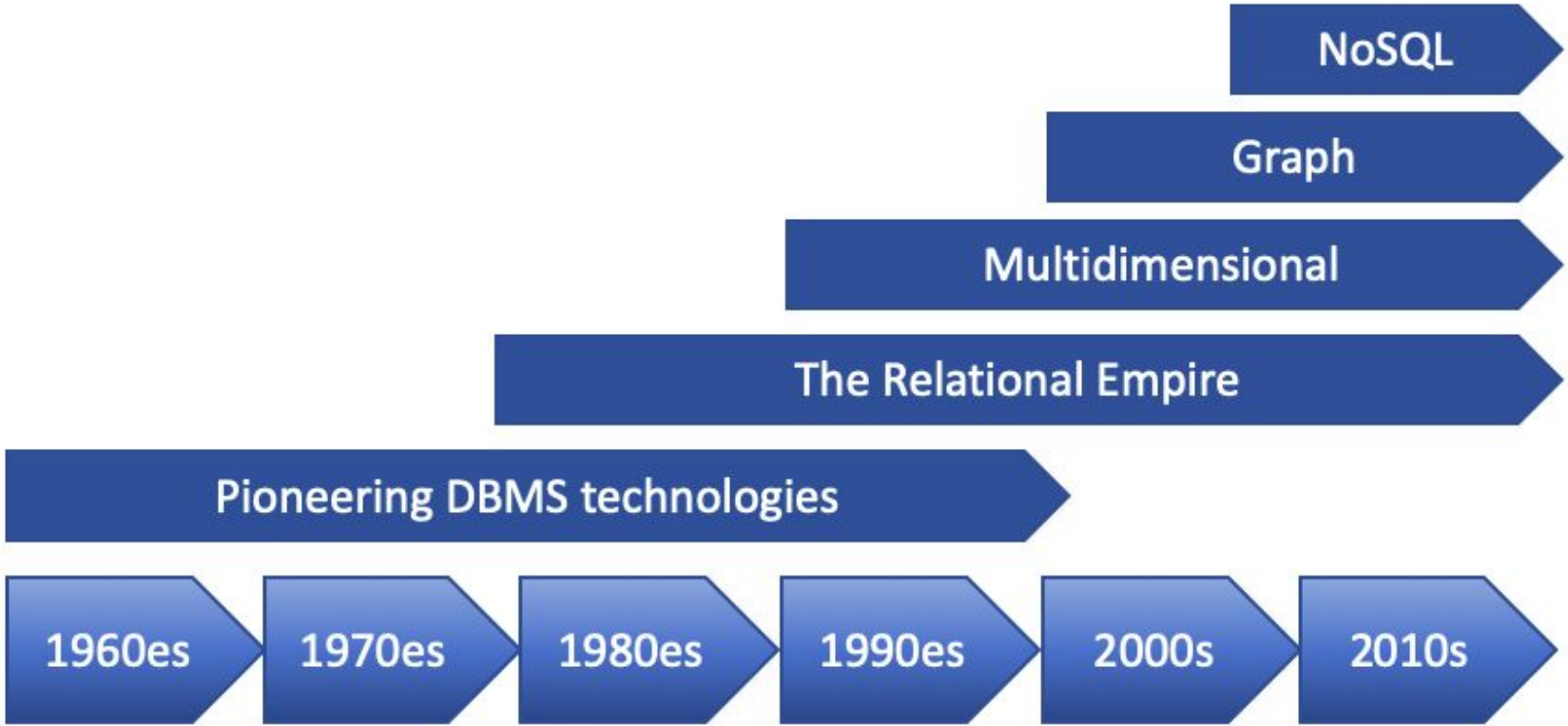
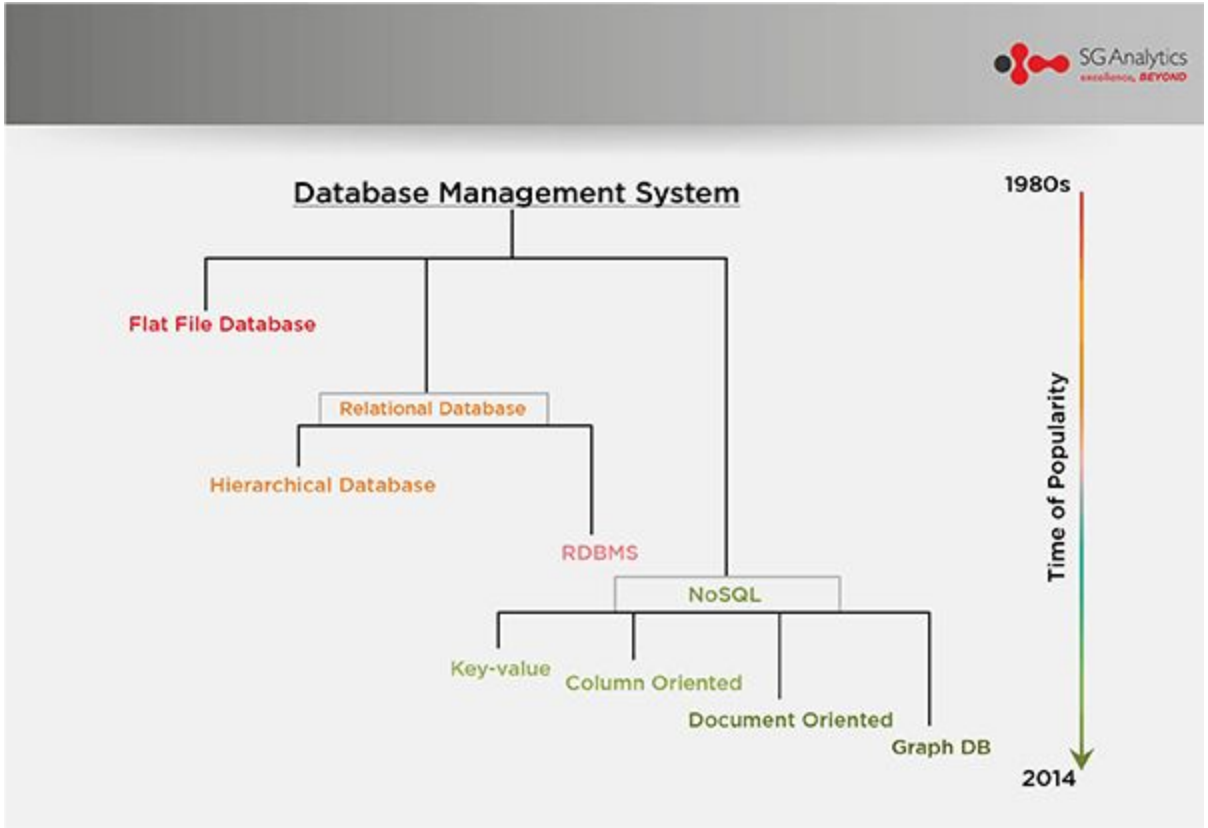
Data Engineering: DataBase


Database History

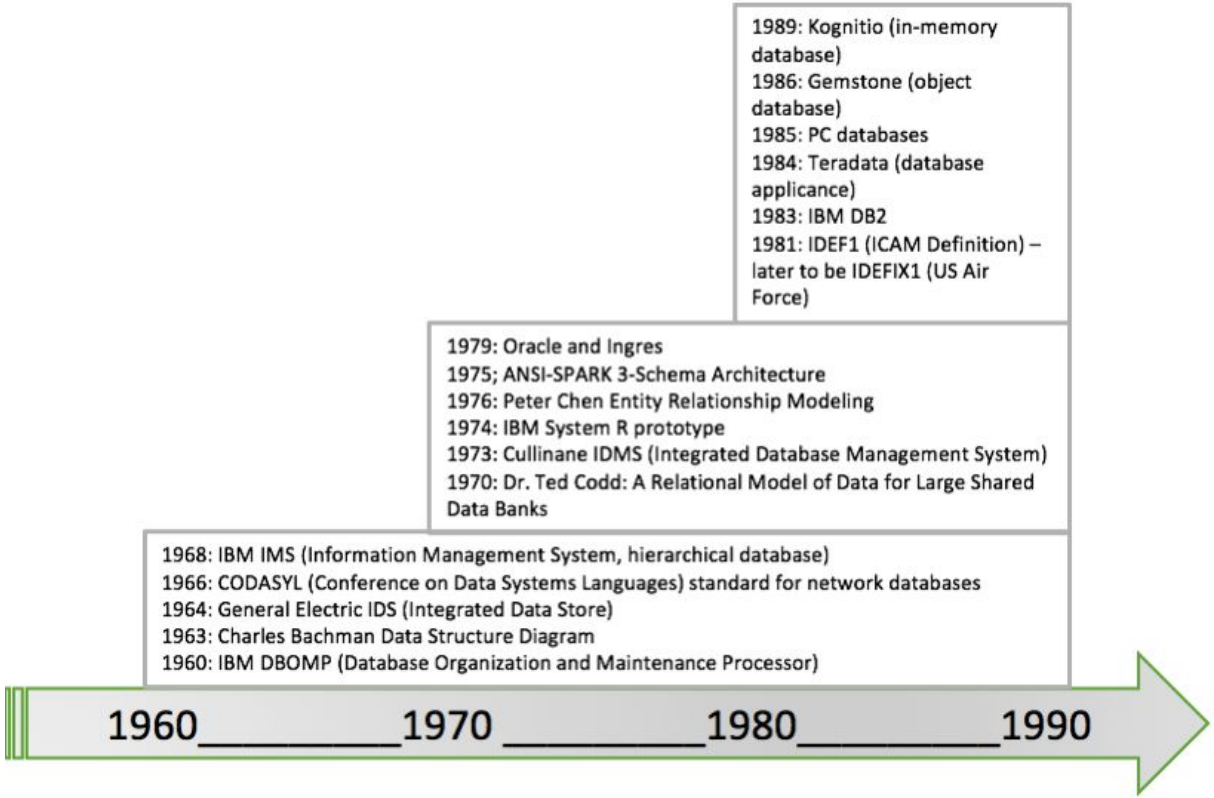
A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

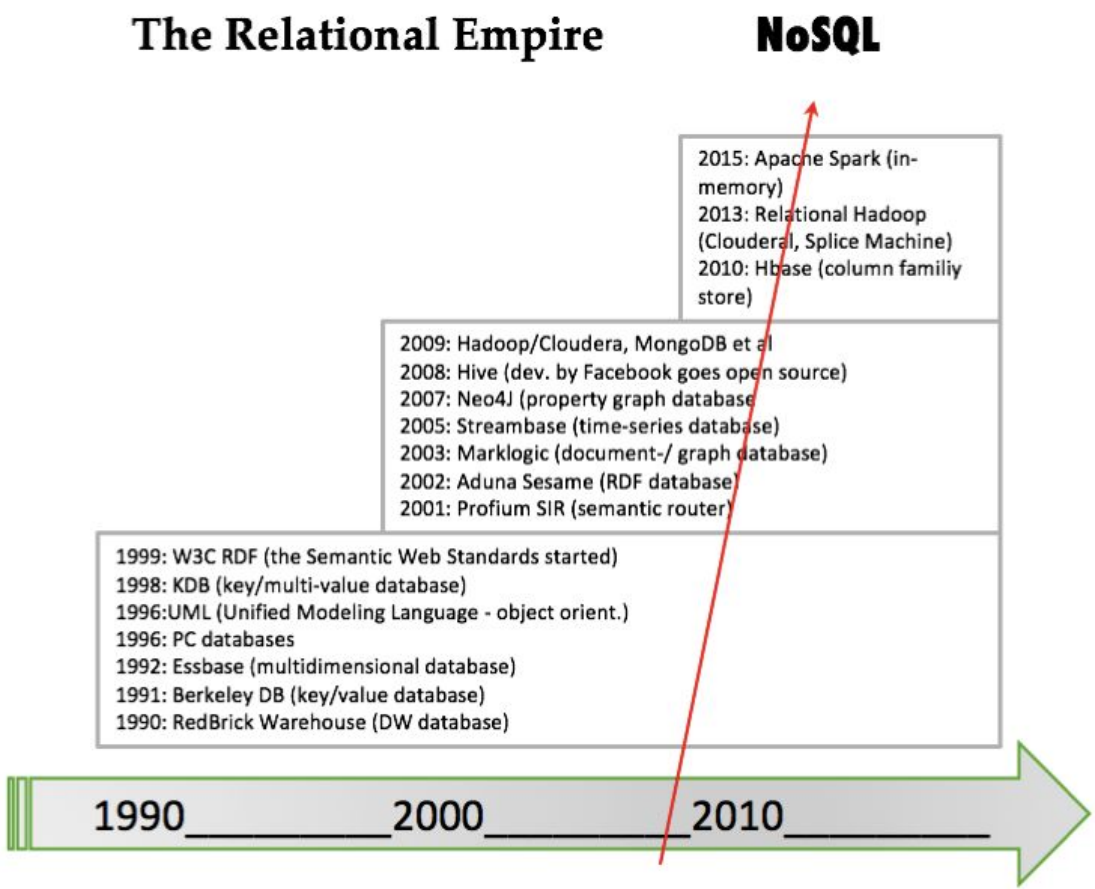

NoSQL
Graph
Multidimensional
The Relational Empire
Pioneering DBMS technologies
1960es
1970es
1980es
1990es
2000s
2010s

A database management system (DBMS) is a software package designed to define, manipulate, retrieve and manage data in a database. A DBMS generally manipulates the data itself, the data format, field names, record structure and file structure. It also defines rules to validate and manipulate this data.


SG Analytics
Database Management System
1980s
Flat File Database
Relational Database
Hierarchical Database
RDBMS
NoSQL
Key-value
Column Oriented
Document Oriented
Graph DB
Time of Popularity
2014

The Pioneers of the DBMS Trail




The Relational Empire
NoSQL
2015: Apache Spark (in-memory)
2013: Relational Hadoop (Clouderal, Splice Machine)
2010: Hbase (column familiy store)
2009: Hadoop/Cloudera, MongoDB et al
2008: Hive (dev. by Facebook goes open source)
2007: Neo4J (property graph database)
2005: Streambase (time-series database)
2003: Marklogic (document-/ graph database)
2002: Aduna Sesame (RDF database)
2001: Profium SIR (semantic router)
1999: W3C RDF (the Semantic Web Standards started)
1998: KDB (key/multi-value database)
1996:UML (Unified Modeling Language - object orient.)
1996: PC databases
1992: Essbase (multidimensional database)
1991: Berkeley DB (key/value database)
1990: RedBrick Warehouse (DW database)
1990
2000
2010

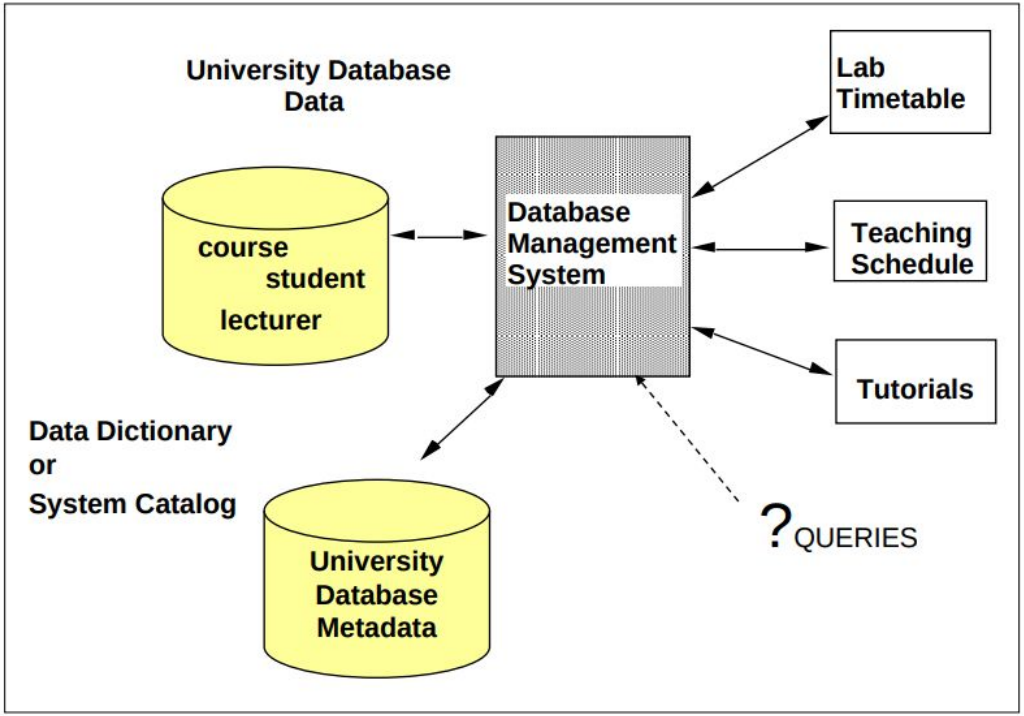
Operations with Databases

- **Design**
  - *Define* structure and types of data
- **Construction**
  - *Create* data structures of DB, *populate* DB with data
- **Manipulation of Data**
  - *Insert*, *delete*, *update*
  - *Query*: “Which department pays the highest salary?”
  - Create *reports*:
    “List monthly salaries of employees, organised by department, with average salary and total sum of salaries for each dept”

