Structured Big Data 2: NoSQL Systems

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Lecture material is mostly home-grown, partly taken with permission and courtesy from Professor Shih-Wei Liao of NTU.

Recap from Last Lecture

- Why using NoSQL instead of RDBMS?
  - As data scales, RDBMS cannot handle it
- The schema from RDBMS will hinder the scalability
- Need the data model with loosen schema

  → **NoSQL**

Objectives of this lecture

- Deep dive into several NoSQL databases
- NoSQL Database Systems to be discussed
  - DynamoDB
  - Cassandra
  - MongoDB
- You may study other similar systems in your independent study

NoSQL: The Name

- "SQL" = Traditional relational DBMS
- Recognition over past decade or so:

  Not every data management/analysis problem is best solved using a traditional relational DBMS
- "NoSQL" = "No SQL" =
  Not using traditional relational DBMS
- "No SQL" ≠ Don't use SQL language
- "NoSQL" = "Not Only SQL"

What's Wrong with RDBMS

- Nothing. One size fits all? Not really.
- Impedance mismatch.
  - Object Relational Mapping doesn't work quite well.
- Rigid schema design.
- Harder to scale.
- Replication.
- Joins across multiple nodes? Hard.
- How does RDMS handle data growth? Hard.
- Need for a DBA.
- Many programmers are already familiar with it.
- Transactions and ACID make development easy.
- Lots of tools to use.

NoSQL Systems

- Alternative to traditional relational DBMS
  - \+ Flexible schema
  - \+ Quicker/cheaper to set up
  - \+ Massive scalability (scale horizontally instead of vertically)
  - \+ Relaxed consistency → higher performance & availability
  - – No declarative query language → more programming
  - – Relaxed consistency → fewer guarantees

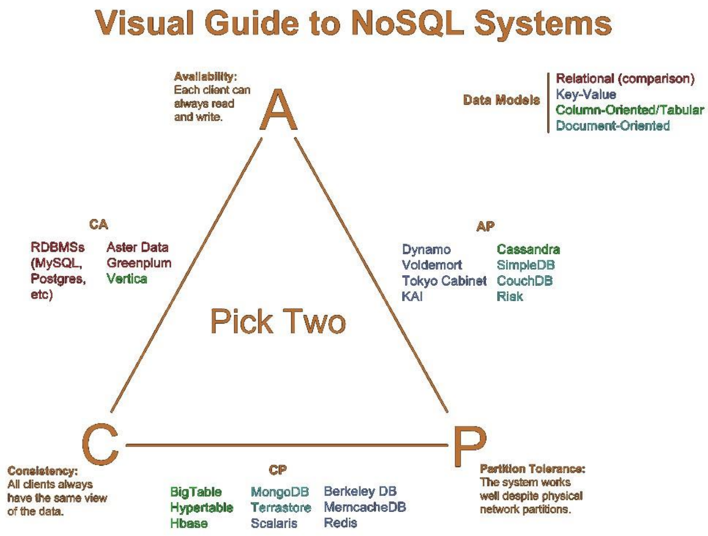
How did we get here?

- Explosion of social media sites (Facebook, Twitter) with **large data needs**
- Rise of **cloud-based solutions** such as Amazon S3 (Simple Storage Solution)
- Just as moving to dynamically-typed languages (Ruby/Groovy), a shift to **dynamically-typed data** with frequent schema changes
- **Open-source** community

Seeds of the NoSQL Movement

- Three major development were the seeds of the NoSQL movement
  - BigTable (Google)
  - Dynamo (Amazon)
    - Gossip protocol (discovery and error detection)
    - Distributed key-value data store
    - Eventual consistency
  - CAP Theorem

CAP Theorem Revisited



The Perfect Storm

- Large datasets, acceptance of alternatives, and dynamically-typed data has come together in a perfect storm
- Not a backlash/rebellion against RDBMS
- SQL is a rich query language that cannot be rivaled by the current list of NoSQL offerings

- "NoSQL" = "Not Only SQL"

Why NoSQL?

Example #1: Web log analysis

Each record: UserID, URL, timestamp, additional-info

Task: Load into database system

Why NoSQL?

Example #1: Web log analysis

Each record: UserID, URL, timestamp, additional-info

Task: Find all records for…

- Given UserID
- Given URL
- Given timestamp
- Certain construct appearing in additional-info

Why NoSQL?

Example #1: Web log analysis

Each record: UserID, URL, timestamp, additional-info
Separate records: UserID, name, age, gender, …

Task: Find average age of user accessing given URL

Why NoSQL?

Example #2: Social-network graph

Each record: $UserID_1$, $UserID_2$
Separate records: UserID, name, age, gender, …

Task: Find all friends of friends of friends of … friends of given user

Why NoSQL?

Example #3: Wikipedia pages

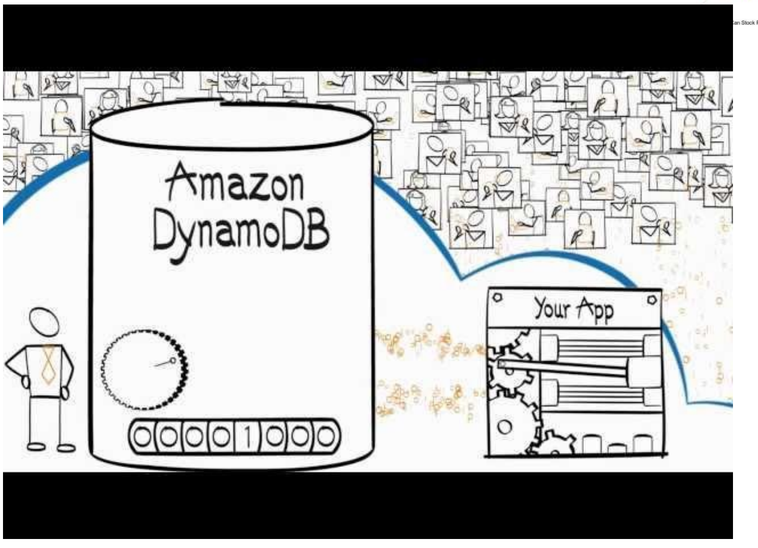
Large collection of documents
Combination of structured and unstructured data

