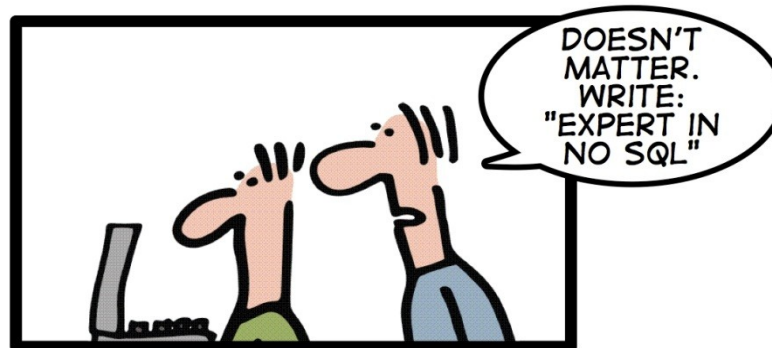
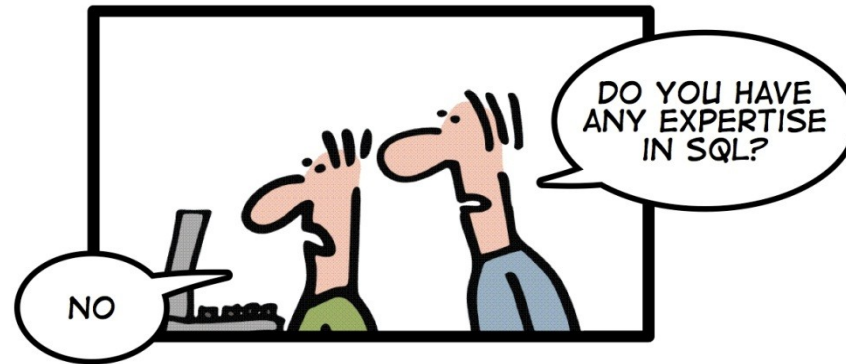


HOW TO WRITE A CV



Leverage the NoSQL boom



Not Only SQL (NoSQL) Databases

Amir H. Payberah
amir@sics.se

SQL is Good

- Relational Databases Management Systems (**RDBMSs**) – mainstay of business



- **SQL** is good
 - Rich language
 - Easy to use and integrate
 - Rich toolset
 - Many vendors



- They promise: **ACID**

ACID Properties



- **Atomicity**: all included statements in a transaction are either executed or the whole transaction is aborted without affecting the database.
- **Consistency**: a database is in a consistent state before and after a transaction.
- **Isolation**: transactions can not see uncommitted changes in the database.
- **Durability**: changes are written to a disk before a database commits a transaction so that committed data cannot be lost through a power failure.

SQL is Good

- SQL is good, ...





BUT...

SQL Challenges

- **Web-based applications** caused spikes.
 - Internet-scale data size
 - High read-write rates
 - Frequent schema changes
 - Large data



The Past and the Moment

	 Circa 1975 “Online Applications”	 Circa 2011 “Interactive Web Applications”
Users	2,000 “online” users = End Point	2,000 “online” users = Starting Point
	Static user population	Dynamic user population
Applications	Business process automation	Business process innovation
	Highly structured data records	Structured, semi-structured and unstructured data
Infrastructure	Data networking in its infancy	Universal high-speed data networking
	Centralized computing (Mainframes and minicomputers)	Distributed computing (Network servers and virtual machines)
	Memory scarce and expensive	Memory plentiful and cheap

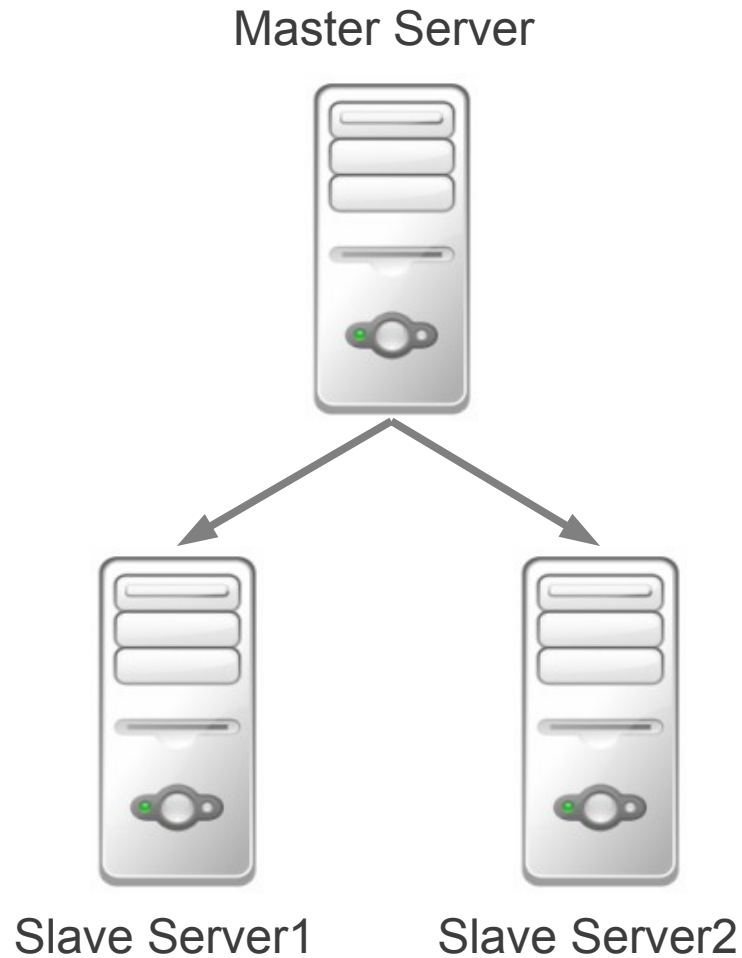
<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf>

Let's Scale RDBMSs

- RDBMS were **not** designed to be **distributed**.
- Possible solutions:
 - Replication
 - Sharding

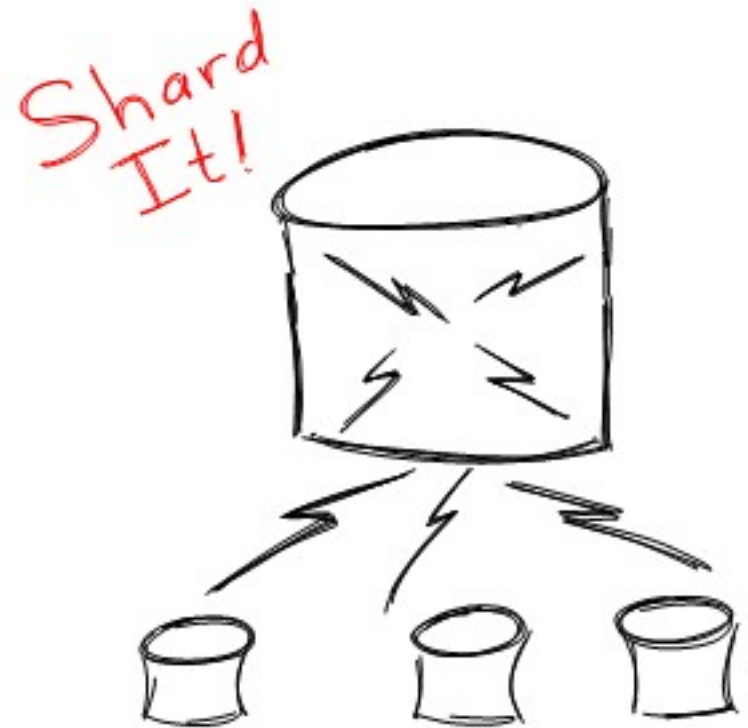
Let's Scale RDBMSs - Replication

- Master/Slave architecture
- It scales **read operations**

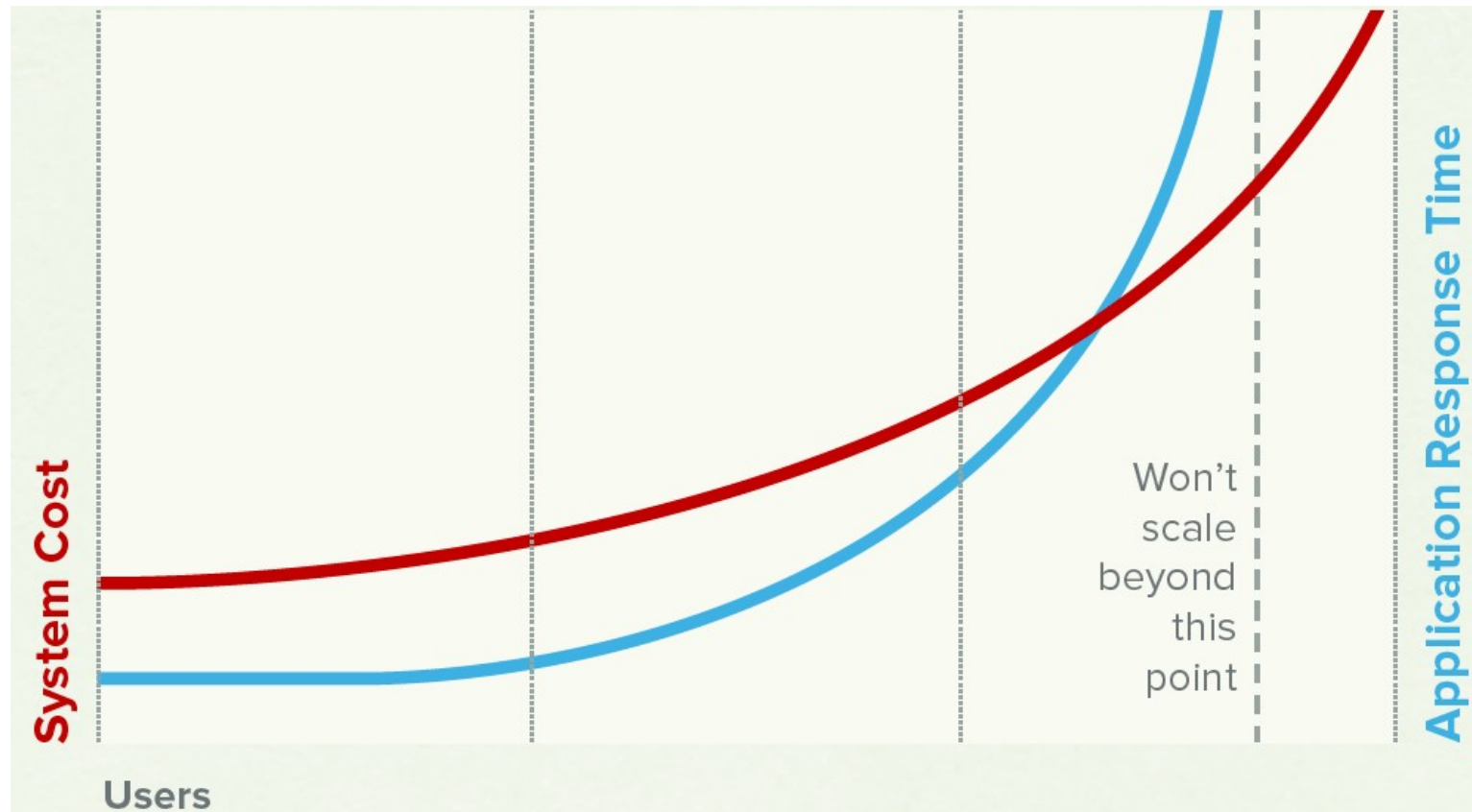


Let's Scale RDBMSs - Sharding

- Scaling out (**horizontal scaling**) based on **data partitioning**, i.e. dividing the database across many (inexpensive) machines.
- This is how youtube, facebook, yahoo all started. With sharded mysql.
- It **scales read and write** operations, but you can't execute transactions across shards (partitions).