An Ideal DB Implementation Should Support:

- Structure
  - data types
  - data behaviour
- Persistence
  - store data on secondary storage
- Retrieval
  - a declarative query language
  - a procedural database programming language
- Performance
  - retrieve and store data quickly
- Data Integrity
- Sharing
  - concurrency
- Reliability and resilience
- Large data volumes

DBMS: A Logical Interface



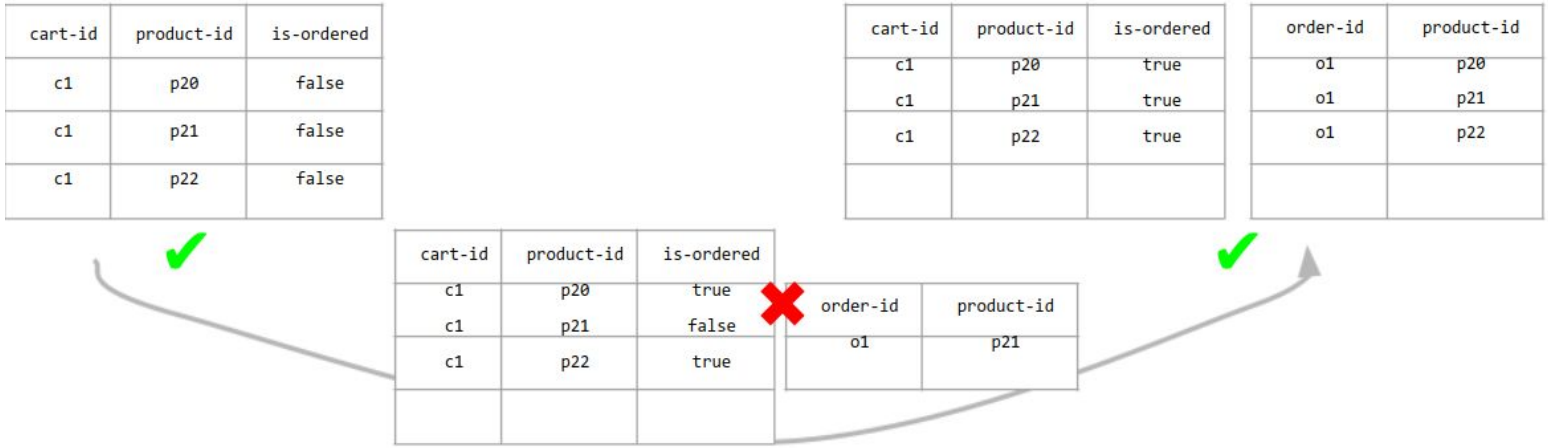
Transactions & ACID properties

Why are transactions needed?

- Failures that cause inconsistency
- Concurrency that causes loss of isolation

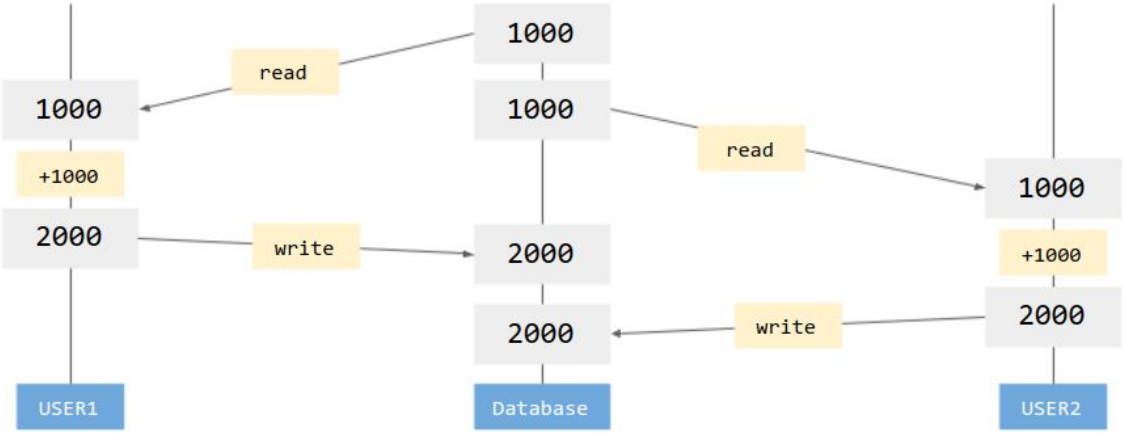
Moving from one valid state to another

1. Consistency even if failures happen
    a. Suppose data is being read, written, read, written and suddenly there is a system failure (say machine powers off), actions may remain incomplete with an invalid status
    b. Eg: An e-commerce order is being placed. While processing the order (calculating price, applying offers) something goes wrong and only half the items are marked as ordered. What should happen?
    c. Ideally, the entire order should ‘rollback’



Isolation during concurrent usage

1. Isolation during concurrent usage
    a. User1 reads a value (say a bank balance) and then writes an updated value which is +1000 of the old value
    b. However, while user1 was reading, user2 also read the value and was adding +1000 to the old value
    c. Now, the final result will be whichever write is last!



What are transactions?

1. Database transactions provide an ‘all-or-nothing’ feature
2. Transaction:
    a. BEGIN TRANSACTION
    b. SELECT
    c. INSERT
    d. UPDATE
    e. SELECT
    f. DELETE
    g. END TRANSACTION
3. **Any failure in the middle, and each operation is rolled-back (like an undo)**
4. **During the time a transaction happens, no change in the values to the same rows will be allowed**
5. Transactions are extremely valuable for financial or transactional applications

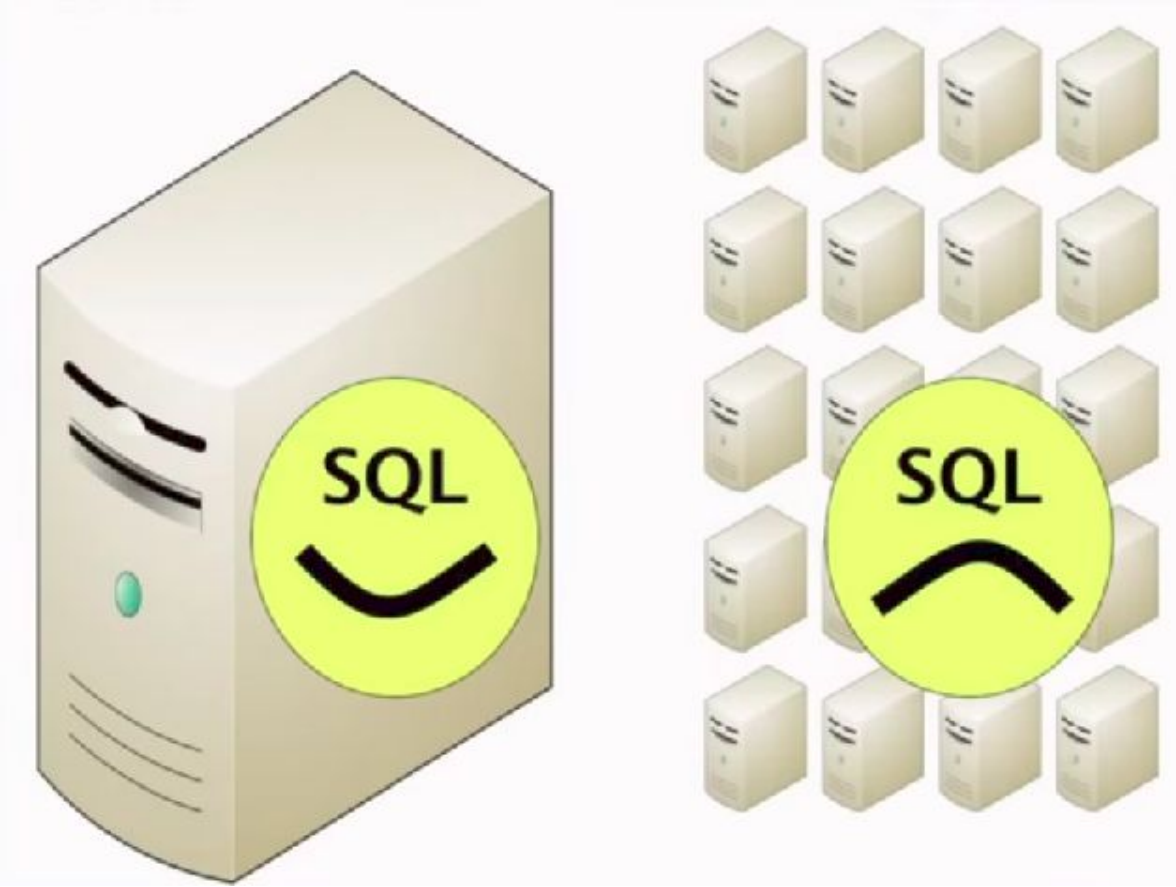
What are transactions?

1. Database transactions provide an ‘all-or-nothing’ feature
2. Transaction:
    a. BEGIN TRANSACTION
    b. SELECT
    c. INSERT
    d. UPDATE
    e. SELECT
    f. DELETE
    g. END TRANSACTION
3. **Any failure in the middle, and each operation is rolled-back (like an undo)**
4. **During the time a transaction happens, no change in the values to the same rows will be allowed**
5. Transactions are extremely valuable for financial or transactional applications

ACID properties

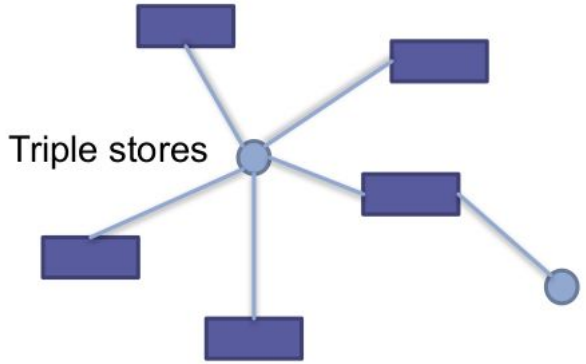
Database transactions exhibit the following properties:

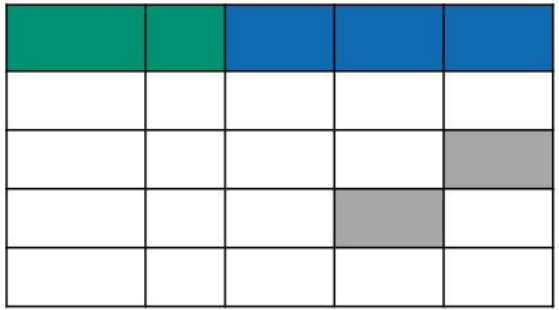
1. Atomicity: Operations are all-or-nothing.
2. Consistency: The entirely of the database (constraints, indexes etc.) moves from one consistent state to another. It is never left in an inconsistent state.
3. Isolation: Concurrent transactions happen as if they had happened without other transactions existing. As if they had happened in a perfect sequence one after another.
4. Durability: Transactions should be committed so that once a transaction is marked as completed by the database, it remains done whether there is a system failure or a power failure etc.


SQL
SQL

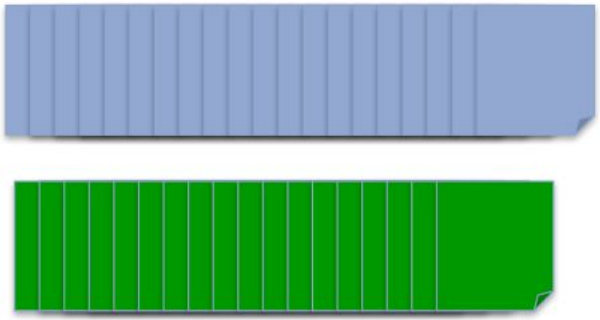
2000s: The NoSQL Era

Key-value stores





Column stores



Document stores


APACHE
HBASE


cassandra


DYNOMITE


mongoDB®


redis

Characteristics of NoSQL

non-relational

open-source

cluster-friendly

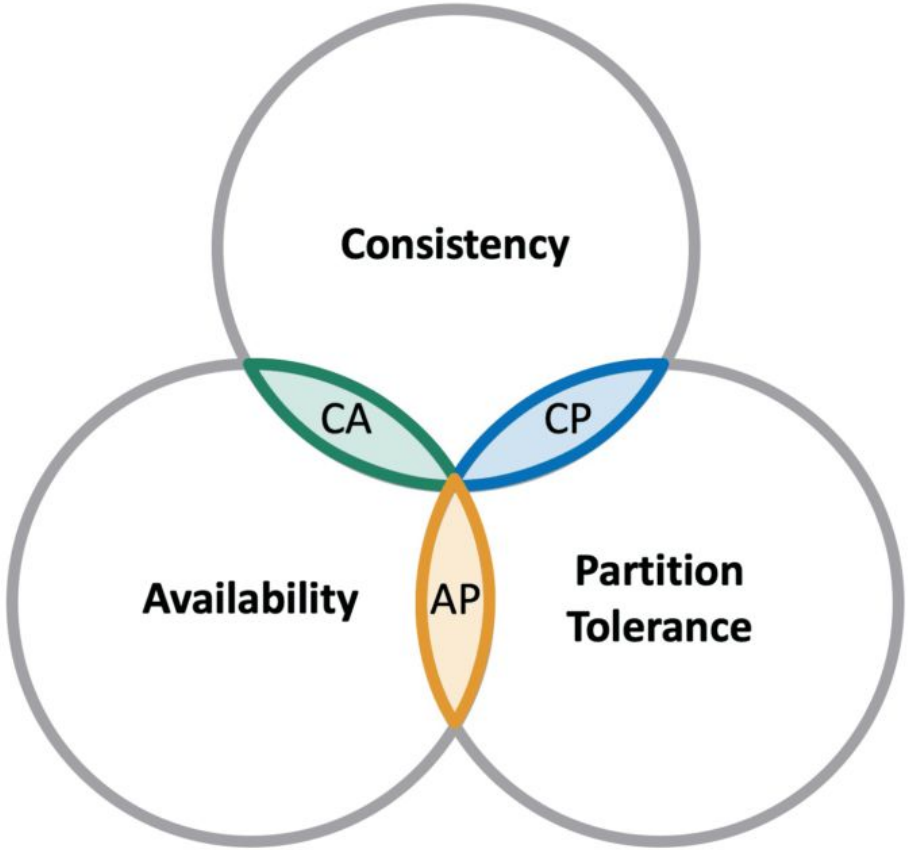
21st Century Web

schema-less

Big Storage

- Performance → Sharding
- Faults → Tolerance
- Tolerance → Replication
- Replication → InConsistency
- Consistency → Low Performance

Cap Theorem




Google
Bigtable
amazon.com
Dynamo

BigTable

- Development began in 2004 at Google (published 2006)
- A Bigtable is a sparse, distributed, persistent multidimensional sorted map.

Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands of q/sec
  - 100TB+ of satellite image data

Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls

BigTable

- Distributed multi-level map
  - With an interesting data model
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

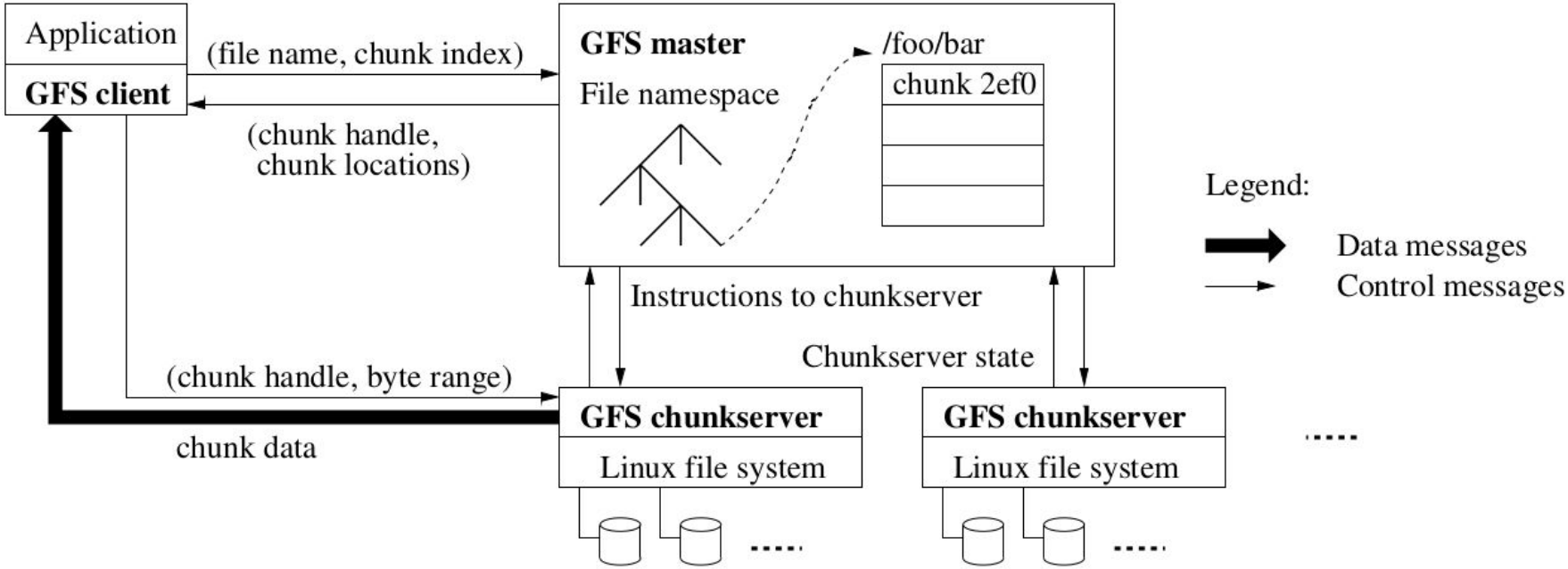
Background: Building Blocks

Building blocks:

- Google File System (GFS): Raw storage
- Scheduler: schedules jobs onto machines
- Lock service: distributed lock manager
  - also can reliably hold tiny files (100s of bytes) w/ high availability
- MapReduce: simplified large-scale data processing

BigTable uses of building blocks:

- GFS: stores persistent state
- Scheduler: schedules jobs involved in BigTable serving
- Lock service: master election, location bootstrapping
- MapReduce: often used to read/write BigTable data


Application
GFS client
(file name, chunk index)
(chunk handle,
chunk locations)
GFS master
File namespace
/foo/bar
chunk 2ef0
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
GFS chunkserver
Linux file system
GFS chunkserver
Linux file system
Legend:
Data messages
Control messages

GFS Architecture

- Single Master
- Chunckservers
- Clients
- Fix-Sized chunks
- HeartBeat
- Replication
- No Caching

Interface

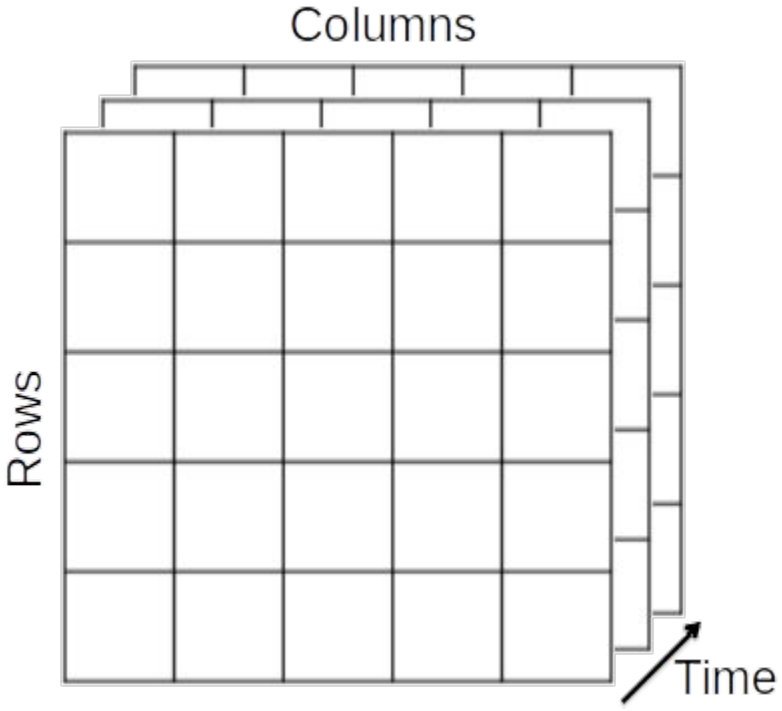
- Create
- Delete
- Open
- Close
- Read
- Write

MetaData

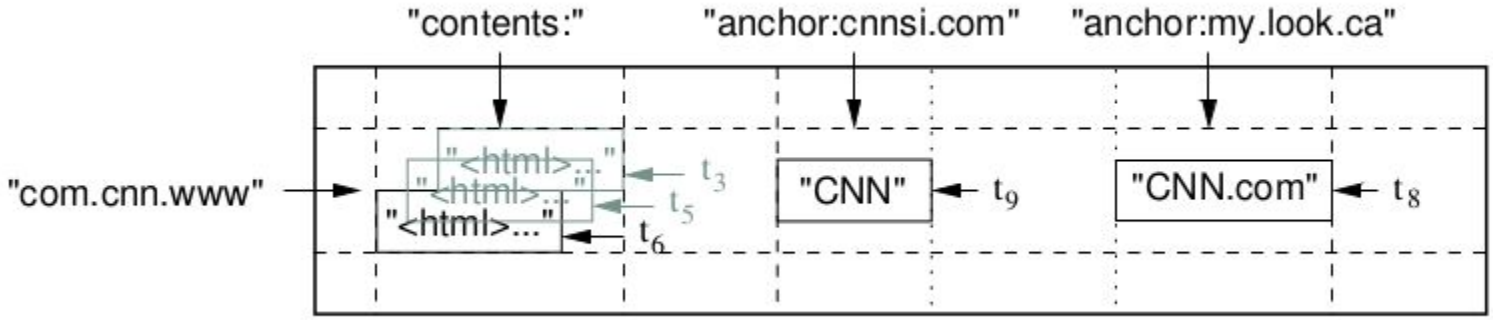
- File and Chunk namespace
- Mapping from Files to Chunks
- Locations of Chunk’s Replicas

The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted arrays of bytes.

(row:string, column:string, time:int64) → string


Columns
Rows
Time

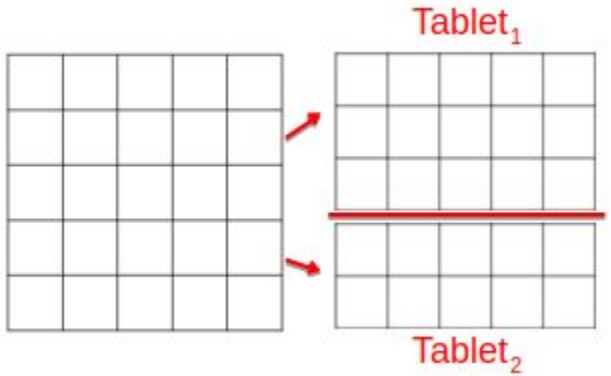
Webtable


"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"<html>..."
"<html>..."
t3
t5
t6
"CNN"
t9
"CNN.com"
t8

Rows

- Name is an arbitrary string
  - Access to data in a row is atomic
  - Row creation is implicit upon storing data
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines

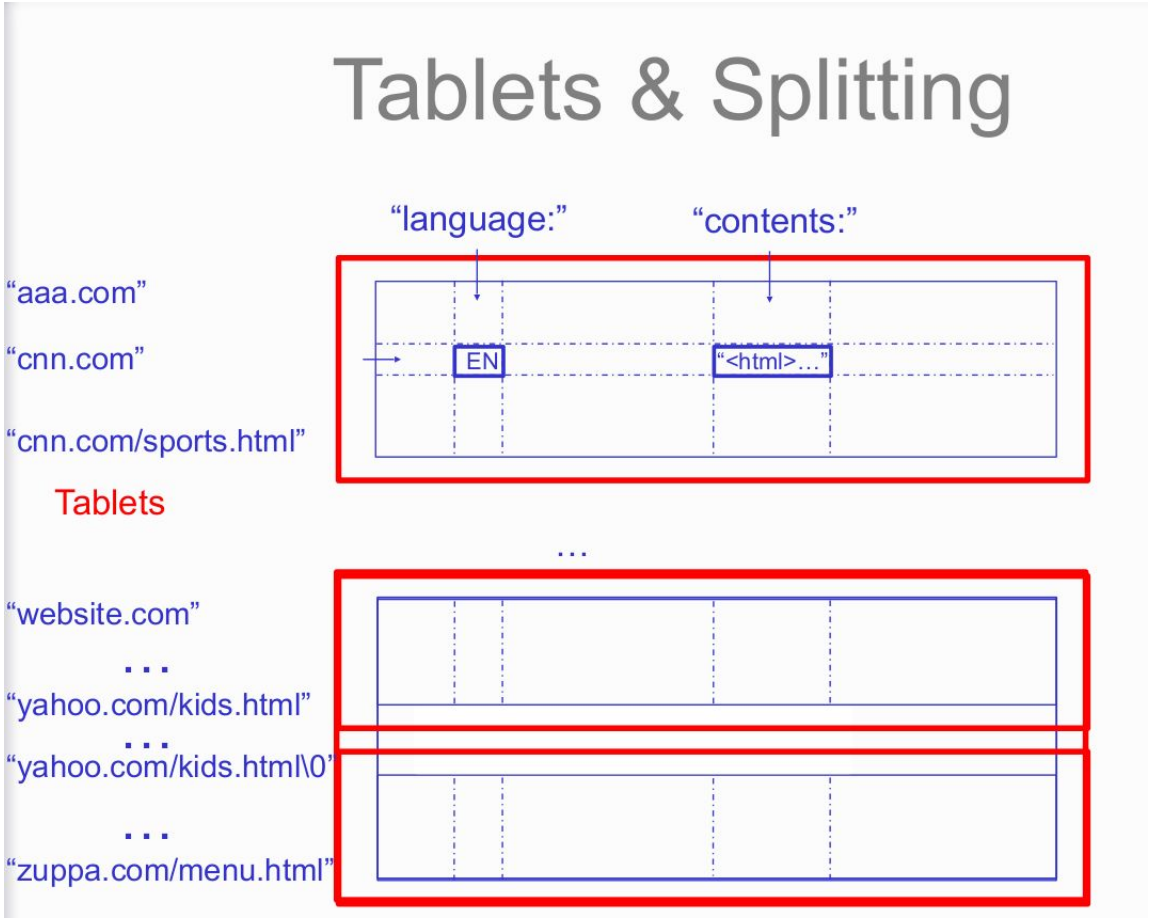
The entire BigTable is split into tablets of continuous ranges of rows

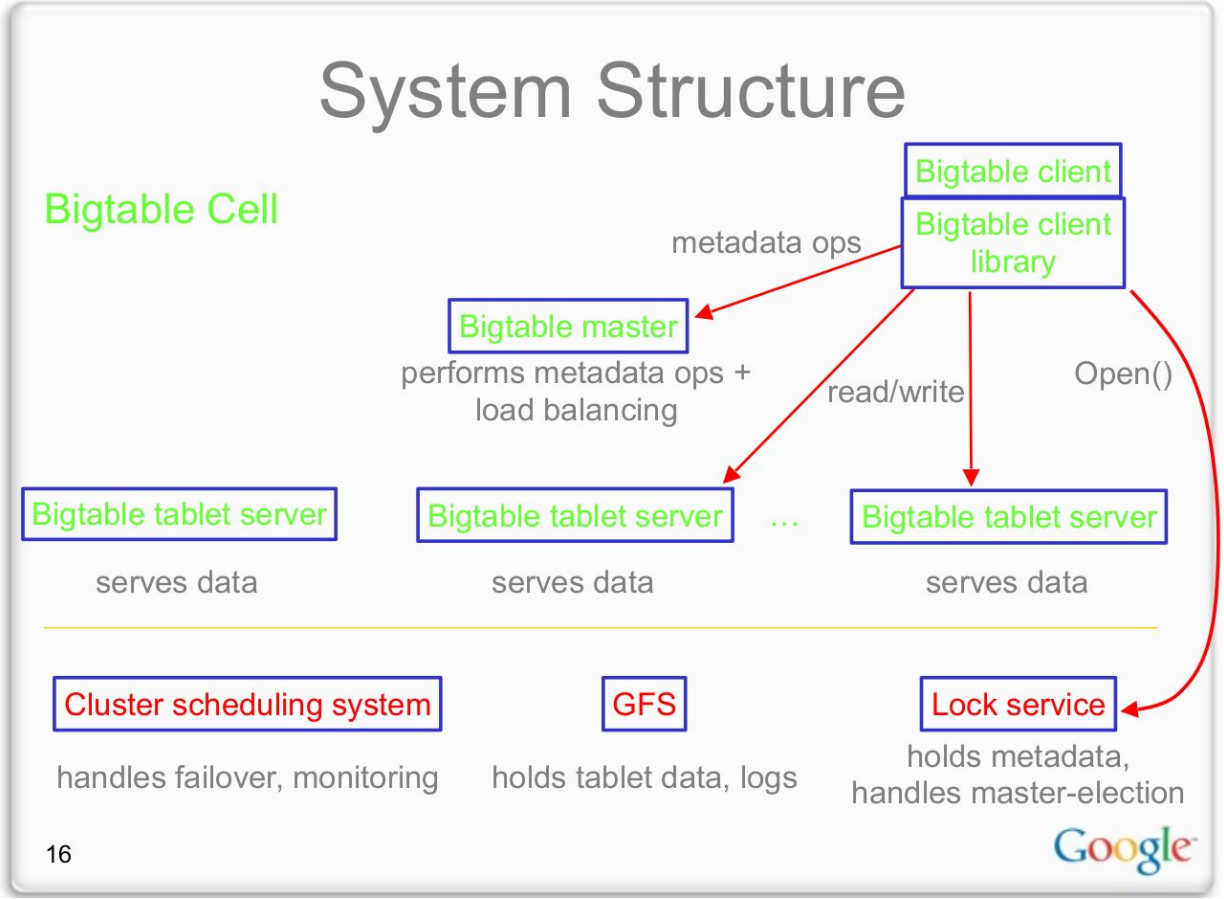


Tablets

- Large tables broken into ***tablets*** at row boundaries
  - Tablet holds contiguous range of rows
    - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

