THE CAP THEOREM

Leonardo Querzoni
querzoni@dis.uniroma1.it

(Thanks to Leonardo Aniello for a large part of these slides!)

SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA "A. RUBERTI"

MIDLAB MIDDLEWARE LABORATORY

Why CAP is relevant today

- Large web based applications
  - Maximum availability
  - Support for huge user bases

- Availability ⟹ replication

- Scalability ⟹ geographic distribution
  - Horizontal scalability vs. vertical scalability

WHY CAP IS RELEVANT TODAY

- Consistency + Availability
  - Single hosted database
  - Services deployed on highly reliable clusters

- If we consider a distributed system with standard network connections there is no way to guarantee a partition-free execution
  - strong latency = network partition
  - routing errors = network partition
  - router failures = network partition
- Partition tolerance is not optional !

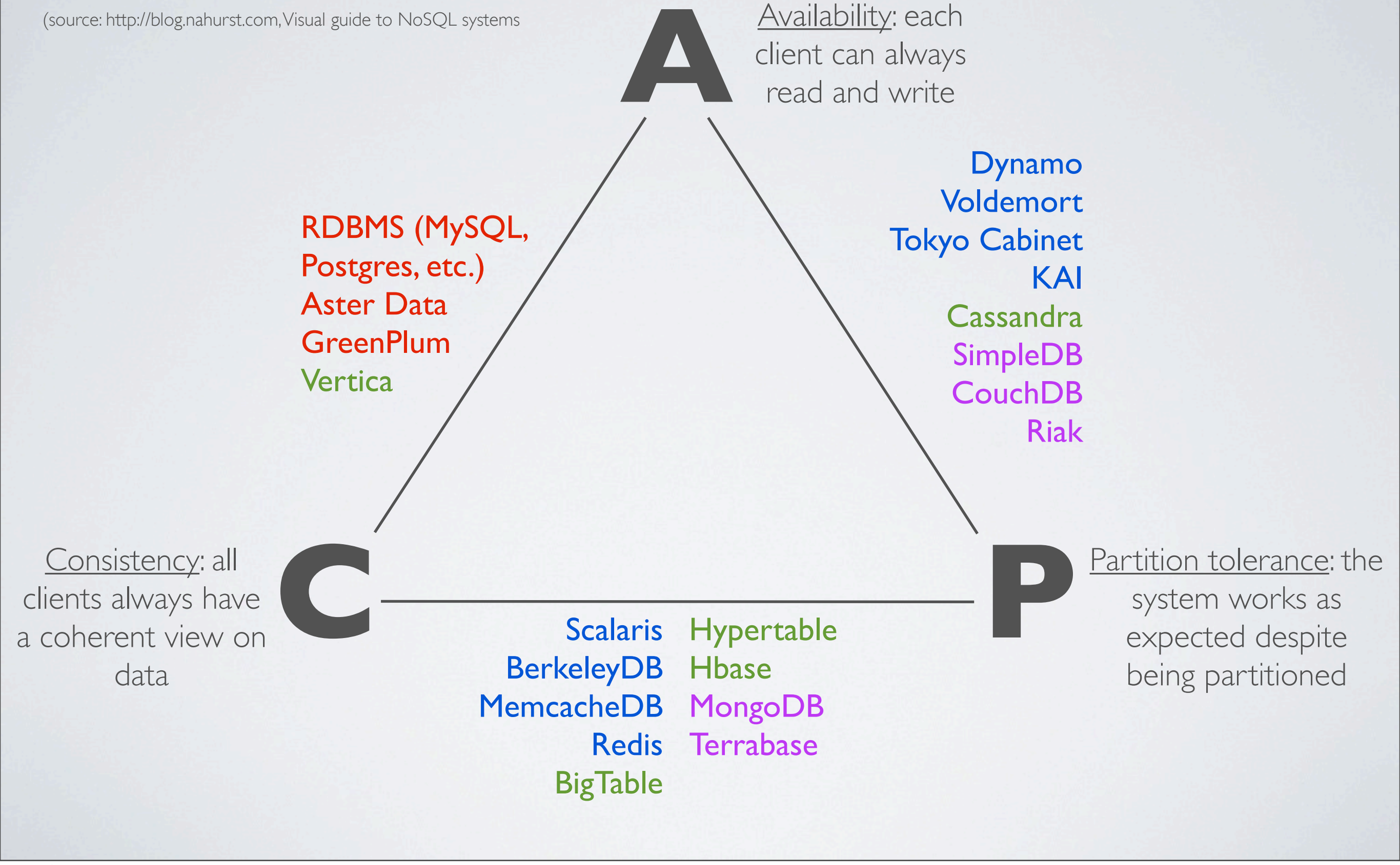
CAP Theorem

- Consistency + Partition tolerance
  - Distributed databases
  - Active/passive replication
  - Quorum-based systems

- Availability + Partition tolerance
  - Web caches
  - Stateless systems or systems without consistency criterions
  - Eventual consistency

NoSQL systems

(source: http://blog.nahurst.com, Visual guide to NoSQL systems



CAP CONSEQUENCES

- CAP limits what we can and cannot guarantee
- CAP does not limit completely what we can or can't do !
- Ex: assume you have a CP database
  - as long as there is a single partition the system is fully available
    - clients can write and read
    - data is correctly replicated while consistency is maintained
    - system admins enjoy some relax with a good drink... ;)
  - when a link goes down and two partitions appear:
    - some degree of availability is lost to guarantee full consistency
    - e.g. writes are blocked, reads are allowed
    - system admins are hard at work trying to fix the issue

CAP CONSEQUENCES

- The need for partition tolerance opens a tradeoff between availability and consistency
  - All the space among the two is available !

  - Linearizability, ACID, ...
  - Timeline consistency
  - BASE: basically available, soft state, eventually consistent
  - No consistency, ...

ACID vs. BASE

ACID

Atomicity, Consistency, Isolation, Durability

- Strong consistency (no stale data)
- Isolation
- Focus on “commit”
- Reduced availability
- Conservative (pessimistic)
- Safe

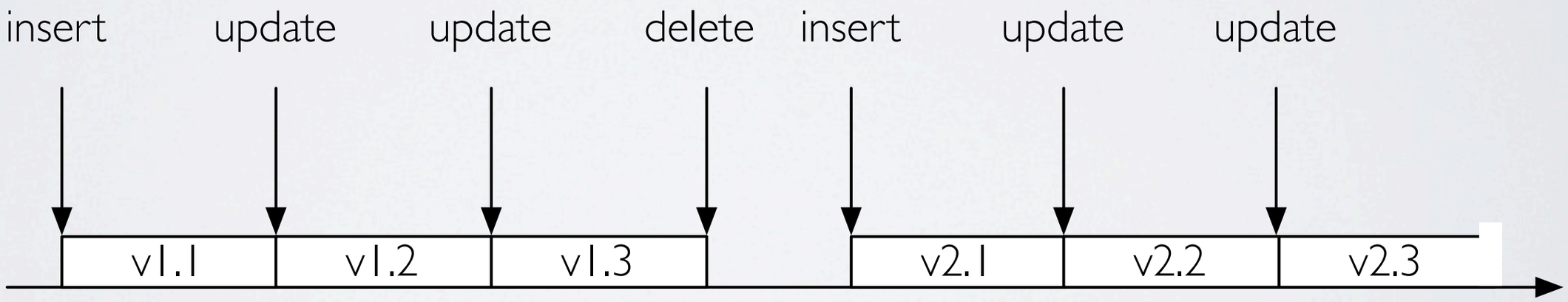
BASE

Basically Available, soft-state, eventually consistent

- Weak consistency (stale data is ok)
- Availability first
- Best effort
- Approximate answers OK
- Aggressive (optimistic)
- Fast

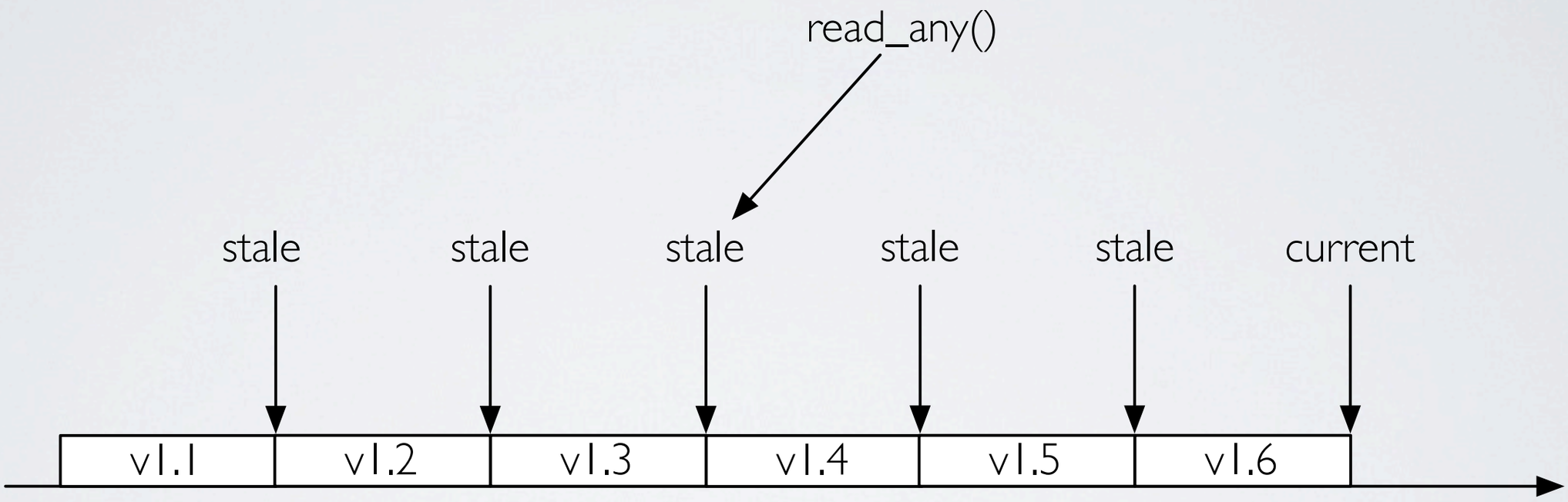
TIMELINE CONSISTENCY

- Introduced with PNUTS (Yahoo!):
  - 3 events: insert, update, delete
  - Every object has a version number
    - major: insert
    - minor: update
  - Uses a master replica



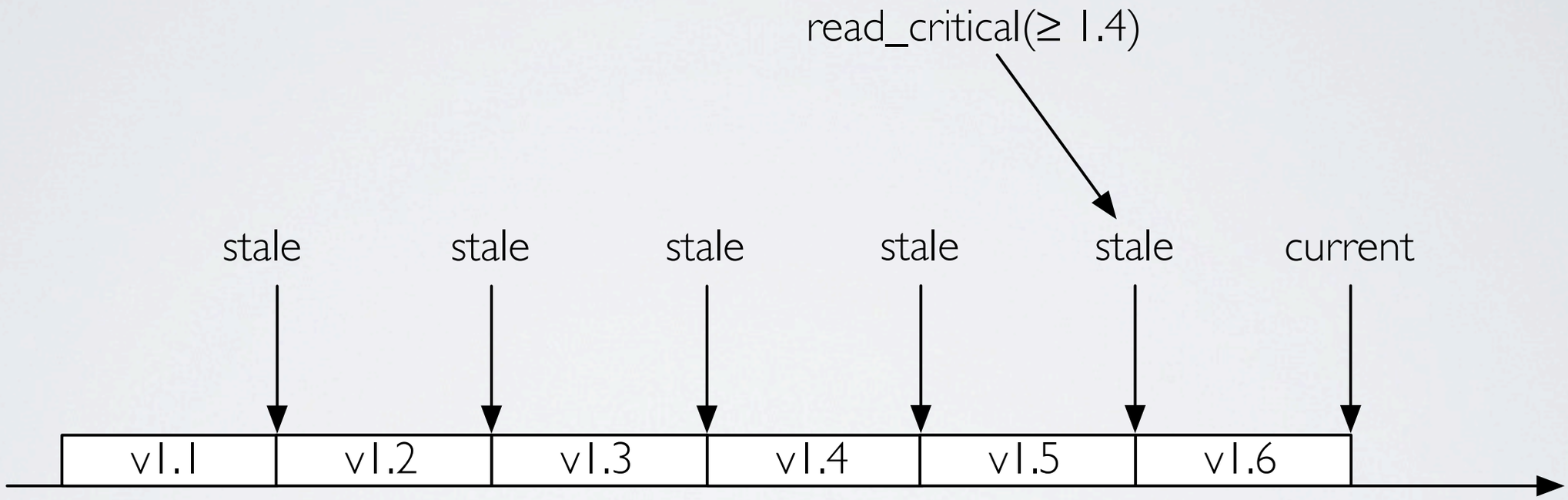
TIMELINE CONSISTENCY

- Various consistency criterions for read operations:
  - *Read Any*: any object version can be returned



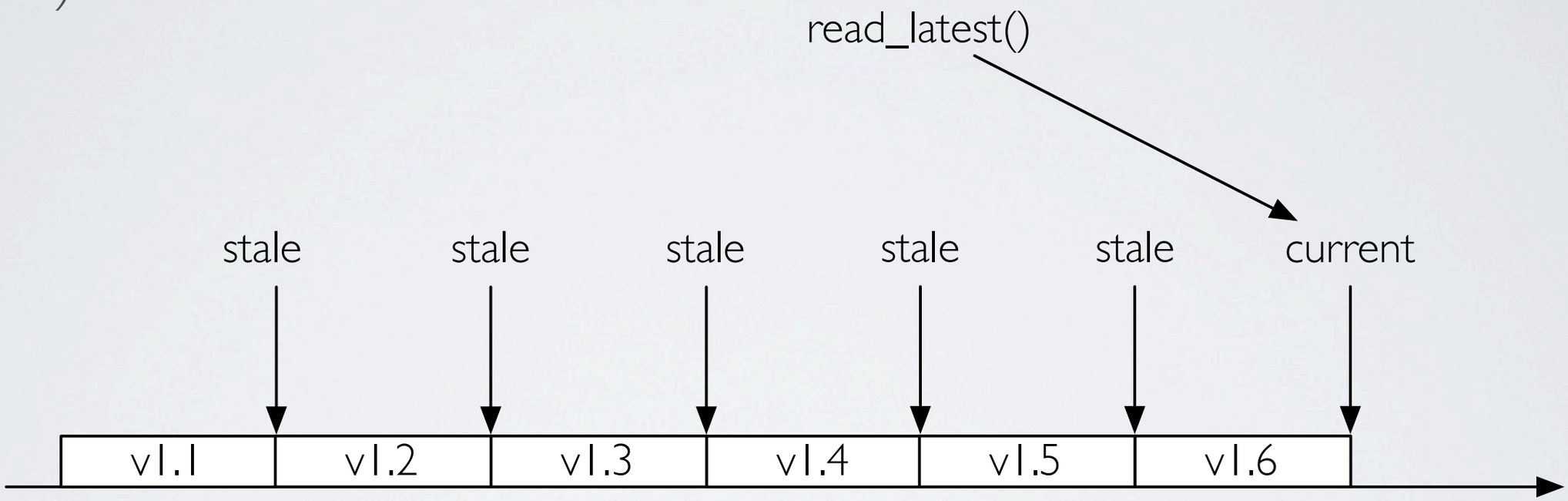
TIMELINE CONSISTENCY

- Various consistency criterions for read operations:
  - *Read Critical* (monotonic read): a read operation returns the last read version or a newer version.