Scaling RDBMSs is **Expensive and Inefficient**



[<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf>]

Not Only SQL

Not **S**QL
Only **Q**uery **L**anguage

What is NoSQL?

- Class of **non-relational** data storage systems.
- All NoSQL offerings **relax** one or more of the **ACID properties**.
 - **Social applications** are **not banks** and they **don't need** the same level of ACID.

NoSQL History

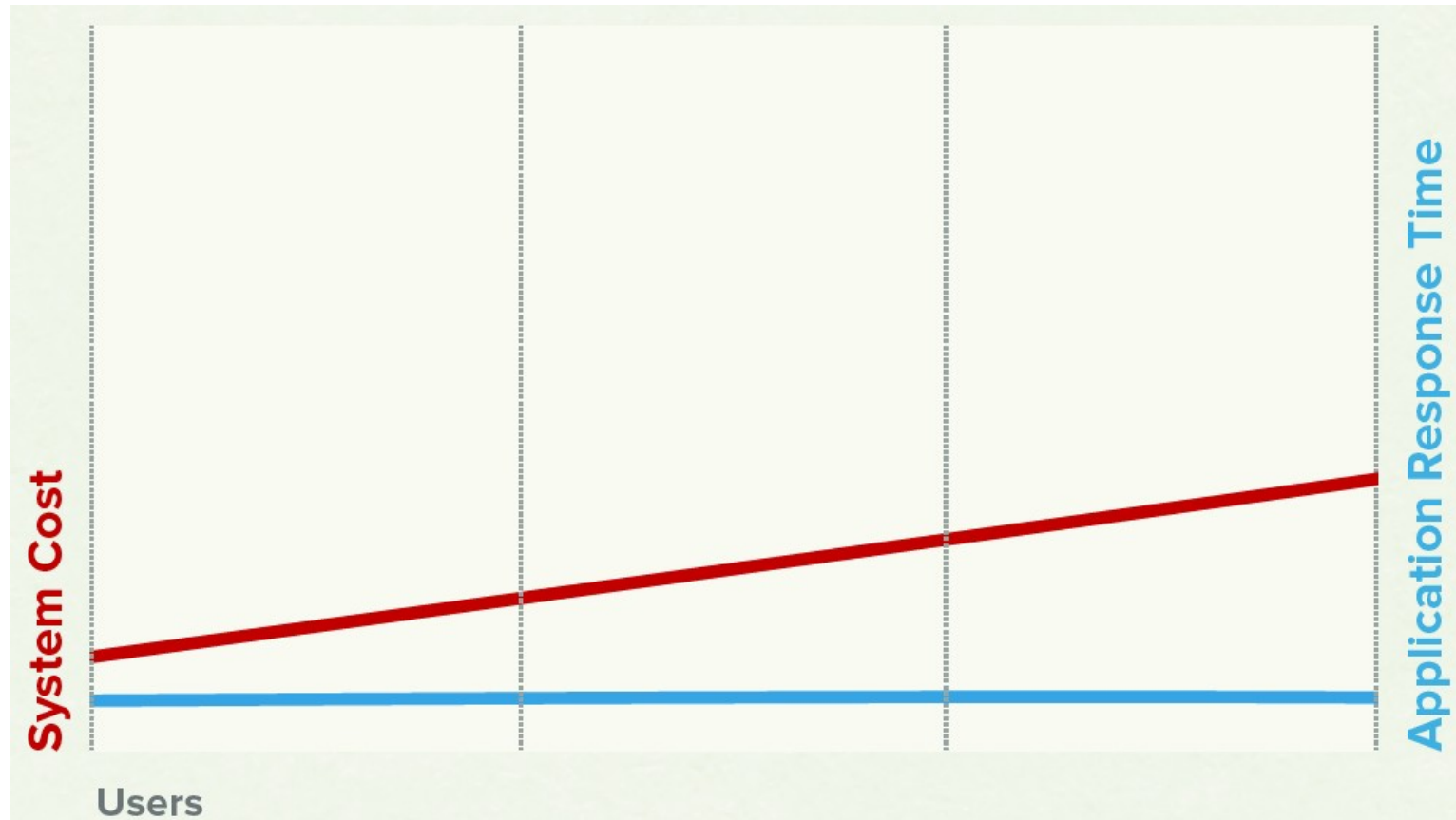
- It was first used in **1998** by Carlo Strozzi to name his relational database that did not expose the standard SQL interface.
- The term was picked up again in **2009** when a **Last.fm** developer, Johan Oskarsson, wanted to organize an event to discuss open-source distributed databases.
- The name attempted to label the emergence of a growing number of non-relational, **distributed data stores that often did not attempt to provide ACID.**



Categories of NoSQL Databases

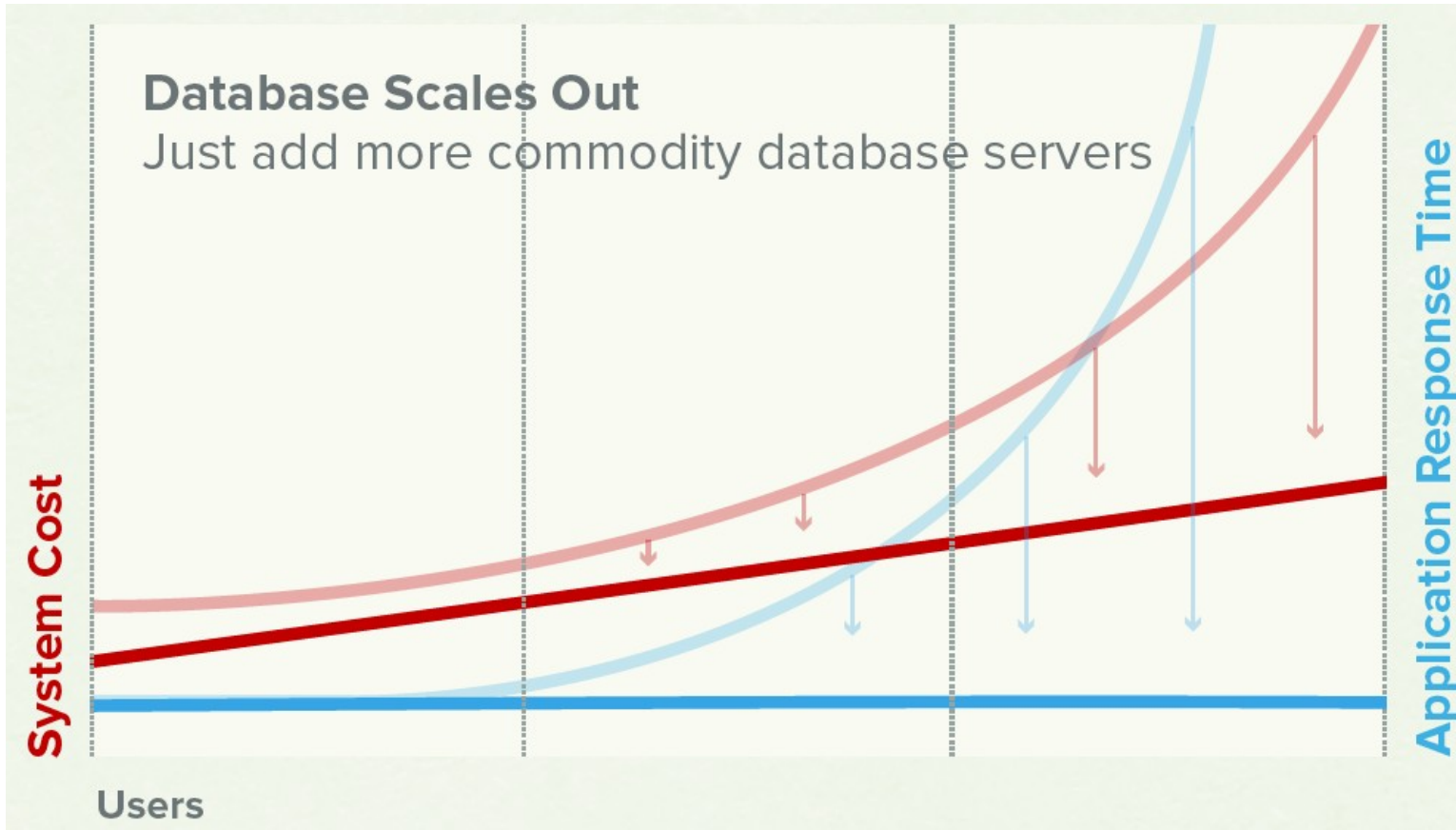
- Key/Value stores
 - **Dynamo**, Scalaris, Berkeley DB, ...
- Column-oriented databases
 - **BigTable**, Hbase, **Cassandra**, ...
- Document databases
 - MongoDB, Terrastore, SimpleDB, ...

NoSQL Cost



[<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf>]

SQL vs. NoSQL

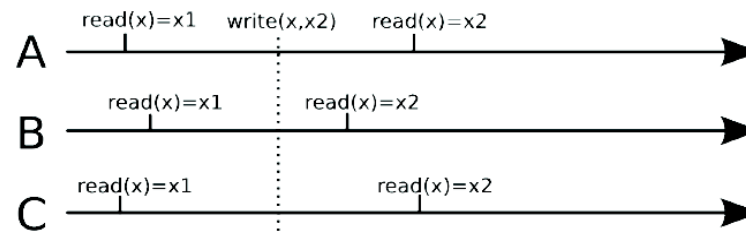


[<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf>]

Consistency

- Strong consistency

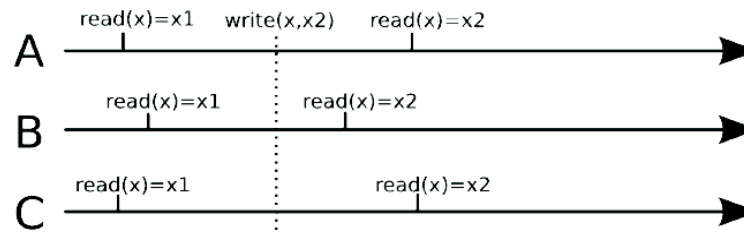
- Single storage image. Informally, after an update completes, any subsequent access will return the updated value.



Consistency

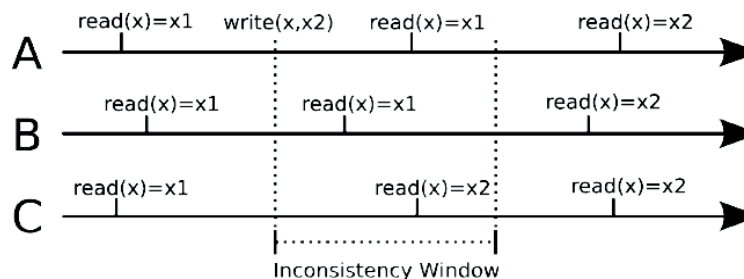
- Strong consistency

- Single storage image. Informally, after an update completes, any subsequent access will return the updated value.