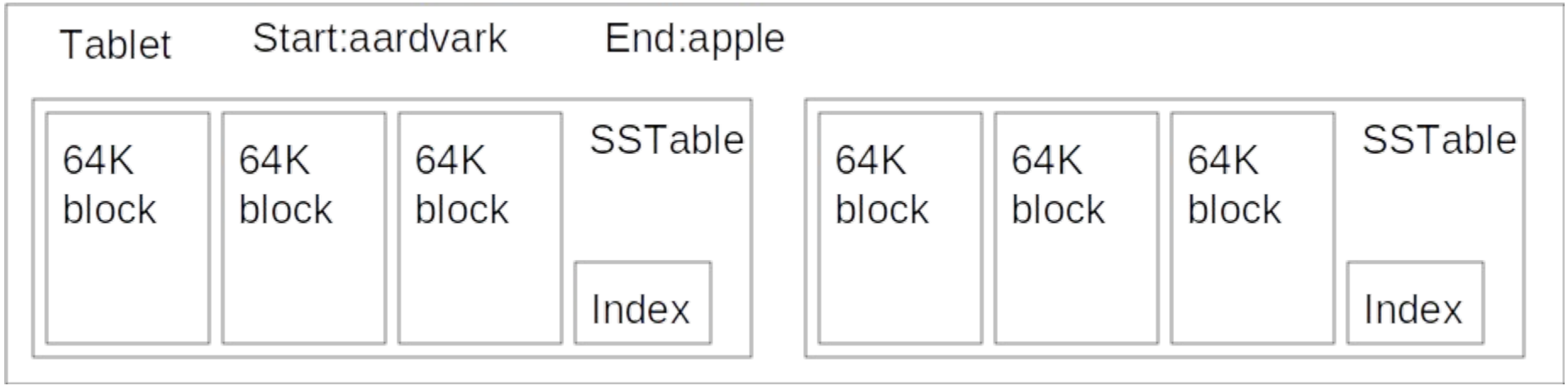
Tablets & Splitting




System Structure
Bigtable Cell
Bigtable client
Bigtable client library
metadata ops
Bigtable master
performs metadata ops + load balancing
read/write
Open()
Bigtable tablet server
serves data
Bigtable tablet server
serves data
...
Bigtable tablet server
serves data
Cluster scheduling system
handles failover, monitoring
GFS
holds tablet data, logs
Lock service
holds metadata, handles master-election
16
Google

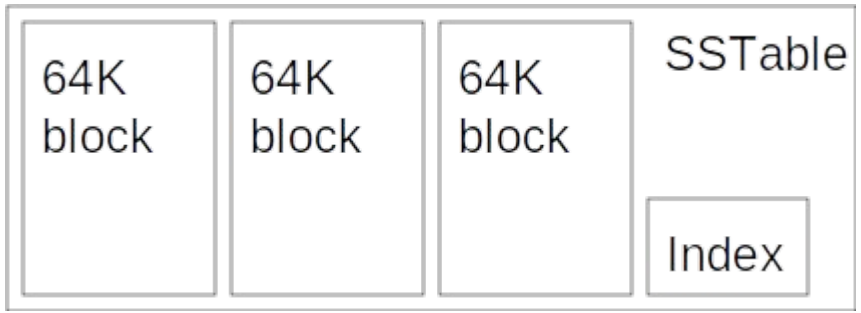
Tablet

- Contains some range of rows of the table
- Built out of multiple SSTables



SSTable

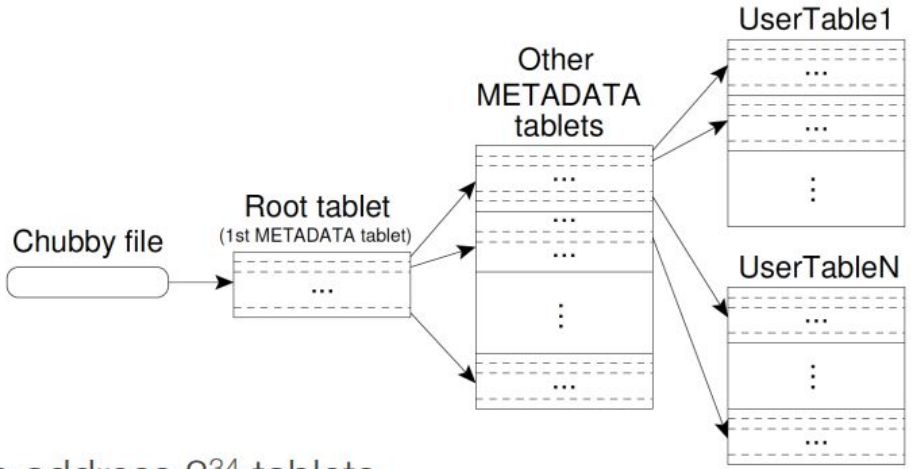
- Immutable, sorted file of key-value pairs
- Chunks of data plus an index



Locating Tablets

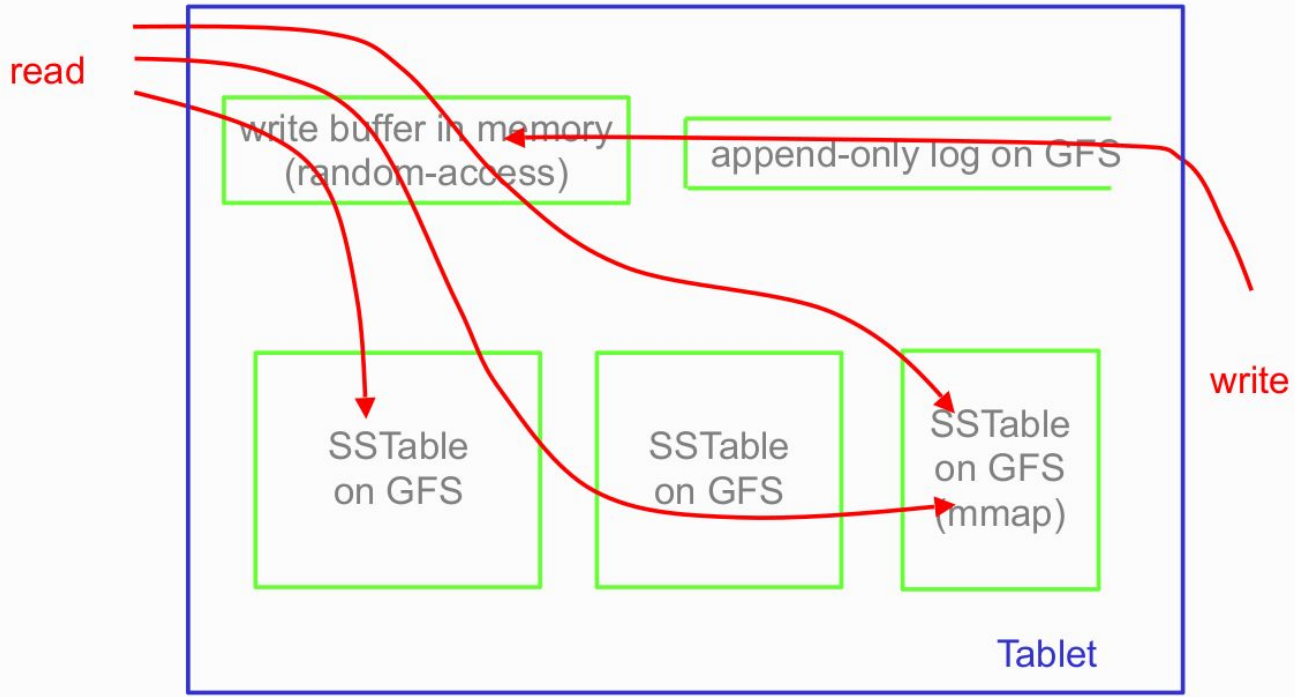
- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row
- One approach: could use the BigTable master
  - Central server almost certainly would be bottleneck in large system
- Instead: store special tables containing tablet location info in BigTable cell itself

Tablet Location Hierarchy



Able to address 2^{34} tablets

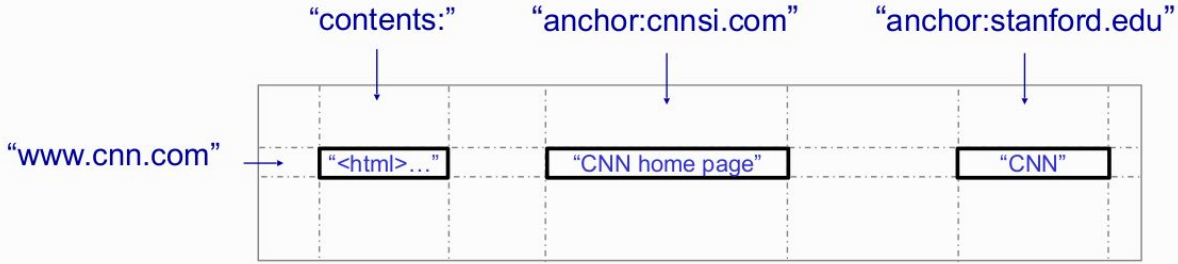
Tablet Representation



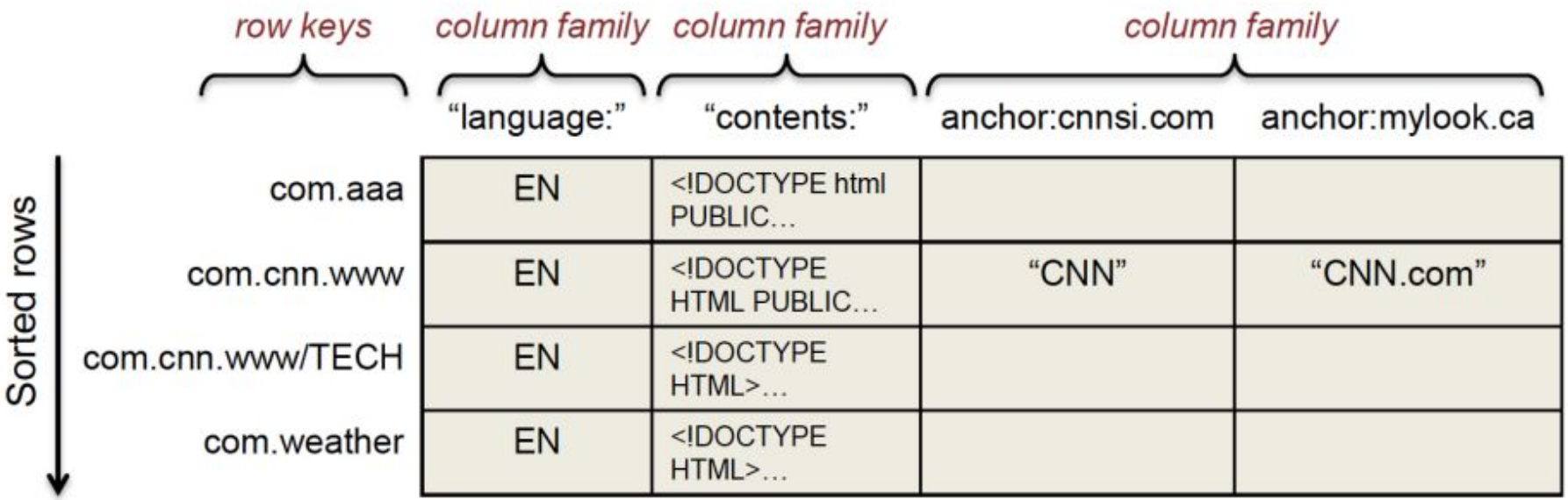
Compactions

- Tablet state represented as set of immutable compacted SSTable files, plus tail of log (buffered in memory)
- Minor compaction:
  - When in-memory state fills up, pick tablet with most data and write contents to SSTables stored in GFS
    - Separate file for each locality group for each tablet
- Major compaction:
  - Periodically compact all SSTables for tablet into new base SSTable on GFS
    - Storage reclaimed from deletions at this point

Columns



- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional level of indexing, if desired


row keys
column family
column family
column family
"language:"
"contents:"
anchor:cnnsi.com
anchor:mylook.ca
Sorted rows
com.aaa
EN
<!DOCTYPE html PUBLIC…
com.cnn.www
EN
<!DOCTYPE HTML PUBLIC…
"CNN"
"CNN.com"
com.cnn.www/TECH
EN
<!DOCTYPE HTML>…
com.weather
EN
<!DOCTYPE HTML>…

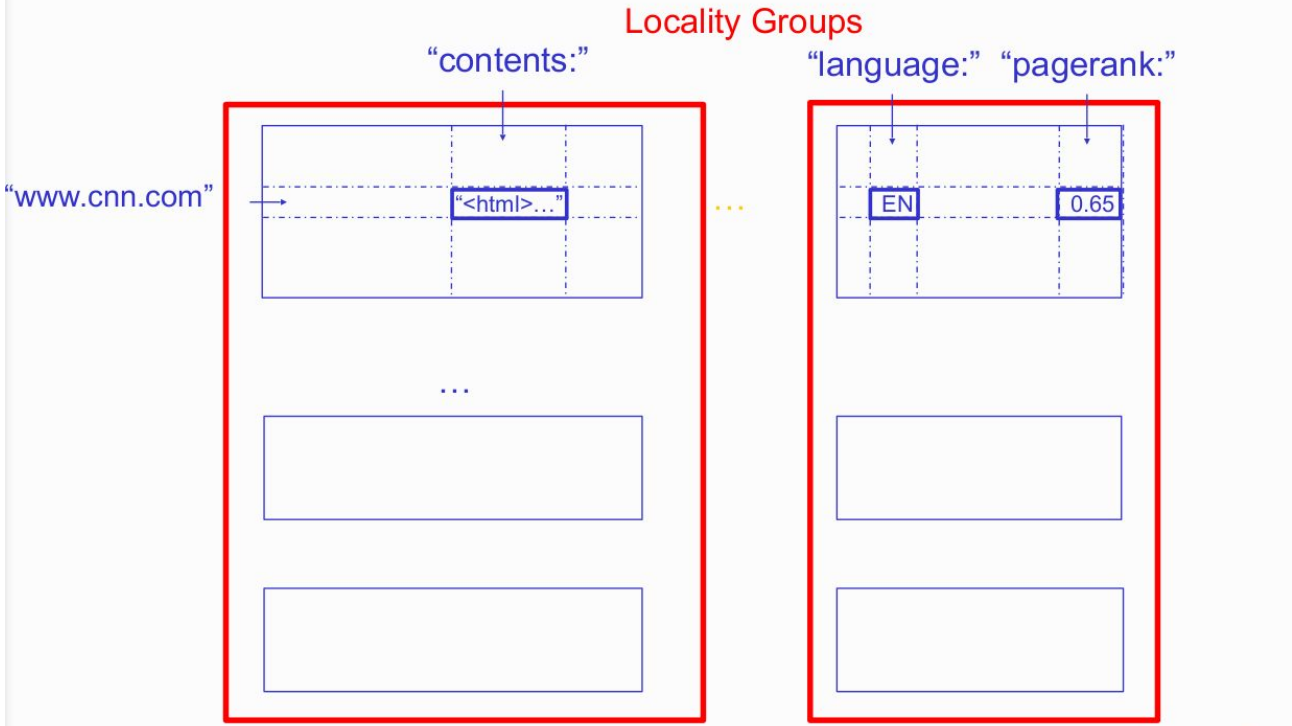
Timestamps

- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column familes can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

Locality Groups

- Column families can be assigned to a locality group
  - Used to organize underlying storage representation for performance
    - scans over one locality group are *O(bytes_in_locality_group)* , not *O(bytes_in_table)*
  - Data in a locality group can be explicitly memory-mapped

Locality Groups

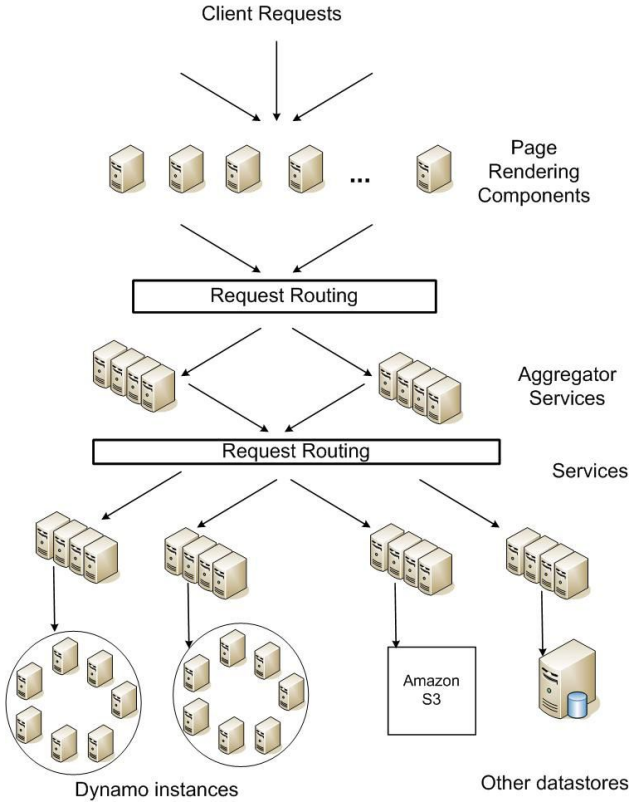


Dynamo: Amazon’s Highly Available Key-value Store

System assumptions and requirements

- Query model
- ACID properties
- Efficiency
- Other assumptions

Service Level Agreements (SLA)