Task: Retrieve introductory paragraph of all pages about U.S. presidents before 1900

Dynamo: Outline

- Background & motivation
- Implementation
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall and Werner Vogels, “Dynamo: Amazon's Highly Available Key-Value Store”, in the *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

Amazon DynamoDB



Background

- Amazon's eCommerce platform architecture
- Composed of highly decentralized, loosely coupled, service-oriented architecture
- Service based on a well-defined interface accessible over the network
- hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide

Amazon Services

- Many services store and retrieve data based on key (called key-value access)
- Examples of key-value access in Amazon
  - best seller lists, shopping carts, customer preferences, sales rank
- Traditional RDBMS as persistent store is not suitable
  - No need for strong consistency
  - No use of schema
  - No need of complex querying and optimization
  - No need for complex management functionalities
  - Scale up v.s. scale out

Motivation

- Focus on reliability and scalability
- Need a highly-available storage system instead of consistency
- Consistency v.s. Availability
  - High availability is more important
  - Client-perceived consistency
  - Tradeoff consistency in favor of higher availability

Requirements and Assumptions

- Query model:
  - Simple read and write data based on key
  - Data stored as a blob (Binary Large Object)
  - Object size small (less than 1MB)
- ACID properties
  - Weaker consistency: Eventual consistency
  - No isolation guarantee
  - Only single key updates

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to **ACID**

Requirements and Assumptions

- Efficiency
  - Based on commodity hardware
  - Stringent SLA requirements (next slide)
  - Tradeoffs: performance, cost efficiency, availability, and durability
- Other: non-hostile environment, no security-related requirements (used only by Amazon's internal services)

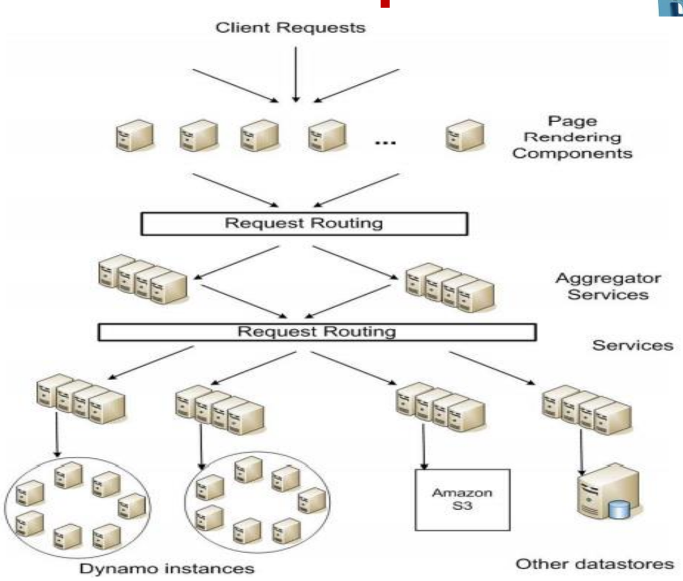
Service Level Agreements (SLA)

- **Definition**: a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions
- Example: response time within 300ms for 99.9% of its requests for a peak client load of 500 req/sec
- SLAs expresses as 99.9th percentile of the distribution
  - Not the traditional mean or average
  - Why? What is the implication of this?

Amazon's Service Oriented Infrastructure

- Decentralized SOA (next slide)
- a page request to a e-commerce site typically requires the rendering engine to construct its response by sending requests to over 150 services
- Services often have multiple dependencies (call chains)
- To ensure a clear bound on page delivery each service within the call chain must obey its performance contract

SOA of Amazon's platform



Implementation

Problem	Technique	Advantage
Partitioning of data	Consistent Hashing	Incremental Scalability
Handling temporary failures	Sloppy Quorum	Provides high availability and durability guarantee when some of the replicas are not available.
High availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates

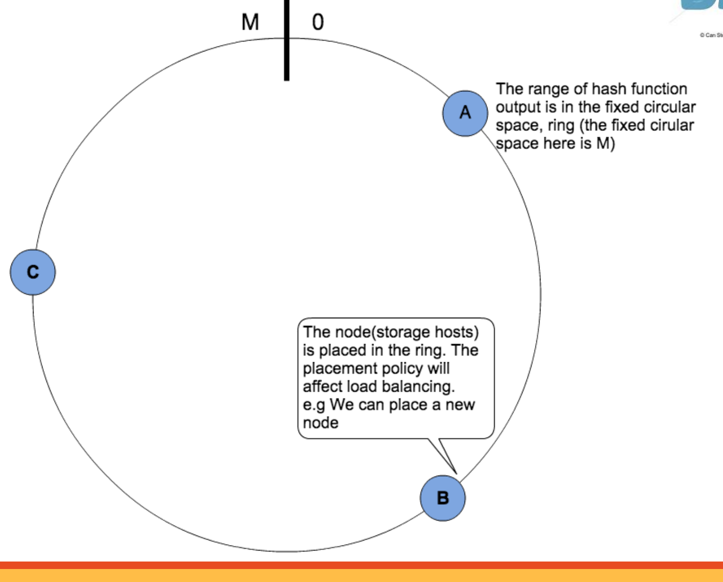
Implementation

- Partition: must be balanced
- Why ?
  - Design requirement: to scale incrementally
  - Need to partition data over the set of nodes(e.g storage host) dynamically
  - balanced distribution of data
  - =>Consistent hashing