- Eventual consistency

- The system does **not guarantee** that subsequent accesses will return the updated value.
- **Inconsistency window.**
- If no new updates are made to the object, **eventually** all accesses will return the last updated value.



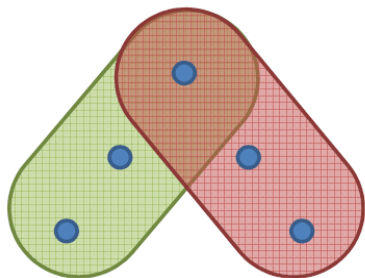
Quorum Model

- **N**: the number of nodes to which a data item is replicated.
- **R**: the number of nodes a value has to be read from to be accepted.
- **W**: the number of nodes a new value has to be written to before the write operation is finished.
- To enforce strong consistency: $R + W > N$

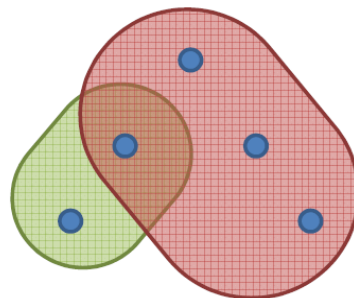


Quorum Model

- **N**: the number of nodes to which a data item is replicated.
- **R**: the number of nodes a value has to be read from to be accepted.
- **W**: the number of nodes a new value has to be written to before the write operation is finished.
- To enforce strong consistency: $R + W > N$



$$R = 3, W = 3, N = 5$$



$$R = 4, W = 2, N = 5$$



Relaxing ACID Properties

- The large-scale applications have to be **reliable**: **availability + redundancy**
- These properties are **difficult** to achieve with **ACID** properties.
- The **BASE** approach forfeits the ACID properties of **consistency and isolation** in favour of **availability, graceful degradation, and performance**.

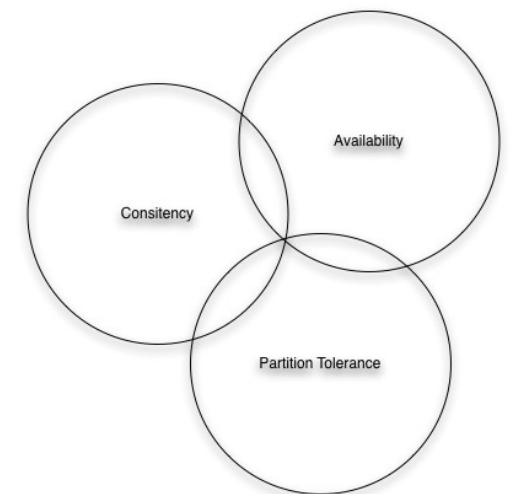
BASE Properties

- **Basically Available**: possibilities of faults but not a fault of the whole system.
- **Soft state**: copies of a data item may be inconsistent.
- **Eventually consistent**: copies becomes consistent at some later time if there are no more updates to that data item.

CAP Theorem

- **Consistency**: how a a system is in a consistent state after the execution of an operation.
- **Availability**: clients can always read and write data in a specific period of time.
- **Partition Tolerance**: the ability of the system to continue operation in the presence of network partitions.

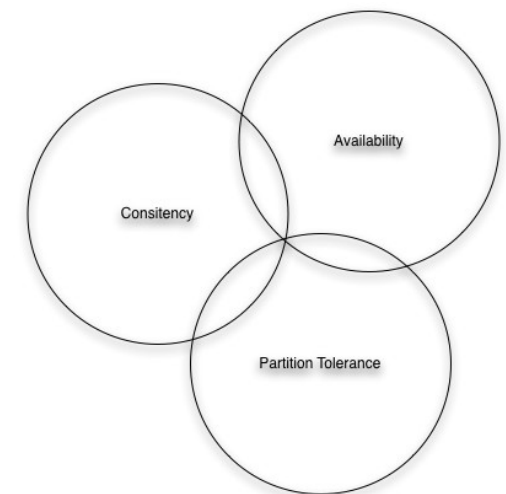
You can choose only two!



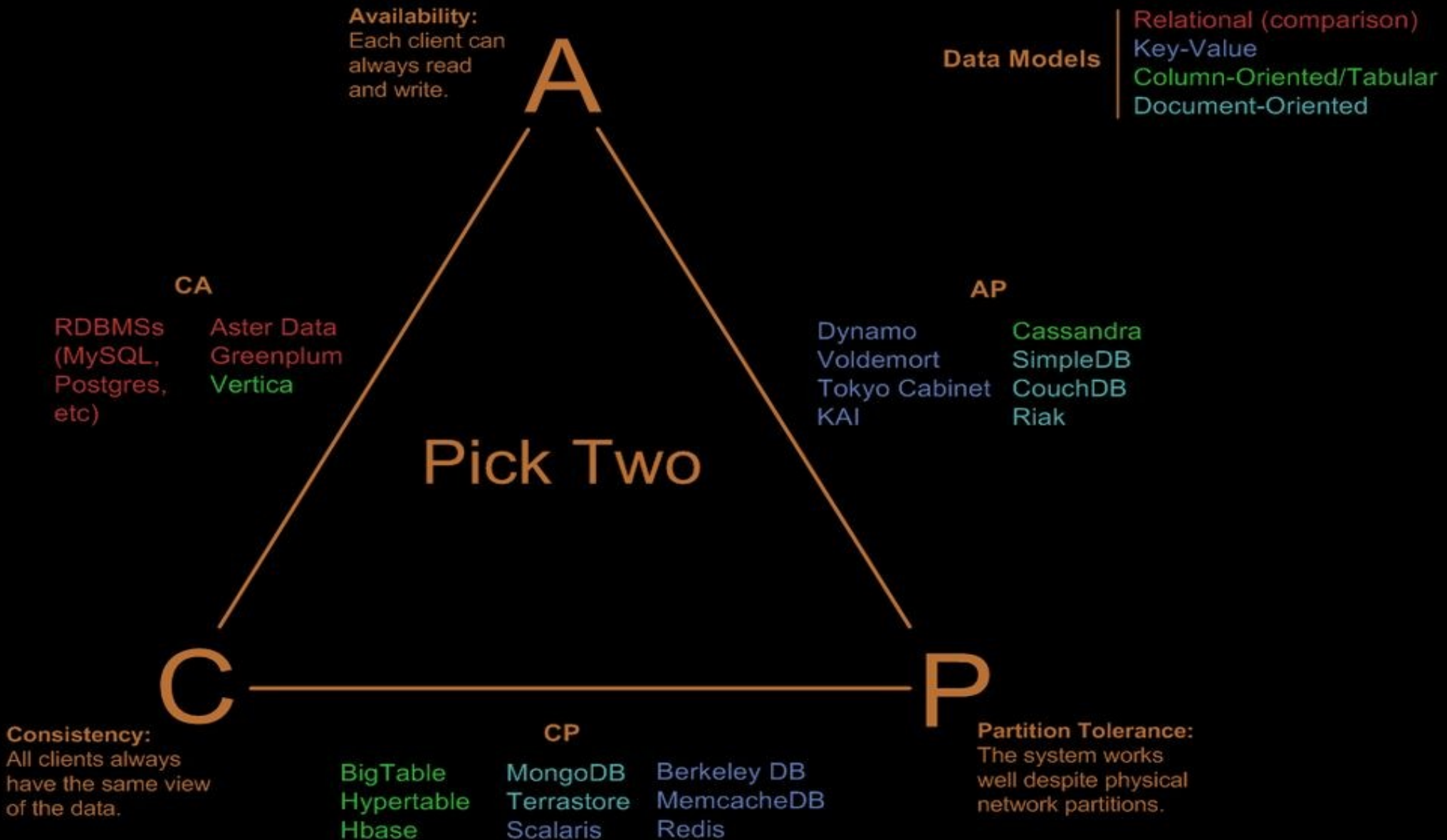
CAP Theorem

- **Consistency**: how a a system is in a consistent state after the execution of an operation.
- **Availability**: clients can always read and write data in a specific period of time.
- **Partition Tolerance**: the ability of the system to continue operation in the presence of network partitions.
- **Very large systems will partition** at some point.
 - it is necessary to decide between **C** and **A**.
 - traditional DBMS prefer **C** over **A** and **P**.
 - most Web applications choose **A**.

You can choose only two!



Visual Guide to NoSQL Systems



Dynamo



Dynamo

- Build a distributed storage system:
 - Scalability
 - Simple: key-value (put/get operations)
 - Highly available
 - Guarantee Service Level Agreements (SLA)

Design Consideration

- It sacrifices strong consistency for availability
 - Always writeable
- Conflict resolution
 - Who: data store or application
 - When: during read operation instead of write operation
- Incremental scalability
- Symmetry
 - Every node should have the same set of responsibilities as its peers
- Decentralization
- Heterogeneity

API

- `get(key)`
 - Return single object or list of objects with conflicting version and context
- `put(key, context, object)`
 - Store object and context under key
 - Context encodes system meta-data, e.g., version number

Dynamo Implementation

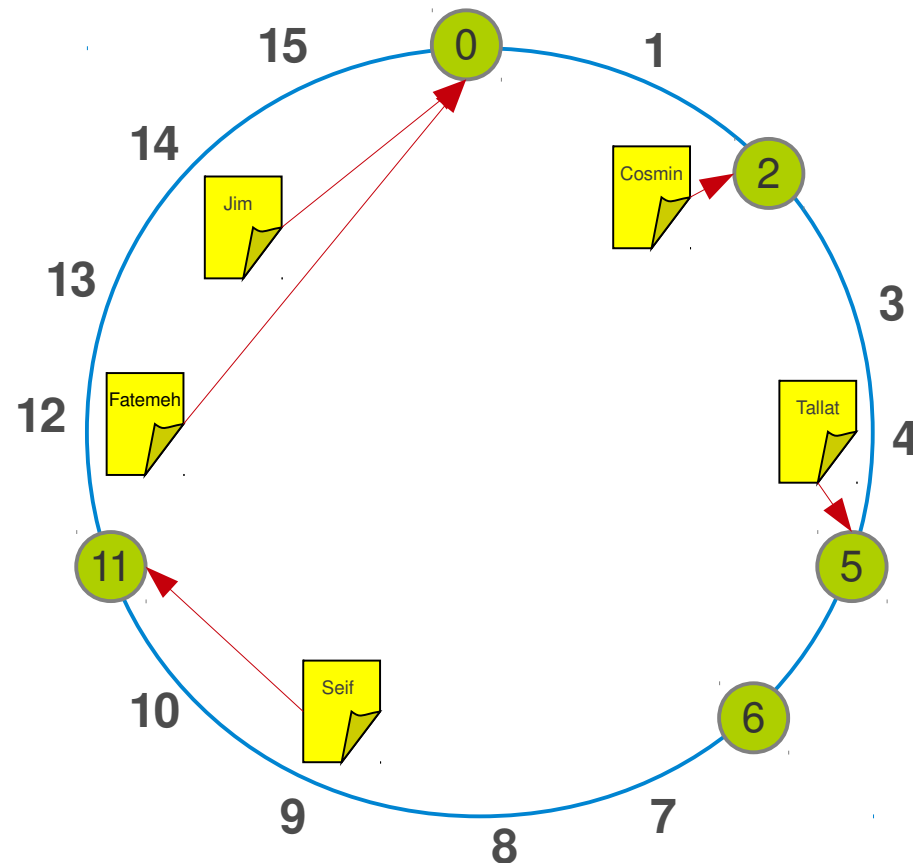
- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- Membership
- Handling failure