Design Consideration

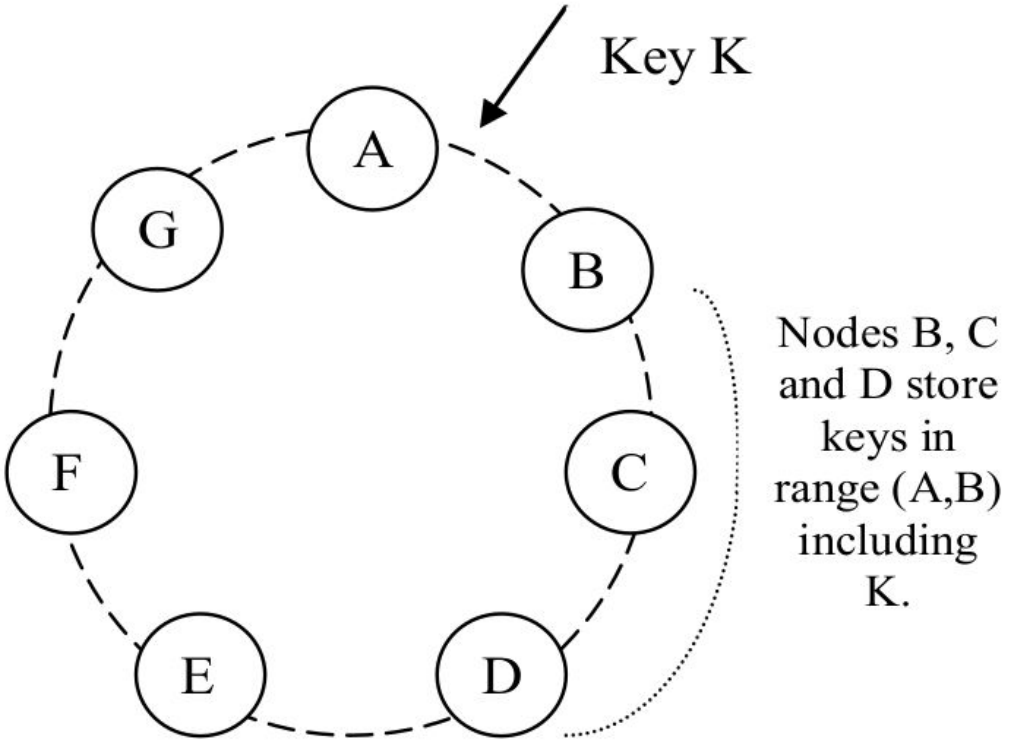
- Sacrifice strong consistency for availability
- Conflict resolution is executed during read instead of write
- Incremental scalability
- Symmetry
- Decentralization
- Heterogeneity

System Architecture

- Partitioning
- High availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

Partitioning Algorithm

- Consistent hashing
- Virtual Nodes

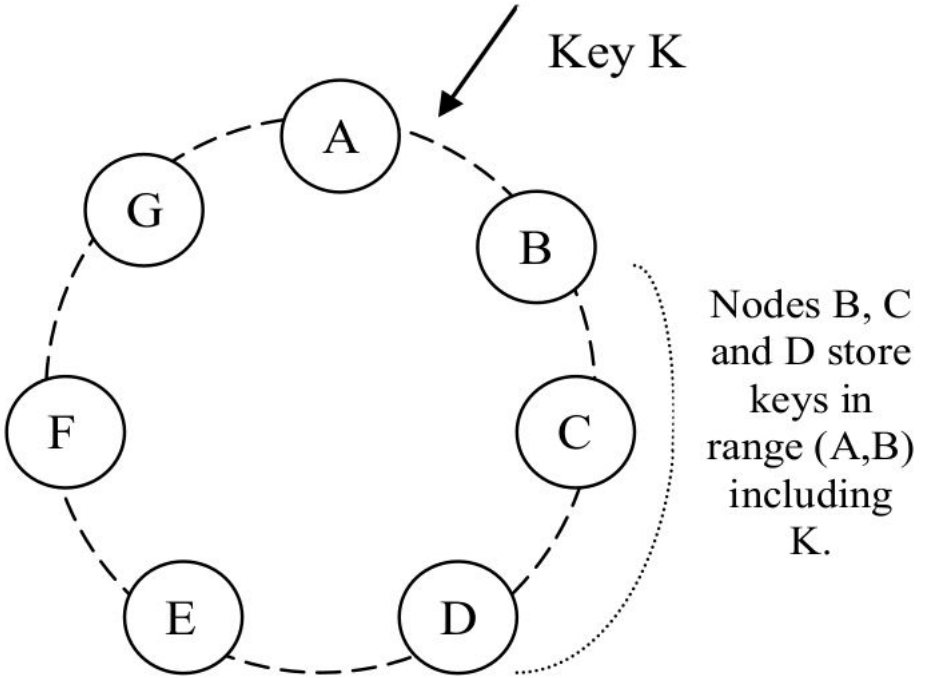


Virtual Nodes

- Node becomes unavailable
- Added new node
- Heterogeneity

Replication

- Each data is replicated at N hosts.
- Preference list



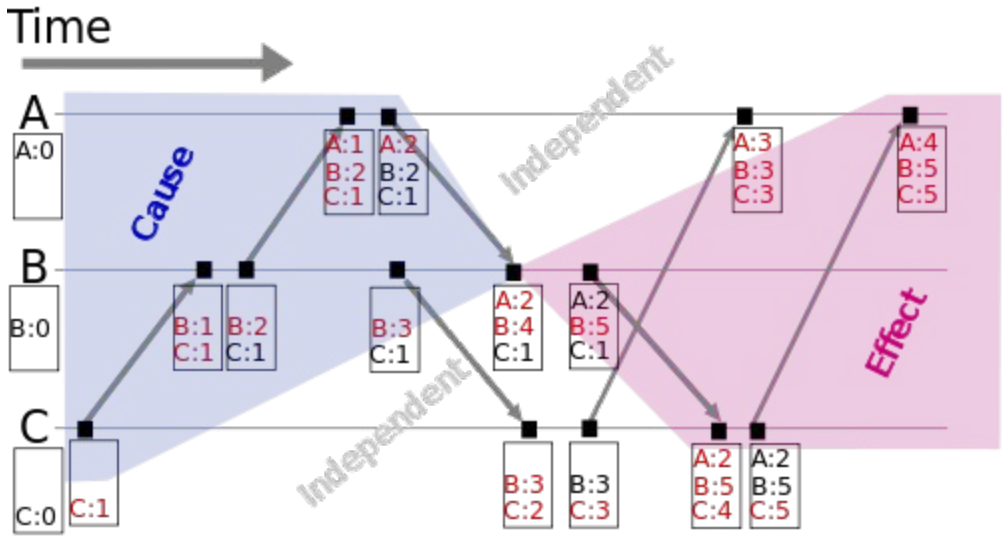
Data Versioning

- Put()
- get()

Data Versioning

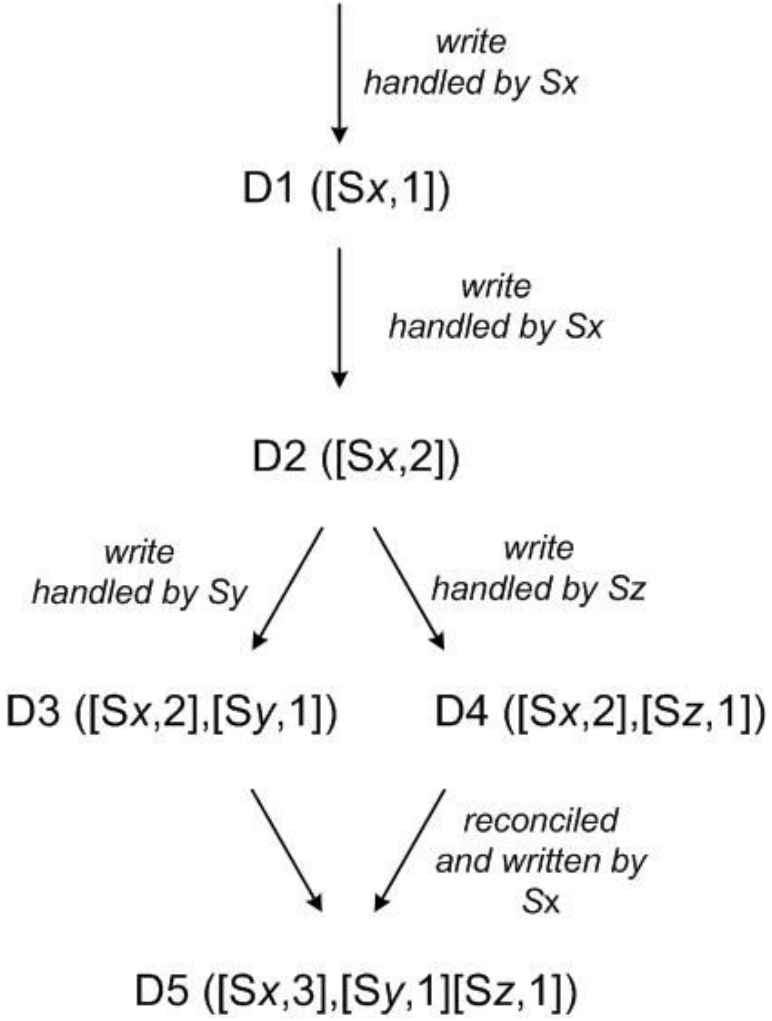
- Key Challenge: distinct version sub-histories - need to be reconciled.
- Solution: uses vector clocks in order to capture causality between different versions of the same object.

Vector Clock



Vector Clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.

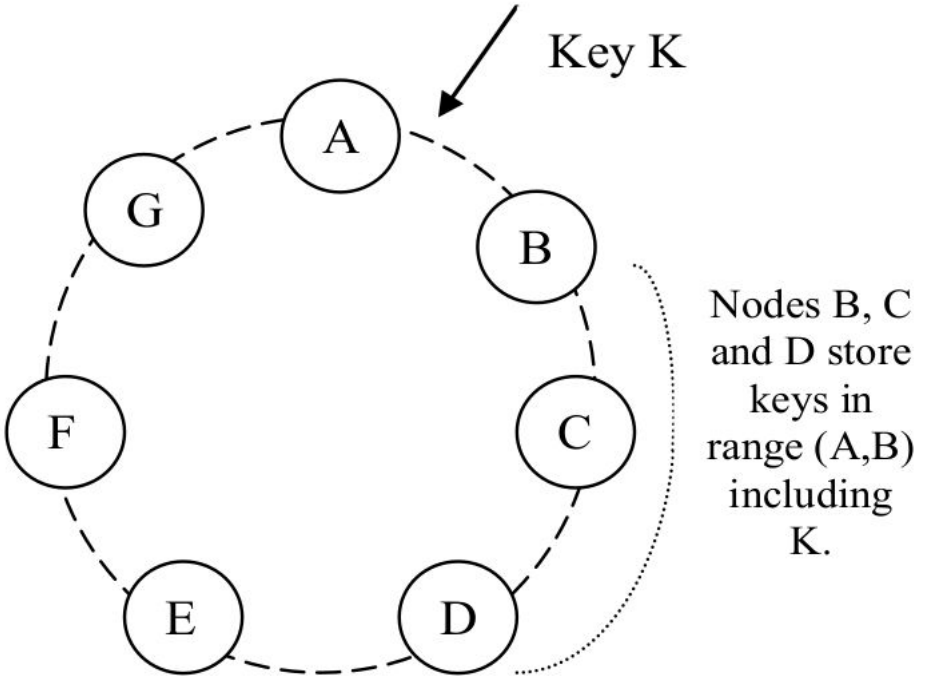

write
handled by Sx
D1 ([Sx,1])
write
handled by Sx
D2 ([Sx,2])
write
handled by Sy
write
handled by Sz
D3 ([Sx,2],[Sy,1])
D4 ([Sx,2],[Sz,1])
reconciled
and written by
Sx
D5 ([Sx,3],[Sy,1][Sz,1])

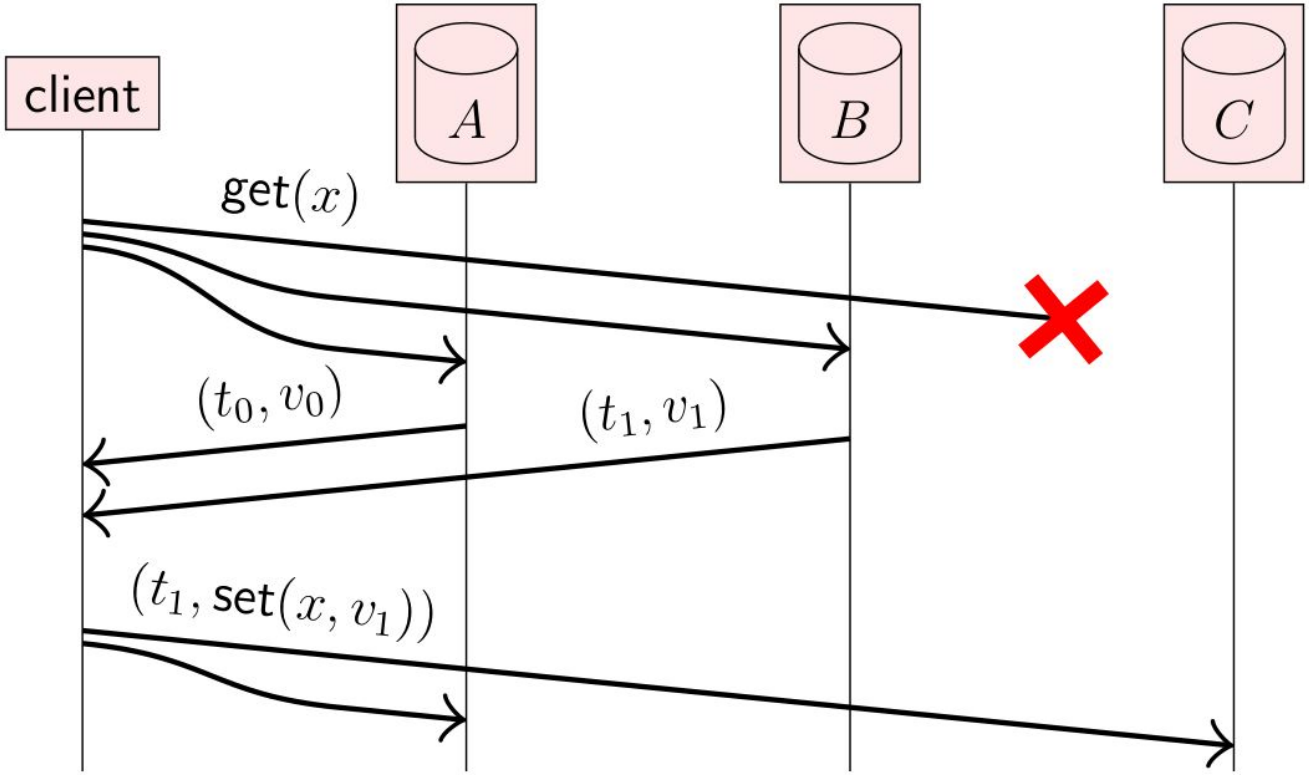
Execution of get () and put () Operations