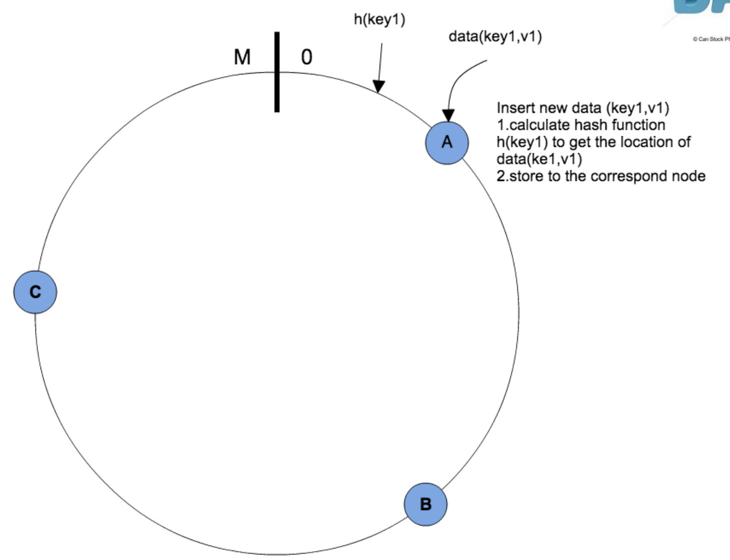
Basic Consistent Hashing

- Hash keys to a fixed circular space or "ring"
- Each node is assigned a random position in the ring
- Each data is assigned to a node by hashing its key and walking clockwise
- Each node is responsible for the **region** between it and its predecessor
- Departure or arrival of a node only affects its immediate neighbors