Dynamo Implementation

- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- Membership
- Handling failure

Data Partitioning

- Based on **consistent hashing**
- Hash key and put on **responsible node**
 - $H(\text{"Fatemeh"}) = 12$
 - $H(\text{"Cosmin"}) = 2$
 - $H(\text{"Seif"}) = 9$
 - $H(\text{"Jim"}) = 14$
 - $H(\text{"Tallat"}) = 4$

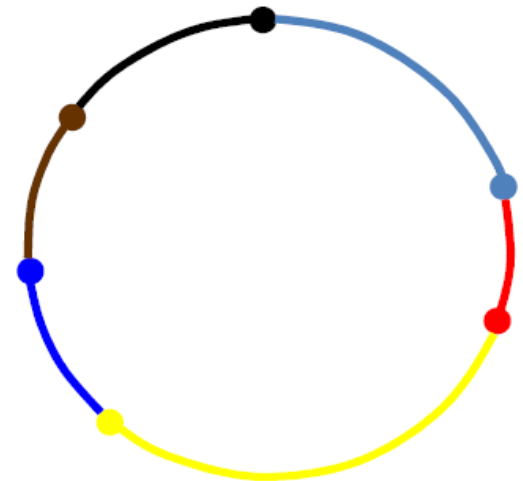


Load Imbalance

- Consistent hashing may lead to **imbalance**

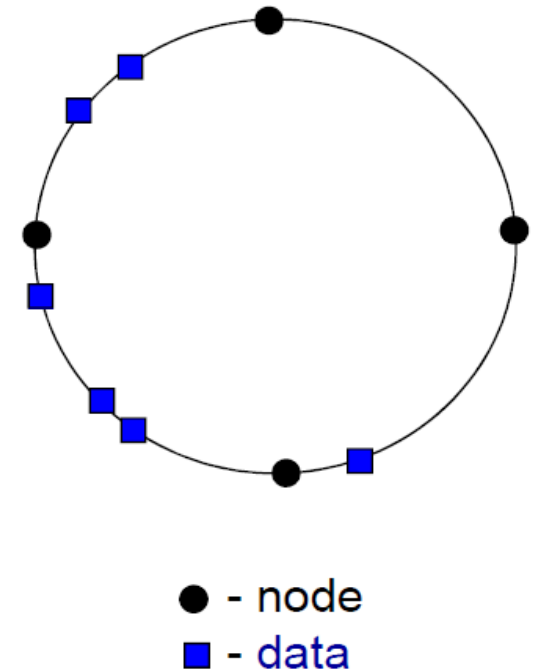
Load Imbalance

- Consistent hashing may lead to **imbalance**
 - Node identifiers may not be balanced



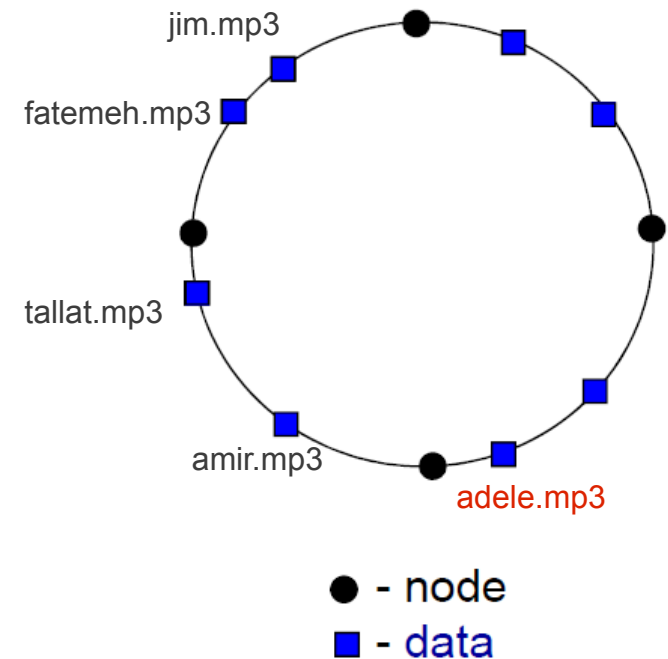
Load Imbalance

- Consistent hashing may lead to **imbalance**
 - Node identifiers may not be balanced
 - Data identifiers may not be balanced



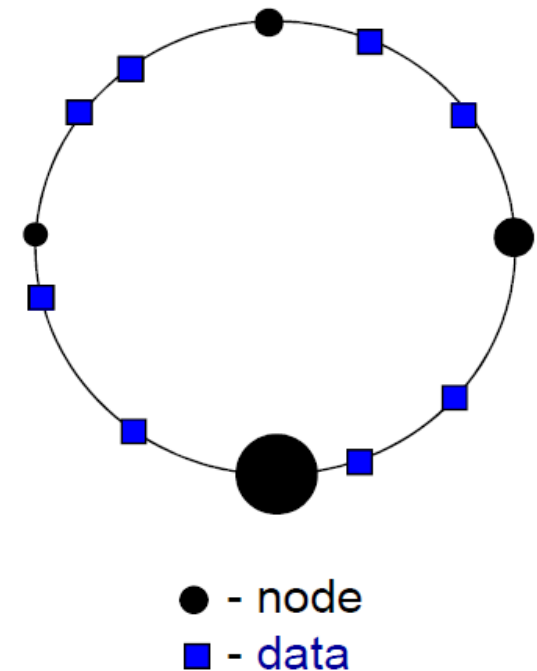
Load Imbalance

- Consistent hashing may lead to **imbalance**
 - Node identifiers may not be balanced
 - Data identifiers may not be balanced
 - Hot spots



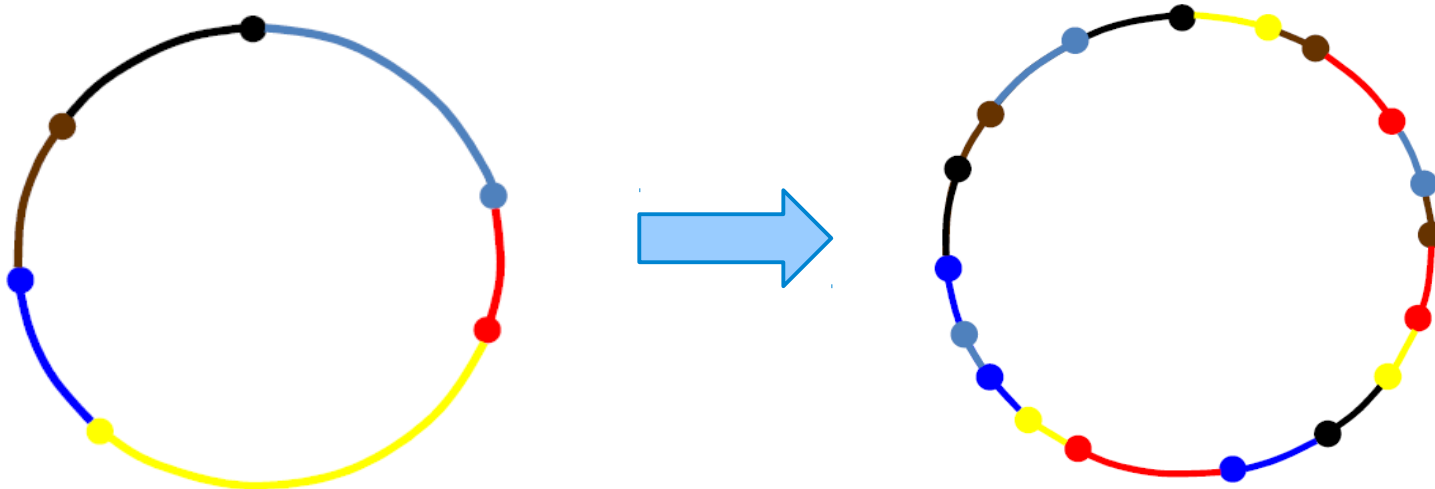
Load Imbalance

- Consistent hashing may lead to **imbalance**
 - Node identifiers may not be balanced
 - Data identifiers may not be balanced
 - Hot spots
 - Heterogeneous nodes



Load Balancing via Virtual Servers

- Each physical node picks **multiple random identifiers**.
 - Each identifier represents a **virtual server**
 - Each node runs **multiple** virtual servers
- Each node responsible for non-contiguous regions.

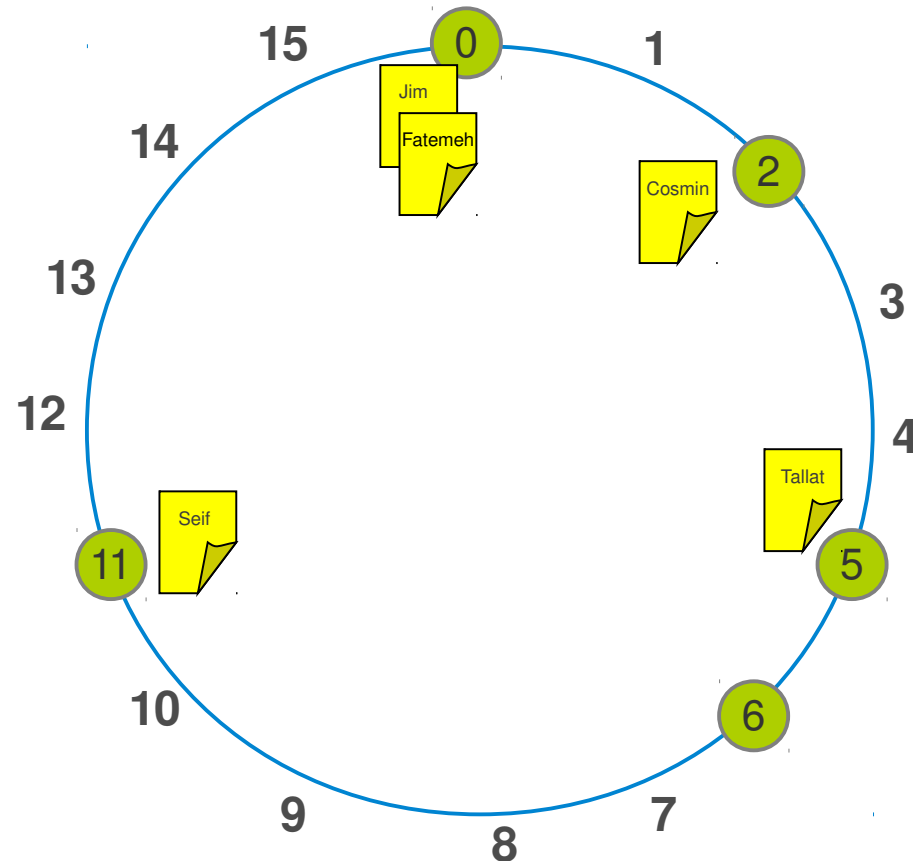


Dynamo Implementation

- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- Membership
- Handling failure

