)

Favor writes
(vs. reads)

Why favor availability over consistency?

"even the slightest outage has significant financial consequences and impacts customer trust"

- … consistency violations may as well have financial consequences and impact customer trust
  - But not in (a majority of) Amazon's services
  - NB: Billing is a separate story

Requirements: High availability / always writeable data store

Key ideas

- Multiple replicas …
- … but avoid <u>*synchronous*</u> replica coordination …

  [used by solutions that provide strong consistency]
- Tradeoff: Consistency □□ Availability *(maintain part. tolerance)*
- … 'weak consistency' makes it possible to provide availability
- However need subsequent decisions:
  - **WHEN** to resolve possible consistency conflicts,
    - Dynamo: at read time (allows providing an "always writeable" data store)
  - **WHO** should solve them
    - Dynamo: the data store [if it can], **OR** [if that fails] the application (<u>configurable, app specific</u>)

Key issues

- Partitioning the key/data space. Request routing
- High availability for writes
- Handling temporary node failures
- Recovering from permanent failures
- Membership and failure detection

Things to remember from last time …

- Vector clocks
  - Can be used to model causality dependence.

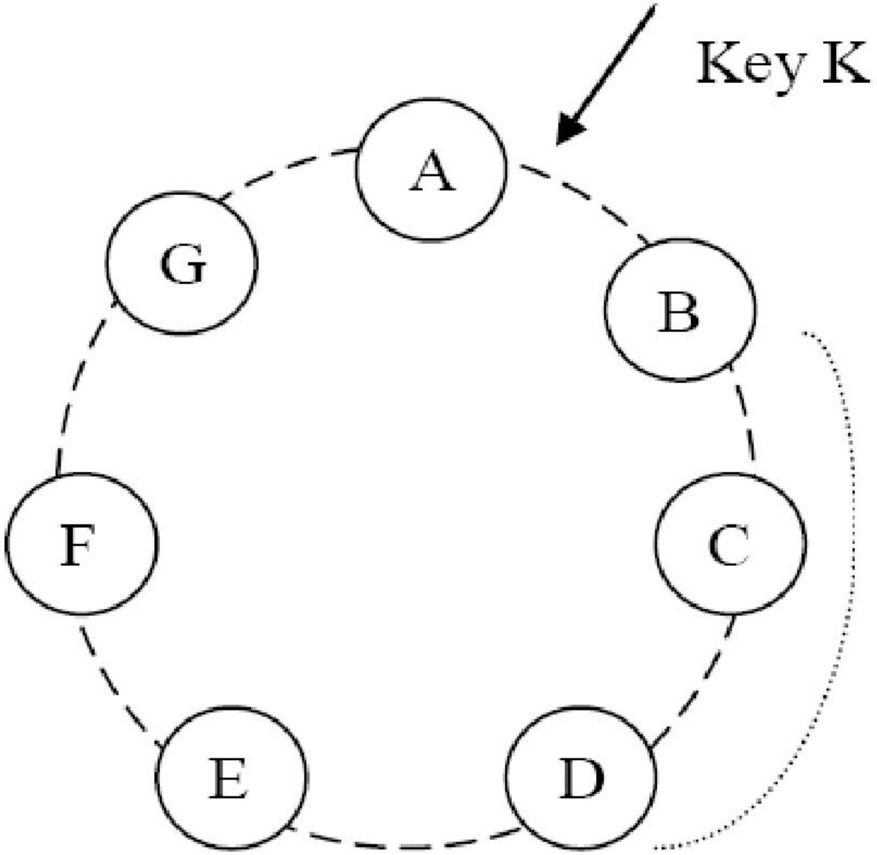
    So far two usage examples
    - [Causally ordered] Group communication
    - [Replica management] Determine replica divergence and the causally related stream of updates

- Voting-based / Quorum-based protocols
  - More examples later

Problem	Technique	Advantage
Partitioning / Request routing	▪ Consistent hashing	Incremental scalability, load balancing, etc.
High availability for writes	▪ Eventual consistency ▪ Reconciliation during reads (uses vector clocks) ▪ Quorum protocol	Availability
Handling temporary failures	▪‘Sloppy’ quorum protocol and hinted handoff	Provides availability and durability when some of the replicas are not available.
Recovering from permanent failures	▪ Anti-entropy using Merkle trees	Synchronizes divergent nodes in the background.
Membership and failure detection	▪ epidemic-based membership protocol	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partition Algorithm: Consistent hashing

- Each data item is replicated at N hosts (successors)



Nodes B, C and D store keys in range (A,B) including K.

Problem	Technique	Advantage
Partitioning / Request routing	▪ Consistent hashing	Incremental scalability, load balancing, etc.
High availability for writes	▪ Eventual consistency ▪ Reconciliation during reads (uses vector clocks) ▪ Quorum protocol	Availability
Handling temporary failures	▪'Sloppy' quorum protocol and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	▪ Anti-entropy using Merkle trees	Synchronizes divergent nodes in the background.
Membership and failure detection	▪ epidemic-based membership protocol	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Quorum systems

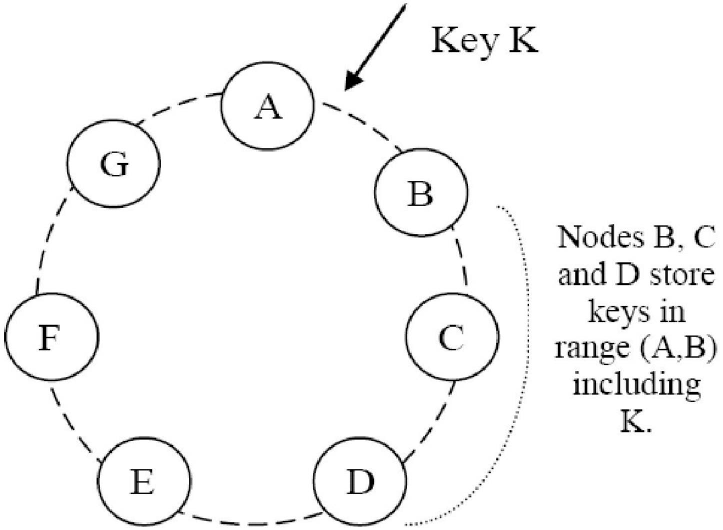
- Multiple replicas to provide data durability (and availability) …
  - … but avoid synchronous replica coordination …

Traditional quorum system:

- R/W: the minimum number of nodes that must participate in a successful read/write operation.
  - Latency of a read/write operation: the slowest of the R (or W) replicas
  - To improve latency R and W are usually configured to be less than N.
- R + W > N and W > N/2 yields a quorum-like system.
  - 'Sloppy quorum' in Dynamo: sometimes these are violated

A second factor for inconsistencies: Dealing with temporary failures through "Hinted handoff"

- Assume replications factor R = 3.
- Replicas on next R nodes.
- When A is temporarily down
  - (or unreachable) during a write,
  - send the write to D.
- D is 'hinted' that the replica belongs to A and will deliver it A when A recovers.