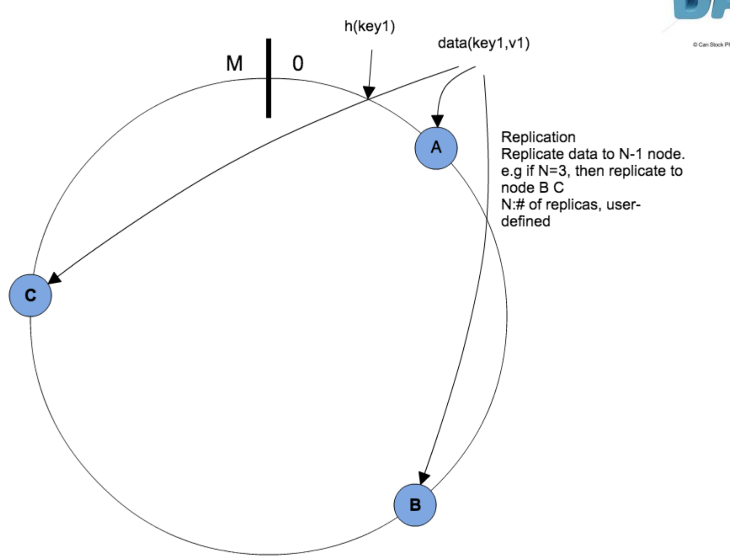
Partition: Consistent



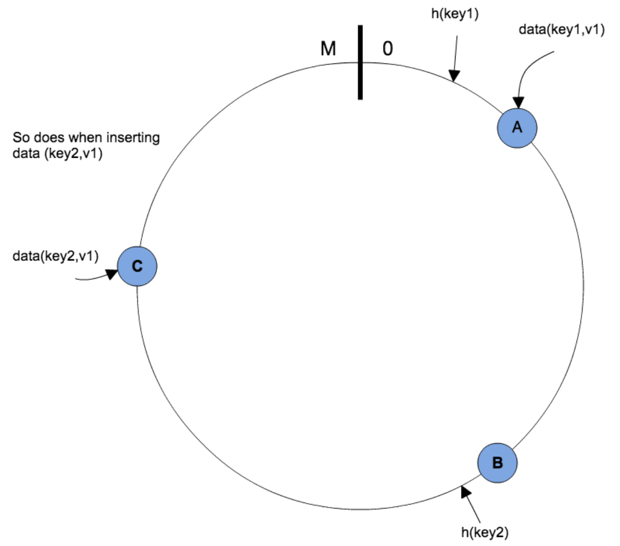
Insert New Data



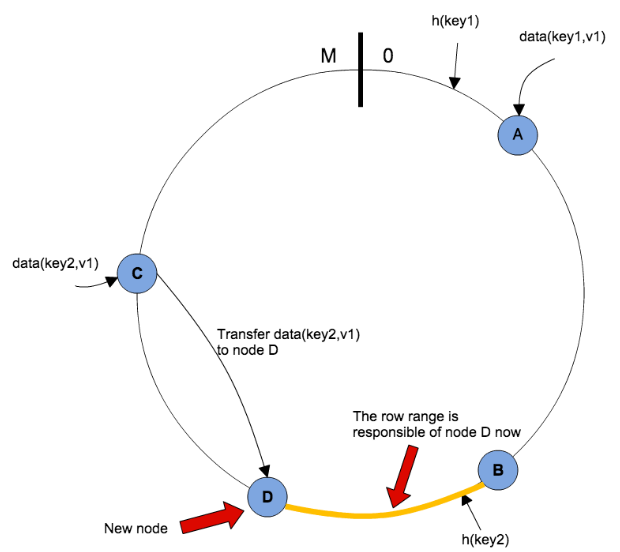
Insert new data: Replication



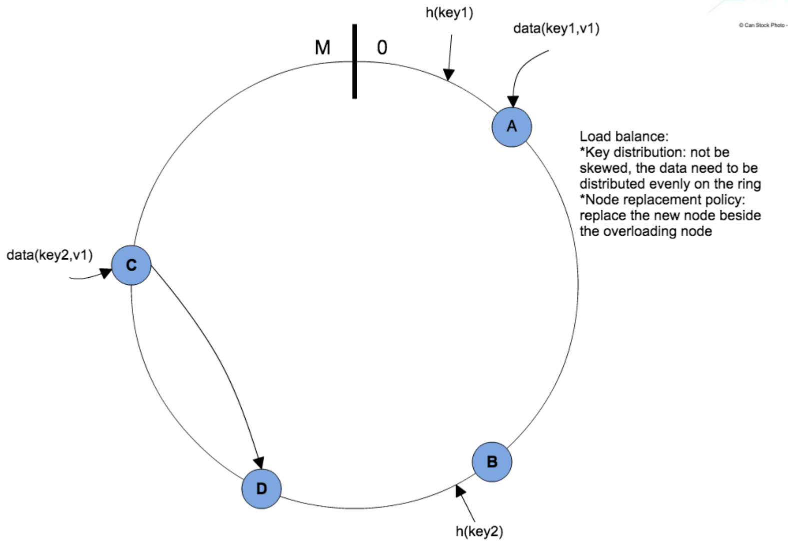
Insert New Data



Adding New Node



Load Balancing



Implementation

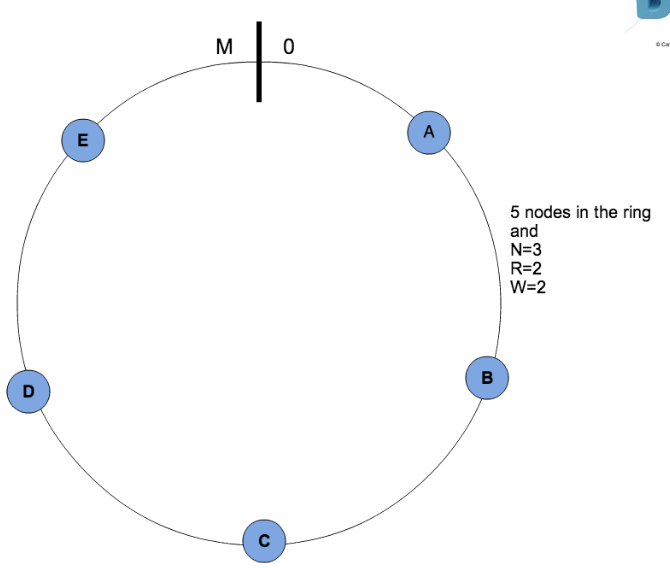
Handling temporary failures

- Sloppy Quorem
  - Availability too high will reduce durability even under the simplest failure
  - Sloppy Quorem is to control the tradeoff between availability and consistency
  - To get enough durability to handle temporary failures

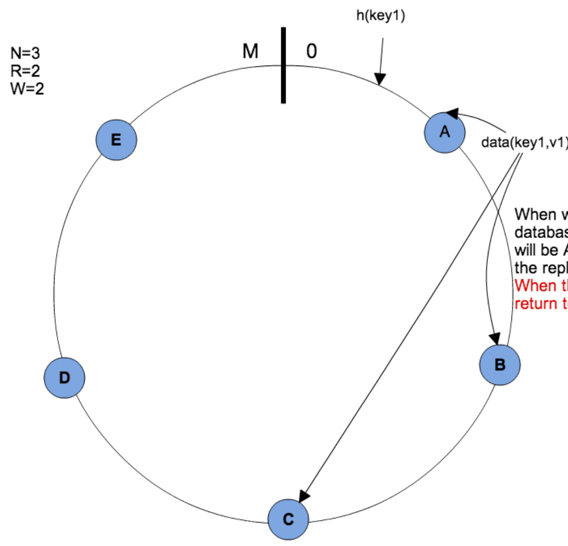
Sloppy Quorem

- Configurable N, R, W
  - **N**: number of successful copies in ideal state
  - **R**: number of successful reads nodes for successful read
  - **W**: number of successful writes nodes for successful write

Sloppy Quorem

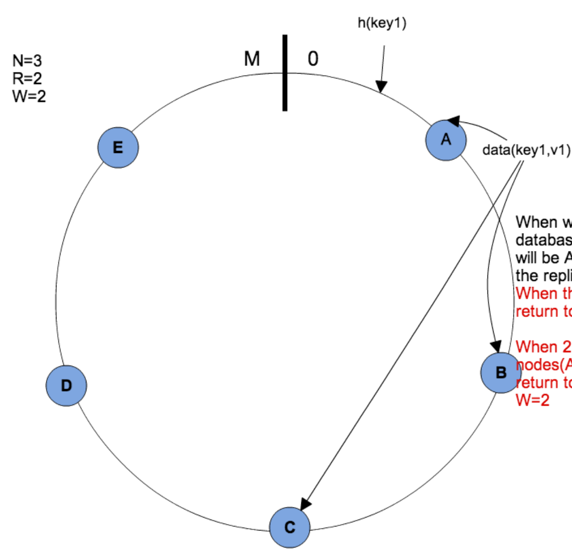


Sloppy Quorem: Write



When write data(key1,v1) in database.The write quorem will be A,B,C,which B,C are the replicas of the data
When the write operation return to the caller?

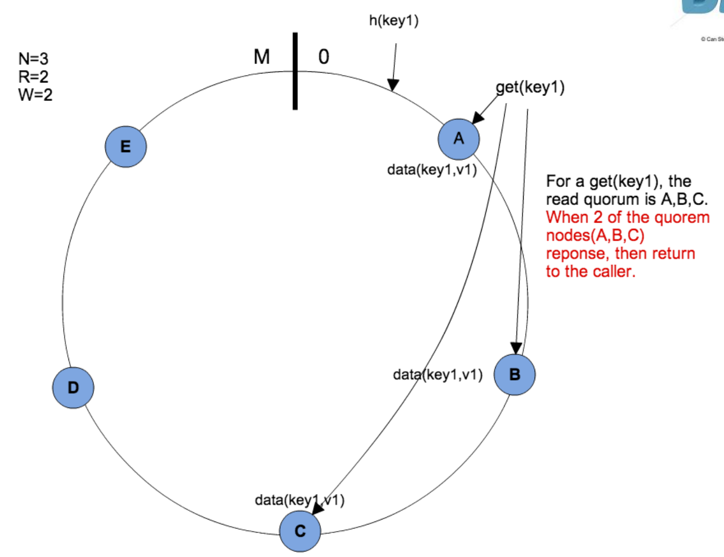
Sloppy Quorem: Write



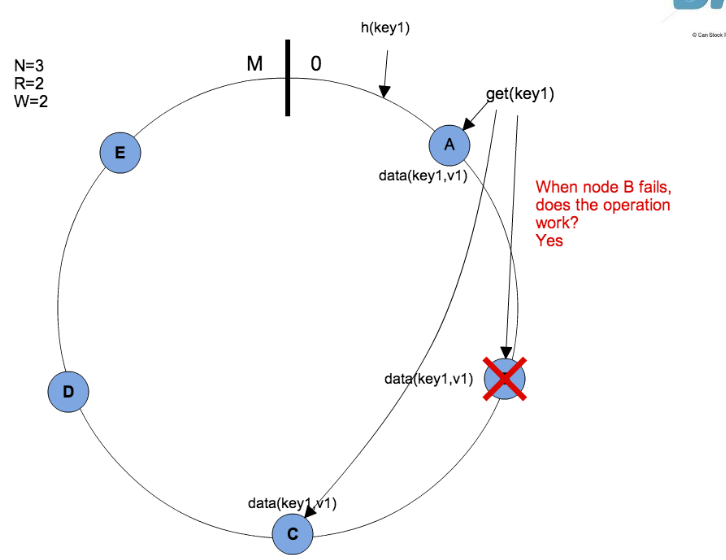
When write data(key1,v1) in database.The write quorem will be A,B,C,which B,C are the replicas of the data
When the write operation return to the caller?