- Generic load balancer
- Use a partition-aware client library

Hinted Handoff

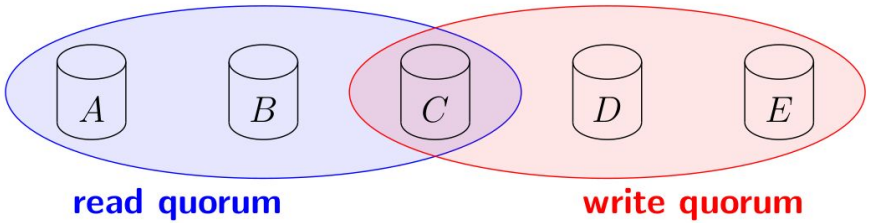
- Assume N = 3. When A is Temporarily down or unreachable during a write, send replica to D.
- D is hinted that the replica belongs to A and it will deliver to A when A is recovered.




client
A
B
C
get(x)
(t0, v0)
(t1, v1)
(t1, set(x, v1))

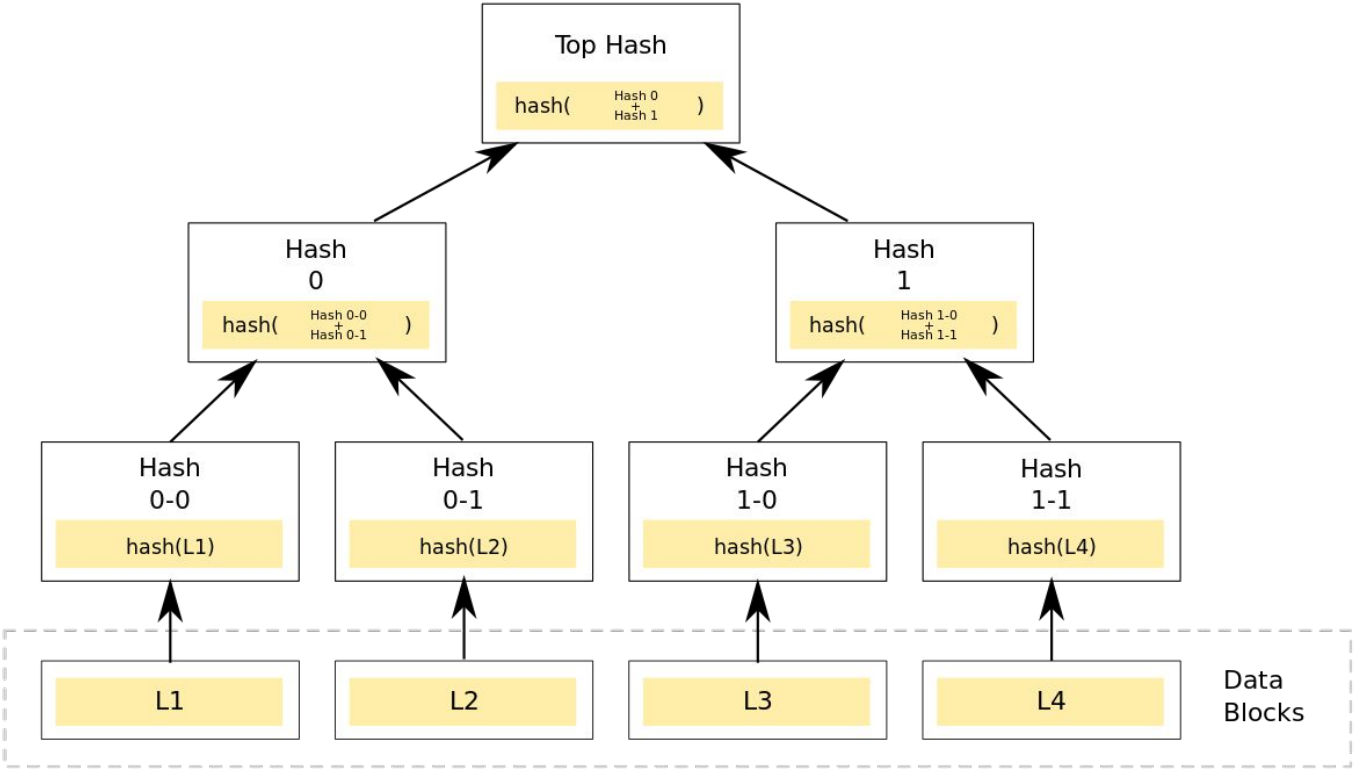
Quorum

- R/W is the minimum number of nodes that must participate in a successful read/write operation.
- Setting R + W > N yields a quorum-like system.
- In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.



Handling Permanent Failures: Replica Synchronization

Merkle Tree


Top Hash
hash(Hash 0 + Hash 1)
Hash 0
hash(Hash 0-0 + Hash 0-1)
Hash 1
hash(Hash 1-0 + Hash 1-1)
Hash 0-0
hash(L1)
Hash 0-1
hash(L2)
Hash 1-0
hash(L3)
Hash 1-1
hash(L4)
L1
L2
L3
L4
Data Blocks

Merkle Tree

- A hash tree where leaves are hashes of the values of individual keys.
- Parent nodes higher in the tree are hashes of their respective children.

Merkle Tree

- Each branch of the tree can be checked independently without requiring nodes to download the entire tree.
- Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas.

Membership and Failure Detection

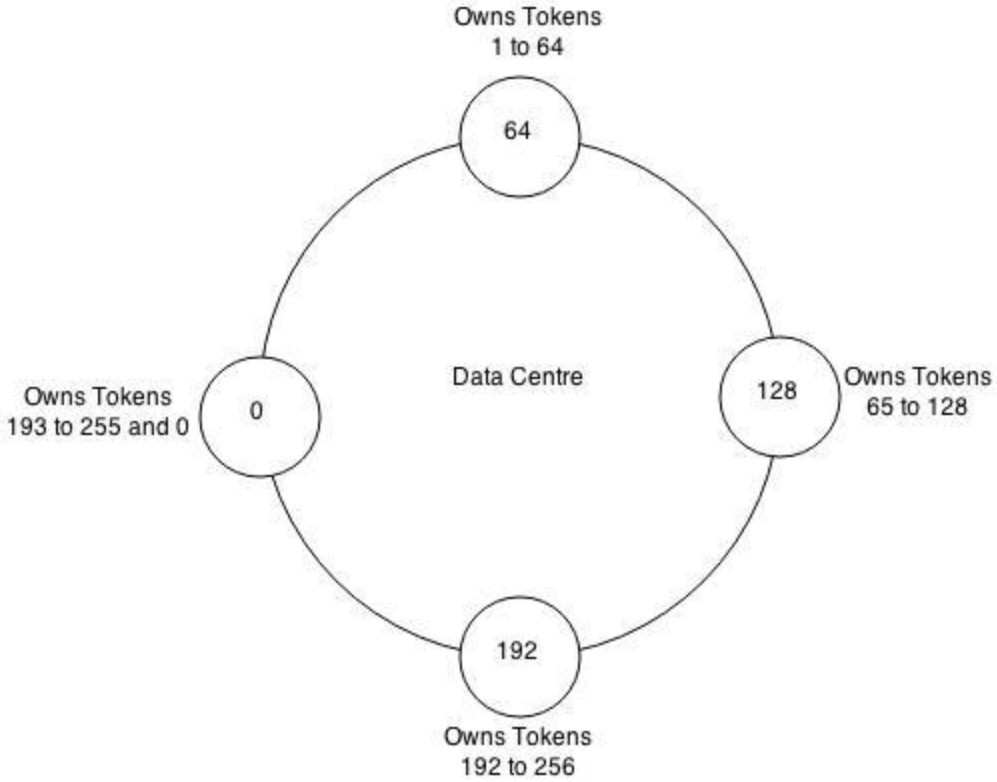
- A gossip-based protocol

https://flopezluis.github.io/gossip-simulator/

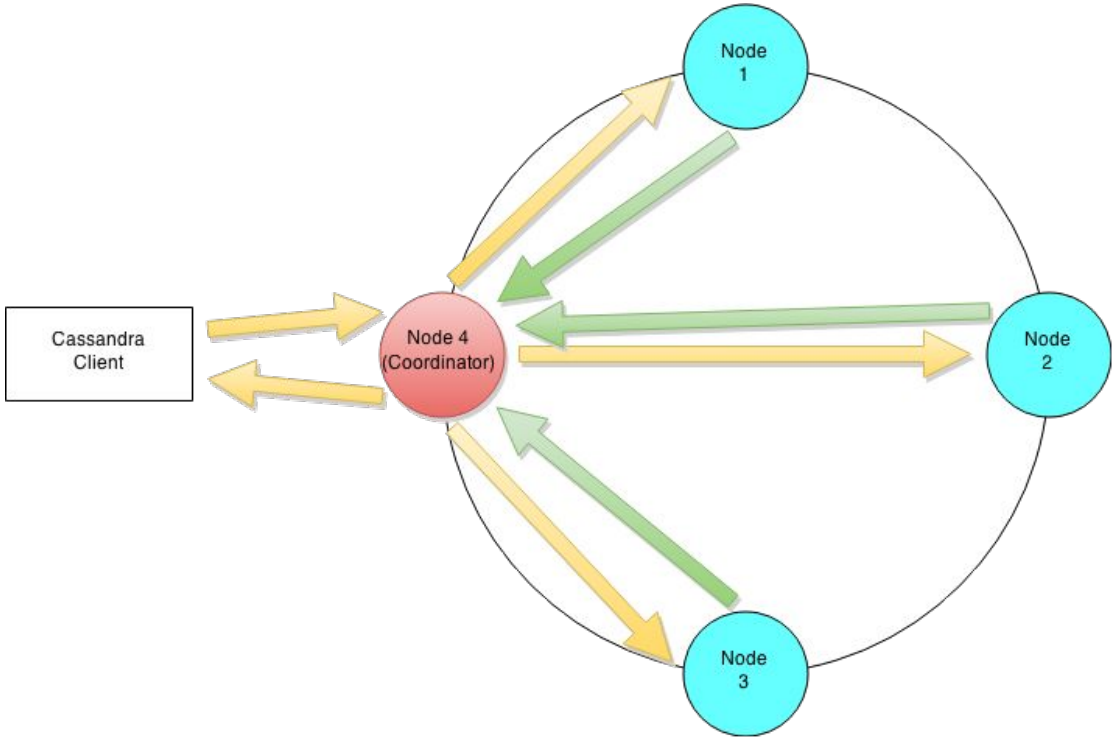
Gossip Applications

- Database Replication
- Cluster Membership
- Failure Detection
- Information Dissemination

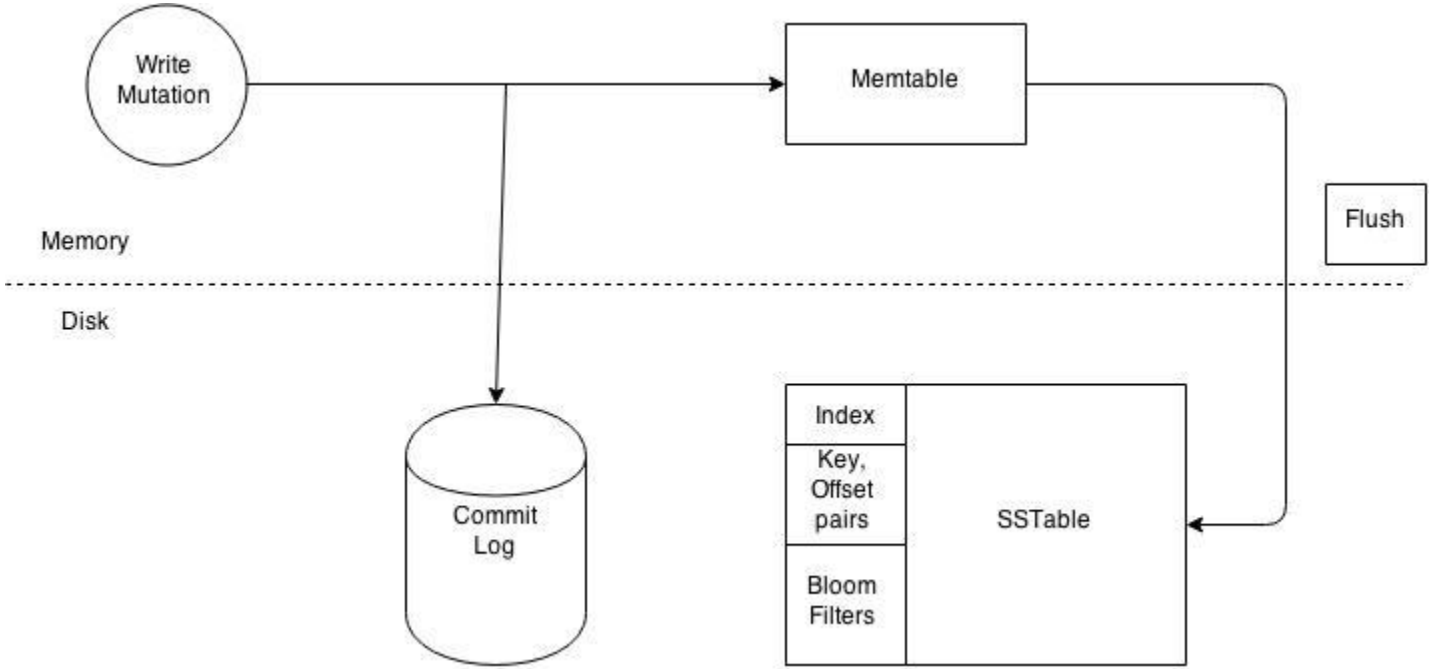
Cassandra


Owns Tokens
1 to 64
64
Owns Tokens
193 to 255 and 0
0
Data Centre
128
Owns Tokens
65 to 128
192
Owns Tokens
192 to 256