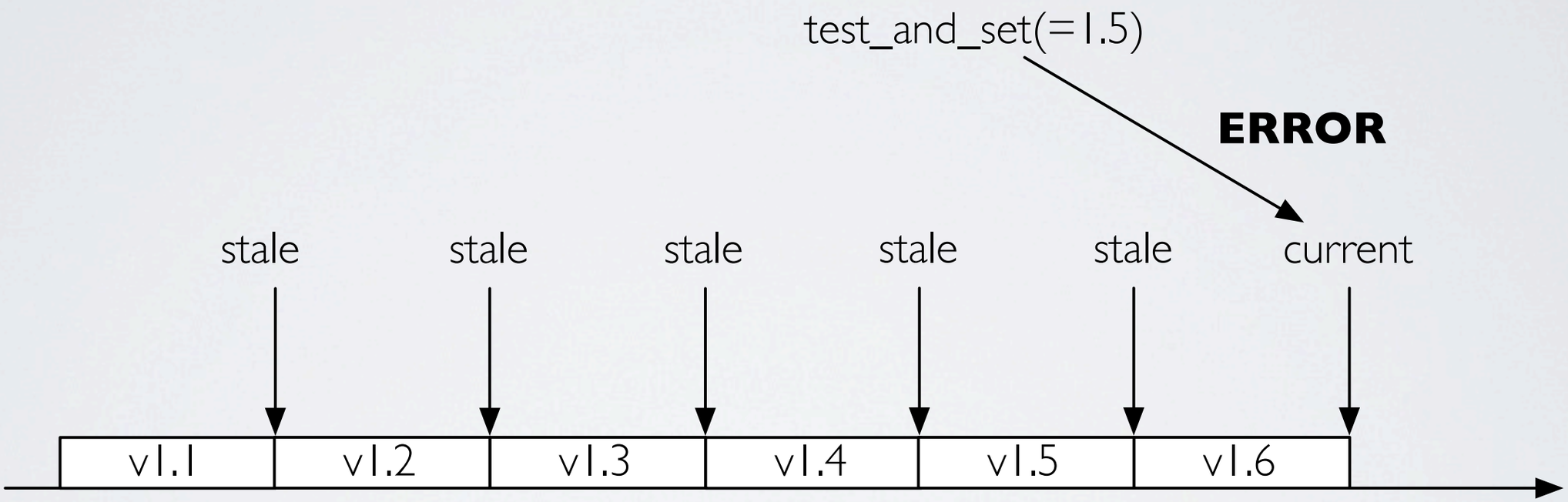
TIMELINE CONSISTENCY

- Various consistency criterions for read operations:
  - *Read Latest*: a read returns the current object version (requires access to the master).



TIMELINE CONSISTENCY

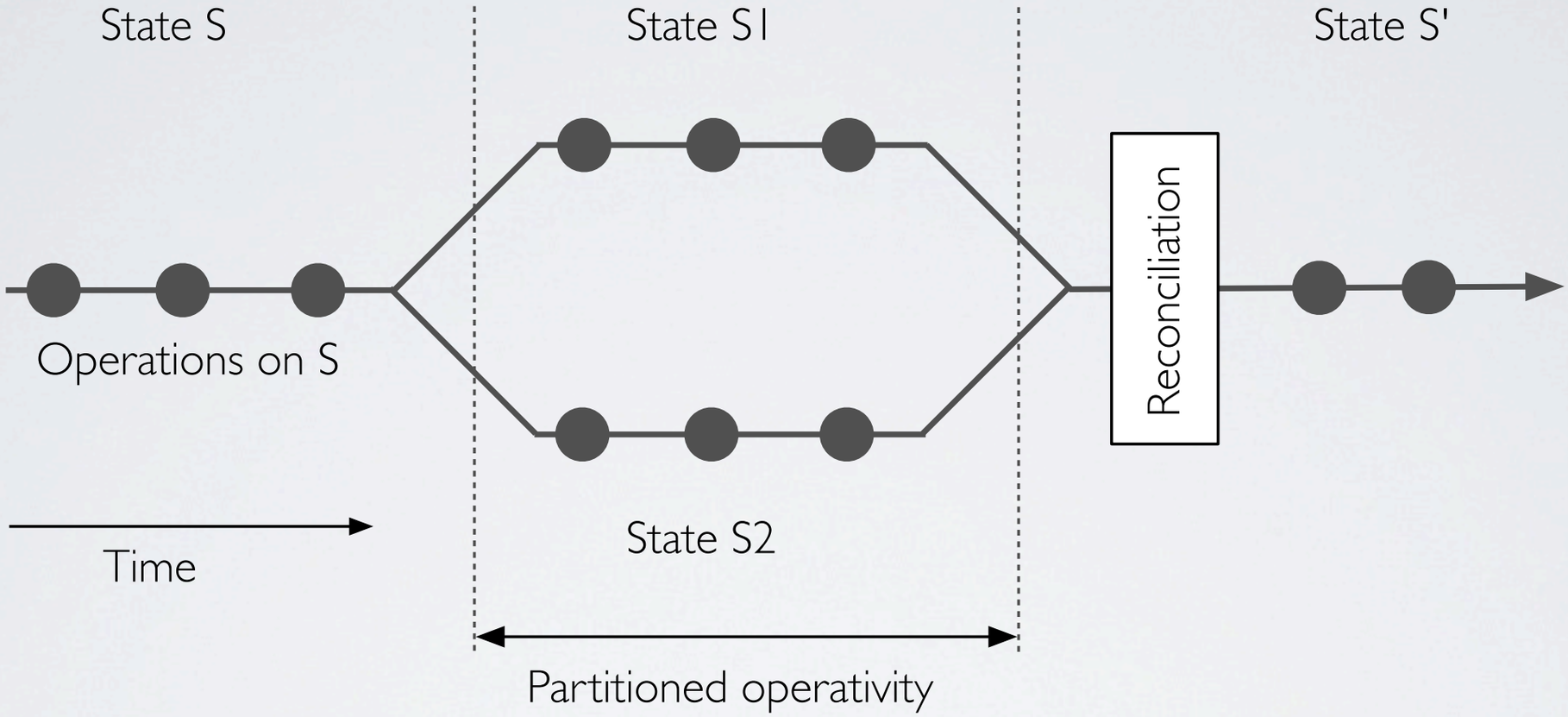
- Various consistency criterions for read operations:
  - *Test-and-Set*: is a conditional write where a new version is written only if it overwrites the last read version.



  - It is enough to guarantee ACIDity on a single record

EVENTUAL CONSISTENCY

- We allow inconsistency periods



- Several approaches to reconciliation
  - Read repair: slow read
  - Write repair: slow write
  - Asynchronous/Periodic/Off-line

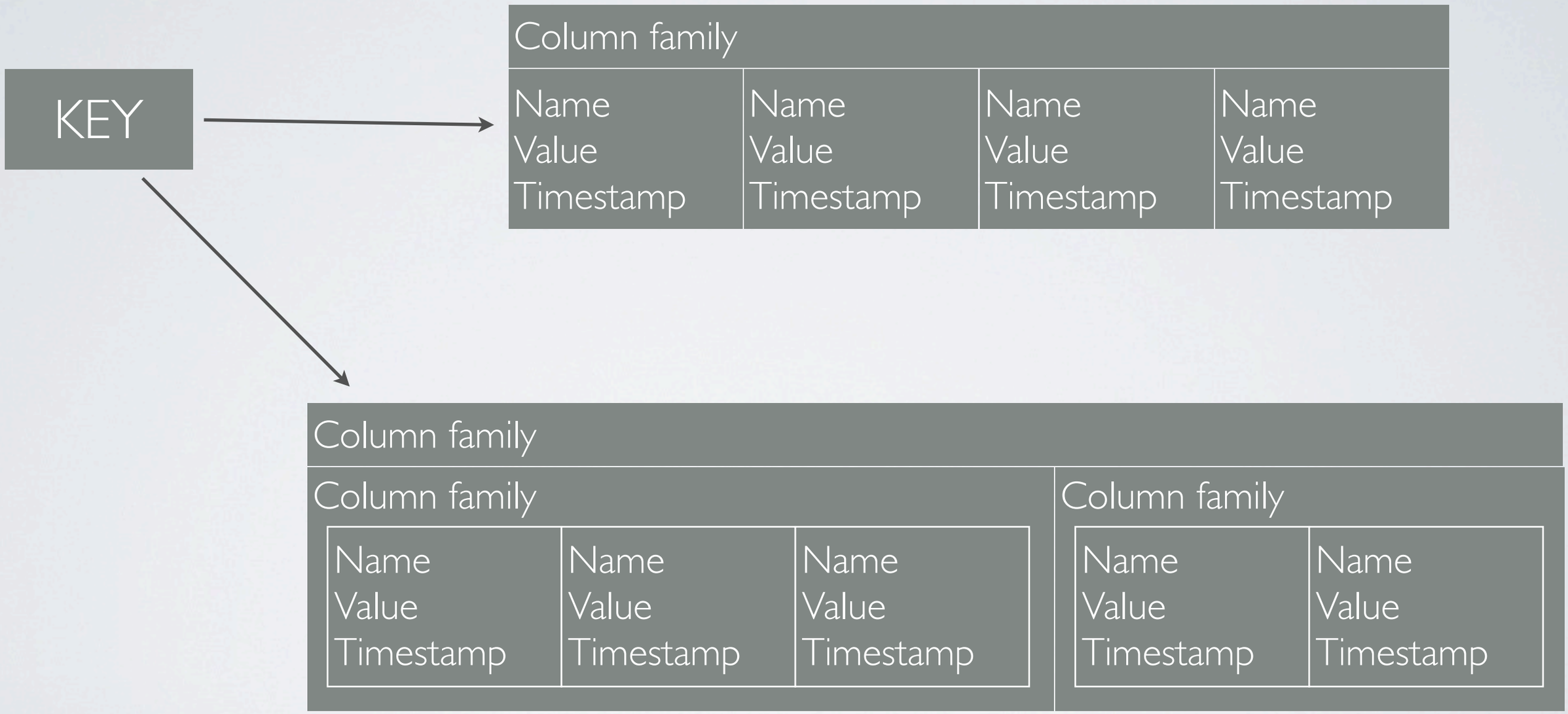
CASSANDRA

- Cassandra is an extremely scalable distributed data storage
  - Developed at Facebook
    - Used to manage user inbox
  - Now managed by the Apache foundation

- Main points
  - Fully distributed (p2p approach)
  - Extremely scalable (DHT)
  - Tunable consistency

CASSANDRA

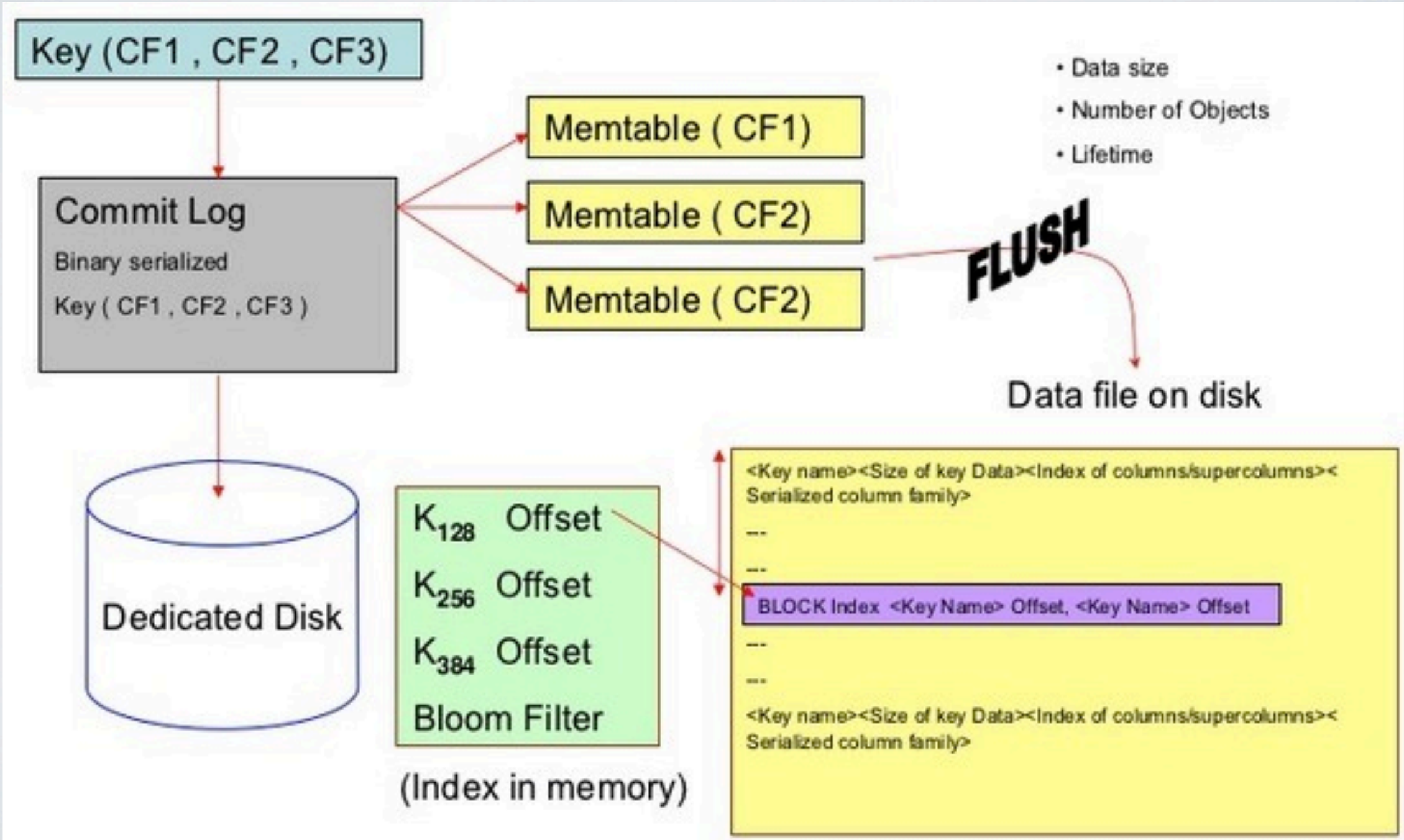
- Data is organized in a key/value store



Cassandra

- Write operations
  - Clients issue write request to any node in the Cassandra cluster
  - The "Partitioner" determines which node is responsible for the requested operation
  - Locally, writes are executed on a log and cached in memory
  - The commit log is written in a dedicated disk
  - Cached data is periodically written to stable storage