When 2 of these nodes(A,B,C) response, then return to the caller, because W=2

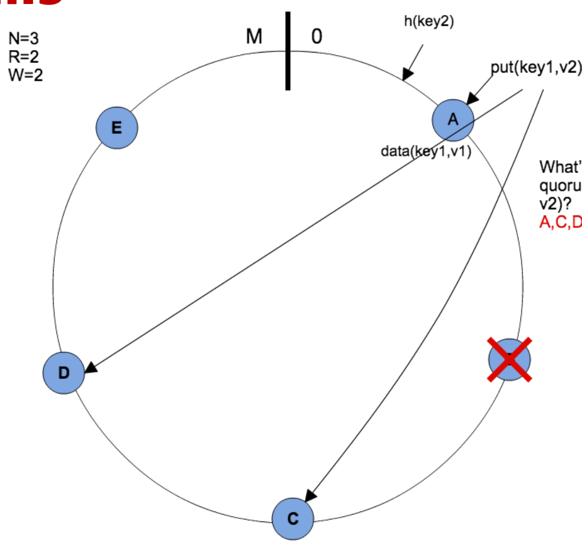
Sloppy Quorem: Read



Sloppy Quorem: Read

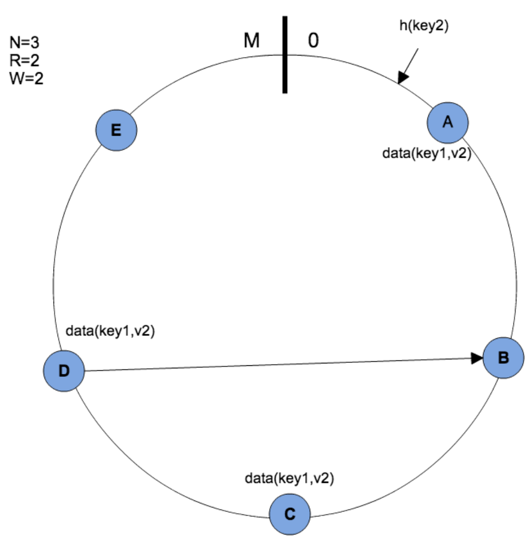


Sloppy Quorum: write after B fails



What's the write quorum for put(key1, v2)?
A,C,D

Sloppy Quorum: After B Recover



After B recovers, the quorem become A,B,C. D transfer data(key1,v2) to B as a replica

Sloppy Quorem

N	R	W	Affection
3	2	2	Typical configuration,Consistent, durable, interactive user state
n	1	n	Strong consistency while poor availability
n	1	1	High availability while weak consistency

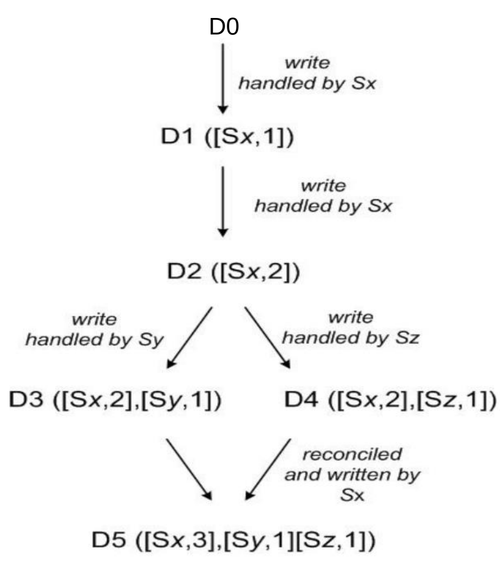
Implementation

- Data Version
  - Dynamo provides fully availibility
  - Consistency => eventually consistency
  - To guarantee eventually consistency

Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
  - **Put(key, context, object)**: context contains metadata & version
  - Each put operation is a new immutable version
- A get() call may return many versions of the same object.
  - **Get(key)**
- **Challenge**: an object having distinct version sub-histories, which the system will need to reconcile in the future.
- **Solution**: uses vector clocks in order to capture causality between different versions of the same object.

Data Versioning



Gossip

- Admin issue command to join/remove node
- Serving node records in its local membership history
- Gossip based protocol used to agree on the memberships
- Partition and Placement information sent during gossip

READ Operation

- Send read requests to nodes
- Wait for minimum no of responses (R)
- Too few replies fail within time bound
- Gather and find conflicting versions
- Create context (opaque to caller)
- Read repair

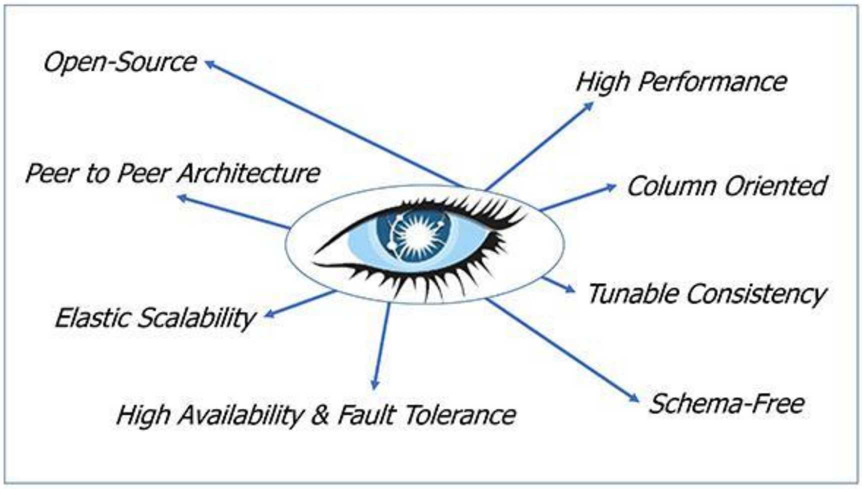
Values of N, R and W

- N represents durability
  - Typical value 3
- W and R affect durability, availability, consistency
- What if W is low?
- Durability and Availability go hand-in-hand?