- Objective: "always writeable"

Data versioning

- Multiple replicas …
  - … but with focus on availability they may diverge
- The issues this introduces:
  - when to resolve possible conflicts?
    - Dynamo’s solution: at read time (allows providing an “always writeable” data store)
      - A *put()* may return before the update has been applied at all the replicas
      - A *get()* call may return different versions of the same object.
  - who should solve them?
    - the data store □ use *vector clocks to capture causality* between different versions of the same object.
    - the application □ uses application specific logic.

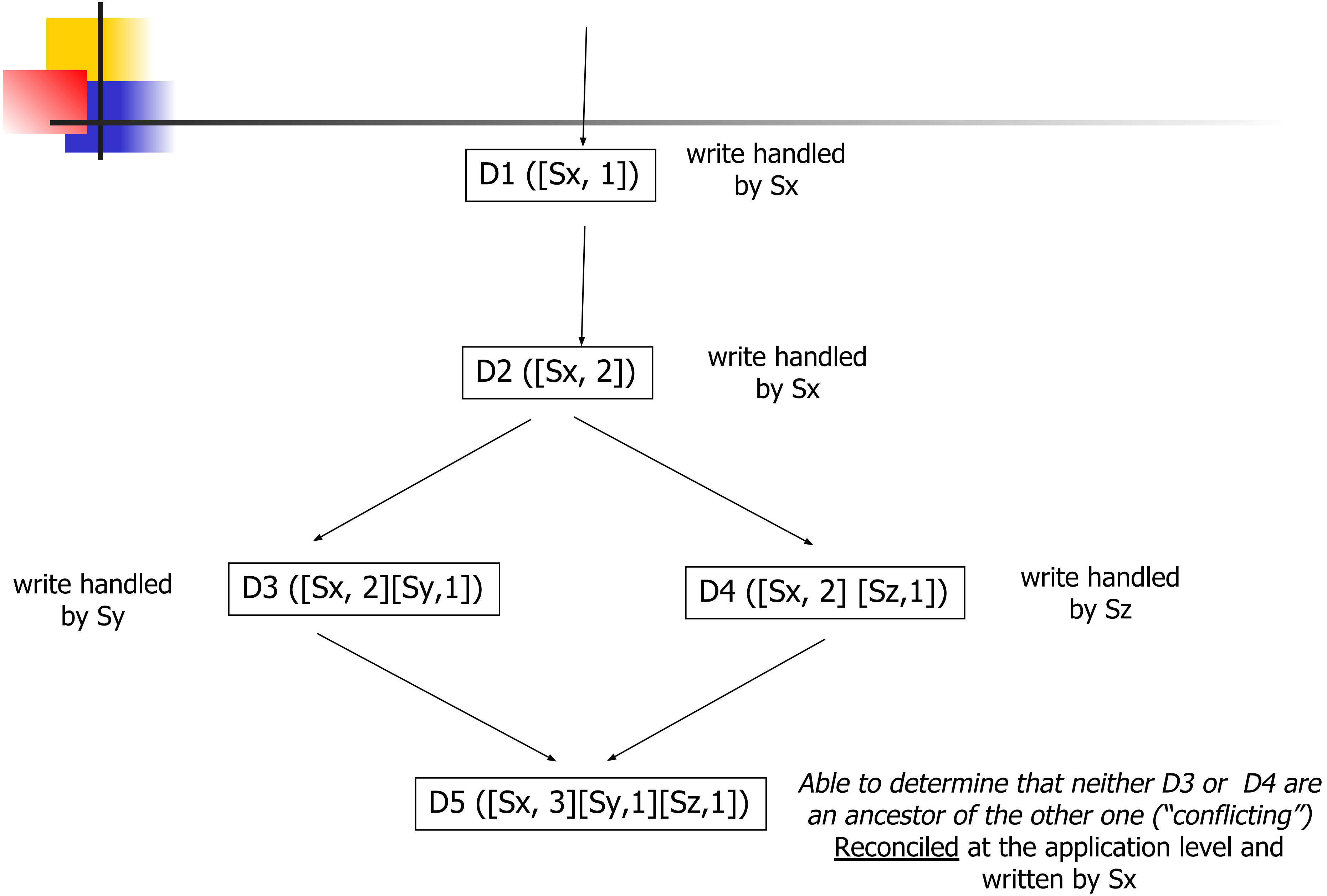
How does the data-store solve conflicts: Vector Clocks?

Each version of each object has one associated vector clock.

- list of (node, counter) pairs.

Data store: Which version to keep V’ or V”?

- If V’ is a direct ancestor of V” then keep V”
  - direct ancestor: each counter on the V’ vector clock is each less-or-equal than corresponding counter in V”
- Otherwise: application-level reconciliation



D1 ([Sx, 1])

write handled
by Sx

D2 ([Sx, 2])

write handled
by Sx

write handled
by Sy

D3 ([Sx, 2][Sy,1])

D4 ([Sx, 2] [Sz,1])

write handled
by Sz

D5 ([Sx, 3][Sy,1][Sz,1])

*Able to determine that neither D3 or D4 are
an ancestor of the other one ("conflicting")*
Reconciled at the application level and
written by Sx

Divergent versions rarely created in practice

1 version □ 99.94%

2 versions □ 0.0057%

3 versions □ 0.00047%

4 versions □ 0.00007%

Live production environment; % versions reconciled using application logic

Source: Clients with high volume of concurrent writes (not failures)

… these may be robots

Problem	Technique	Advantage
Partitioning	▪ Consistent hashing	Incremental scalability, load balancing, etc.
High availability for writes	▪ Eventual consistency ▪ ***Vector clocks with reconciliation during reads*** ▪ Quorum protocol	Availability
Handling temporary failures	▪ 'Sloppy' quorum protocol and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	▪ Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	▪epidemic-based membership protocol	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