CASSANDRA

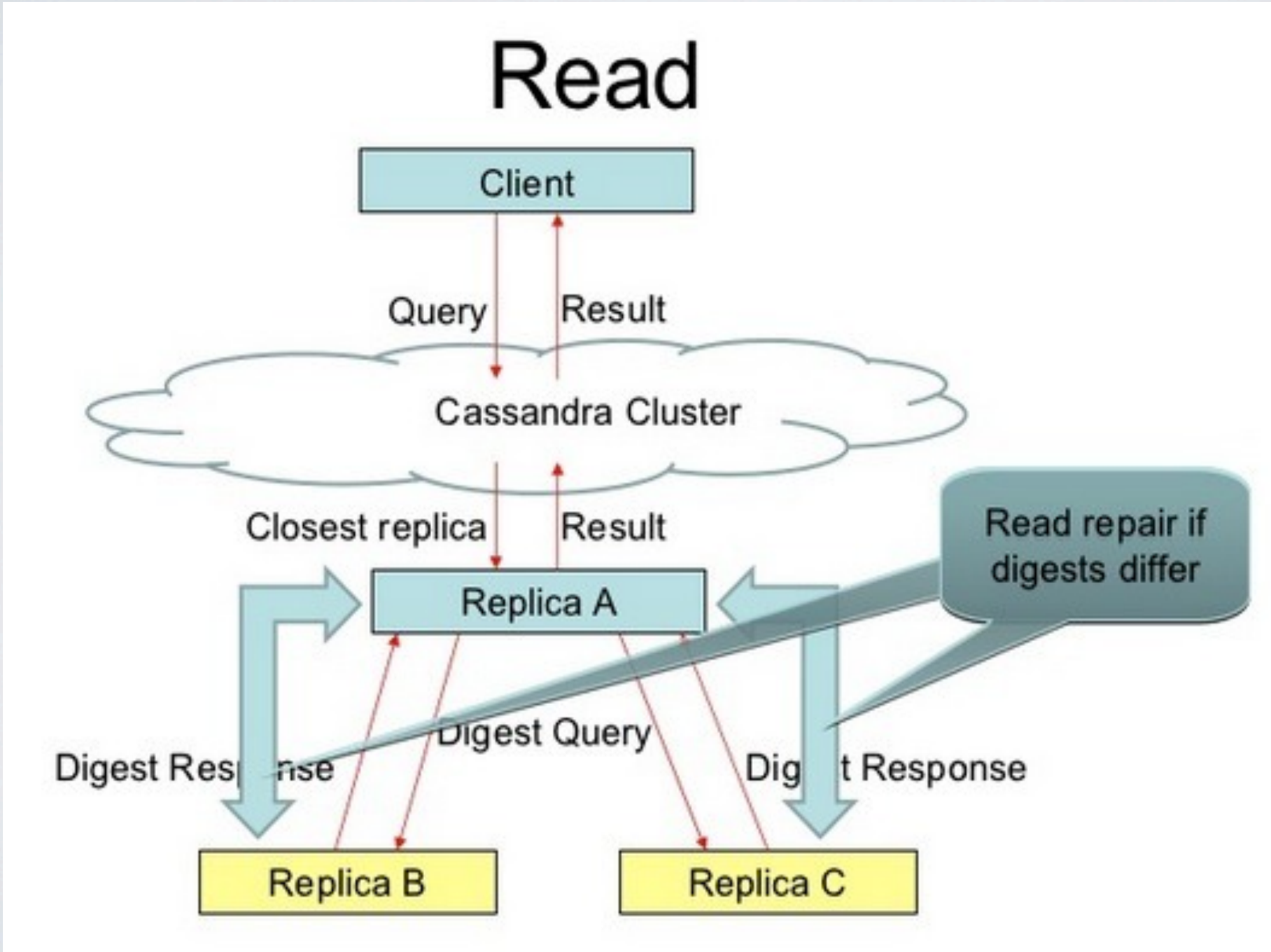

Key (CF1 , CF2 , CF3)
Commit Log
Binary serialized
Key (CF1 , CF2 , CF3)
Memtable (CF1)
Memtable (CF2)
Memtable (CF2)
• Data size
• Number of Objects
• Lifetime
FLUSH
Data file on disk
Dedicated Disk
K128 Offset
K256 Offset
K384 Offset
Bloom Filter
(Index in memory)
<Key name><Size of key Data><Index of columns/supercolumns><
Serialized column family>
--
--
BLOCK Index <Key Name> Offset, <Key Name> Offset
--
--
<Key name><Size of key Data><Index of columns/supercolumns><
Serialized column family>

CASSANDRA

- Write properties:
  - No locks
  - No read-before-write
  - Sequential access to disks
  - Behaves like a write-back cache
  - Guarantees atomic access to a single key
  - Always writable (also during failures)

CASSANDRA

- Reads



CASSANDRA

- Consistency

WRITE

Level	Description
ZERO	no consistency
ANY	1st response (including HH)
ONE	1st response
QUORUM	N/2+1 replicas
ALL	All replicas

Weak / Strong

READ

Level	Description
ONE	1st response
QUORUM	N/2+1 replicas
ALL	All replicas

BIGTABLE

- Google's distributed data storage for structured data
- OSDI 2006
- Used by more than 60 Google's services
  - Web indexing
  - Google Earth
  - Google Finance
  - Google Analytics
  - Orkut
  - ...

BIGTABLE - REQUIREMENTS

- Scales to petabytes of data
- Deployed on thousands of machines
- Wide applicability, fits to a variety of demanding workloads
  - from batch processing (need for high throughput)
  - to timely interactions with users (need for low latencies)
- High availability
- High performance

BIGTABLE - DATA MODEL

- "sparse, distributed, persistent multi-dimensional sorted map"
  - **sparse**: NULL values are not stored, save space
  - **distributed**: data set managed by multiple physical nodes
  - **persistent**: data saved on stable storage
  - **multi-dimensional**: data indexed by row, column, timestamp
  - **sorted**: data is stored to disk according to a pre-defined order

BIGTABLE - DATA MODEL

- Classic tabular data view:

EmpID	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	40000

- row oriented representation (typical in RDBMS)
  - all the values of a row are serialized together

    1:Smith:Joe:40000::2:Jones:Mary:50000::3:Johnson:Cathy:40000
  - to get the values of a few columns, all the values of the rows have to be read
  - many disk seeks

BIGTABLE - DATA MODEL

- Classic tabular data view:

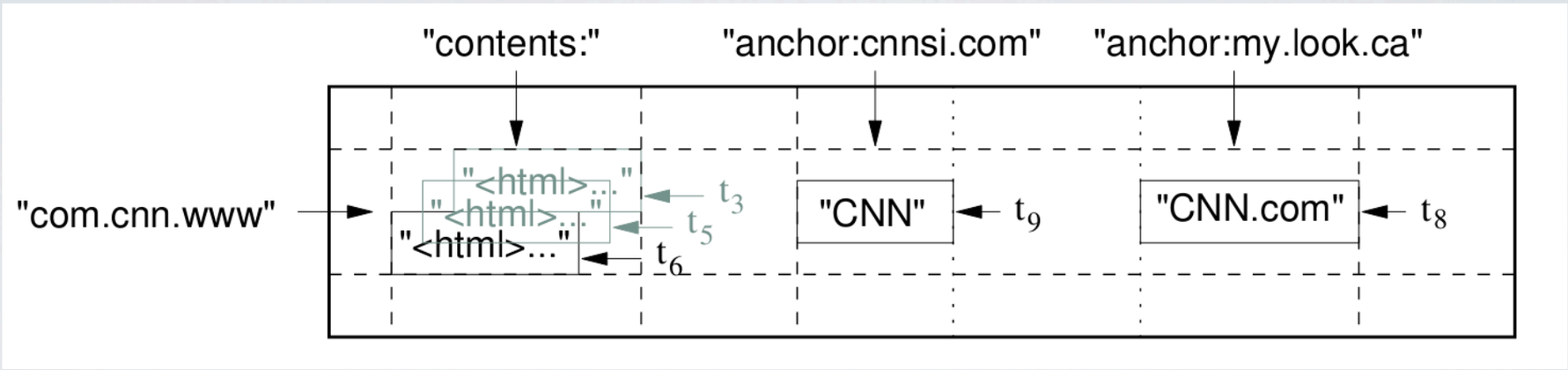
EmpID	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

- column oriented representation (e.g. Bigtable)
  - values for a same column are serialized together

    1,001:2,002:3,003::Smith,001:Jones,002: Johnson,003::Joe,001:Mary,002:Cathy:
    003::40000,001,003:50000,002

  - to get the values of a few columns, only interested columns have to be read
  - disk seeks are heavily reduced

BIGTABLE - DATA MODEL

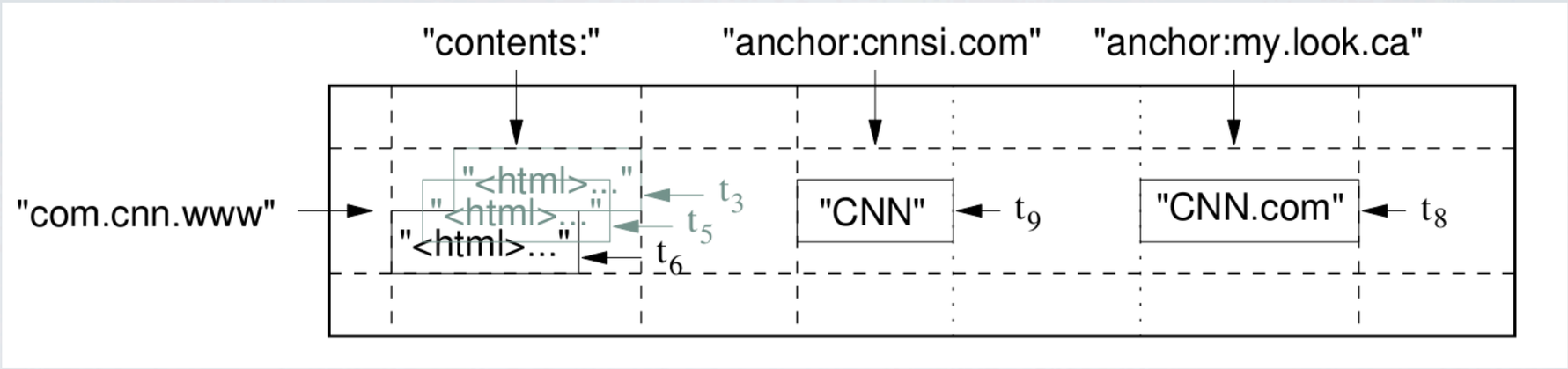


(source: F. Chang et al. "Bigtable: a distributed storage system for structured data", ACM TOCS 2008)

- Data is indexed by a triple

  (row, column, timestamp) → *string*

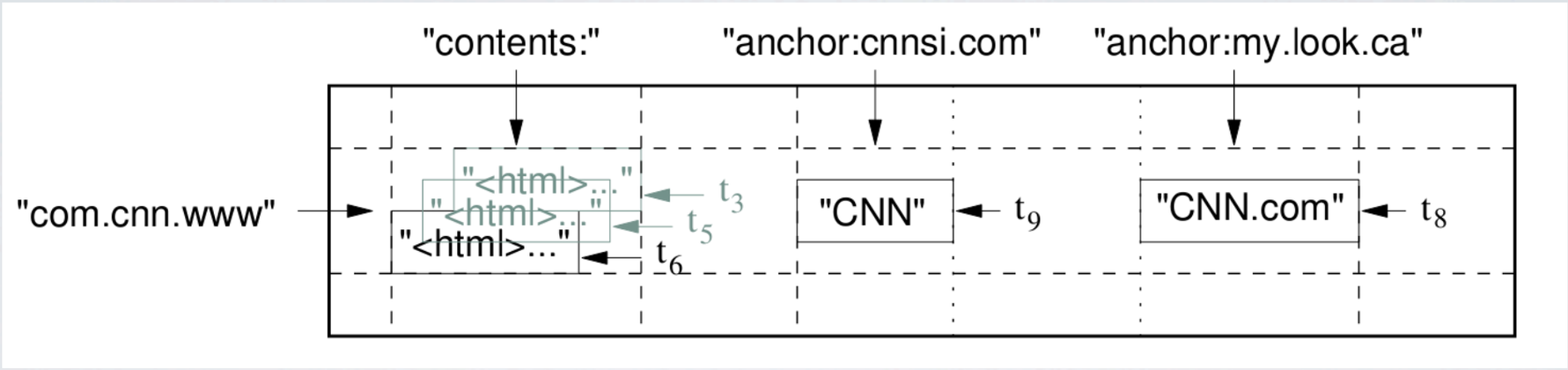
BIGTABLE - DATA MODEL



(source: F. Chang et al. "Bigtable: a distributed storage system for structured data", ACM TOCS 2008)

- ROWS
  - rows are arbitrary strings
  - reads/writes to a single row are atomic
    - easier to manage concurrent requests
  - data sorted by row key in a lexicographic order

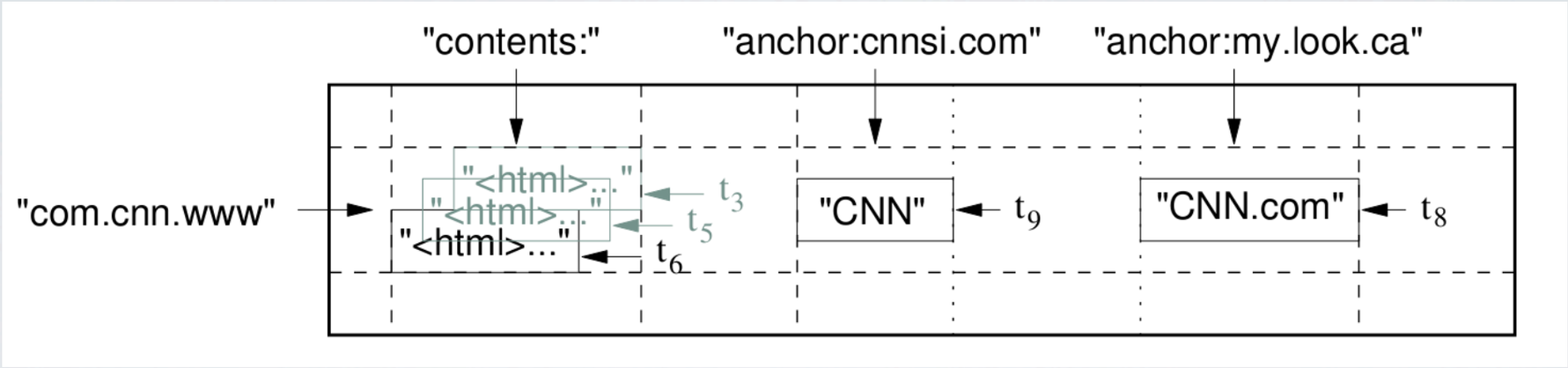
BIGTABLE - DATA MODEL



(source: F. Chang et al. "Bigtable: a distributed storage system for structured data", ACM TOCS 2008)

- COLUMNS
  - grouped in *column families*
  - access control and disk/memory management are performed at column family level
  - design choices
    - small number of column families, unbounded number of columns
    - use a column family for each required query
  - column key syntax → family:qualifier