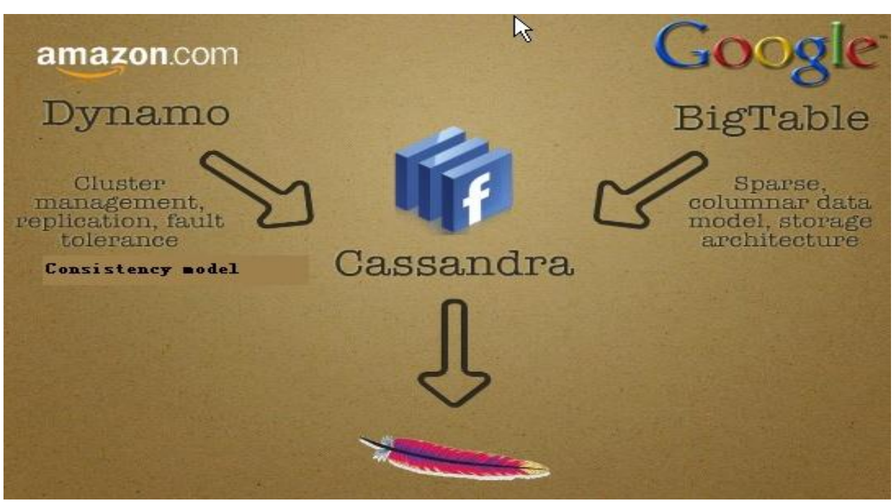
Conclusion and Influence

- Dynamo has provided high availability and fault tolerance
- Provides owners to customize according to their SLA requirements
- Decentralized techniques can provide highly available system
- Some of the principles used by S3
- Open source implementation
  - Cassandra
  - Voldemort

Apache Cassandra



A Picture is worth 1000 words



Proven

- The Facebook stores 150TB of data on 150 nodes
- Used at Twitter, Rackspace, Mahalo, Reddit, Cloudkick, Cisco, Digg, SimpleGeo, Ooyala, OpenX, others

Dynamo vs BigTable

	Dynamo	BigTable
Architecture	decentralized	centralized
Data model	key-value	sorted map
API	get, put	get, put, scan, delete
Security	no	access control
Partitioning	consistent hashing	key range based
Replication	successor nodes in the ring	chunkservers in GFS
Storage	Plug-in	SSTables in GFS
Membership and failure detection	Gossip-based protocol	Handshakes initiated by master

What is Cassandra

- A distributed data store for big data applications
- A schema free NoSQL distributed DBMS
- A hybrid between a key-value and a column-oriented data model
- High availability with no single point of failure
- Symmetric architecture to scale horizontally with automatic cluster maintenance
- Tunable consistency
- Open source

Design Goals

- High availability
- Flexible consistency
  - trade-off strong consistency in favor of high availability
- Incremental scalability
- Optimistic Replication
- “Knobs” to tune tradeoffs between consistency, durability and latency
- Low total cost of ownership
- Minimal administration

Best of Both Worlds

- From BigTable
  - Sparse , ”columnar” data model
    - Optional,2-level maps Called Super-Column Families
  - SSTable Disk Storage
    - Append-only Commit Log
    - MemTable (Buffer & Sort)
    - Immutable SSTable Files
  - Hadoop Integration
- From Dynamo
  - Symmetric,P2P architecture
    - No Special nodes, No SPOF(Single Point Of Failure)
  - Gossip Based cluster management
  - Distributed hash table for data placement
    - Pluggable partitioning
    - Pluggable topology discovery
    - Pluggable placement strategies
  - Tunable, Eventual Consistency

Data Model

- The whole cluster contains several keyspaces
- Keyspace
  - Typically, a cluster has one keyspace per application
- Data is stored as a multi dimensional map indexed by key (row key)
- ColumnFamily
  - Contains several simple columns or super columns
- SuperColumn
  - Consists of several columns
- Column
  - Described by **name**, **value**, **timestamp**

Simple column family

- *column_family : column*

keyA	column1	column2	column3
keyC	column1	column7	column11

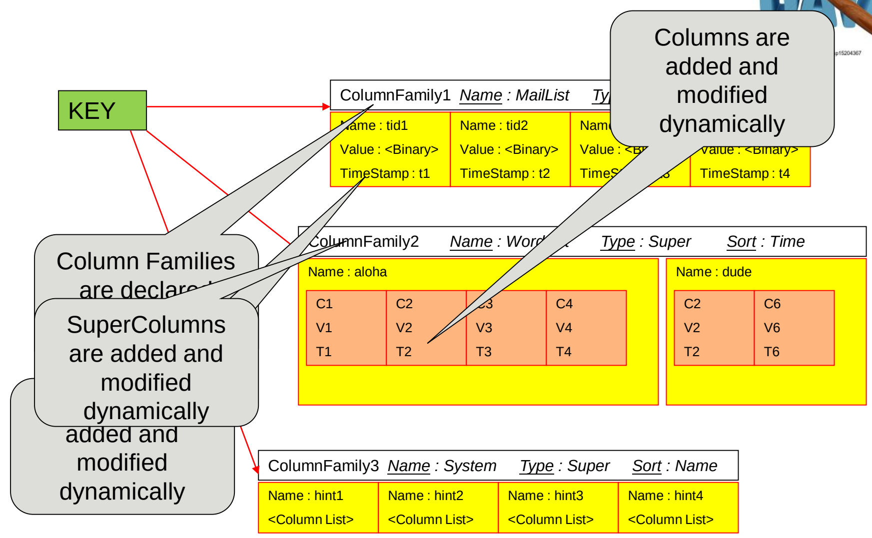
Column
Byte[] Name
Byte[] Value
I64 timestamp