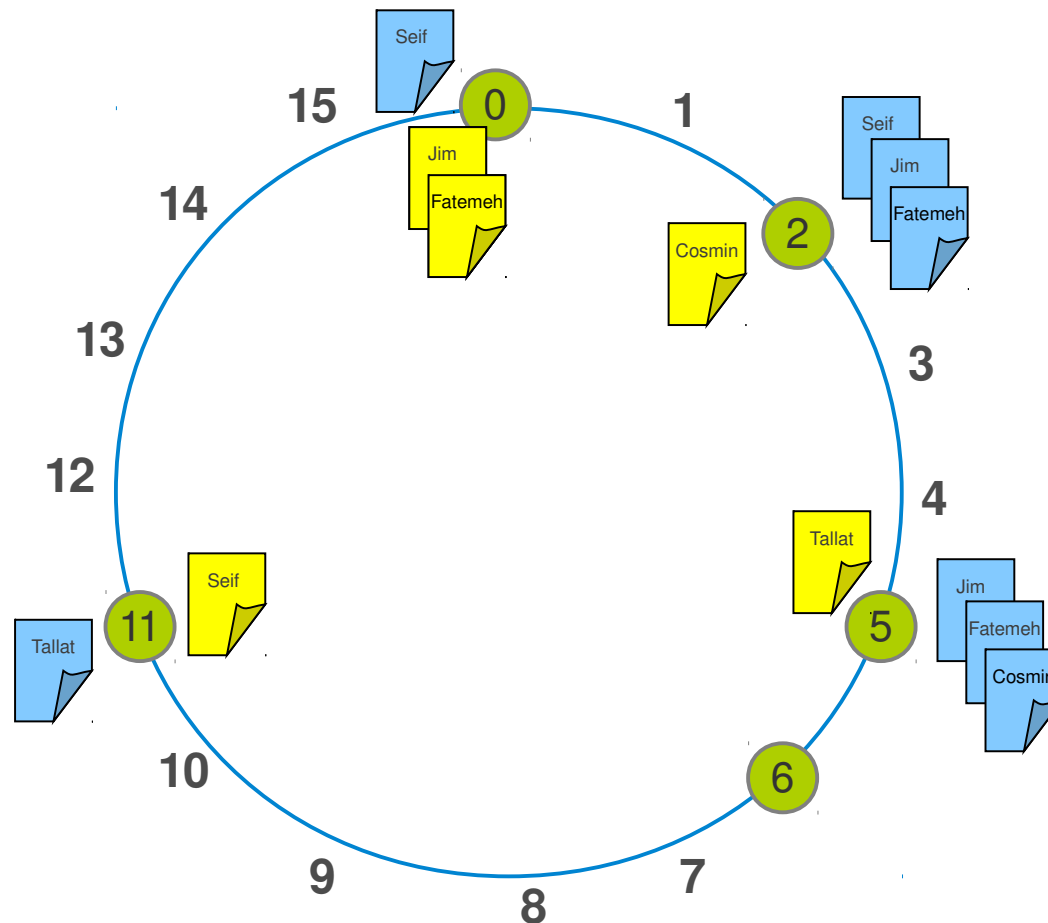
Replication

- To achieve high availability and durability, Dynamo replicates its data on **multiple hosts**.
- The list of nodes that is responsible for storing a particular key is called the **preference list**.



Replication

- To achieve high availability and durability, Dynamo replicates its data on **multiple hosts**.
- The list of nodes that is responsible for storing a particular key is called the **preference list**.



Dynamo Implementation

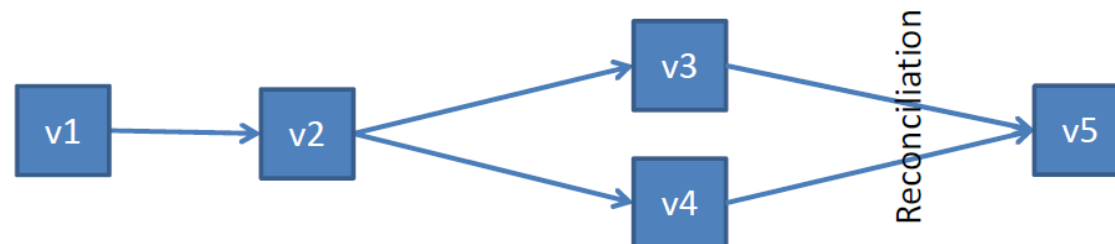
- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- Membership
- Handling failure

Data Versioning

- Updates are propagated asynchronously.
 - Replicas **eventually** become consistent.
- Each update/modification of an item results in a new and immutable version of the data.
 - Multiple versions of an object may exist.
- New versions can subsume older versions.

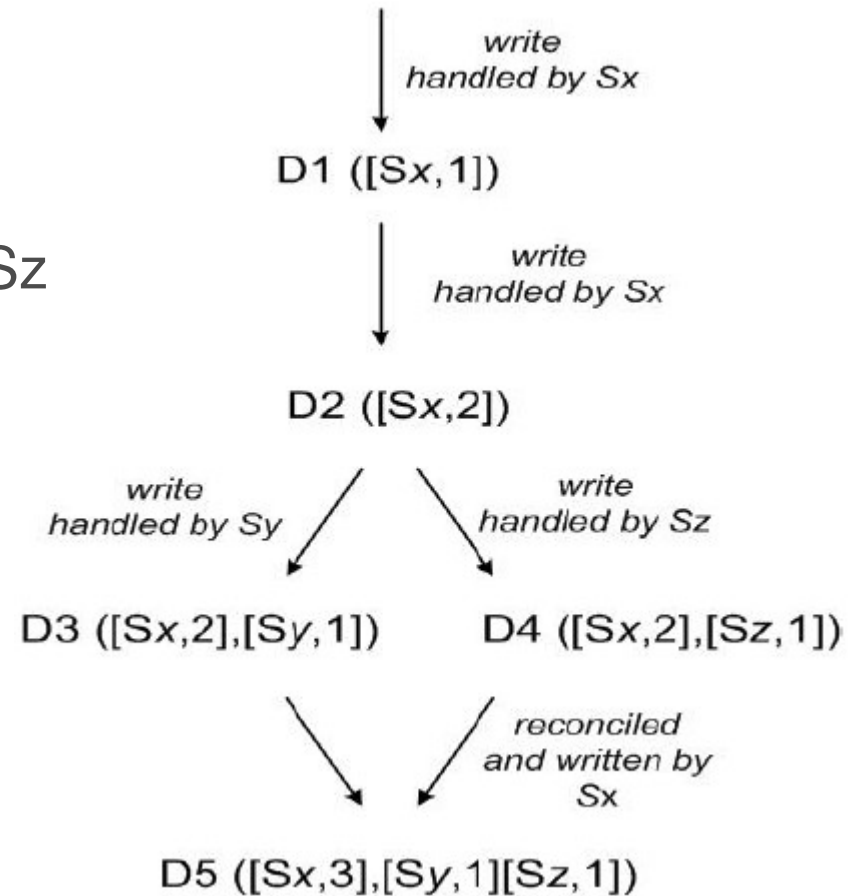
Data Versioning

- Version branching can happen due to node failures, network failures/partitions, etc.
 - Target applications are aware that multiple versions can exist.
- Use **vector clocks** for capturing **causality**, in the form of **(node, counter)**
 - If **causal**: older version can be forgotten
 - If **concurrent**: conflict exists, requiring reconciliation
- A put requires a context, i.e., which version to update



Data Versioning

- Client C1 writes new object via S_x
- C1 updates the object via S_x
- C1 updates the object via S_y
- C2 reads D2 and updates the object via S_z
- C3 reads D3 and D4 via S_x
 - The read's context is a summary of the clocks of D3 and D4: $[(S_x, 2), (S_y, 1), (S_z, 1)]$
- Reconciliation



Dynamo Implementation

- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- Membership
- Handling failure

Execution of Operations

- put and get operations
- Client can send the request
 - to the node responsible for the data (**coordinator**)
 - Save on latency, code on client
 - to a generic load balancer
 - Extra hope



Put

- Coordinator generates new vector clock and
 - writes the new version locally
- Send to N nodes
- Wait for response from $W-1$ nodes
- Using $W=1$
 - High availability for writes
 - Low durability

Get

- Coordinator requests existing versions from N
 - Wait for response from R nodes
- If multiple versions, return all versions that are causally unrelated
- Divergent versions are then reconciled
- Reconciled version written back
- Using $R=1$
 - High performance read engine

Dynamo Implementation

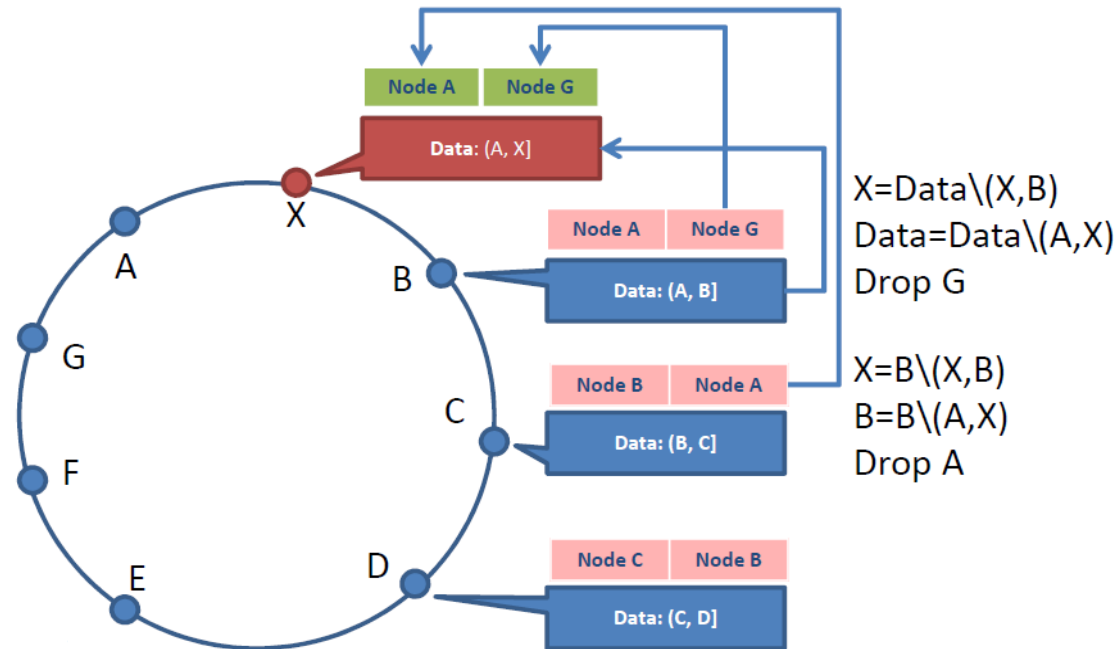
- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- **Membership**
- Handling failure

Membership Management

- Administrator **explicitly** adds and removes nodes.
- Receiving node stores changes with time stamp.
- **Gossiping** to propagate membership changes.
 - Eventually consistent view
 - $O(1)$ hop overlay

Adding Node

- A new node X added to system
 - X is assigned key ranges w.r.t. its virtual servers
 - For each key range, it transfers the data items



Failure Detection

- Passive failure detection
 - Use **pings** only for detection from failed to alive
 - A detects B as failed if it doesn't respond to a message
 - A periodically checks if B is alive again