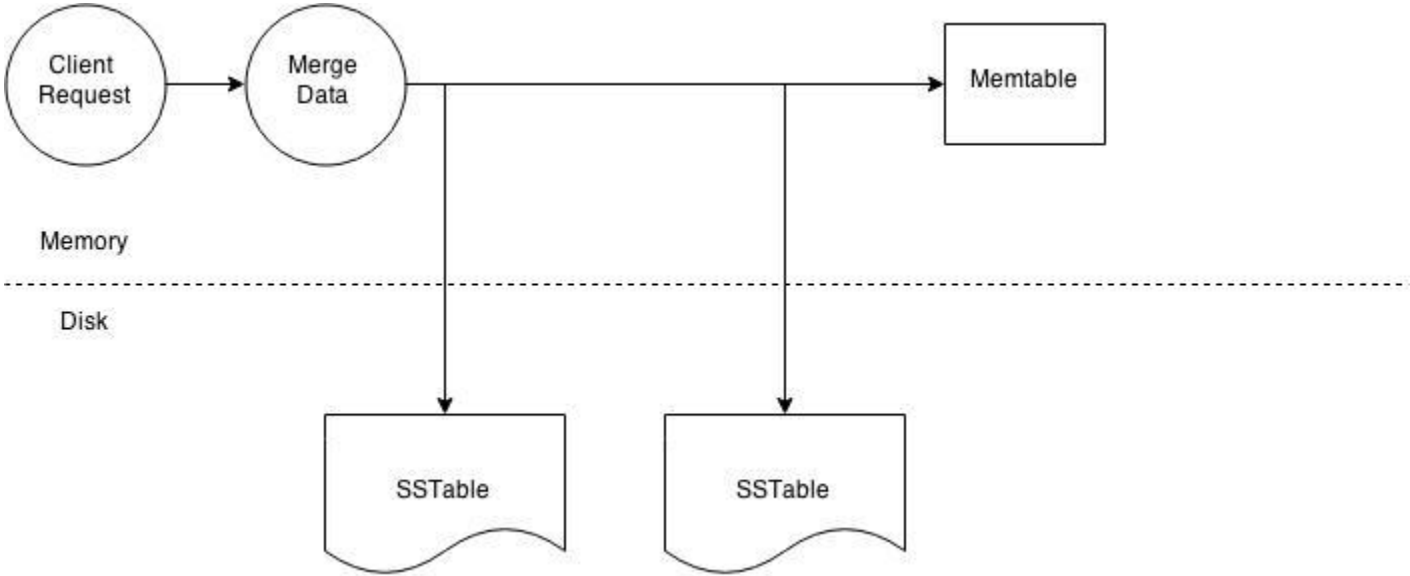
Write Path: Cluster Level



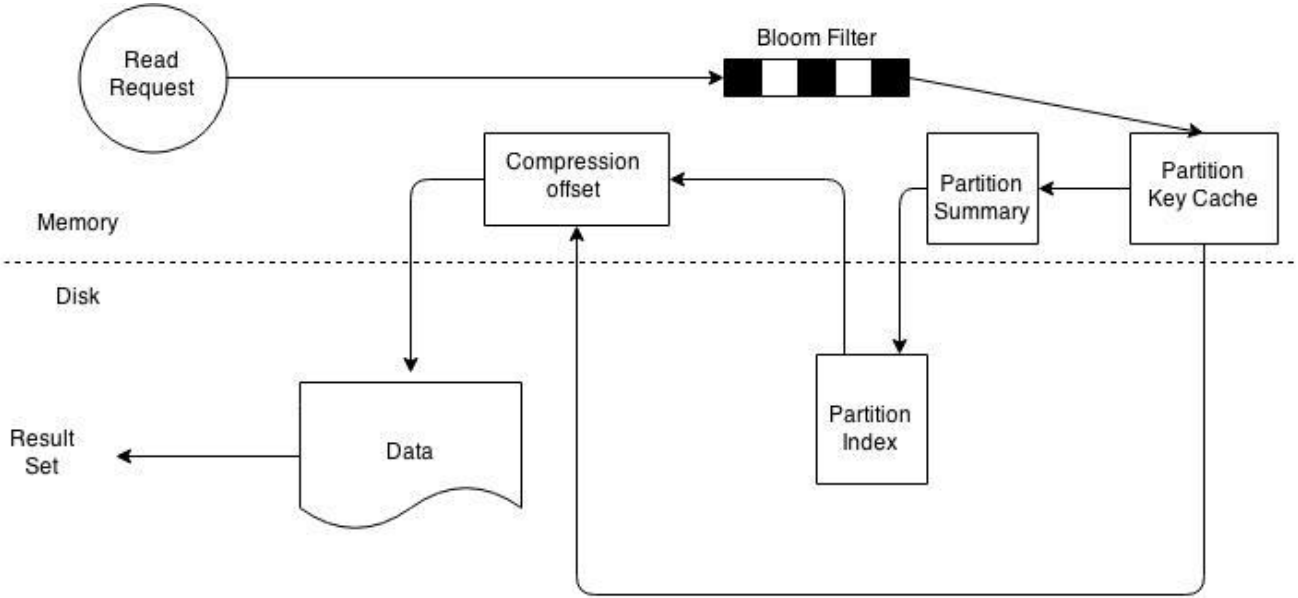
Write Path: Node Level



Read Path: Node Level



Read Path: SSTable



Consistency

A word that means many different things in different contexts!

- ACID: a transaction transforms the database from one “consistent” state to another
- Read-after-write consistency
- Replication: replica should be “consistent” with other replicas
  - “consistent” = in the same state? (when exactly?)
  - “consistent” = read operations return same result?
- Consistency model: many to choose from

Distributed Transactions

Recall atomicity in the context of ACID transactions:

- A transaction either commits or aborts
- If it commits, its updates are durable
- If it aborts, it has no visible side-effects
- ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies:

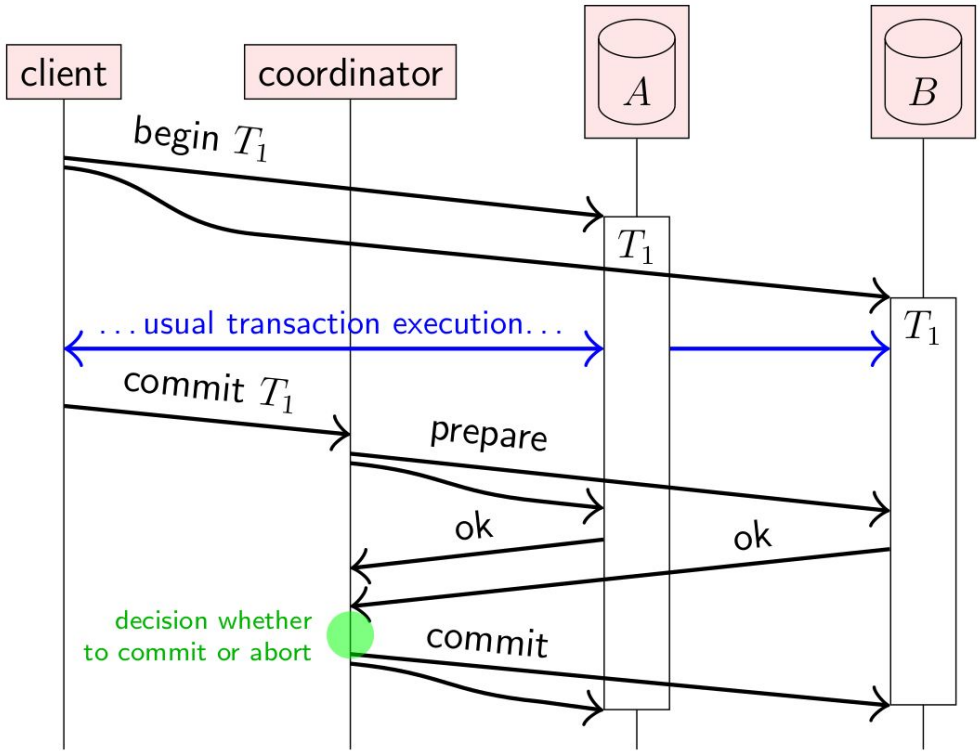
- Either all nodes must commit, or all must abort
- If any node crashes, all must abort

Ensuring this is the **atomic commitment** problem.

Atomic commit versus consensus

Consensus	**Atomic commit**
One or more nodes propose a value	Every node votes whether to commit or abort
Any one of the proposed values is decided	Must commit if all nodes vote to commit; must abort if ≥ 1 nodes vote to abort
Crashed nodes can be tolerated, as long as a quorum is working	Must abort if a participating node crashes

Two-phase commit (2PC)



The coordinator in two-phase commit

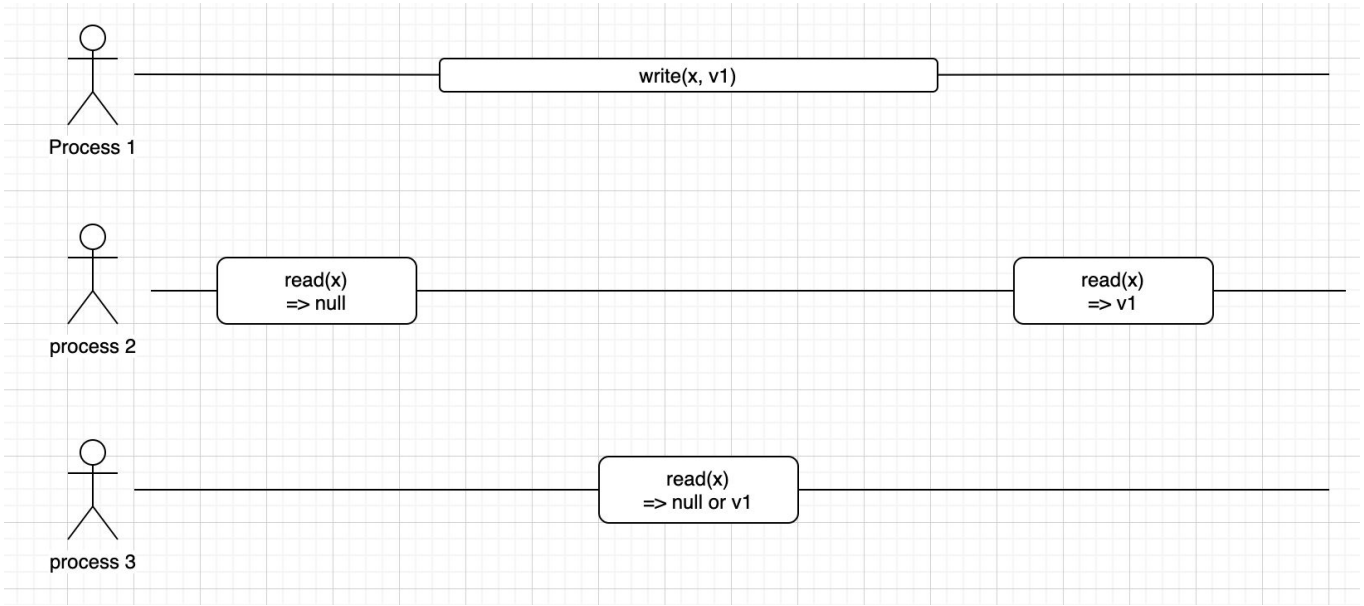
What if the coordinator crashes?

- Coordinator writes its decision to disk
- When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided
- Replicas participating in transaction cannot commit or abort after responding "ok" to the *prepare* request (otherwise we risk violating atomicity)
- Algorithm is blocked until coordinator recovers

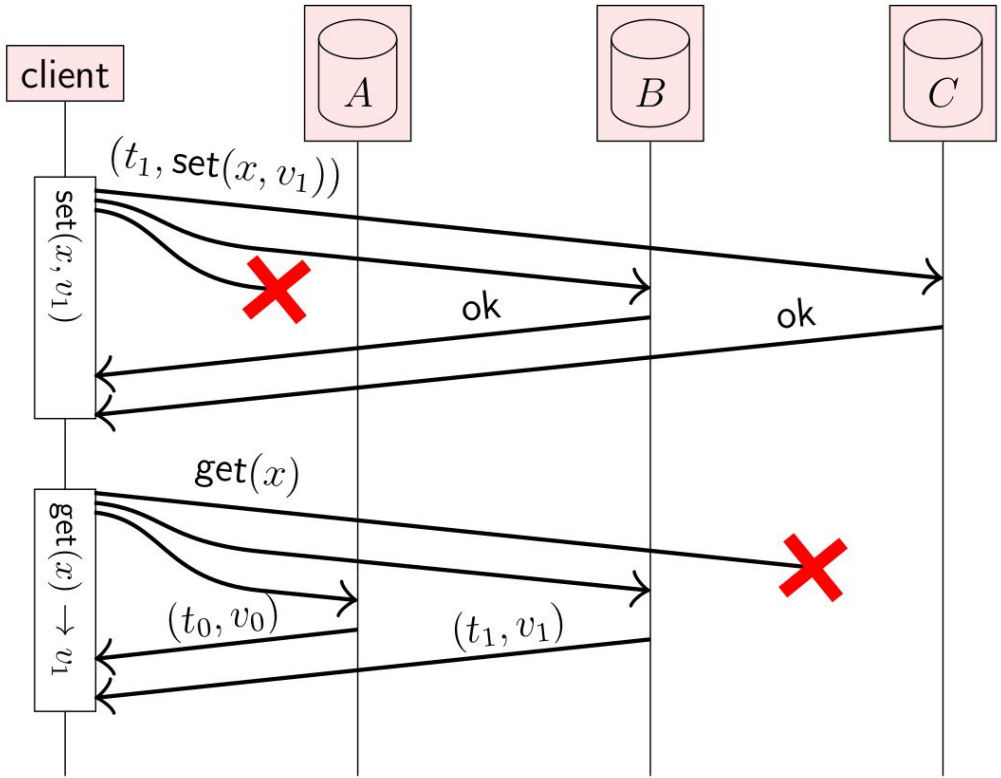
Linearizability

Linearizability is the strongest form of consistency model in the distributed system.

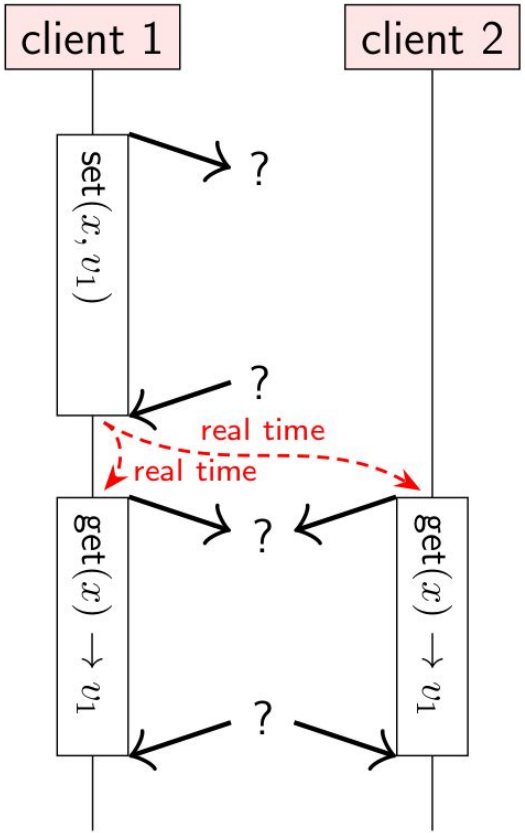
- Informally: every operation takes effect atomically sometime after it started and before it finished.
- All operations behave as if executed on a single copy of the data (even if there are in fact multiple replicas)


write(x, v1)
Process 1
read(x)
=> null
read(x)
=> v1
process 2
read(x)
=> null or v1
process 3