BIGTABLE - DATA MODEL



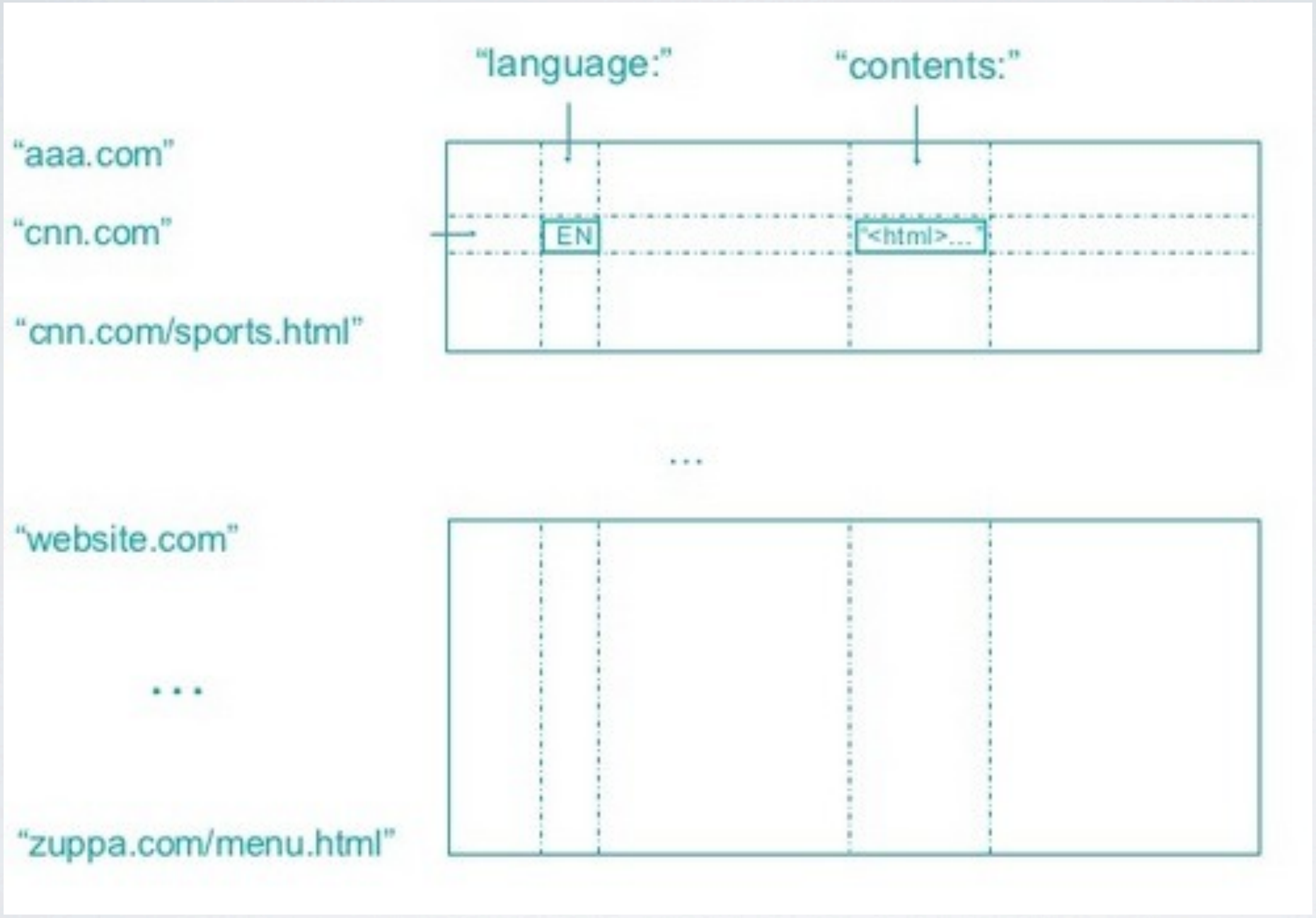
(source: F. Chang et al. "Bigtable: a distributed storage system for structured data", ACM TOCS 2008)

- TIMESTAMP
  - a cell in table is indexed by row key and column key and can include several versions of the data
    - these versions are indexed by timestamp
    - timestamps are assigned by
      - BigTable: they represent real time in microseconds
      - client: they are chosen on the basis of application-specific needs
    - versions sorted by decreasing timestamp: most recent version is read first
  - garbage-collection mechanisms
    - keep only the last N versions
    - keep only the versions written in the last N days

BIGTABLE - LOCALITY

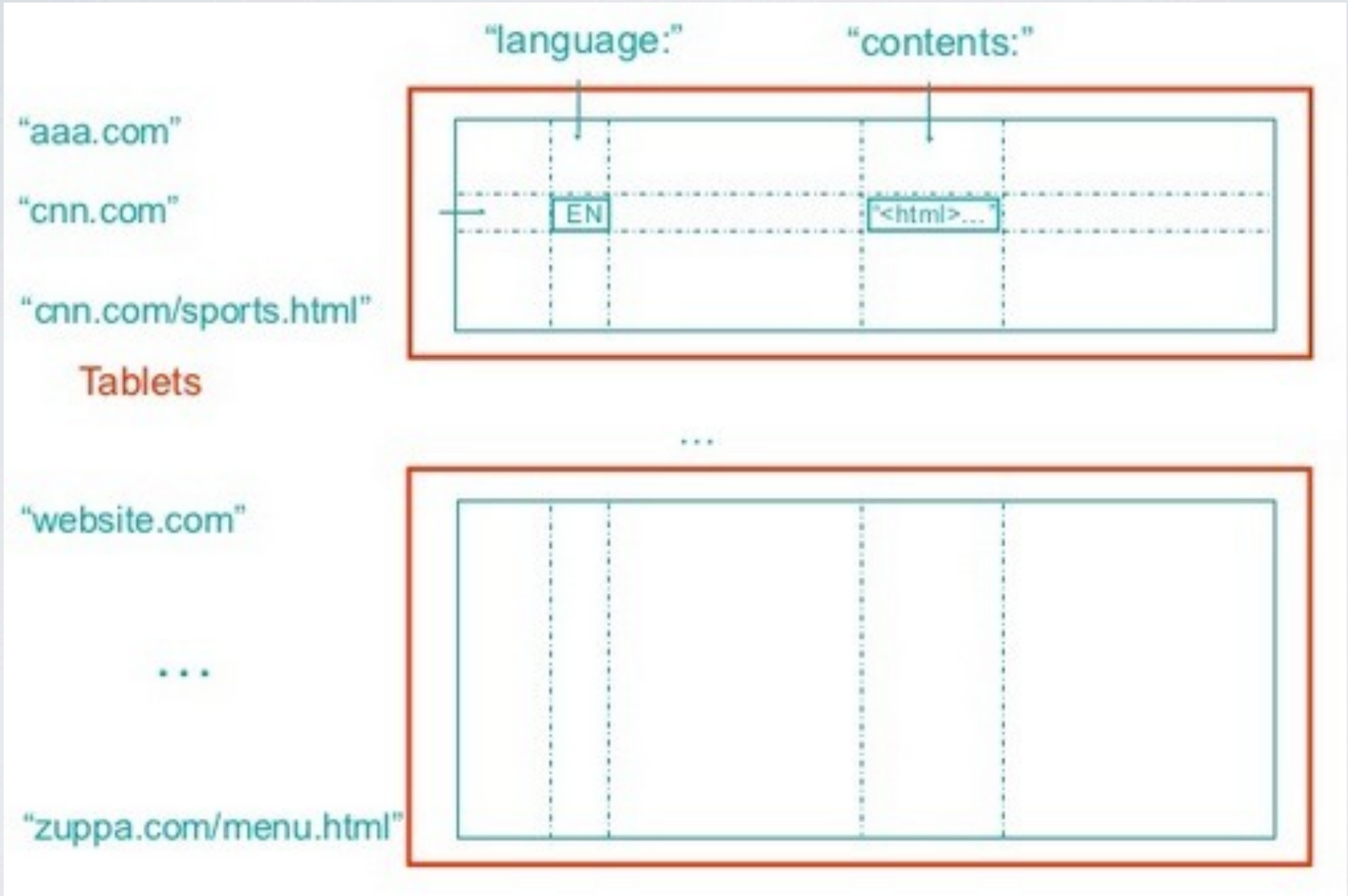
- dynamic partitioning of the rows of a table
- **tablet**: a row range, unit of distribution and load balancing
- rows with contiguous keys are stored in the same tablet
- all the rows of a tablet are stored in the same node
- clients can leverage this property to get good locality for data accesses

BIGTABLE - LOCALITY



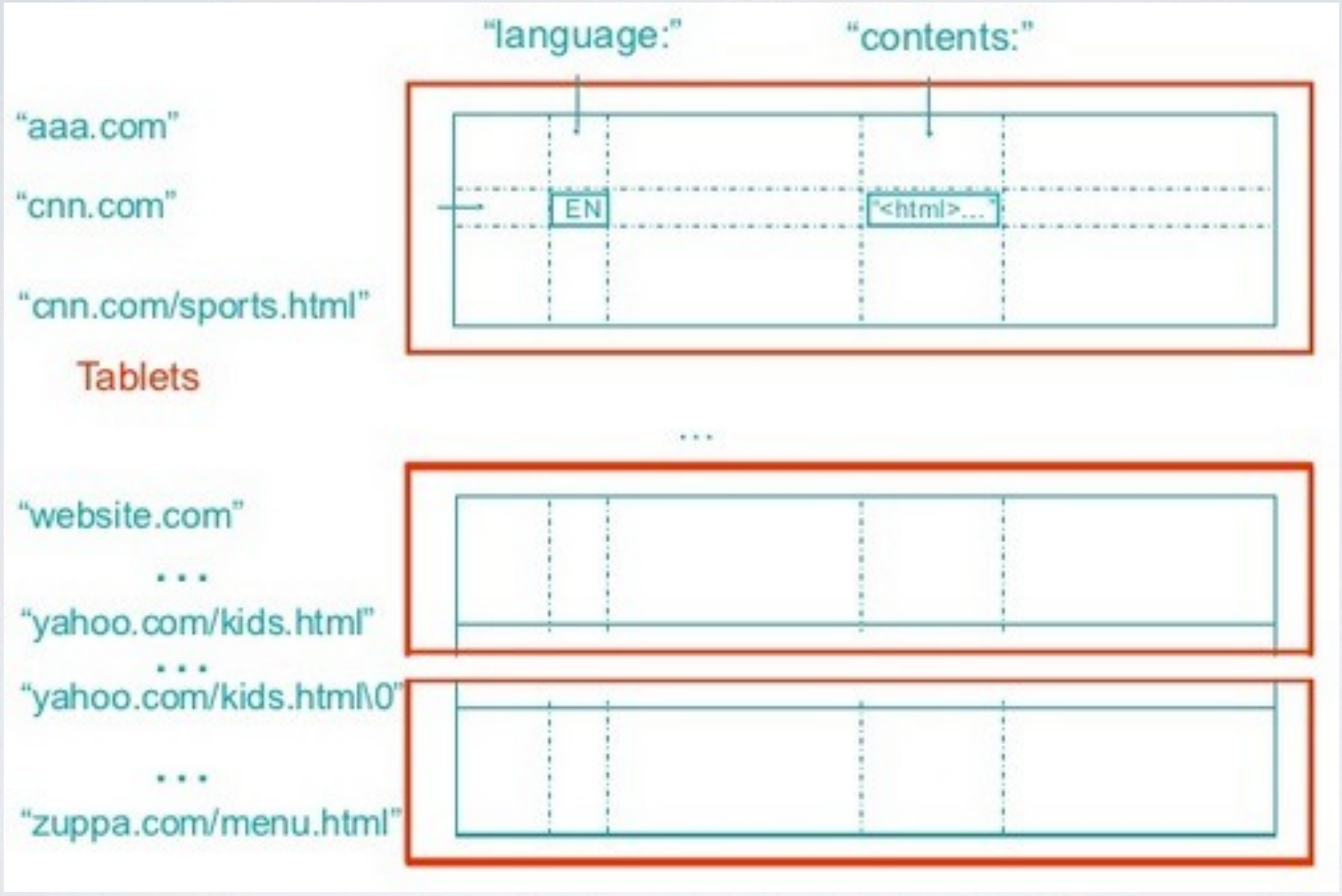
(source: J.Dean, "Designs, Lessons and Advice from Building Large Distributed Systems"

BIGTABLE - LOCALITY



(source: J.Dean, "Designs, Lessons and Advice from Building Large Distributed Systems"

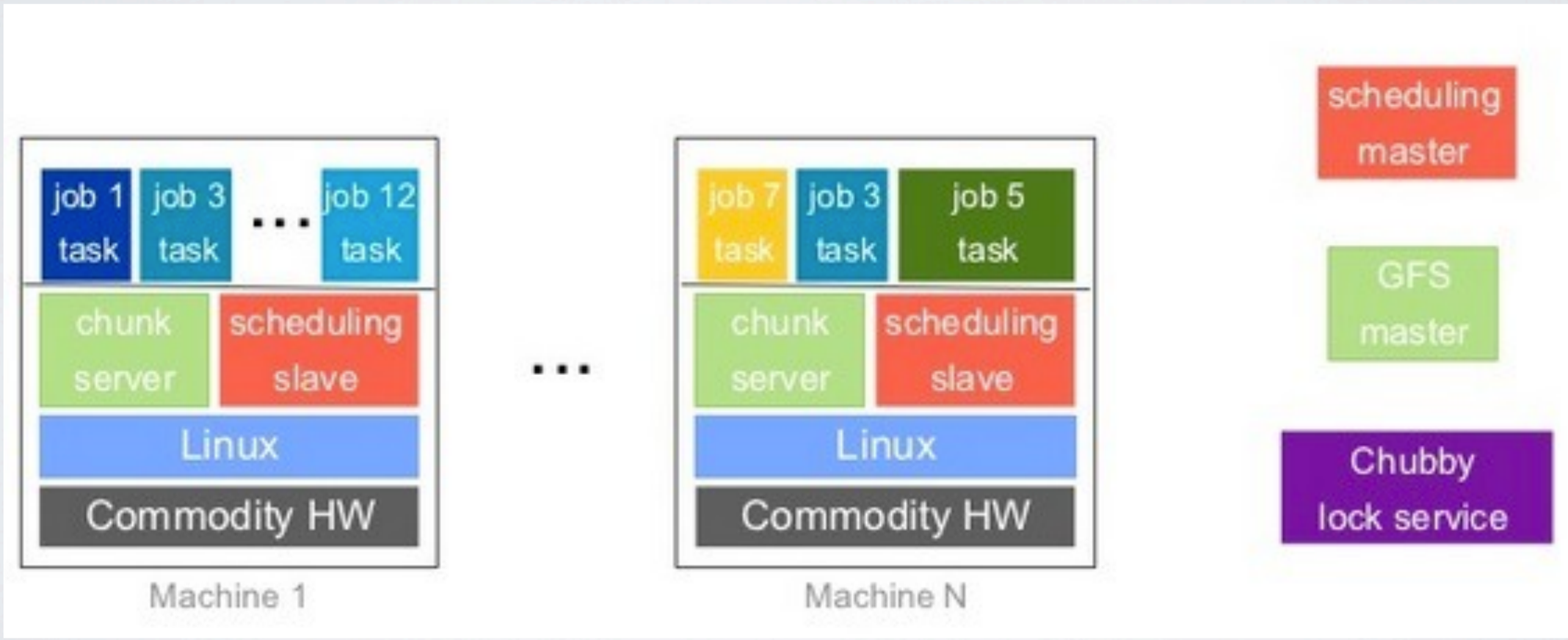
BIGTABLE - LOCALITY



(source: J.Dean, "Designs, Lessons and Advice from Building Large Distributed Systems"