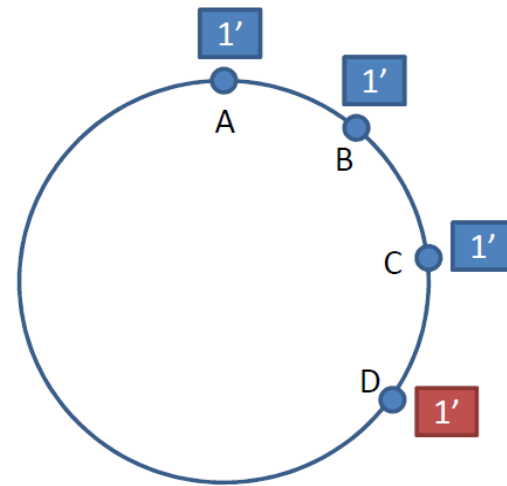
- In the absence of client requests, A doesn't need
 - to know if B is alive

Dynamo Implementation

- Data partitioning
- Replication
- Data versioning
- Execution of put and get operations
- Membership
- Handling failure

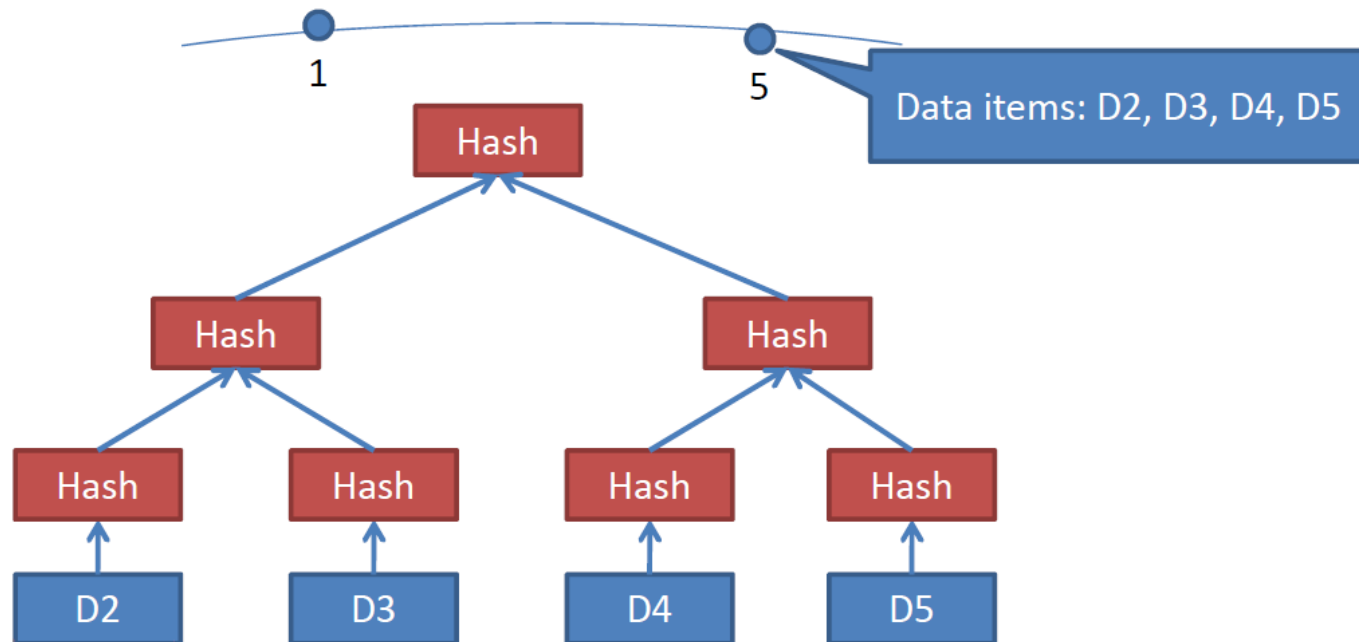
Handling Transient Failures

- Due to partitions, quorums might not exist
 - Sloppy quorum
 - Create transient replicas
 - **N healthy nodes** from the **preference list**
 - Reconcile after partition heals
- Say A is unreachable
- “put” will use D
- Later, D detects A is alive
 - send the replica to A
 - remove the replica



Handling Permanent Failure

- **Anti-entropy** for replica synchronization.
- Use **Merkle trees** for fast **inconsistency detection** and minimum transfer of data.
 - Nodes maintain Merkle tree of each key range.
 - Exchange root of Merkle tree to check if the key ranges are up-to-date.



Dynamo Summary

- CAP
- Key/Value storage: [put](#) and [get](#)
- Data partitioning: [consistent hashing](#)
- Load balancing: [virtual server](#)
- Replication: several nodes, [preference list](#)
- Data versioning: vector clock, [resolve conflict at read time](#) by the application
- Membership management: join/leave by [admin](#), [gossip-based](#) to update the nodes' views, [ping](#) to detect failure
- Handling transient failure: [sloppy quorum](#)
- Handling permanent failure: [Merkle tree](#)

BigTable



BigTable

- Highly available distributed storage for **structured data** that is designed to **scale to a very large size**.
- Built with structured data in mind
 - **URLs**: content, metadata, links, anchors, page rank
 - **User data**: preferences, account info, recent queries
 - **Geography**: roads, satellite images, points of interest, annotations
- Used at:
 - Google Finance
 - Orkut
 - Google Earth & Google Maps
 - Dozens of others...



BigTable 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

Table Model

- Distributed multi-dimensional sparse map
- (row, column, timestamps) → value

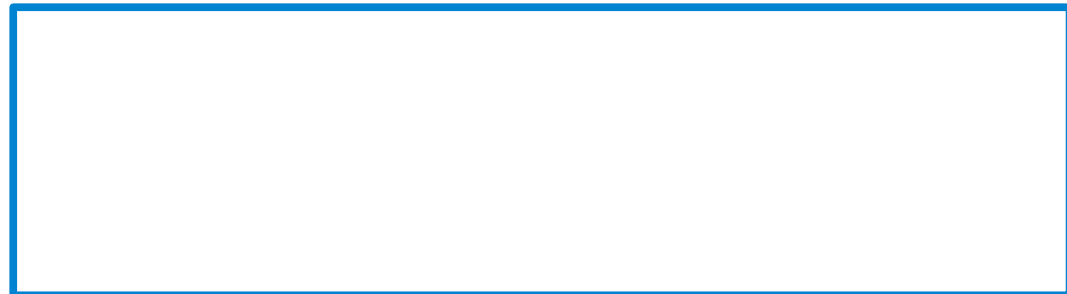


Table Model - Rows

- Every read or write in a row is **atomic**.
- Rows sorted in **lexicographical** order.



Table Model - Columns

- Column families
 - Group of (the same type) column keys
 - The basic unit of data access
 - Created before data being stored
 - Column key naming: `family:qualifier`

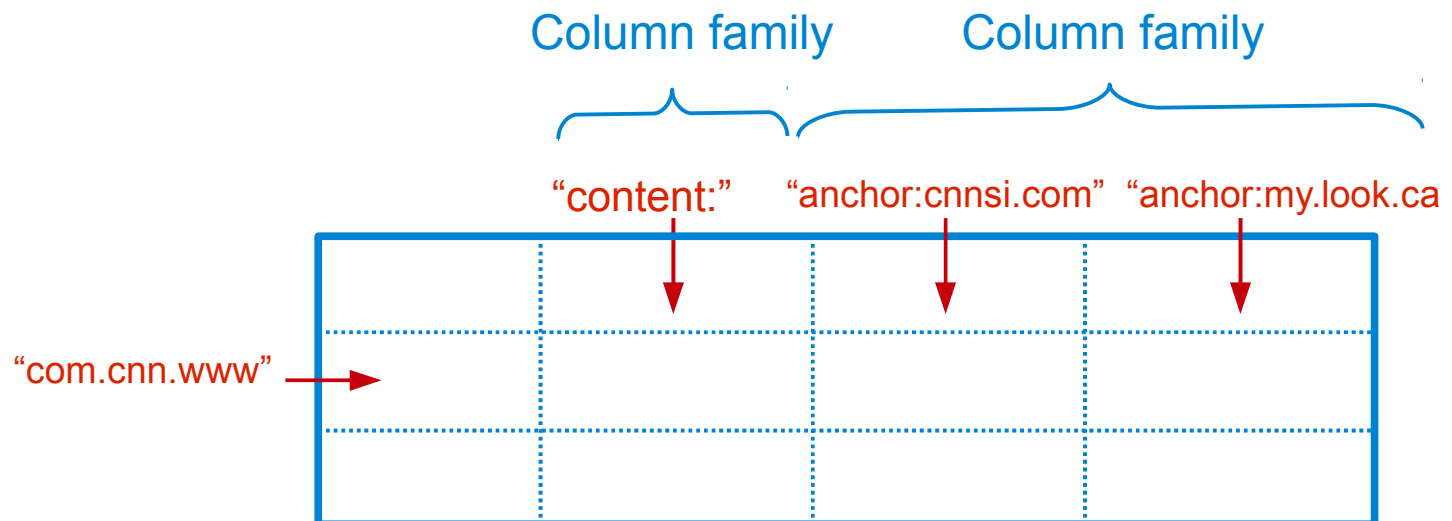
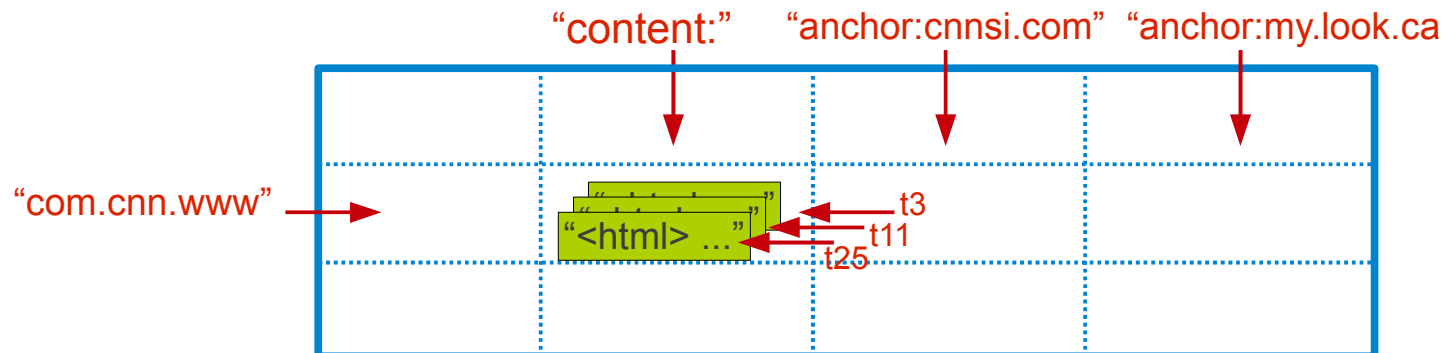