Super column family

- *column_family : super_column : column*

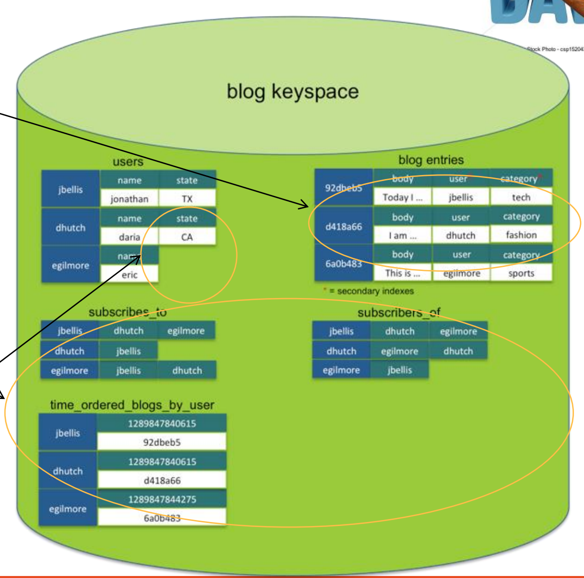
keyF	**Super1** column | column | column	**Super2** column | column | column
keyJ	**Super1** column | column | column	**Super5** column | column | column

Data Model



Data Model Example

- **Column Families**:
  - Like SQL tables
  - but may be unstructured (client-specified)
  - Can have index tables
- Hence "column-oriented databases" / "NoSQL"
  - No schemas
  - Some columns missing from some entries
  - "Not Only SQL"
  - Supports get(key) and put(key, value) operations
  - Often write-heavy workloads



Consistency Model

- Consistency level is based on replication factor N (usually 3)
- Can set read quorum R (usually 2) and write quorum W (usually 2)
- Different levels of consistency are allowed (next two slides)
- R + W > N means strong consistency

Consistency Levels - Write

Level	Description
ANY	At least one node
ONE	At least one replica node
TWO	At least two replica nodes
THREE	At least three replica nodes
QUORUM	Write to a quorum of replica nodes
LOCAL_QUORUM	Write to a quorum of the current data center as the coordinator
EACH_QUORUM	Write to quorums of all data centers
ALL	Write to all replica nodes in the cluster

Consistency Levels - Read

Level	Description
ONE	Read from the closest replica
TWO	Read from two of the closest replicas
THREE	Read from three of the closest replicas
QUORUM	Read from a quorum of replicas
LOCAL_QUORUM	Read from a quorum of the current data center as the coordinator
EACH_QUORUM	Read from quorums of all data centers
ALL	Read from all replicas in the cluster

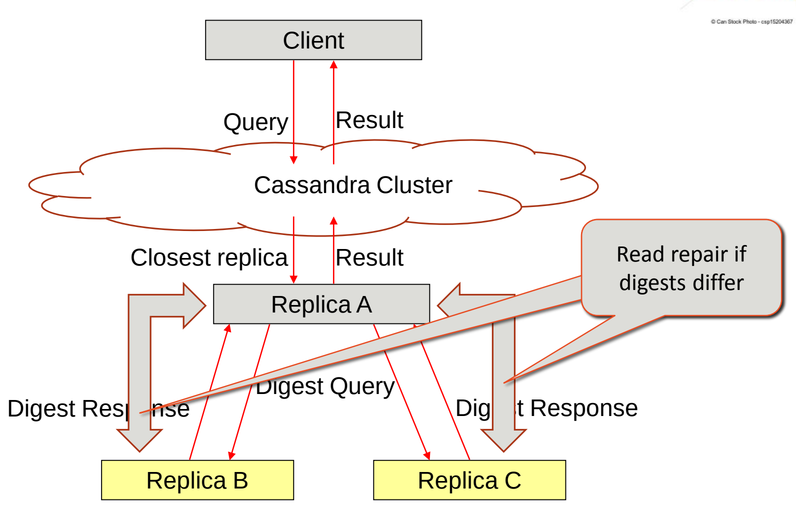
Write Operations

- A client issues a write request to a random node in the Cassandra cluster.
- The "Partitioner" determines the nodes responsible for the data.
- Locally, write operations are logged and then applied to an in-memory version.
- Commit log is stored on a dedicated disk local to the machine.

Write Properties

- No locks in the critical path
- Sequential disk access
- Behaves like a write back cache (vs write through)
- Append support without read ahead
- Atomicity guarantee for a key per replica
- “Always Writable”
  - accept writes during failure scenarios

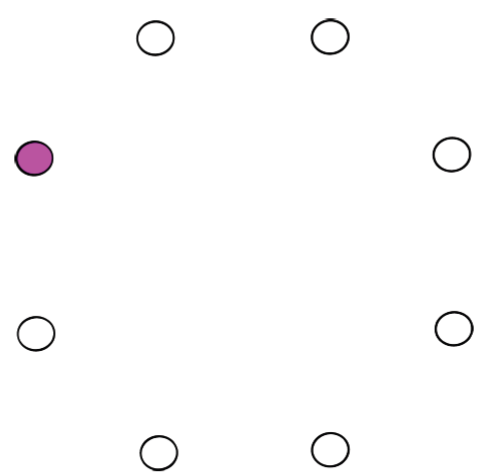
Read



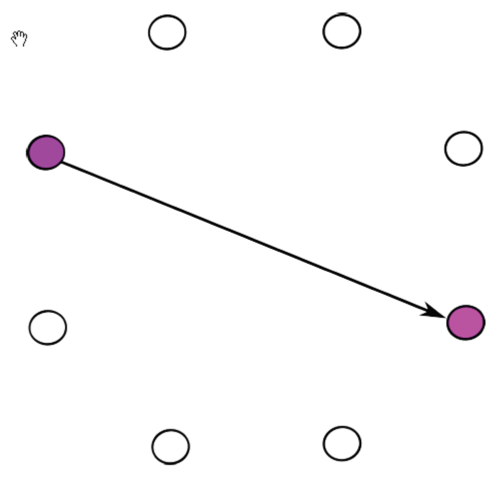
Gossip Protocols

- Network Communication protocols inspired for real life rumour spreading.
- Periodic, Pairwise, inter-node communication.
- Low frequency communication ensures low cost.
- Random selection of peers.
- Example – Node A wish to search for pattern in data
  - Round 1 – Node A searches locally and then gossips with node B.
  - Round 2 – Node A,B gossips with C and D.
  - Round 3 – Nodes A,B,C and D gossips with 4 other nodes ……
- Round by round doubling makes protocol very robust.