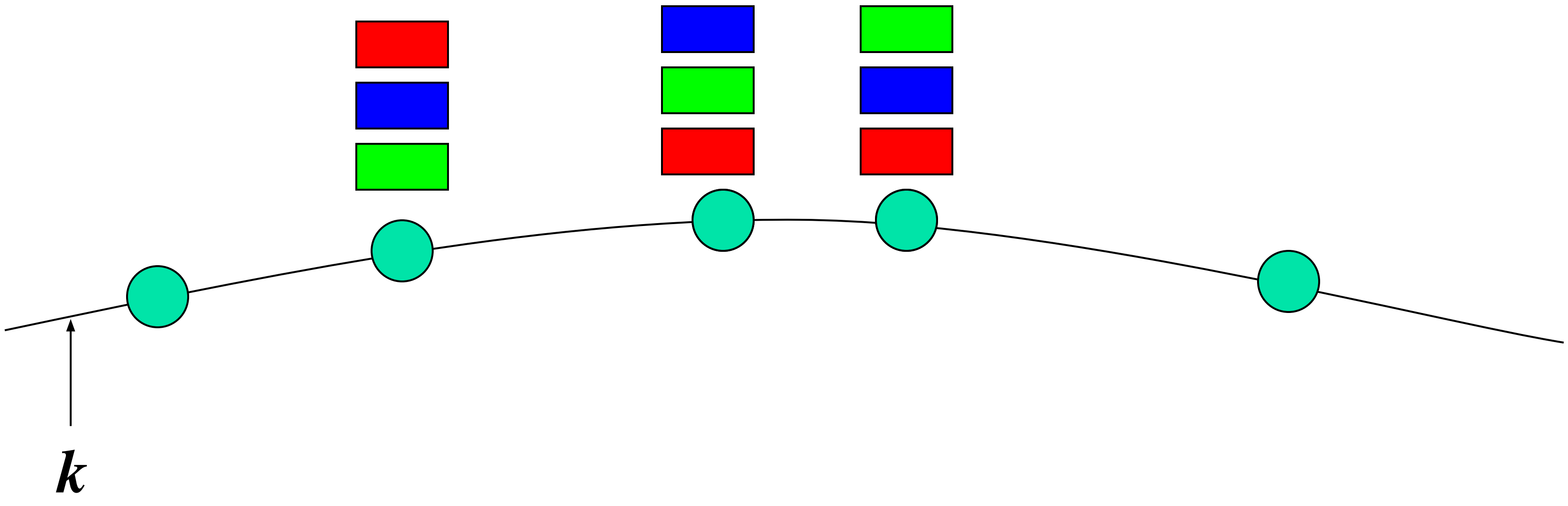
Problem	**Technique**	**Advantage**
Partitioning	▪ Consistent hashing	Incremental scalability, load balancing, etc.
High availability for writes	▪ Eventual consistency ▪ Vector clocks with reconciliation during reads ▪ Quorum protocol	Availability
Handling temporary failures	▪ ‘Sloppy’ quorum protocol and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	▪ **Anti-entropy using Merkle trees**	Synchronizes divergent replicas in the background.
Membership and failure detection	▪epidemic-based membership protocol	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Other techniques

- Node synchronization:
  - Merkle hash tree.

Reconciliation

- Dynamo will replicate each data item on N successors
  - A pair (*k*,*v*) is stored by the node closest to *k* and replicated on N successors of that node
- Why is this may be hard?
  - As nodes may be slow, fail temporarily …sloppy quorum, hinted handoff
  - Need to reconciliate keysets



Replication Meets Epidemics

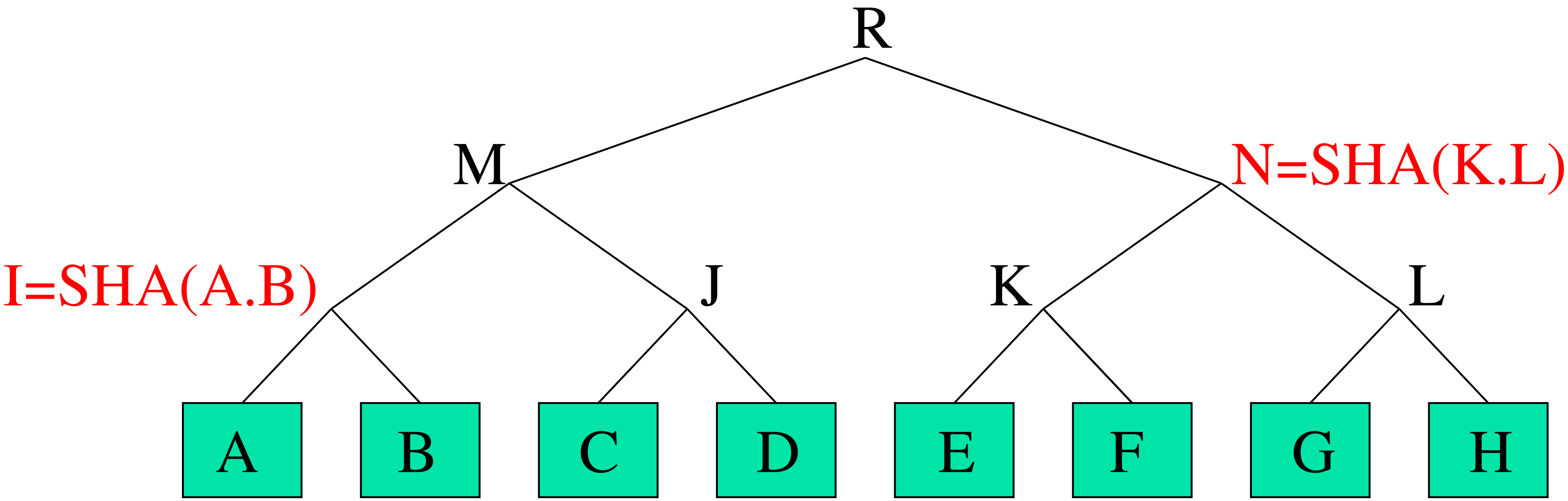
Goal: synchronize the key/value pairs on two nodes

- Candidate Algorithm
  - For each (*k*,*v*) stored locally, compute SHA(*k.v*)
  - Every period, pick a random leaf-set neighbor
  - Ask neighbor for all its hashes
  - For each unrecognized hash, ask for key and value
    - (if need to reconcile use vector timestamps to reason about version freshness)

- This is an epidemic algorithm All *N* members will have all (*k*,*v*) in *log(N)* periods
  - But (above) the cost is O(*C*), where *C* is the size of the set of items stored at the original node