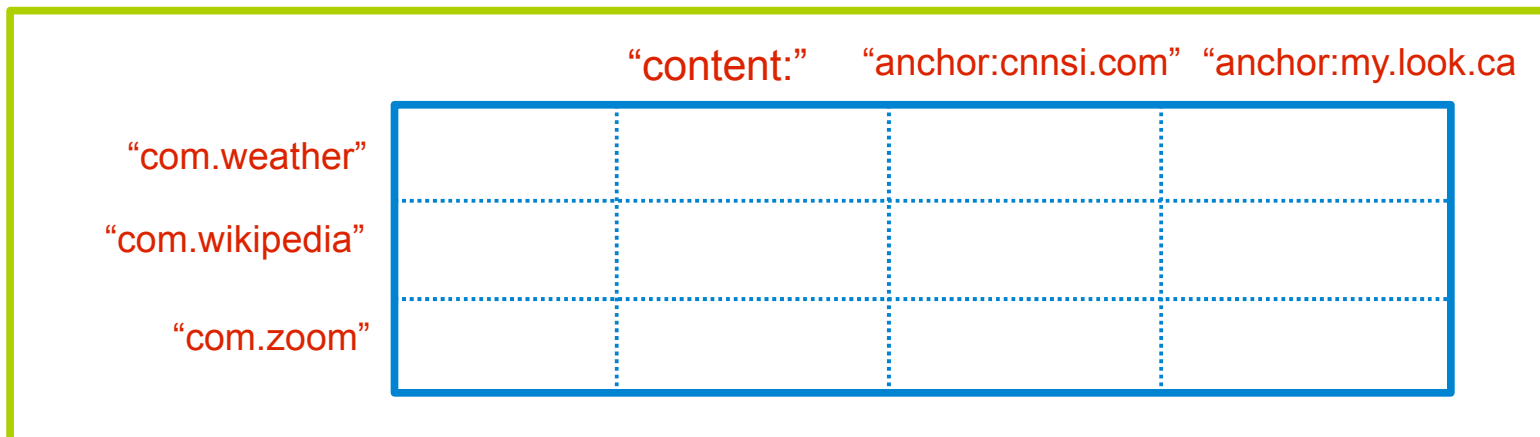
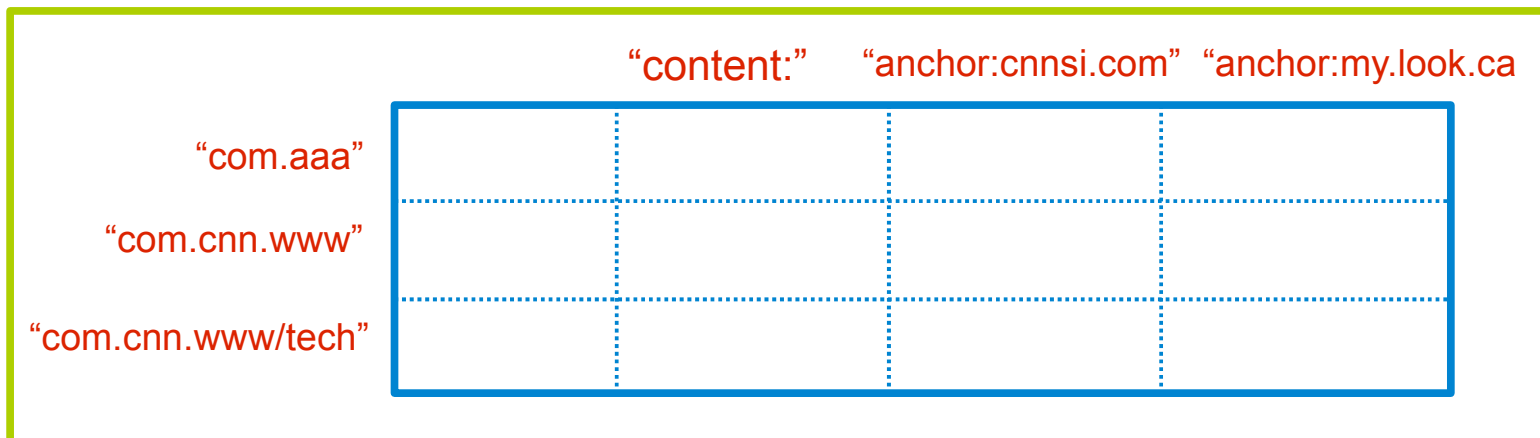
Table Model - Timestamps

- Each column family may contain **multiple versions**
- Version indexed by a 64-bit timestamp
 - Real time or assigned by client
- Per-column-family settings for **garbage collection**
 - Keep only latest n versions
 - Or keep only versions written since time t
- Retrieve most recent version if no version specified



Tablets: Pieces of a Table

- A **table** starts as one **tablet**.
 - As it grows, it is **split** into multiple tablets.
- **Tablet** = range of contiguous rows



API

- Create/delete tables & column families
- Change cluster, table, and column family metadata
- Write or delete values
- Read values from specific rows
- Iterate over a subset of data in a table
- Atomic read-modify-write row operations

API – Writing Example

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

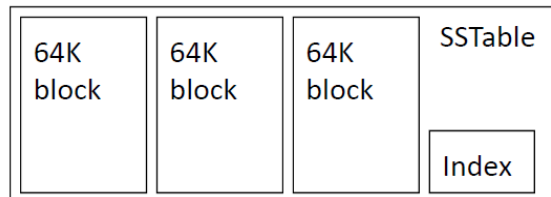
API – Reading Example

```
Scanner scanner (T) ;
scanner.Lookup ("com.cnn.www") ;

ScanStream *stream;
stream = scanner.FetchColumnFamily ("anchor") ;
stream->SetReturnAllVersions () ;
for (; !stream->Done (); stream->Next ()) {
    printf ("%s %s %lld %s\n",
        scanner.RowName (),
        stream->ColumnName (),
        stream->MicroTimestamp (),
        stream->Value ()) ;
}
```

BigTable Supporting Services (1/2)

- Google File System (**GFS**)
 - For storing log and data files
- Cluster management system
 - For scheduling jobs, monitoring health, dealing with failures
- Google **SSTable**
 - Internal file format
 - Provides a persistent, ordered, immutable **map from keys to values**
 - Memory or disk based



BigTable Supporting Services (2/2)

- Chubby
 - Highly-available & persistent distributed lock (lease) service
 - Five active replicas; one elected as master to serve requests
 - Majority must be running
 - Paxos used to keep replicas consistent
- Chubby is used to:
 - Ensure there is only one active master
 - Store bootstrap location of BigTable data
 - Discover tablet servers
 - Store BigTable schema information
 - Store access control lists

BigTable Implementation

- Major components
- Tablet location
- Tablet assignment
- Tablet serving
- Compactions

BigTable Implementation

- Major components
- Tablet location
- Tablet assignment
- Tablet serving
- Compactions

Major Components

- Tablet server
- Master server
- Client library

Major Components – Tablet Server

- **Many** tablet servers
- Can be added or removed dynamically
- Each **manages a set of tablets** (typically 10-1000 tablets/server)
- **Handles read/write requests to tablets**
- **Splits tablets** when too large

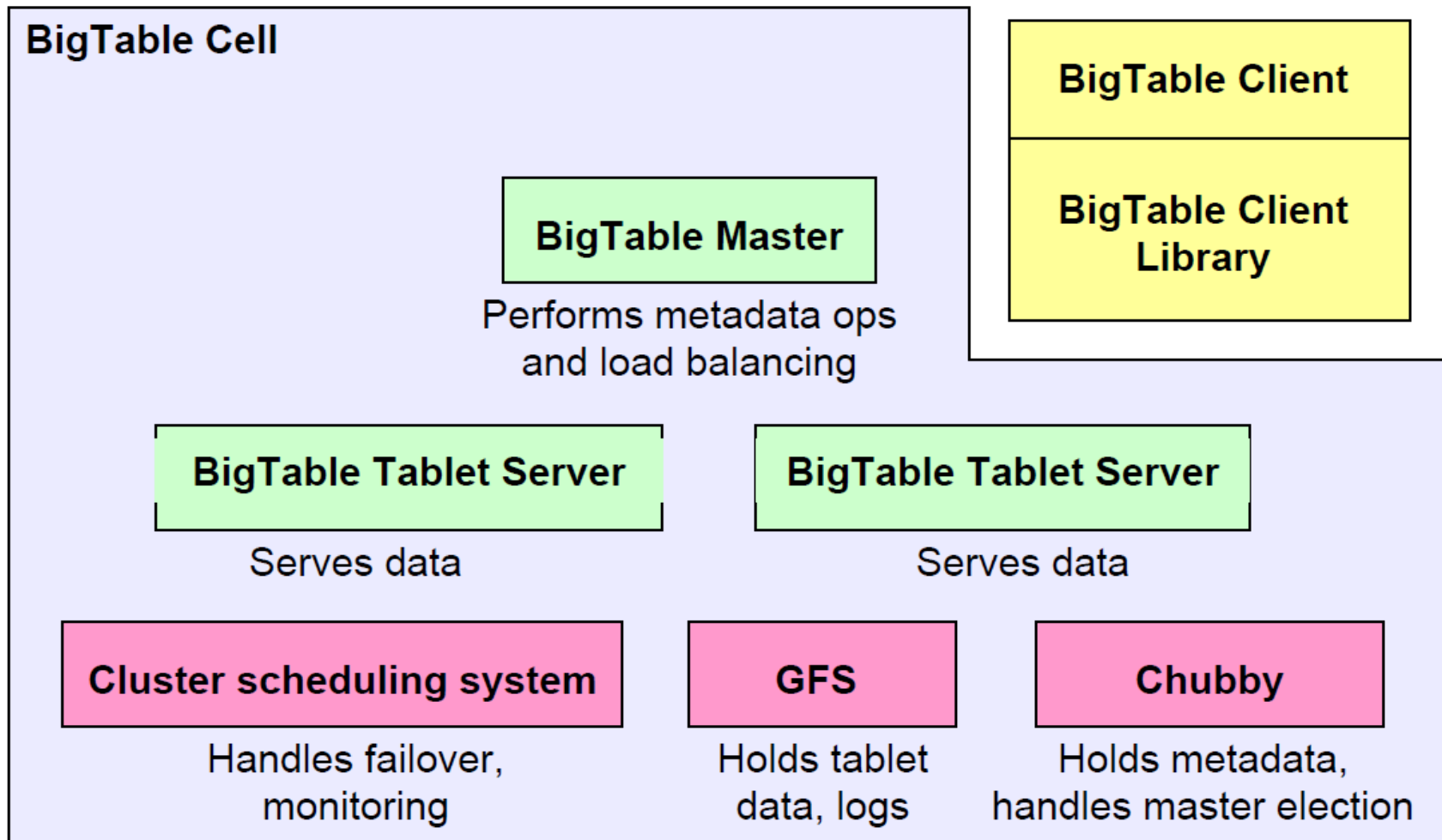
Major Components – Master Server

- One master server
- Assigns tablets to tablet server
- Balances tablet server load
- Garbage collection of unneeded files in GFS

Major Components – Client Library

- Library that is linked into every client
- Client data does not move through the master
- Clients communicate directly with tablet servers for reads/writes