BIGTABLE - ARCHITECTURE

- Google cluster environment
  - 1000s machines, few configurations
  - GFS + Cluster scheduling are core services

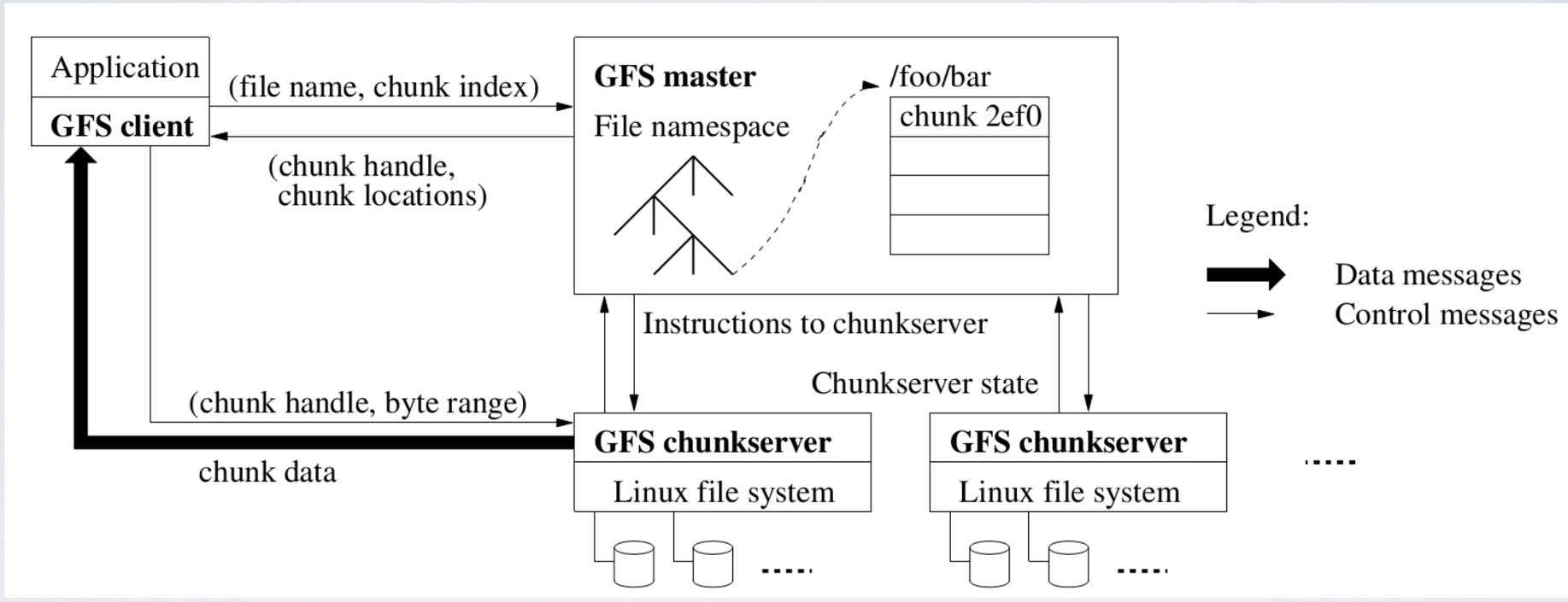


(source: J.Dean, "Designs, Lessons and Advice from Building Large Distributed Systems"

Architecture - GFS

- GFS design rationale
  - store a modest number of large files
  - typical workloads
    - two kinds of reads: large streaming reads and small random reads
    - many large, sequential writes that append data to files
    - small writes at arbitrary positions are supported but do not have to be efficient
  - well-defined semantics for multiple clients concurrently appending to a file
  - high throughput is more important than low latency

ARCHITECTURE - GFS



- files are divided into fixed-size chunks (default 64 MB)
- each chunk is replicated on multiple chunkservers (default 3)
- the master maintains all file system metadata
- clients never read and write file data through the master

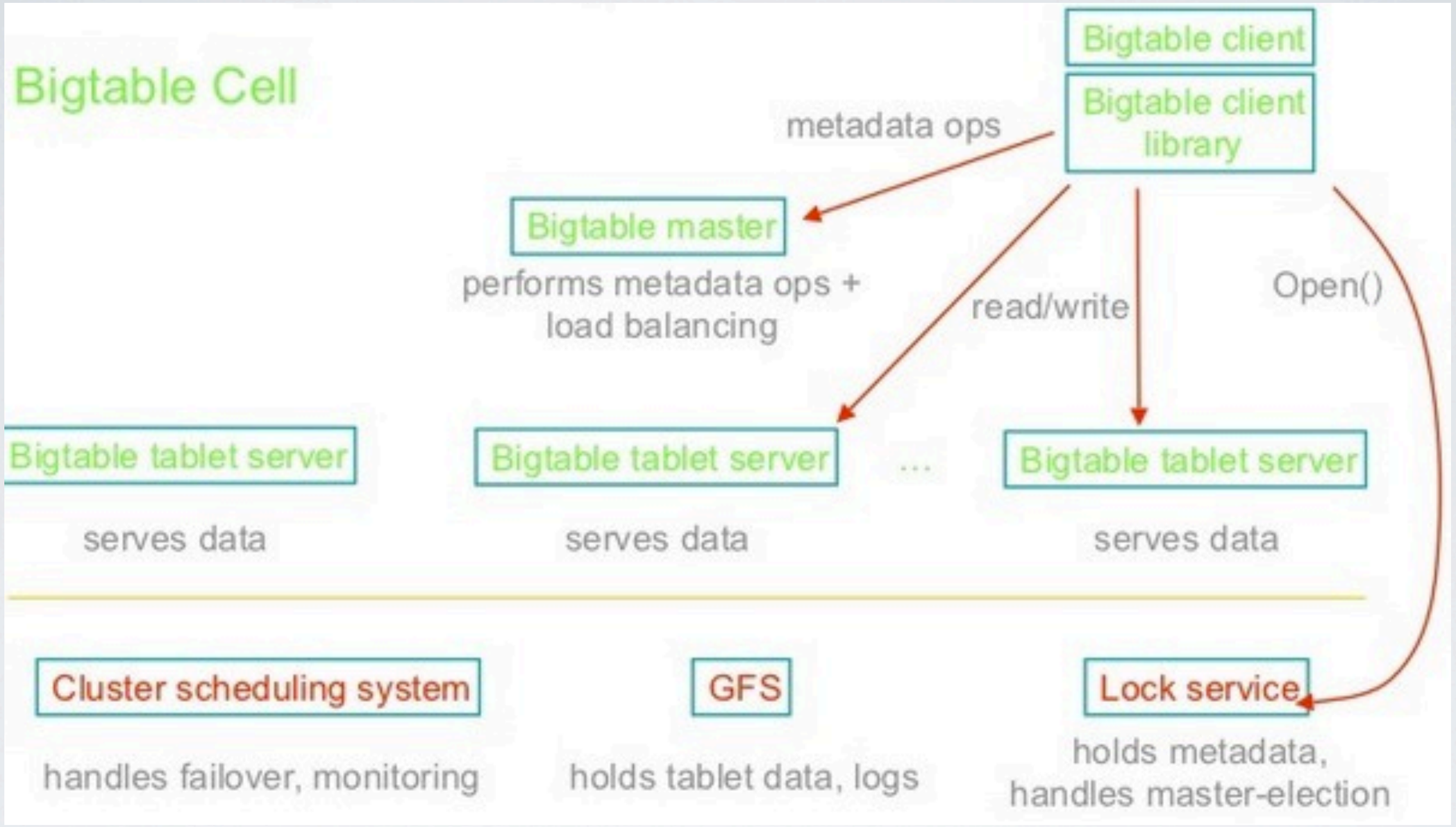
Architecture - Chubby

- A highly-available and persistent distributed lock service
  - five active replicas, one elected as the master that serves requests
  - service is live when a majority of the replicas are running
  - Paxos algorithm to keep replicas consistent despite failures
- Chubby provides a namespace that consists of directories and files
  - dirs and files can be used as a lock, reads/writes to a file are atomic
- Used for several tasks
  - ensure there is at most one active master at any time
  - store the bootstrap location of BigTable data
  - discover tablet servers and finalize tablet server deaths
  - store BigTable schema information
  - store access control lists

BIGTABLE - ARCHITECTURE

- Components:
  - one master server
    - assignment of tablets to table servers
    - detection of addition/removal of tablet servers
    - load balancing of tablet servers
    - garbage-collection of files in GFS
    - handling of schema changes
  - many tablet servers
    - can be dynamically added/removed
    - each server manages a set of tablets
    - handling of read/write requests to data of managed tablets
    - splitting of tablets that have grown too large
  - a library for the clients
    - client data does not move through the master: direct communication with tablet servers ⇒ the master is lightly loaded in practice

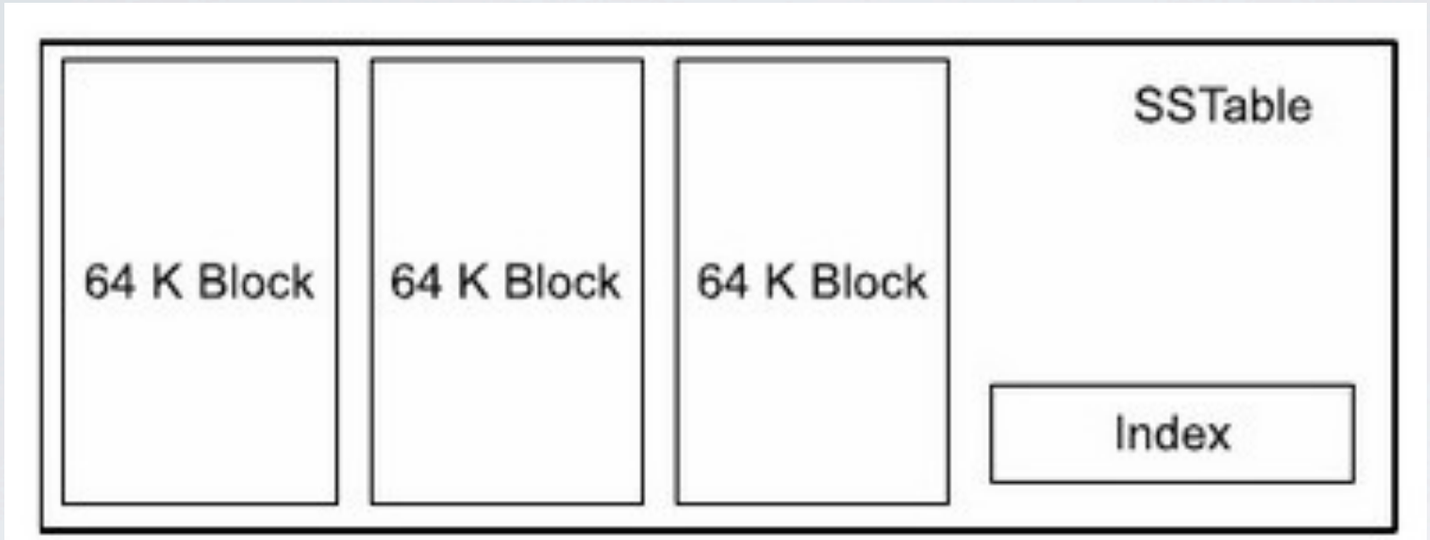
BIGTABLE - ARCHITECTURE



(source: J.Dean, "Designs, Lessons and Advice from Building Large Distributed Systems"

BIGTABLE - ARCHITECTURE

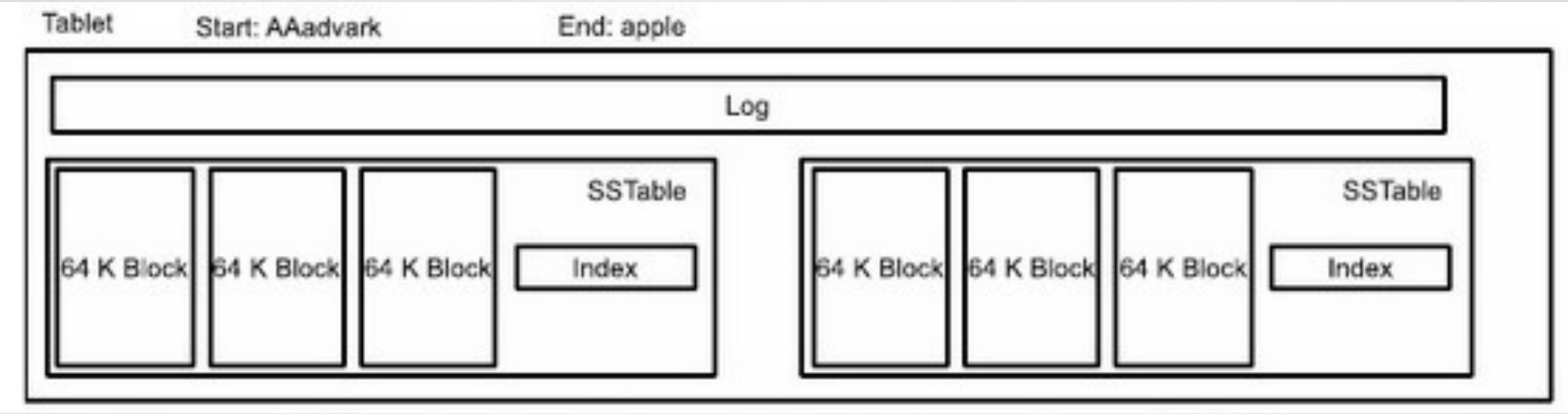
- Data is stored in SSTables



- Stored in GFS
- Immutable, sorted set of key/value pairs
- Data chunks + index of block ranges