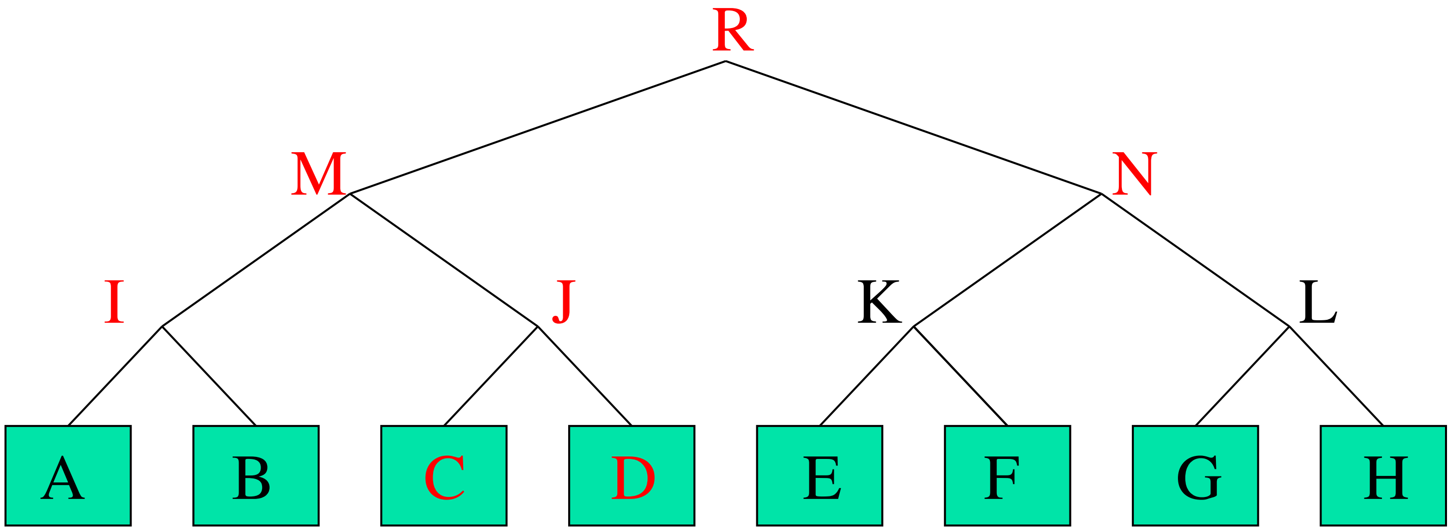
Merkle Trees

- An efficient summarization technique
  - Interior nodes are the secure hashes of their children
  - E.g., I = SHA(A.B), N = SHA(K.L), etc.



Merkle Trees

- Merkle trees are an efficient summary technique
  - If the top node is signed and distributed, this signature can later be used to verify any individual block, using only O(log *n*) nodes, where *n* = # of leaves
  - E.g., to verify block C, need only R, N, I, C, & D



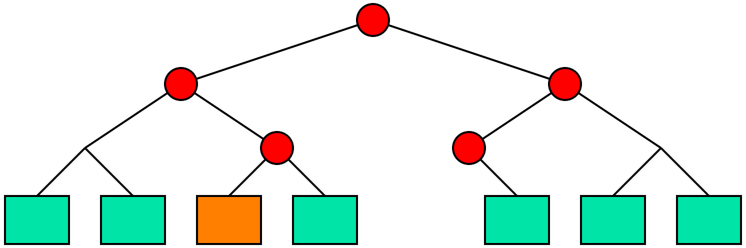
One use: enables client to verify integrity of data stored in a cloud

Using Merkle Trees as Summaries

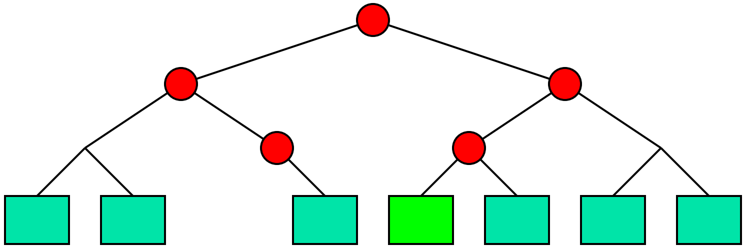
- Use Merkle tree to accelerate set comparison (and identify differences)
  - B gets tree root from A, if same as local root, done
  - Otherwise, recurse down tree to find difference

[assumption: sets diverge little]

A’s values:



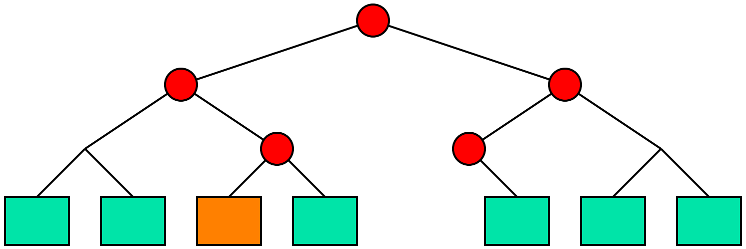
B’s values:



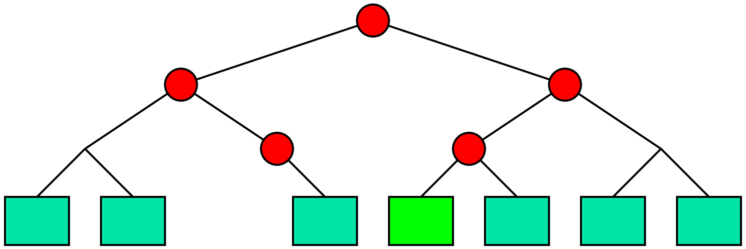
Using Merkle Trees as Summaries

- Use Merkle tree to accelerate set comparison (and identify differences)
  - B gets tree root from A, if same as local root, done
  - Otherwise, recurse down tree to find difference
- New cost is O(*d* log *C*)
  - *d* = number of differences, *C* = size of disk

A's values:

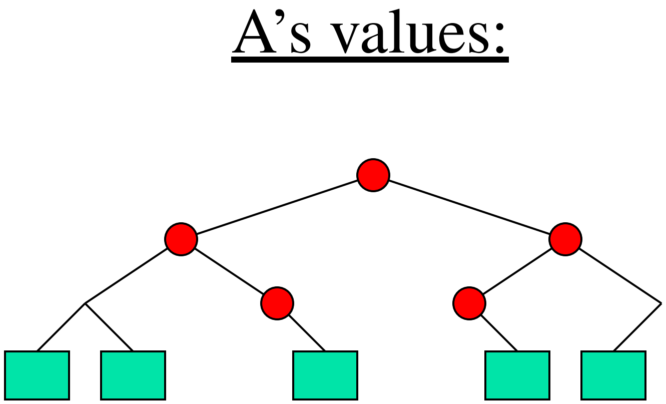


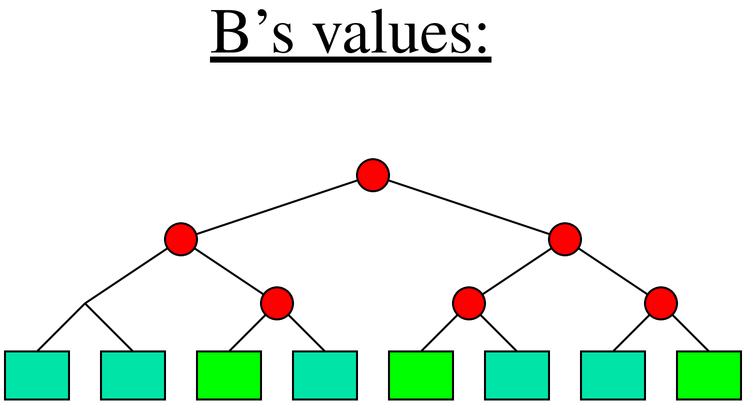
B's values:



Using Merkle Trees as Summaries

- Still too costly:
  - If A is down for an hour, then comes back, changes will be randomly scattered throughout tree

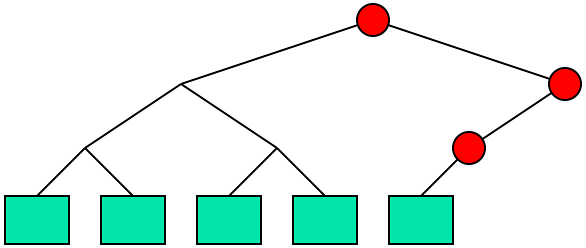




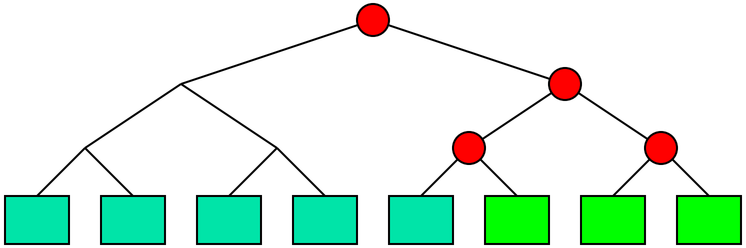
Using Merkle Trees as Summaries

- Still too costly:
  - If A is down for an hour, then comes back, changes will be randomly scattered throughout tree
- Solution: order values by time instead of hash
  - Localizes values to one side of tree

A’s values:



B’s values:



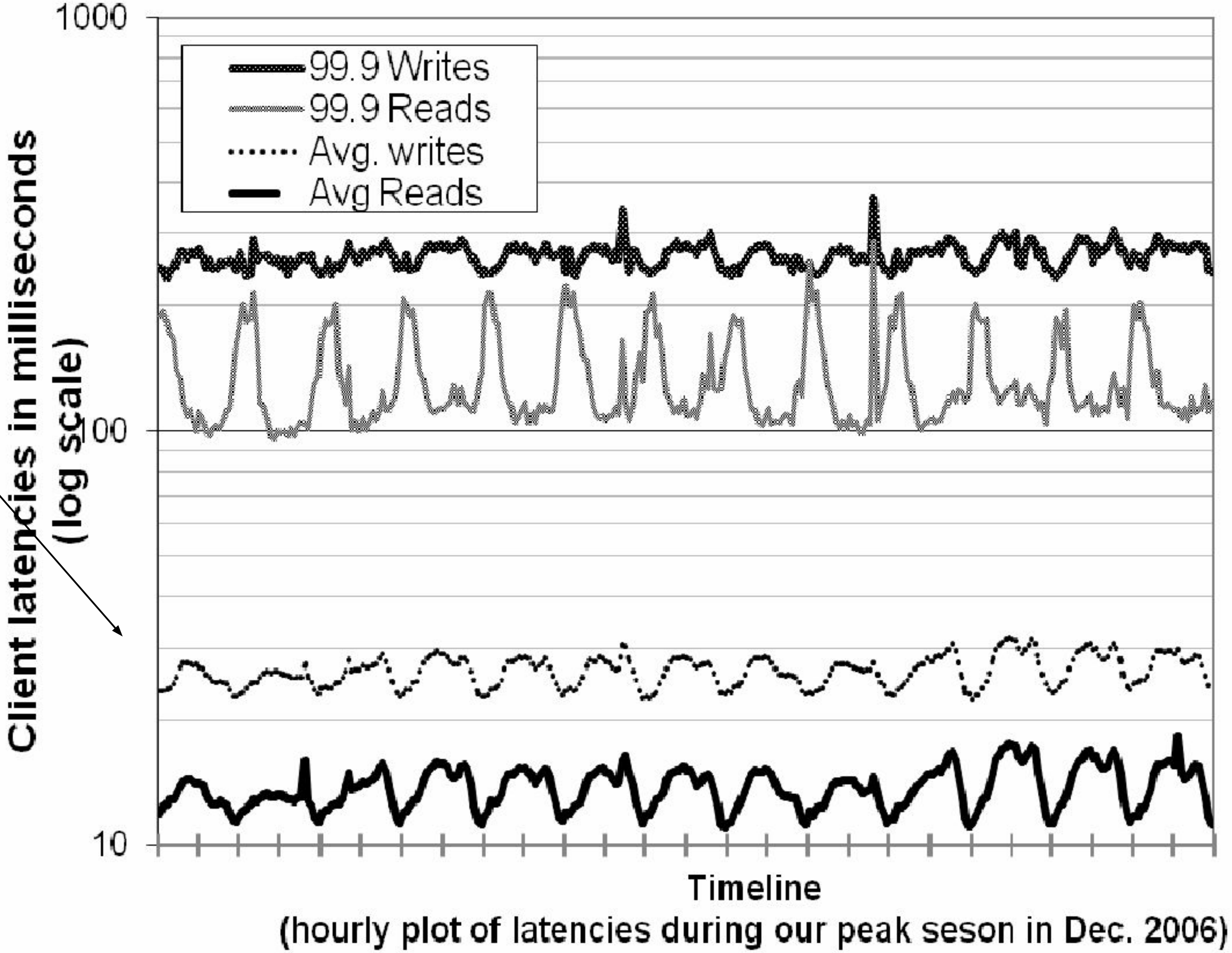
Problem	**Technique**	**Advantage**
Partitioning	▪ Consistent hashing	Incremental scalability, load balancing, etc.
High availability for writes	▪ Eventual consistency ▪ Vector clocks with reconciliation during reads ▪ Quorum protocol	Availability
Handling temporary failures	▪ ‘Sloppy’ quorum protocol and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	▪ **Anti-entropy using Merkle trees**	Synchronizes divergent replicas in the background.
Membership and failure detection	▪ epidemic-based membership protocol	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Dynamo Implementation