High-level Structure

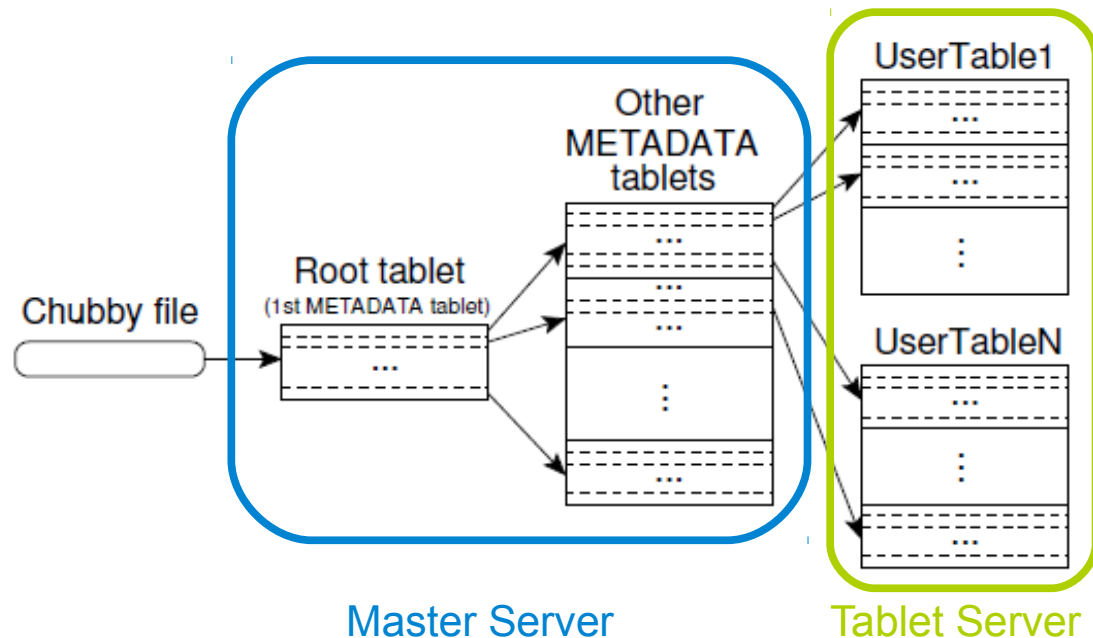


BigTable Implementation

- Major components
- Tablet location
- Tablet assignment
- Tablet serving
- Compactions

Table Location – Finding a Tablet

- **Three-level** hierarchy
- **Root tablet** contains location of all tablets in a special METADATA table
- **METADATA** table contains location of each tablet under a row
key = f(tablet table ID, end row)
- The client library **caches** tablet locations.



BigTable Implementation

- Major components
- Tablet location
- Tablet assignment
- Tablet serving
- Compactions

Tablet Assignment

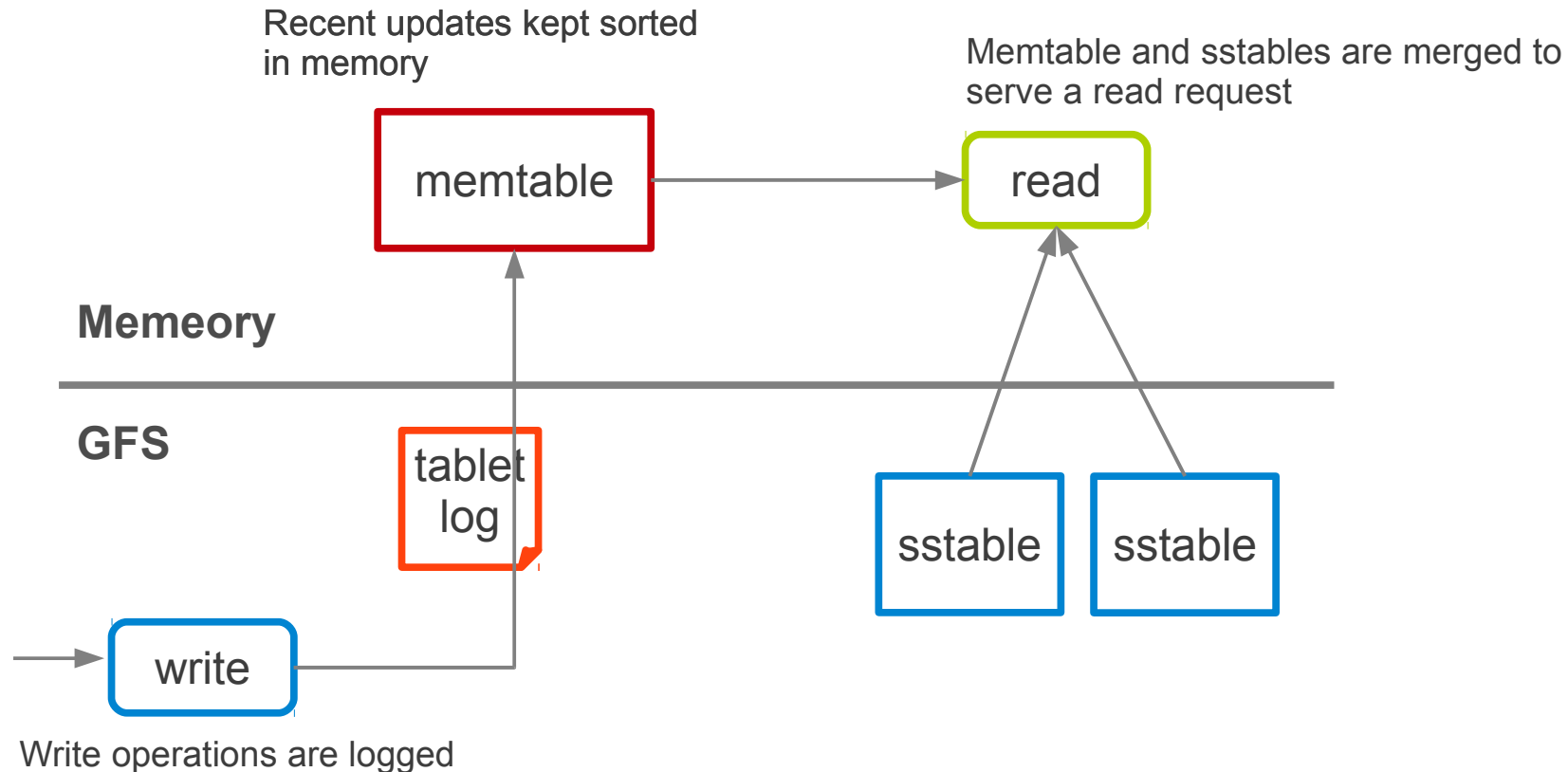
- 1 tablet → 1 tablet server
- Master keeps tracks of set of **live tablet serves** and **unassigned tablets**.
- Master sends a **tablet load request** for unassigned tablet to the tablet server.
- BigTable uses **Chubby** to keep track of tablet servers.
- Master detects the status of the lock of each tablet server by **checking periodically**.
 - Master is responsible for finding when tablet server is no longer serving its tablets and reassigning those tablets as soon as possible.

BigTable Implementation

- Major components
- Tablet location
- Tablet assignment
- **Tablet serving**
- Compactions

Tablet Serving

- Updates committed to a commit log.
- Recently committed updates are stored in memory – **memtable**
- Older updates are stored in a sequence of **SSTables**.



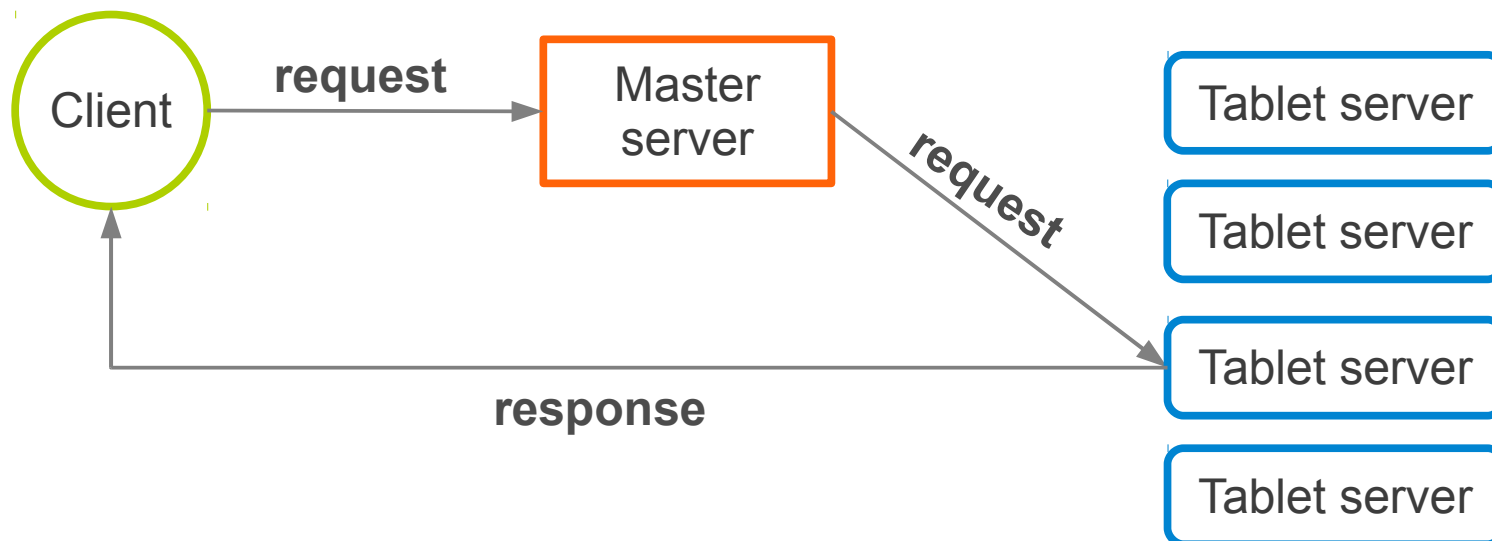
Tablet Serving

- **Strong consistency**

- Only one tablet server is responsible for a given piece of data.
- Replication is handled on the GFS layer

- **Trade-off with availability**

- If a tablet server fails, its portion of data is temporarily unavailable until a new server is assigned.



BigTable Implementation

- Major components
- Tablet location
- Tablet assignment
- Tablet serving
- Compactions

Compactions

- When in-memory is full
- **Minor compaction**
 - convert the **memtable** into an **SSTable**
 - Reduce memory usage and log traffic on restart
- **Merging Compaction**
 - Reduces number of SSTables
 - Reads the contents of **a few SSTables** and the **memtable**, and writes out a new **SSTable**
- **Major Compaction**
 - Merging compaction that **results in only one SSTable**
 - No deleted records, only sensitive live data

BigTable Summary

- CAP
- Column-oriented storage: (row, column, timestamps) → string
- A table is divided into a number of tablets, and each tablet is one or more SSTable file in GFS.
- One master server that communicates only with tablet servers.
- Multiple tablet servers that perform actual client accesses
- Chubby lock service holds metadata, e.g., the location of the root metadata tablet for the table.
- Three-level hierarchy
- Compactions: minor/merging/major



Cassandra

facebook

amazon.com

Dynamo

Cluster management, replication, fault tolerance



Cassandra



Google

BigTable



Sparse, columnar data model, storage architecture

From Dynamo

- Symmetric p2p architecture
- Gossip based discovery and error detection
- Distributed key-value store
 - Partitioning
 - Topology discovery
- Eventual consistency

From BigTable

- Sparse Column oriented sparse array
- SSTable disk storage
 - Append-only commit log
 - Memtable (buffering and sorting)
 - Immutable sstable files
 - Compaction

Any Questions?