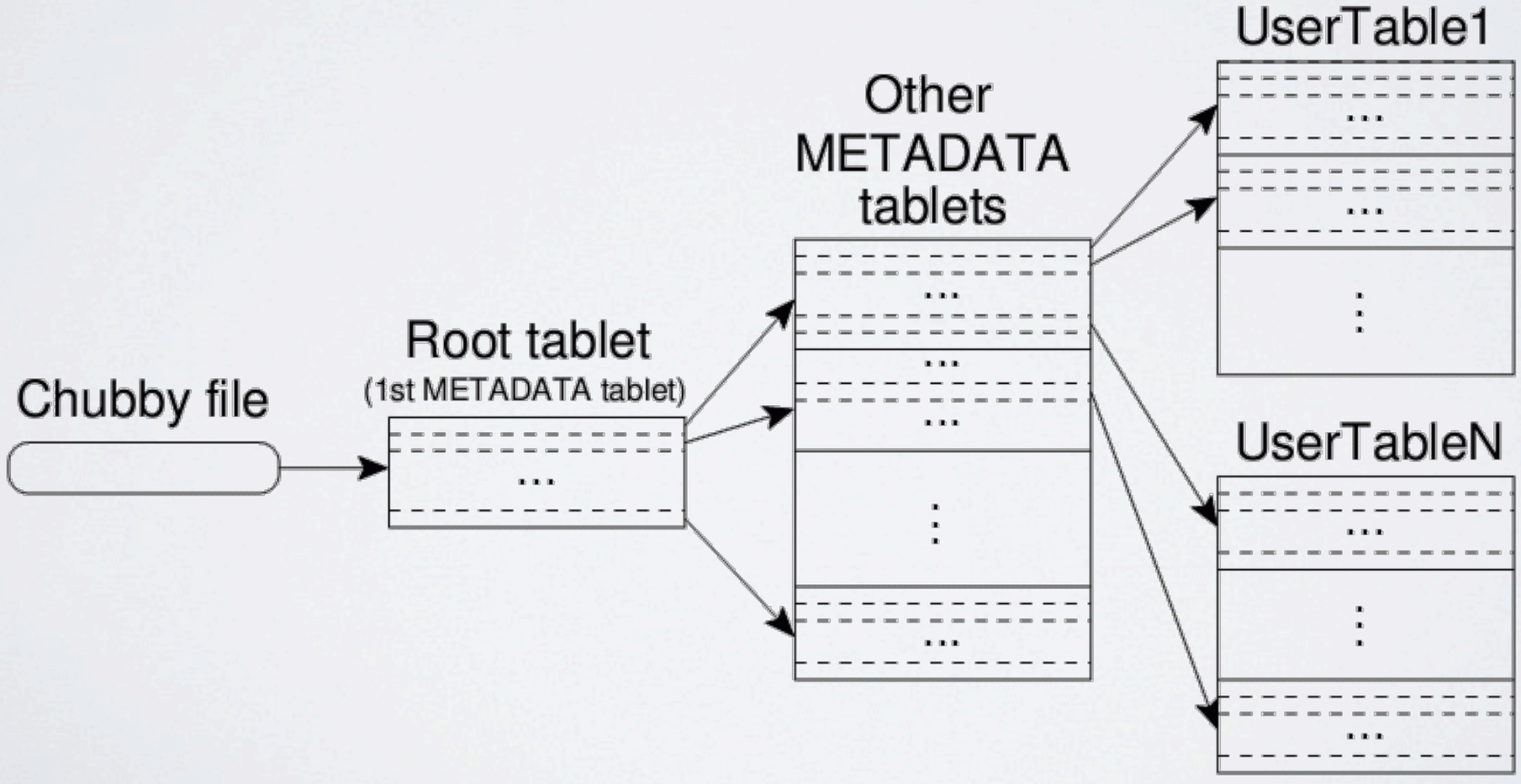
BIGTABLE - ARCHITECTURE

- Tablet structure



BIGTABLE - ARCHITECTURE

- Tablet location: three-level hierarchy
  - first: a file stored in Chubby that contains the location of the root tablet
  - second: the root tablet stores the locations of the tablets of a METADATA table
  - third: a METADATA tablet contains the location of a set of user tablets



BIGTABLE - ARCHITECTURE

- the METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table ID and its end row
- the client library caches tablet locations
  - if the client does not know the location of a tablet, it recursively moves up the tablet location hierarchy starting from the lower levels (optimistic caching)
  - if the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby
  - if the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses
- the client library pre-fetches tablet locations
  - it reads the metadata for more tablets when it reads the METADATA table

BIGTABLE AND CAP

- Bigtable is a CP system
  - if the master dies, the services it provided are no longer functioning until a new master is started
  - if a tablet server dies, client requests to the tablets it managed cannot be served until such tablets are assigned by the master to another tablet server
  - if Chubby fails (a majority of its replicas die), BigTable cannot execute any synchronization or serve any client request
  - in case of GFS failure, SSTables and commit logs are not available until recovery

BIGTABLE AND CAP

- Consistency guarantees:
  - each row is managed by a single tablet server
    - requests for a row are serialized
  - data (SSTables and commit logs) for a specific tablet is written to GFS by a single tablet server
    - reads and writes for tablet are serialized
  - writes to GFS files are always appends
    - no overwrites
  - SSTables are written once and then read
    - writes are not interleaved with reads
  - **strong consistency model**

Dynamo

- Dynamo is a distributed key-value storage used at Amazon to provide data persistence to several core services
  - shopping cart
  - best seller list
  - customer preferences
  - session management
  - sales rank
  - product catalog

Dynamo - Requirements

- many services only need "primary-key" access to a data store
- tens of millions customers at peak times
- always available

  "customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados"

- highly scalable
- reliable
- knobs to tradeoff availability and performances

Dynamo - Data model

- key-value store: any object is associated with a unique key
  - simple interface
    - *get(key)*
    - *put(key, context, object)*
  - keys and objects are treated as opaque arrays of bytes

Dynamo - Design rationale

- high availability + scalability ☐ data replication and distribution
- replication approaches:
  - synchronous replica coordination ☐ strong consistency, tradeoffs availability
  - optimistic replication techniques ☐ increased availability, possible conflicts

changes are allowed to propagate to replicas in the background

conflict resolution
when to resolve conflicts?
who resolves conflicts?

Dynamo - Design rationale

- When to resolve conflicts?
  - **write time:** writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time
  - **at read time:** "always writable" data store, conflicts resolved when data has to be read

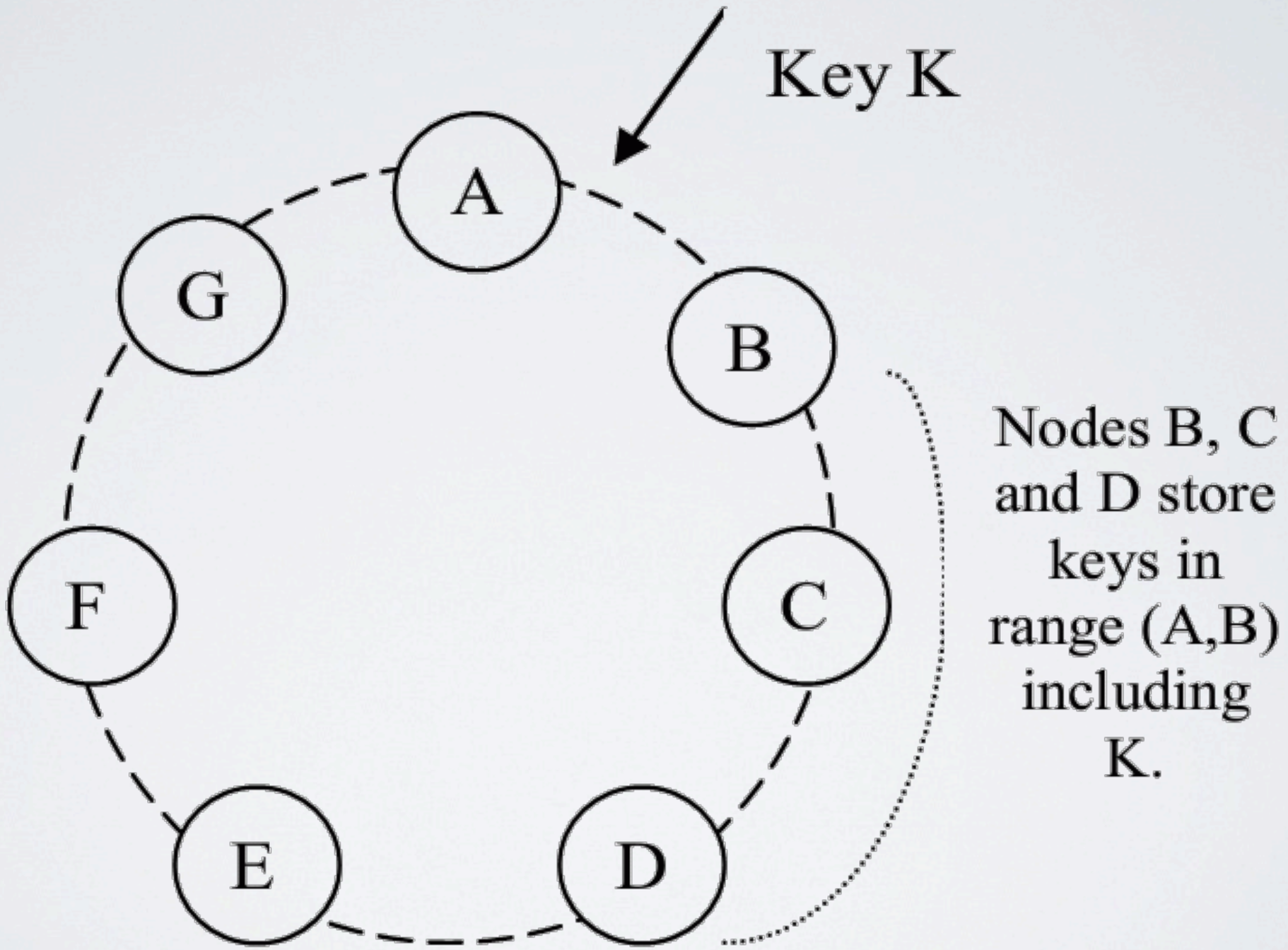
Dynamo - Design rationale

- Who resolves the conflicts?
  - **data store**
    - only simple policies make sense (e.g. "last write wins")
    - easy for developers
    - allow a performance-oriented design
  - **application:**
    - can decide on the best conflict resolution method
    - more complex for developers
- Dynamo takes an hybrid approach
  - tries to reconcile distinct data versions
  - if it fails, sends back to the client the list of conflicting versions

Dynamo - Architecture

- Based on a distributed hash table structure
  - nodes are organized in a ring
    - each node is assigned a random position on the ring
  - each data item is replicated at N nodes
    - the key of the item is mapped to a position on the ring (consistent hashing)
    - the item is replicated at the N nodes met by walking the ring clockwise starting from the position of the key
    - the first node met is referred to as the coordinator for such key
    - each node is responsible for the region between it and its Nth predecessor
  - the list of nodes responsible for key is called the preference list
    - to account for node failures, a preference list contains more than N nodes
    - these additional nodes are chosen by continuing to walk the ring clockwise

DYNAMO - ARCHITECTURE



DYNAMO - ARCHITECTURE

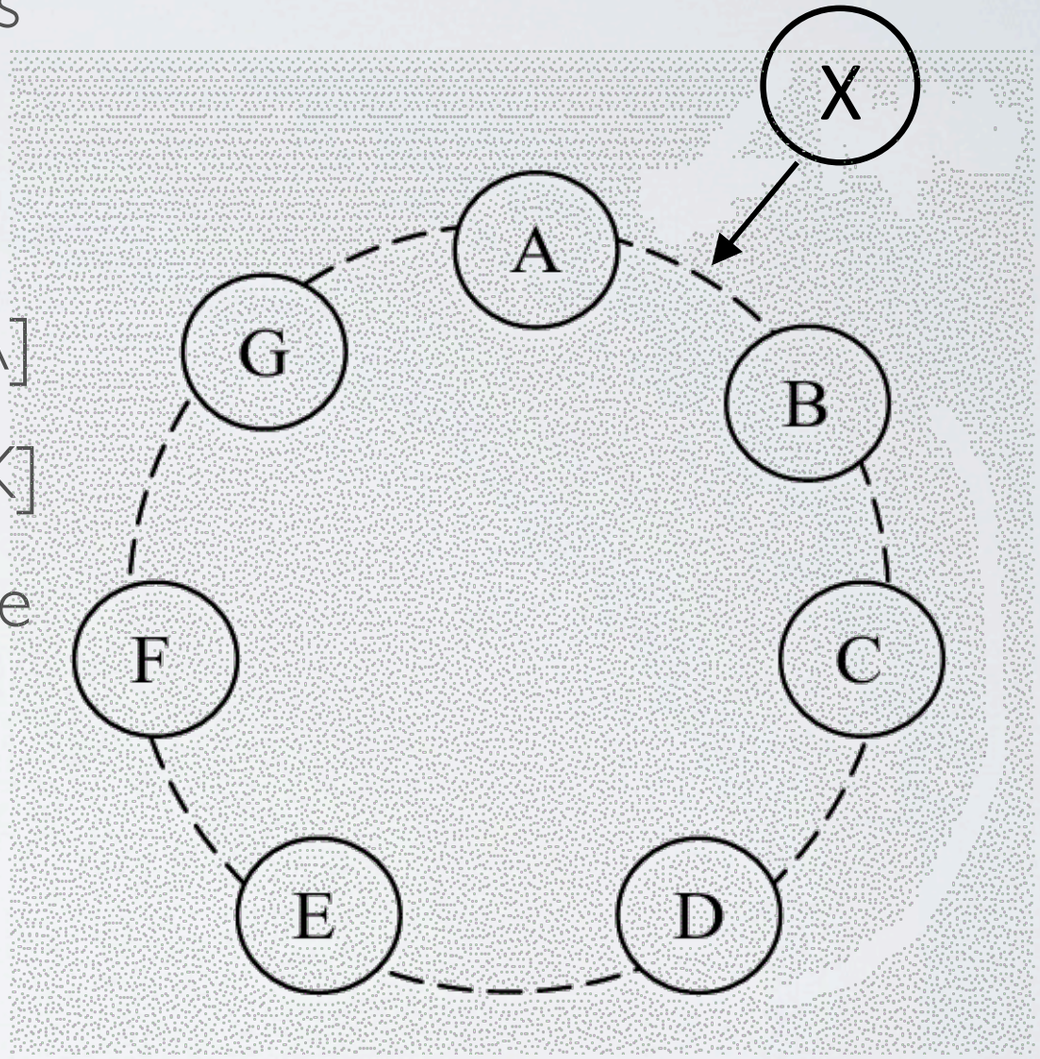
- challenges of random position assignment
  - it leads to non-uniform data and load distribution
  - the basic algorithm is oblivious to the performance heterogeneity of nodes
- map a node to multiple points instead of a single one
  - a virtual node looks like a single node in the system, but each node can be responsible for more virtual nodes
  - when a new node is added, it is assigned multiple positions or tokens
- advantages
  - if a node becomes unavailable, its load is spread across more nodes
  - when a node becomes available again (or in case of a new node), it accepts a roughly equivalent amount of load from the other nodes
  - the number of virtual nodes can be tuned on node's capacity

Dynamo - Architecture

- Node addition
  - when a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring
  - for every key range assigned to X, there are other nodes currently in charge of handling such range
  - due to the allocation of key ranges to X, some nodes no longer have to manage some keys, and transfer them to X
  - when a node is removed from the system, the reallocation of keys happens in a reverse process
  - operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes

Dynamo - Architecture

- X is added to the ring between A and B
- X is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]
- B no longer has to store the keys in (F, G]
- C no longer has to store the keys in (G, A]
- D no longer has to store the keys in (A, X]
- nodes B, C, D transfer to X the keys in the ranges they no longer have to store



Dynamo - Architecture

- Dynamic partitioning
  - addresses the need for horizontal scalability
- consistent hashing
  - keys are mapped to ring positions using a hash function
  - the output range of the hash is treated as a fixed circular space
  - departure or arrival of a node only affects its immediate neighbors
  - there is no need to redistribute all the keys

Dynamo - Architecture

- Dynamic partitioning
  - addresses the need for horizontal scalability
- consistent hashing
  - keys are mapped to ring positions using a hash function
  - the output range of the hash is treated as a fixed circular space
  - departure or arrival of a node only affects its immediate neighbors
  - there is no need to redistribute all the keys