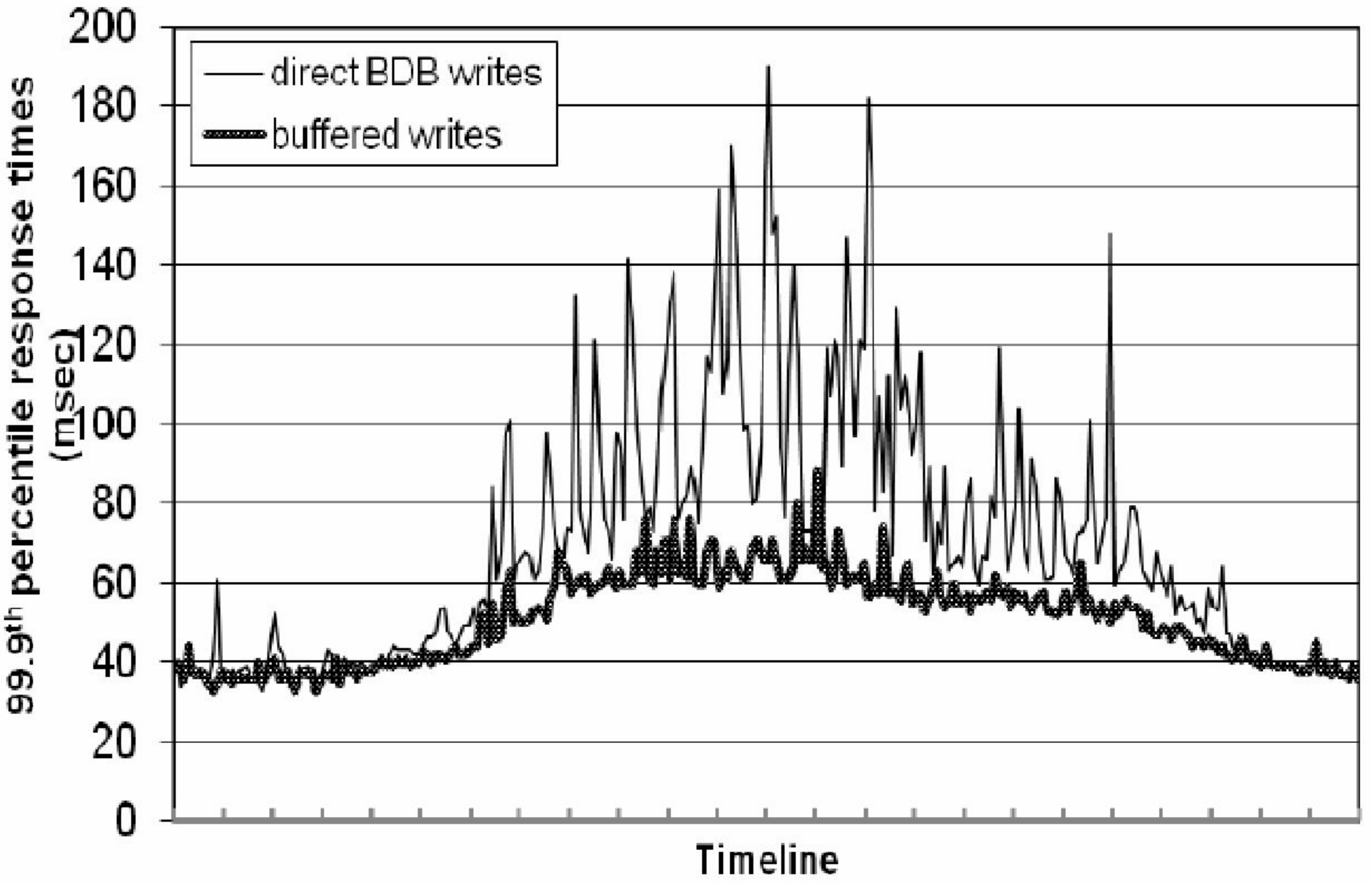
- Java
  - non-blocking IO
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: larger objects
- Quorum choices (N,W,R) □ influence object availability, durability, consistency