Read-after-write consistency

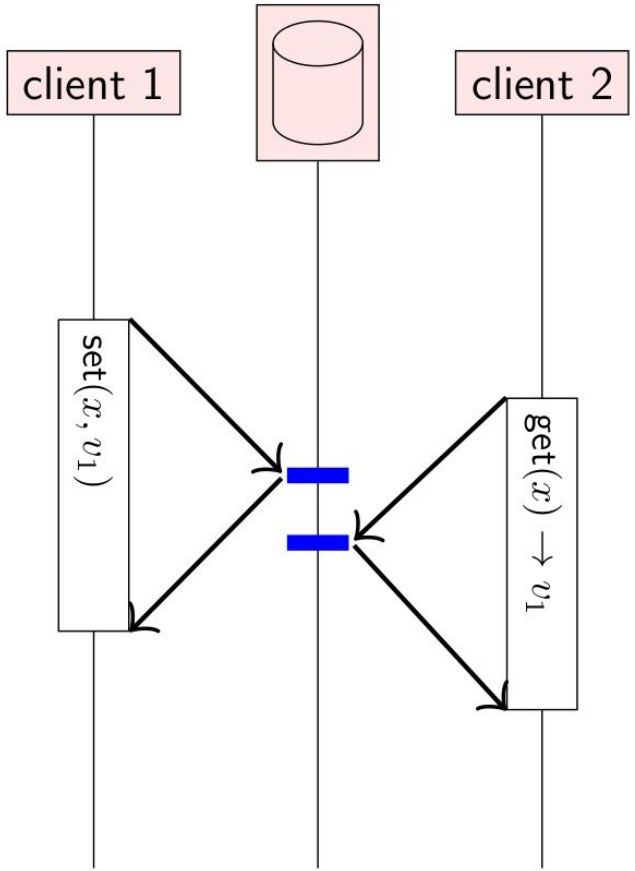


From the client's point of view



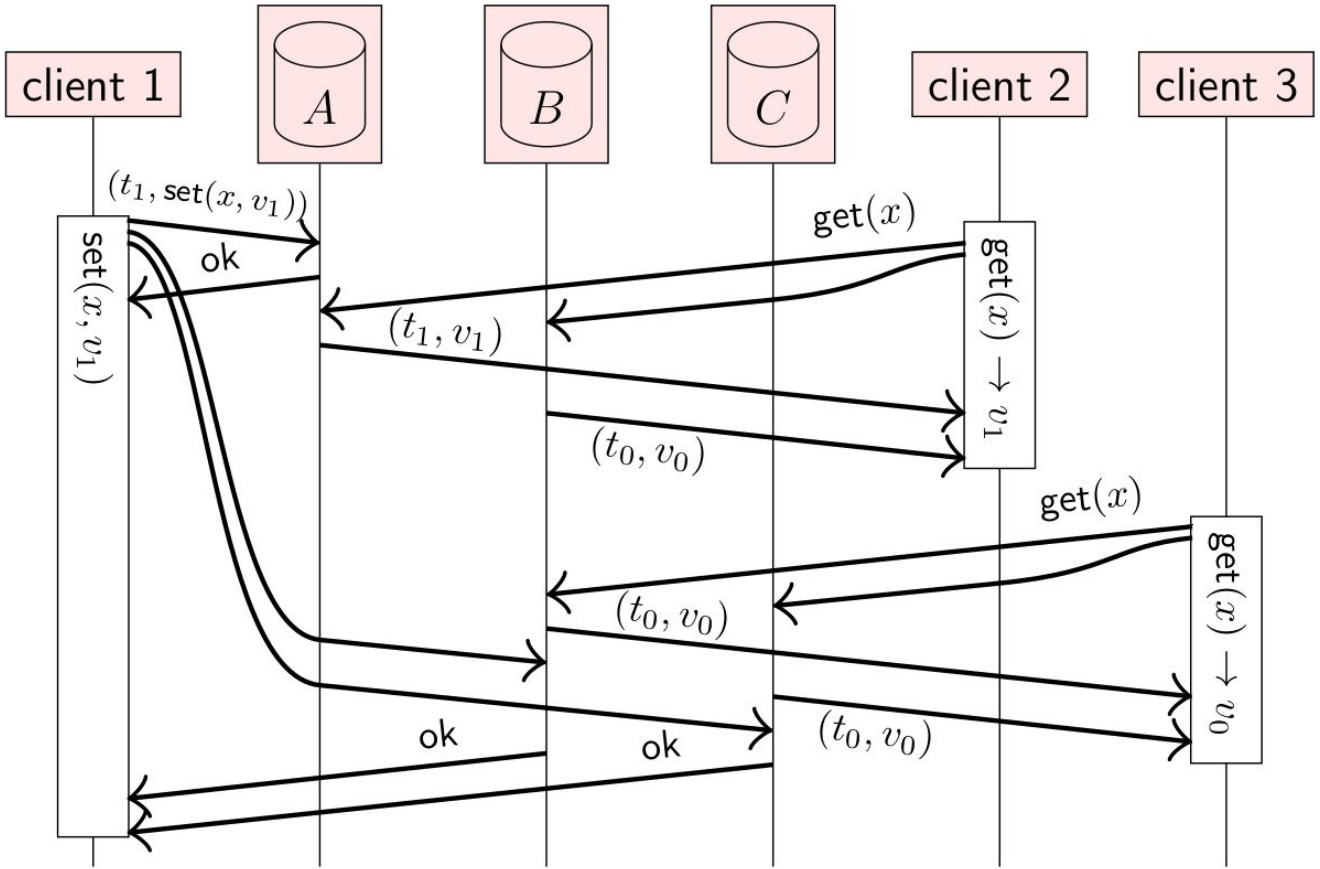
- Focus on client-observable behaviour: when and what an operation returns
- Ignore how the replication system is implemented internally
- Did operation A finish before operation B started?
- Even if the operations are on different nodes?
- **This is not happens-before**: we want client 2 to read value written by client 1, even if the clients have not communicated!

Operations overlapping in time

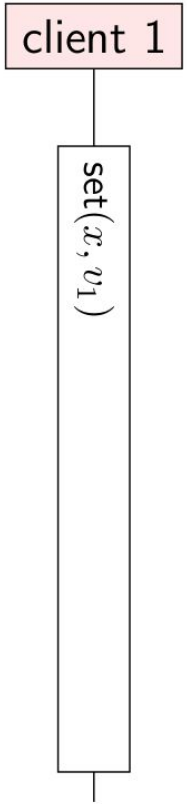


- Client 2's get operation overlaps in time with client 1's set operation
- Maybe the set operation takes effect first?
- Just as likely, the get operation may be executed first
- Either outcome is fine in this case

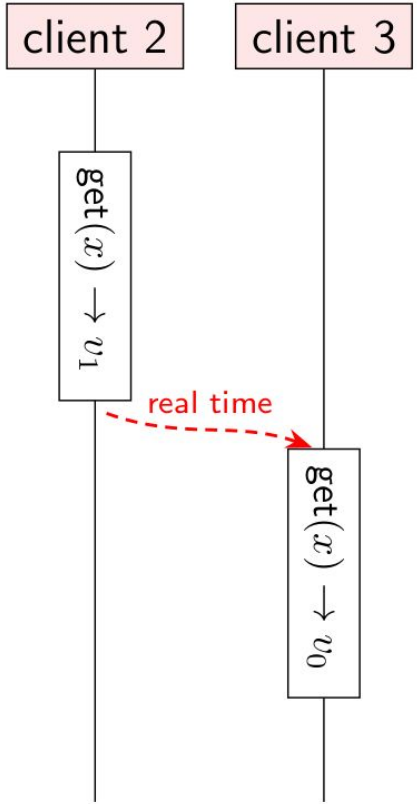
Not linearizable, despite quorum reads/writes



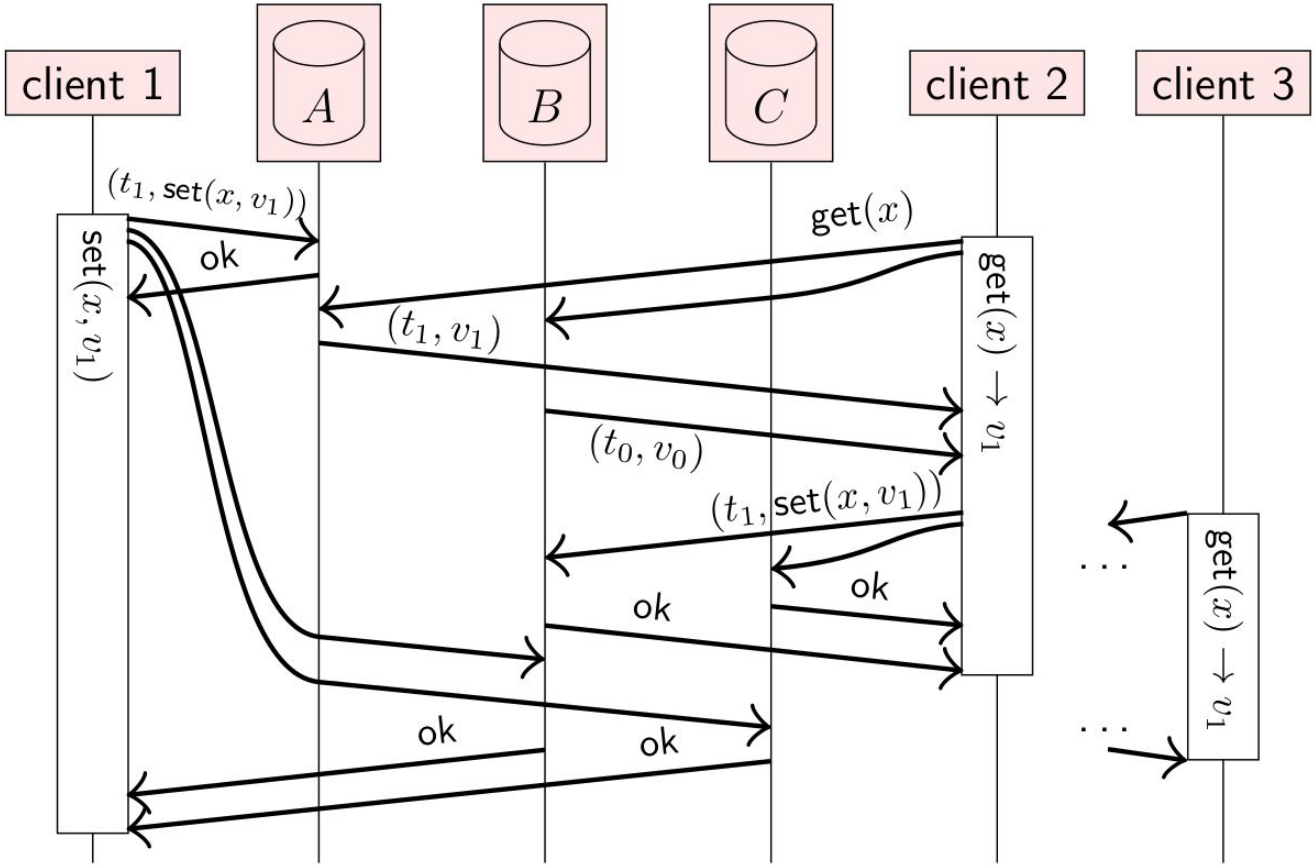
Not linearizable, despite quorum reads/writes



- Client 2's operation finishes before client 3's operation starts
- Linearizability therefore requires client 3's operation to observe a state no older than client 2's operation
- This example violates linearizability because v_0 is older than v_1



ABD: Making quorum reads/writes linearizable



Eventual consistency

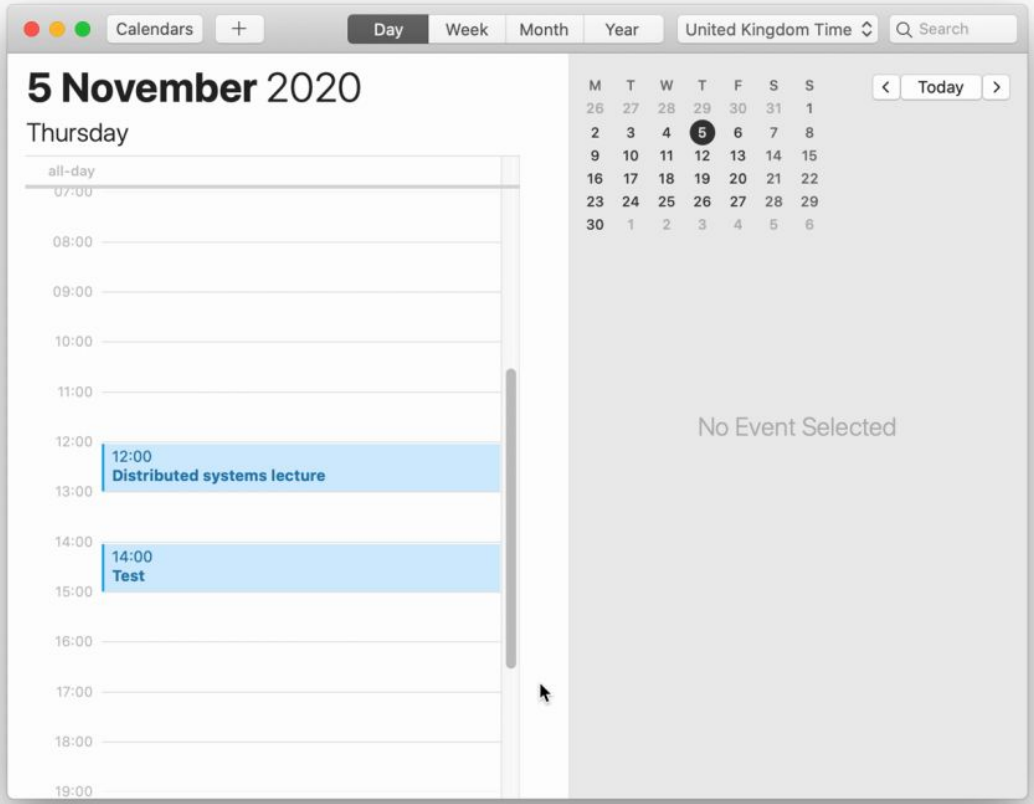
Linearizability advantages:

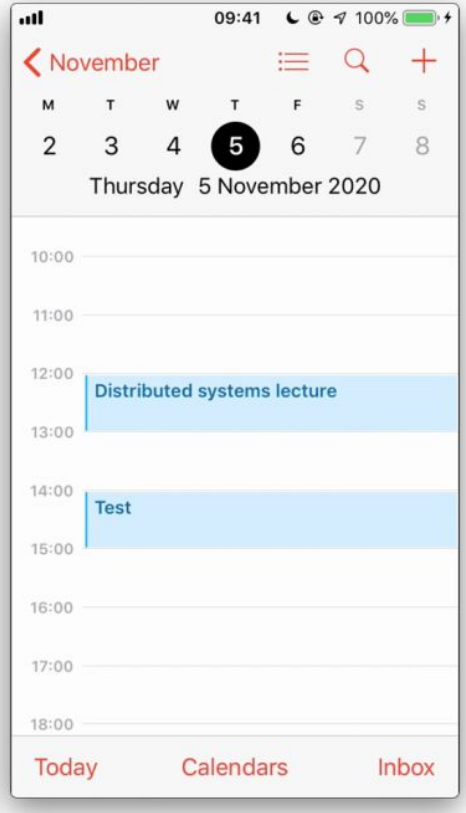
- Makes a distributed system behave as if it were non-distributed
- Simple for applications to use

Downsides:

- **Performance** cost: lots of messages and waiting for responses
- **Scalability** limits: leader can be a bottleneck
- **Availability** problems: if you can't contact a quorum of nodes, you can't process any operations

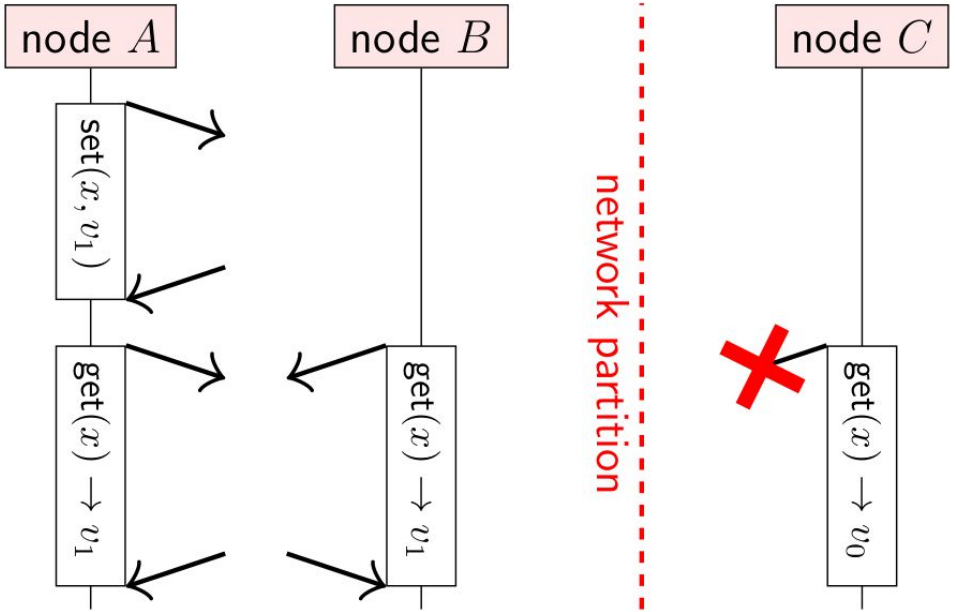
Eventual consistency: a weaker model than linearizability. Different trade-off choices.


Calendars
Day
Week
Month
Year
United Kingdom Time
Search
5 November 2020
Thursday
Today
all-day
12:00
Distributed systems lecture
14:00
Test
No Event Selected


09:41
100%
November
Thursday 5 November 2020
12:00
Distributed systems lecture
14:00
Test
Today
Calendars
Inbox

The CAP theorem

A system can be either strongly **Consistent** (linearizable) or **Available** in the presence of a network **Partition**



C must either wait indefinitely for the network to recover, or return a potentially stale value

Eventual consistency

Replicas process operations based only on their local state.

If there are no more updates, eventually all replicas will be in the same state. (No guarantees how long it might take.)

Strong eventual consistency

- Eventual delivery: every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- Convergence: any two replicas that have processed the same set of updates are in the same state (even if updates were processed in a different order).

References

- Introduction to NoSQL-Martin fowler
- Big Data for Engineers 2018-ETH course
- Bigtable: A Distributed Storage System for Structured Data-Fay Chang, ...
- Dynamo: Amazon’s Highly Available Key-value Store-Giuseppe DeCandia, …
- Google Bigtable-Adapted by S. Sudarshan from a talk by Erik Paulson, UW Madison
- Google’s BigTable-Jeffrey Kendall
- Dynamo: Amazon’s Highly Available Key-value Store
- Cassandra - A Decentralized Structured Storage System-Avinash Lakshman and Prashant Malik

References

- Introduction to Apache Cassandra’s Architecture
- http://www.scs.stanford.edu/20sp-cs244b/
- https://www.cst.cam.ac.uk/teaching/2122/ConcDisSys
- https://www.read.seas.harvard.edu/~kohler/class/cs261-f11/bigtable.html