Dynamo - Architecture

- in Amazon's environment
  - node outages are often transient but may last long
    - a node outage rarely signifies a permanent departure
    - it shouldn't result in rebalancing or repair
  - manual error could result in the unintentional startup of new Dynamo nodes
- use an explicit mechanism to add/remove nodes from the ring
  - an administrator issues a membership change to a node
  - membership changes history nodes can be removed/added many times

Dynamo - Architecture

- gossip-based protocol
  - propagates membership changes
  - maintains an eventually consistent view of membership
  - every second the nodes reconcile
    - their membership change histories
    - the mappings of virtual nodes to ring positions
- each node is aware of the token ranges handled by its peers

Dynamo - Architecture

- any node can receive client get and put operations for any key
- client strategies to select a node
  - route to a load balancer that selects a node based on load information
    - the client does not have to link any code specific to Dynamo
  - use a partition-aware library that routes requests to the coordinators of required keys
    - lower latency because a potential forwarding step is skipped
- the node handling a read/write operation is the coordinator for that operation

Dynamo - Architecture

- if the load balancer chooses a node not in the top N of the preference list, then the request is forwarded to the top node of the preference list
- read/write operations involve the first N healthy nodes in the preference list
  - this is the reason why such list includes more than N nodes
  - when there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed

Dynamo - Consistency

- consistency protocol
  - R: minimum number of nodes that must participate in a read
  - W: minimum number of nodes that must participate in a write
- several combinations
  - $R + W > N$: strong consistency
  - $R + W \leq N$: eventual consistency
  - $R = 1, W = N$: optimize reads
  - $R = N, W = 1$: optimize writes
- the latency of an operation is dictated by the slowest replica
- R and W are usually configured to be < N, to provide better latency

Dynamo - Consistency

- put() request
  - the coordinator of the key writes the new version locally
  - then sends the new version to the other replicas
  - if at least W-1 nodes respond then the write is successful
- read() request
  - the coordinator of the request demands all existing versions from the top N nodes
  - then waits for R responses before returning the result
  - if multiple versions of are found, it returns all those that are causally unrelated

Dynamo - Consistency

- a put() may return before the update is applied to all the replicas
  - a subsequent get() may return an outdated object
- under certain failure scenarios (server outages, network partitions), updates may not arrive at all replicas for an extended period of time
- example: Shopping Cart
  - requirement: an "Add to Cart" operation can never be forgotten or rejected
  - if the most recent state of the cart is unavailable, and a change is made to an older version, that change is still meaningful and should be preserved
  - but it shouldn't supersede the currently unavailable state of the cart, because the latter may contain changes that should be preserved!!
  - in Dynamo, "Add to Cart" and "Delete from Cart" are put() requests

Dynamo - Consistency

- when a customer wants to add an item to the cart and the latest version is not available, the item is added to the older version
- the divergent versions are reconciled later
  - syntactic reconciliation
    - if new version subsumes previous one(s), the reconciliation is easy
  - semantic reconciliation
    - version branching may happen (due to failures combined with concurrent updates), resulting in conflicting versions of an object
    - the client must reconcile the multiple branches
- Shopping Cart example:
  - an "add to cart" is never lost but deleted items can resurface

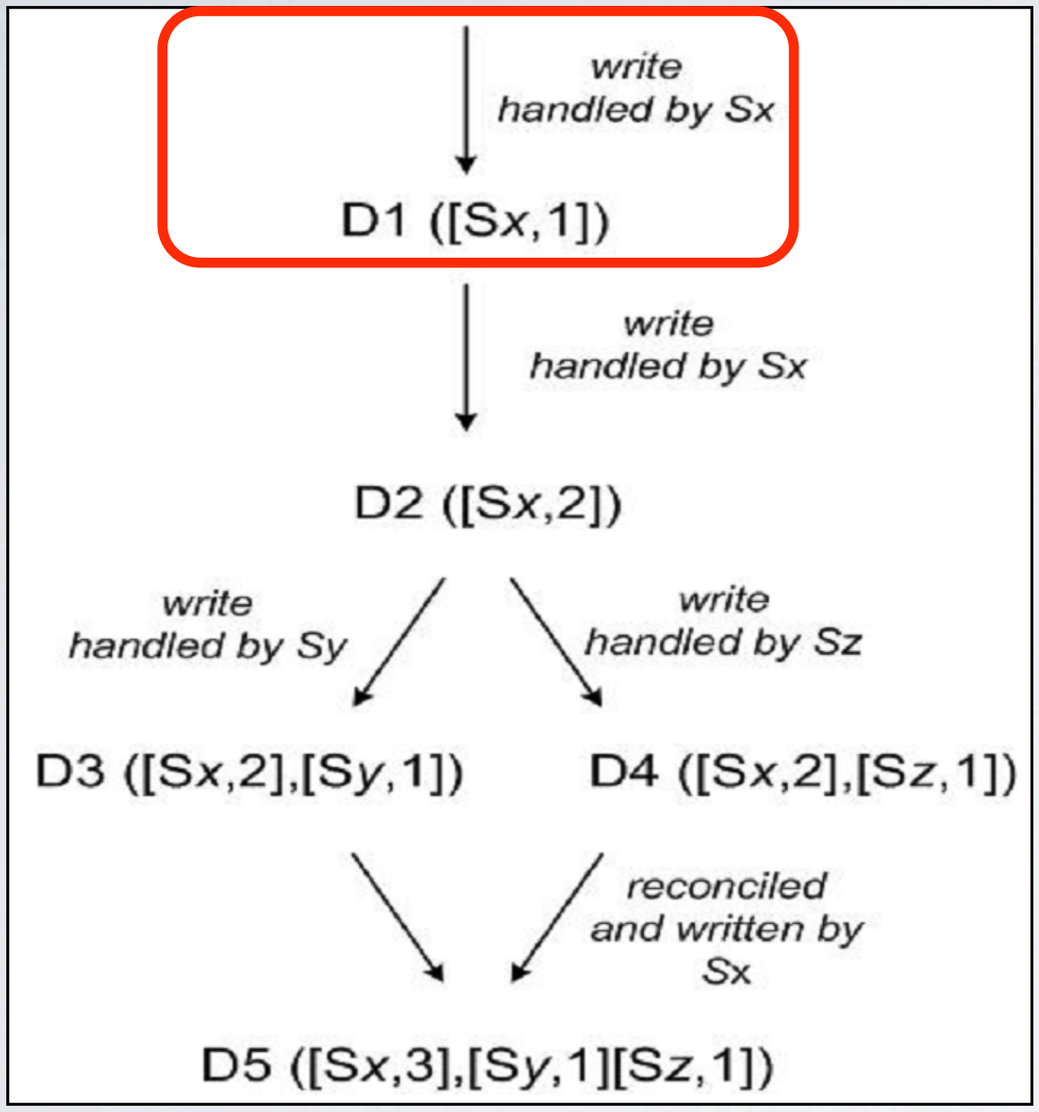
Dynamo - Consistency

- vector clocks are used to capture the causality among versions
  - a vector clock is associated with every version of every object
  - it is a set of pairs [node ID, operation index]
  - to determine if two versions of an object are on parallel branches or have a causal ordering, their vector clocks have to be checked
- when a client wants to update an object, it must specify which version it is updating
  - by passing the context it obtained from an earlier read operation
  - the context contains the vector clock information

Dynamo - Consistency

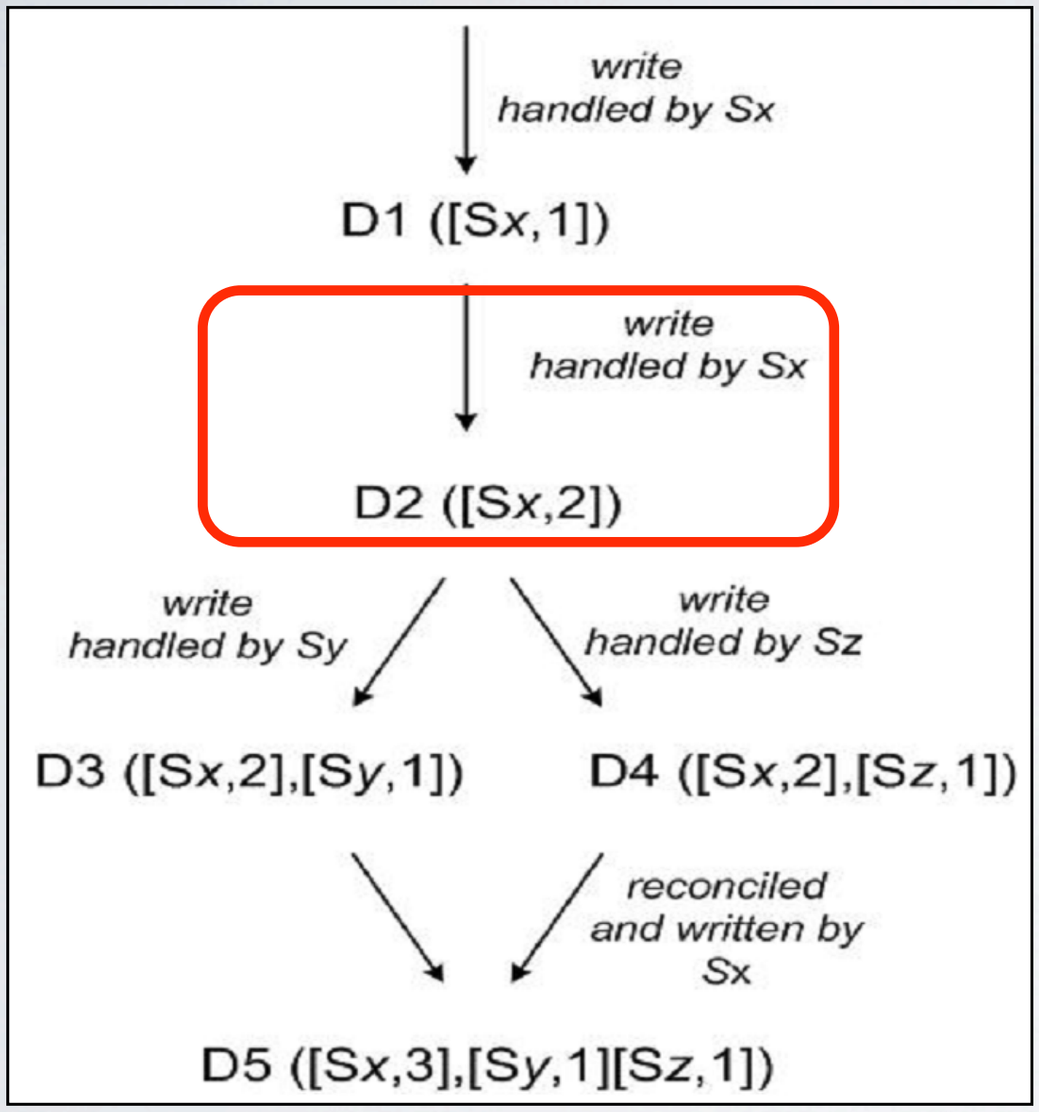
- semantic reconciliation
  - if multiple branches are found that cannot be syntactically reconciled, all the versions are returned by a read operation
  - the context includes the corresponding version information
  - an update using this context is considered to have reconciled the divergent versions and the branches are collapsed

Dynamo - Example



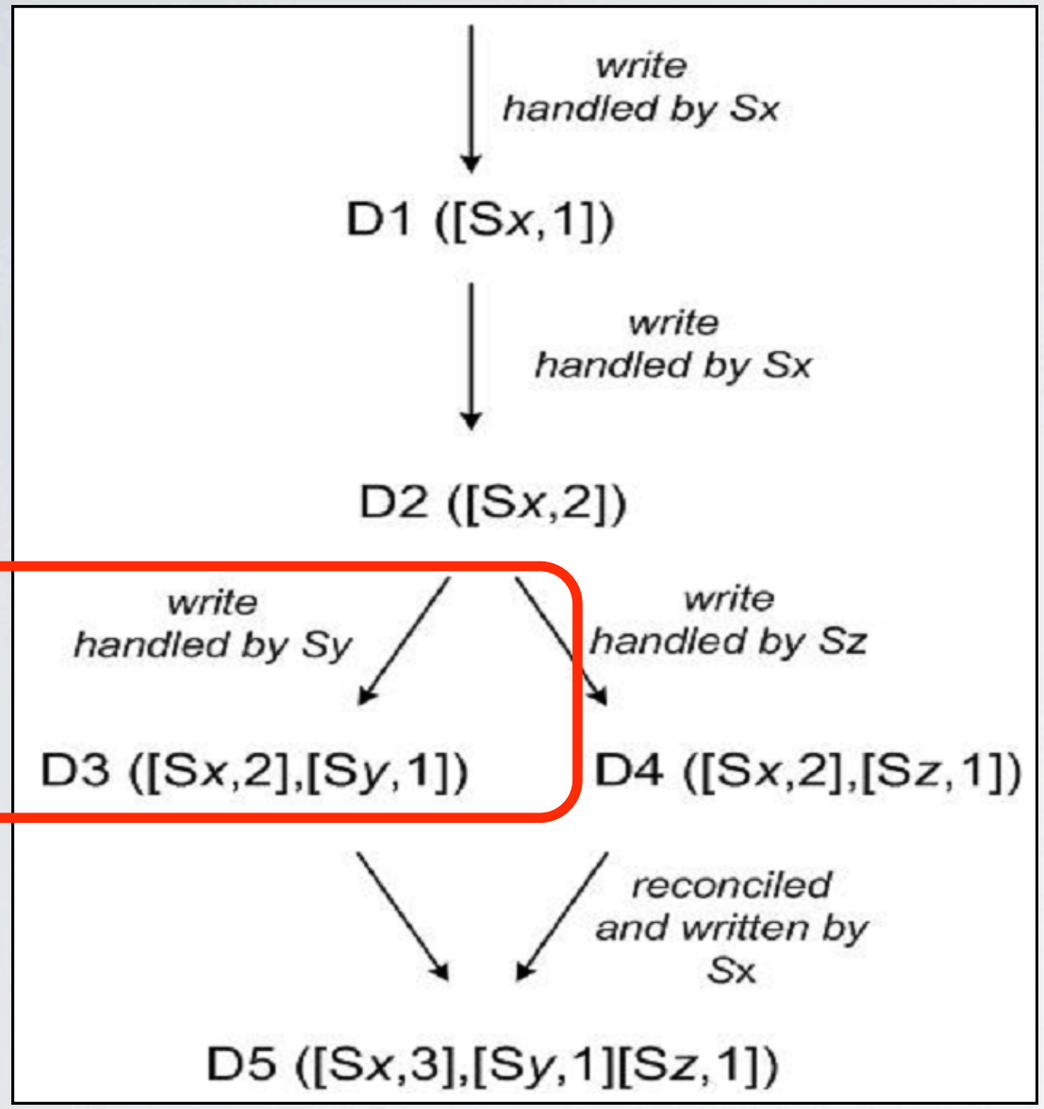
- a client writes a new object
- node Sx handles the write: increases its sequence number and uses it to create the data's vector clock
- the system now has the object D1 and its associated clock ([Sx, 1])