Performance evaluation

Writes



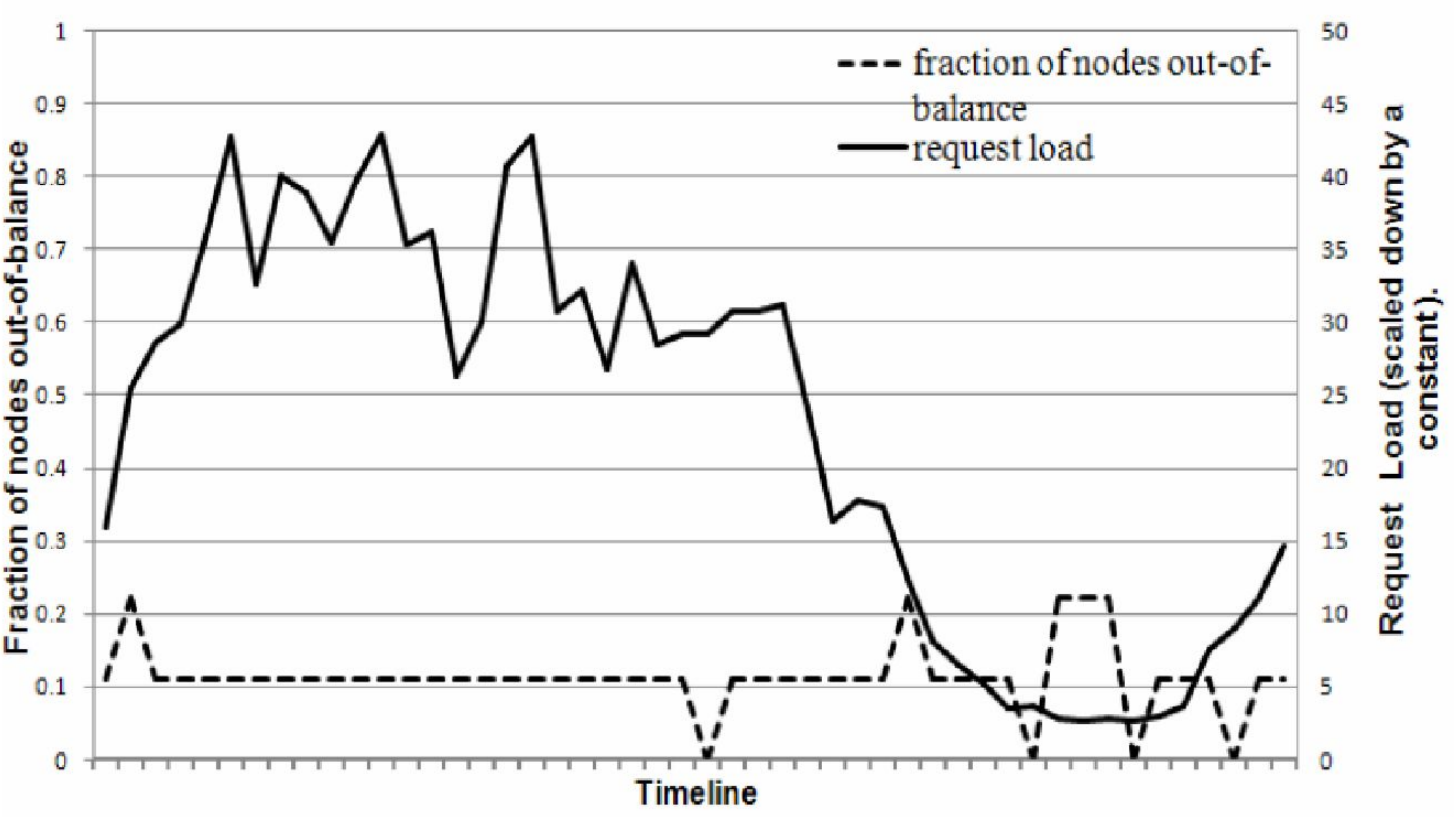
Trading between latency & durability



Comparison of performance of 99.9th %-tile latencies for buffered vs. non-buffered writes over 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

Load balance

~~out-of-balance~~: nodes with request load above 15% from the average system load



Divergent versions rarely created in practice

1 version □ 99.94%

2 versions □ 0.0057%

3 versions □ 0.00047%

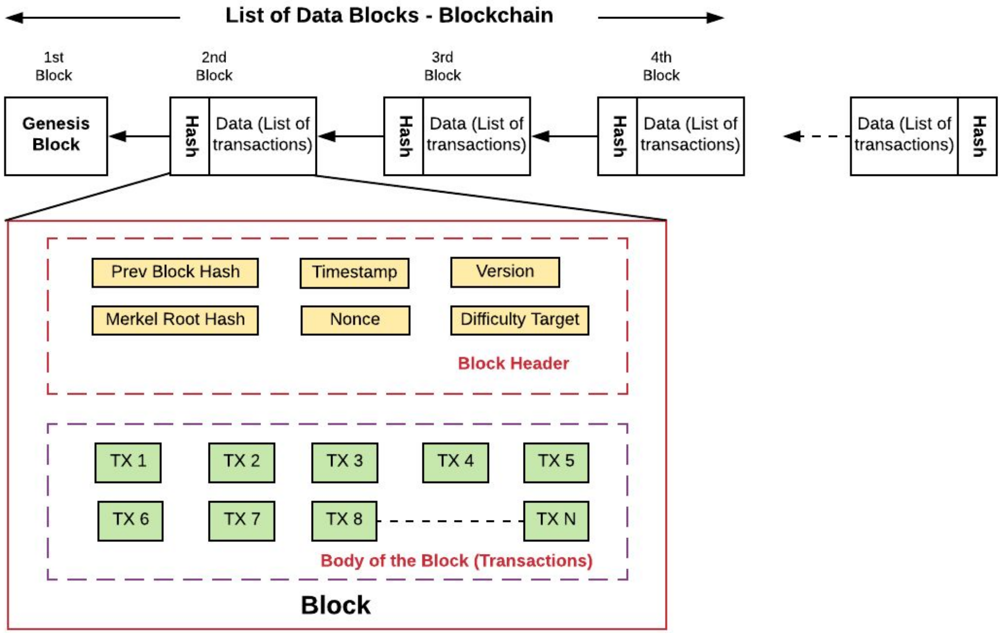
4 versions □ 0.00007

Source: High volume of concurrent writes … robots?

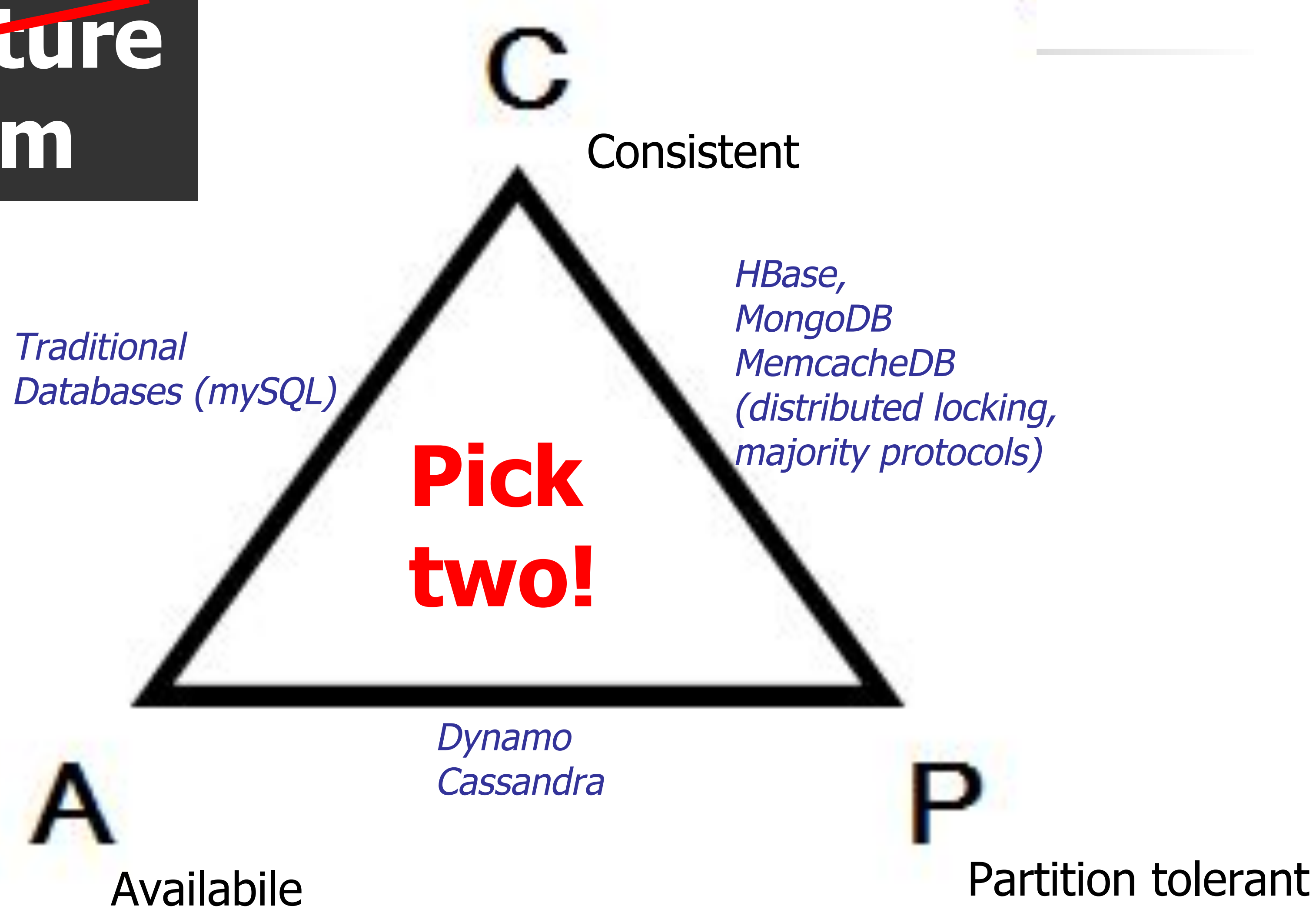
Problem	**Technique**	**Advantage**
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	▪ Eventual consistency ▪ Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	▪ 'Sloppy' quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	▪ Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	▪ epidemic-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Large set *S*: say 1M elements, each element 1KB-1MB