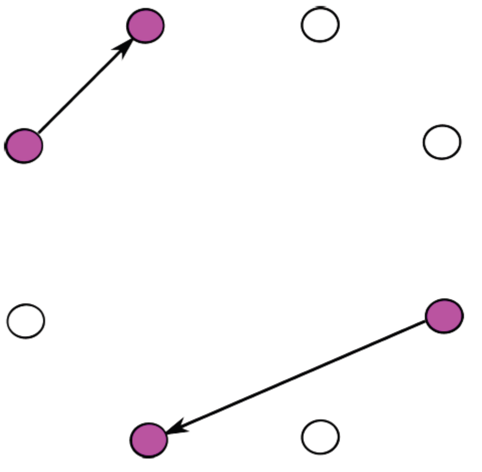
Gossip - Initial State



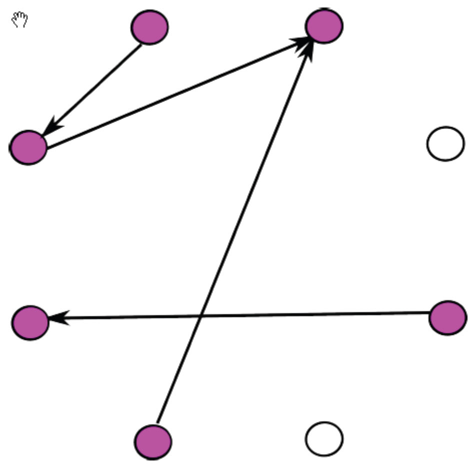
Gossip – 1st Round



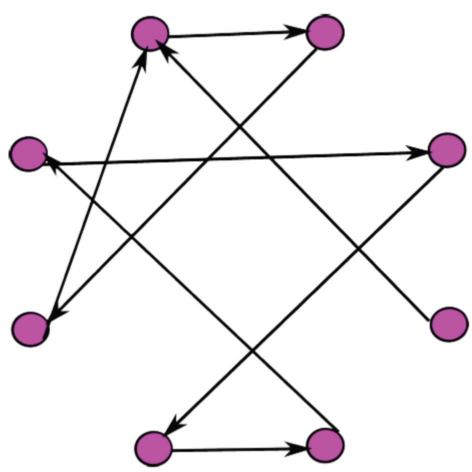
Gossip – 2nd Round



Gossip – 3rd Round

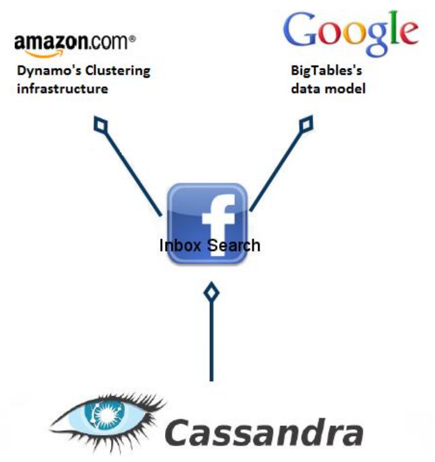


Gossip – 4th Round



Facebook Inbox Search

- Term Search
- Interactions
  a. Given the name of a person
  b. Return all messages that the user might have ever sent or received from that person



Facebook Inbox Search

- Cassandra was developed to address this problem.
- 50+TB of user messages data in 150 node cluster on which Cassandra was tested.
- Search user index of all messages in 2 ways.
  - Term search : search by a key word
  - Interactions search : search by a user id

Latency Stat	Search Interactions	Term Search
Min	7.69 ms	7.78 ms
Median	15.69 ms	18.27 ms
Max	26.13 ms	44.41 ms

Example: Term Search

- Key: User id
- Super column: Words that make up the message
- Column: Individual message identifiers of the messages that contain the word

```
|- Facebook (keyspace)
|----| UserIndexes (CF)
|-------| user_id = 119 (key)
|----------| term = meeting (super column name)
|----------------| docID = 154 => rank = 0.978 (value = standard column)
|----------------| docID = 564 => rank = 0.756
|----------------| docID = 654 => rank = 0.778
|----------| term = computer (super column name)
|----------------| docID = ...
```


Comparison with MySQL

- MySQL > 50 GB Data
  Writes Average : ~300 ms
  Reads Average : ~350 ms

- Cassandra > 50 GB Data
  Writes Average : 0.12 ms
  Reads Average : 15 ms

Why FB pick HBase?

- Cassandra's eventual consistency model
  - Wasn't a good match for their new real-time Facebook Messaging product
- 2 types of data patterns
  - A short set of temporal data that tends to be volatile
  - An ever-growing set of data that rarely gets accessed

Why FB pick HBase? (II)

- HBase
  - Has a simpler consistency model than Cassandra
  - Very good scalability and performance for their data patterns
  - HDFS, the filesystem used by HBase, supports replication, end-to-end checksums, and automatic rebalancing
  - Facebook's operational teams have a lot of experience using HDFS because Facebook is a big user of Hadoop and Hadoop uses HDFS as its distributed file system

Why FB pick HBase? (Ref.)

- The Underlying Technology of Messages (FB)
- Why HBase is a better choice than Cassandra with Hadoop? (StackOverflow)
- HBase vs Cassandra: 我們遷移系統的原因 (Blogger)
- Taking the Bait (Apache HBase)
- Oracle NoSQL Database Compared to Cassandra and HBase (PDF)

Conclusion

- There's no Holy Grail
- Add fancy features only when absolutely needed.
- Many types of failures are possible.
- Need proper systems-level monitoring.
- Value simple designs
- Analyze carefully and choose, or even design your own solution.
  - Data model
  - Consistency
  - Throughput or response time
  - Fault tolerance