DYNAMO - EXAMPLE



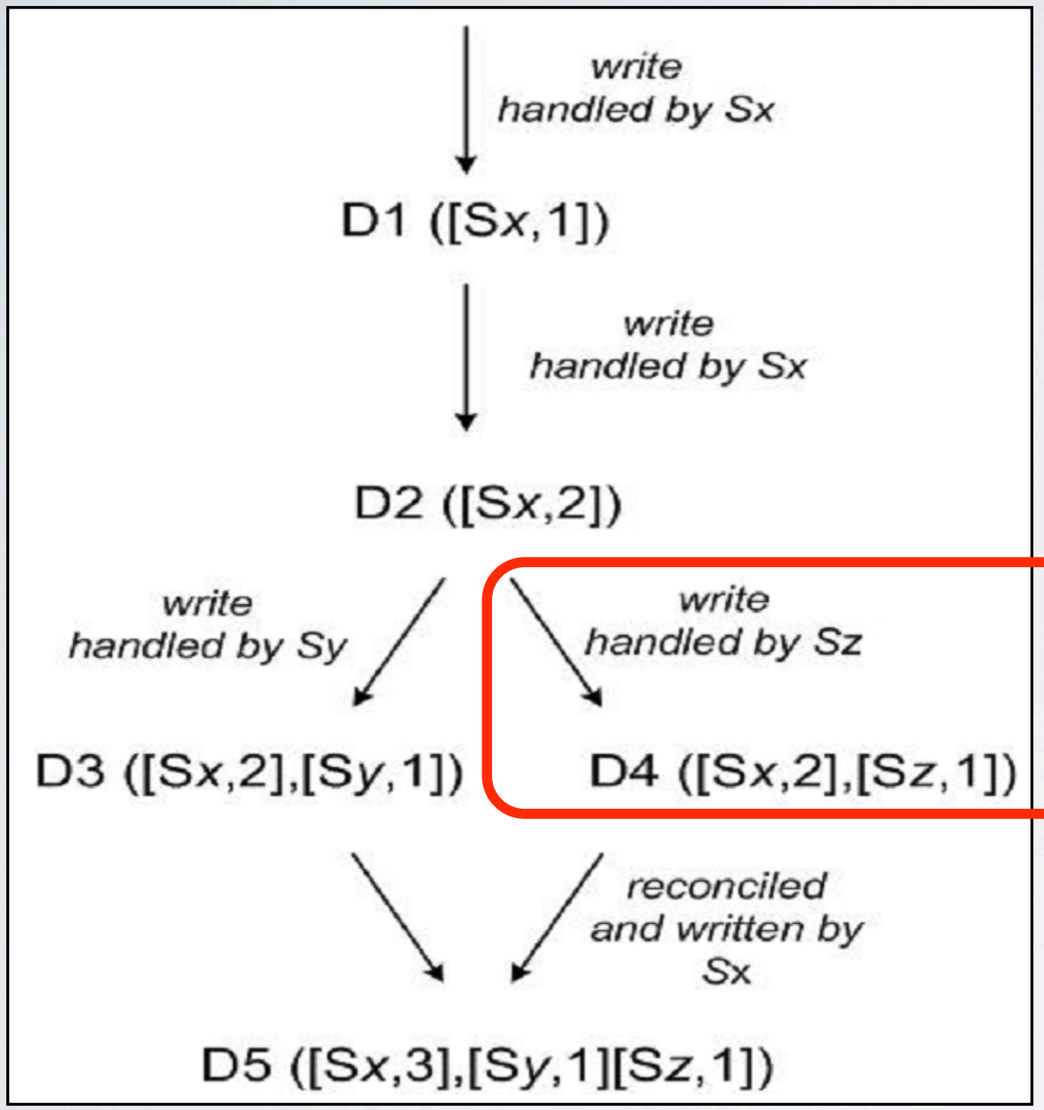
- the same client updates the object
- the same node Sx handles this request
- the system now has the object D2 and its associated clock ([Sx, 2])
- D2 descends from D1 and therefore over-writes D1
- there may be replicas of D1 lingering at nodes that have not yet seen D2

DYNAMO - EXAMPLE



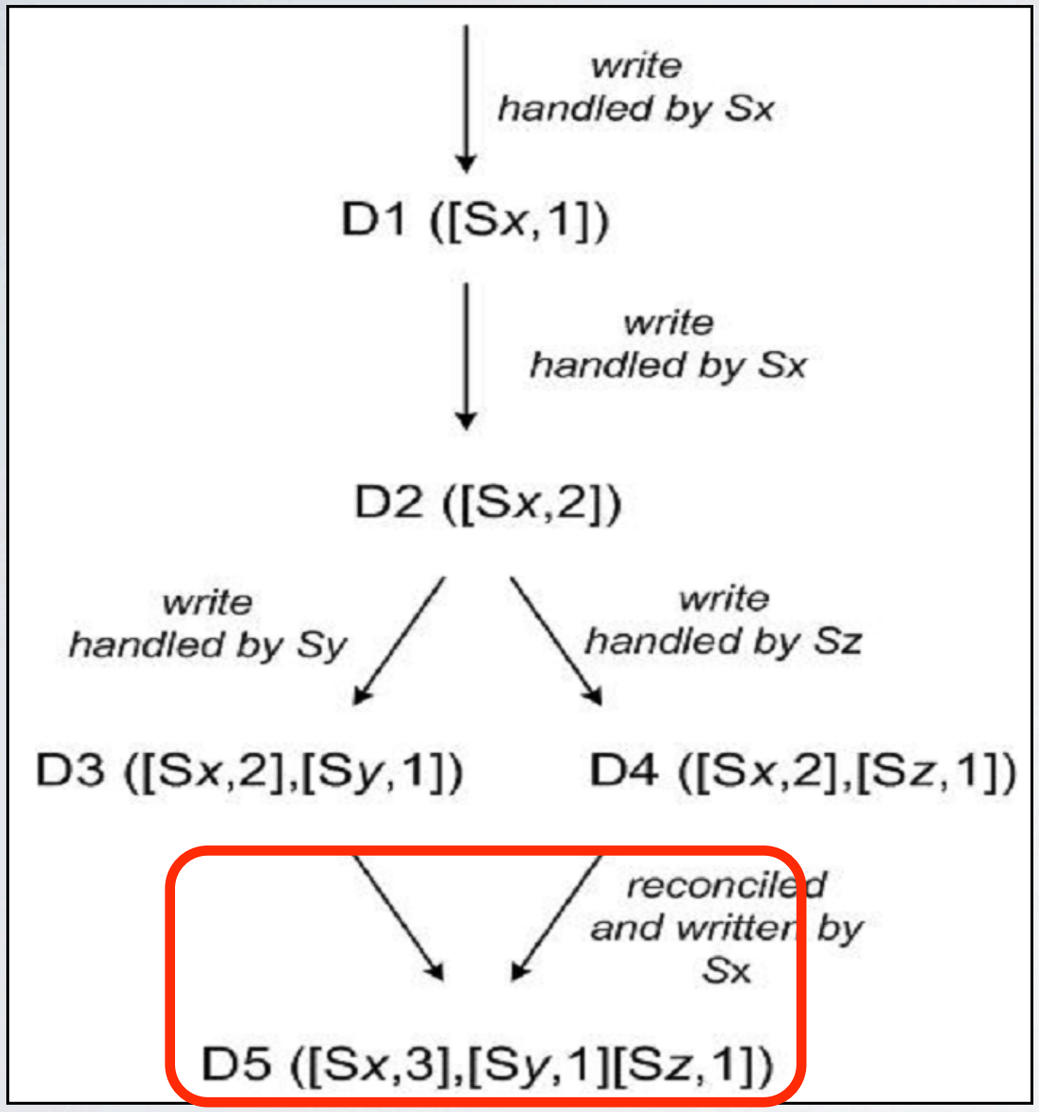
- the same client updates the object again
- a different server Sy handles the request
- the system now has data D3 and its associated clock ([Sx, 2], [Sy, 1])

Dynamo - Example



- a different client reads D2 and then tries to update it
- another server Sz does the write
- the system now has D4 with vector clock ([Sx, 2], [Sz, 1])
- a node that is aware of D1 or D2 and receives D4 could determine that D1 and D2 are overwritten by D4 and can be garbage collected
- a node that is aware of D3 and receives D4 will find that there is no causal relation between them
- there are changes in D3 and D4 that are not reflected in each other and both versions must be kept

DYNAMO - EXAMPLE

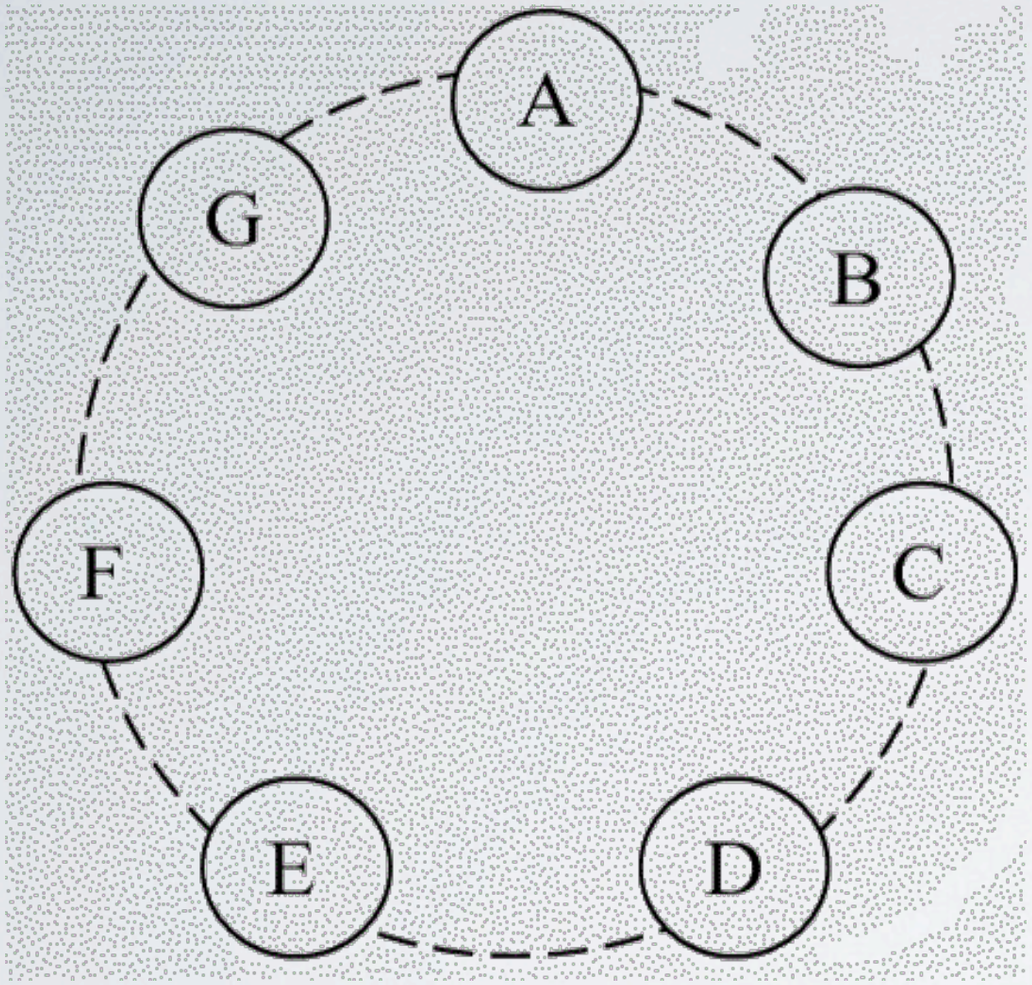


- some client reads both D3 and D4
- the context will reflect that both values were found by the read
- the context is a summary of the clocks of D3 and D4 ([Sx, 2], [Sy, 1], [Sz, 1])
- node Sx handles the write
- the system now has data D5 and its associated clock ([Sx, 3], [Sy, 1], [Sz, 1])
- reconciliation completed

DYNAMO - FAULT TOLERANCE

- with a traditional quorum approach, Dynamo would be unavailable during server failures and network partitions
- use a sloppy quorum
  - read/write are performed on the first N healthy nodes from the preference list
  - they may not be the first N nodes met while walking the ring clockwise

DYNAMO - FAULT TOLERANCE



- **Hinted Handoff**
- if A is temporarily down/ unreachable during a write, then a replica that would normally have lived on A will now be sent to D
- the replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica
- upon detecting that A has recovered, D will attempt to deliver the replica to A

DYNAMO - FAULT TOLERANCE

- using hinted handoff, Dynamo ensures that the r/w are not failed due to temporary node or network failures
- applications needing the highest level of availability can set W to 1
  - this way a write is accepted as long as a single node has written the key
  - the write request is only rejected if all nodes in the system are unavailable
- It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s)
  - data center failures happen due to power outages, cooling failures, network failures, and natural disasters
  - Dynamo replicates each object across multiple data centers
  - preference lists are constructed such that the nodes are spread across multiple data centers

DYNAMO - FAULT TOLERANCE

- problematic scenario: hinted replicas become unavailable before they can be returned to the original replica node
- solution: anti-entropy protocol to keep the replicas synchronized
- Uses Merkle trees
  - hash tree where leaves are hashes of single objects
  - parent nodes higher in the tree are hashes of their children
  - each branch of the tree can be checked independently without requiring nodes to download the entire tree

DYNAMO - FAULT TOLERANCE

- synchronization mechanism
  - if the hash values of the root of two trees are equal then
    - the values of the leaf nodes in the tree are equal
    - the nodes require no synchronization
  - otherwise
    - the nodes exchange the hash values of children
    - the process continues until it reaches the leaves of the trees
    - at that point, "out of sync" keys can be identified

DYNAMO - FAULT TOLERANCE

- advantages of Merkle trees
  - minimize the amount of data to be transferred for synch
  - reduce the number of required disk reads
- Dynamo uses Merkle trees for anti-entropy as follows
  - each node maintains a separate Merkle tree for each key range it hosts
  - two nodes exchange the root of the Merkle tree of the key ranges that they host in common
  - using the synch mechanism described before, the nodes determine if they have any differences and perform the appropriate synch

DYNAMO - FAULT TOLERANCE

- Fault detection:
  - avoid to communicate with unreachable peers during get()/put() operations and transfers of partitions and hinted replicas
  - a purely local notion of failure detection is entirely sufficient
    - node A may consider node B failed if B does not respond to A's messages
    - with a steady rate of client requests, A quickly discovers that B is unresponsive when B fails to respond to a message
    - A then uses alternate nodes to service requests that map to B's partitions
    - A periodically retries B to check for the its recovery
    - in the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive

Dynamo and CAP

- sloppy quorum allows to serve requests if some replicas are not available
- even if all the nodes responsible for a certain key are down, hinted handoff allows to temporarily store the write to another node
- consistency conflicts are resolved
  - at read time through syntactic or semantic reconciliation
  - in the background using Merkel trees
- eventually, all the replicas will converge □ eventual consistency
- ring partitions and concurrent r/w operations can make distinct clients see distinct versions of the same key

BIBLIOGRAPHY

1. Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform, VLDB 2008.
2. Avinash Lakshman, Prashant Malik. Cassandra: a structured storage system on a P2P network. SPAA 2009.
3. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data, OSDI 2006
4. S. Ghemawat, H. Gobioff, S. Leung. The Google File System", SOSP 2003
5. M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems, OSDI 2006
6. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels,. Dynamo: Amazon's Highly Available Key-value Store, SOSP 2007