- P1: Need to test for differences between two sets
  - isDifferent (S1, S2) □ list of differences
- P2: You also know that if there is a difference between S1 and S2, it’s small
- Bonus: Verifiable proof of membership



CAP ~~Conjecture~~ Theorem



Consistent

Availabile

Partition tolerant

Traditional Databases (mySQL)

HBase, MongoDB MemcacheDB (distributed locking, majority protocols)

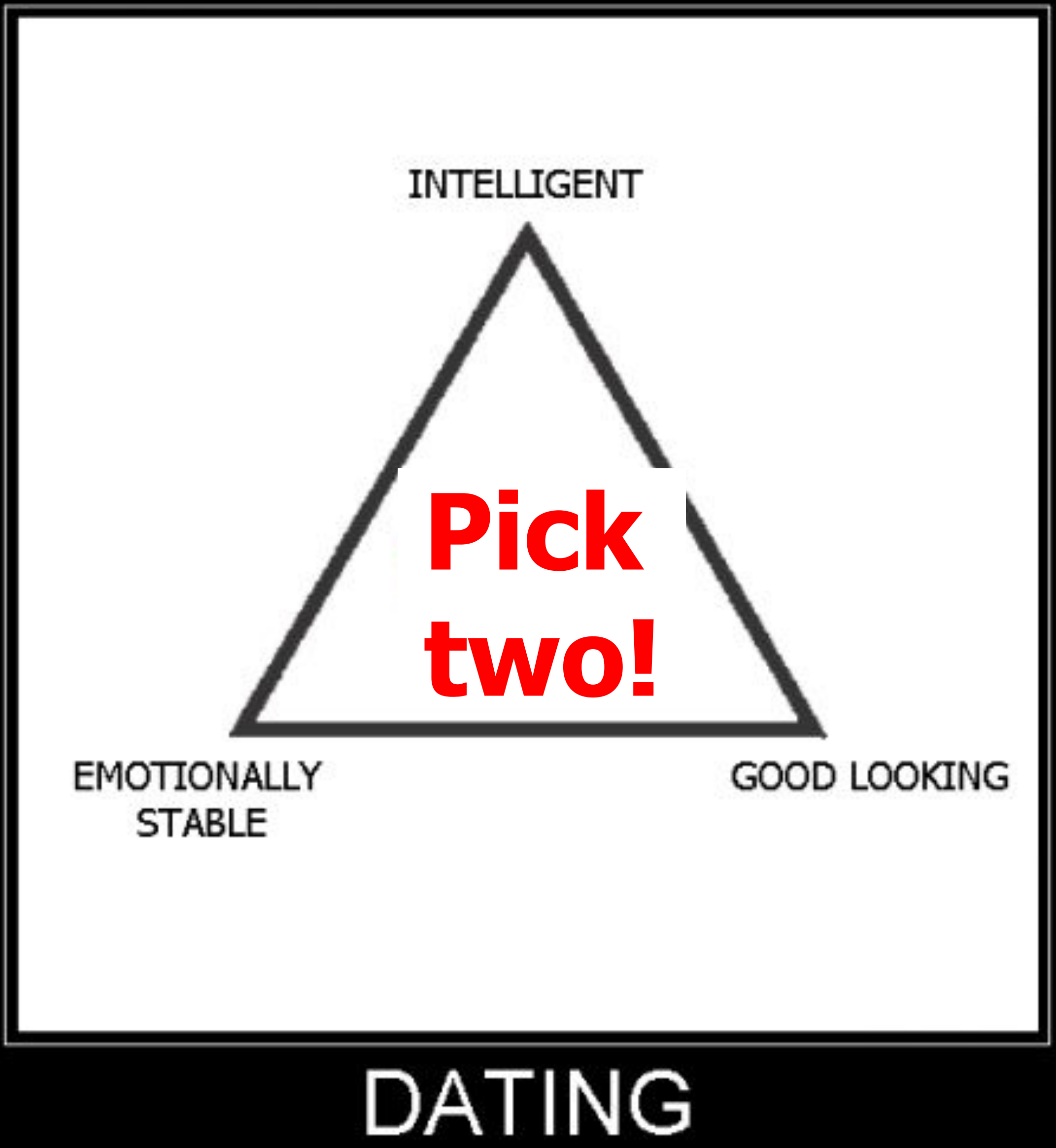
Dynamo Cassandra

Pick two!

NB: As with all impossibility results mind the assumptions: may do nice stuff with different assumptions

“Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”. SIGACT News 33(2): 51-59 (2002)

THE DATING CONJECTURE



Pick two!

Gilbert/Lynch theorems

Theorem 1: It is impossible in the **asynchronous** network model to implement a read/write object store that guarantees

- Availability AND
- Atomic consistency

in all executions (including those in which messages are lost)

asynchronous networks: no clocks, message delays unbounded

Gilbert/Lynch theorems

Theorem 2: It is impossible in the **partially synchronous** network model to implement a read/write object store that guarantees

- Availability AND
- Atomic consistency

in all executions (including those in which messages are lost)

partially synchronous network model. Bounds on:

- a) time it takes to deliver messages that are NOT lost, and
- b) message processing time

exist and are known, **but process clocks are not synchronized**