Structured Big Data 2: NoSQL, NewSQL and Distributed SQL Systems

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Recap from Last Lecture

- Why can't we use **traditional RDBMS**?
  - As data scales, RDBMS cannot handle it
  - The schema from RDBMS will hinder the scalability
- Need the data model with loosen schema and high scalability

  → **NoSQL**
- Want the best of both worlds

  → **NewSQL**

SQL vs NoSQL vs NewSQL

SQL:

- Relies solely on relational tables for storing and accessing transactional data.
- Relies on basic SQL as its primary query language.
- Employs rigid and well-defined data schema.
- Minimizes redundancies via normalization.
- Utilizes traditional **vertical** scalability (up, not out).
- Popular SQL Databases: Microsoft SQL Server, MySQL, Oracle Database, IBM Db2, Informix, MariaDB, PostgreSQL, SQLite, ...

SQL vs NoSQL vs NewSQL

NoSQL:

- Relies on different models, such as key-value, document, wide-column, or graph.
- Relies on high performance writes and huge, **horizontal** scalability for big data.
- Does not rely on a defined schema for writing data.
- Supports a large variety of modern programming languages, tools, and applications.
- Lacks strong consistency (instead, relies on a default "eventual consistency" for higher availability).
- Popular NoSQL Databases: HBase, Cassandra, Amazon DynamoDB, Couchbase Server, CouchDB, MongoDB, Oracle NoSQL, Redis, ...

SQL vs NoSQL vs NewSQL

NewSQL:

- Combines relational model of SQL databases with the versatile scalability and speed of NoSQL databases.
- Uses cluster-native and shared-nothing architecture to provide low latency, high read/write performance.
- Favors consistency over availability (though configurations can be tuned for better balance).
- Variety in schema management, depending on the vendor.
- Popular NewSQL Databases: Apache Trafodion, Altibuse, ClusterixDB, MemSQL, VoltDB, NuoDB, TIBCO ActiveSpaces, ….

SQL vs NoSQL vs NewSQL

Functions	RDBMS	NoSQL	NewSQL
SQL	Supported	Not supported	Supported
Machine dependency	Singe machine	Multi- machine/Distributed	Multi- machine/Distributed
DBMS type	Relational	Non- relational	Relational
Schema	Table	Key-value, column store, document store	Both
Storage	On disk + cache	On disk + cache	On disk + cache
Properties support	ACID	CAP through BASE	ACID
Horizontal scalability	Not supported	Supported	Supported
Query Complexity	Low	High	Very High
Security concern	Very high	Low	Low
Big volume	Less performance	Fully supported	Fully supported
OLTP	Not fully supported	Supported	Fully supported
Cloud support	Not fully supported	Supported	Fully supported

Systems to be discussed

- Deep dive into some NoSQL & NewSQL databases
- **NoSQL systems** to be discussed:
  - DynamoDB
  - Cassandra
  - MongoDB
- **NewSQL systems** to be discussed:
  - VoltDB
  - NuoDB (if time)
  - ClustrixDB (if time)
  - Vitess (if time)

What do we REALLY want?

- What's wrong with SQL, NoSQL and NewSQL?
- Back to basics: Re-examine the fundamental requirements.
- What we REALLY want is Distributed SQL!!
- Google Spanner, the first of its kind
- Distributed SQL DBs:
  - CockroachDB
  - YugabyteDB
  - SkySQL
  - Google F1

NoSQL Databases

NoSQL: The Name

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

  Not every data management/analysis problem is best solved using a traditional RDBMS
- "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 due to conceptual difficulty mapping between the two (relational & OO) logic models.
- 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

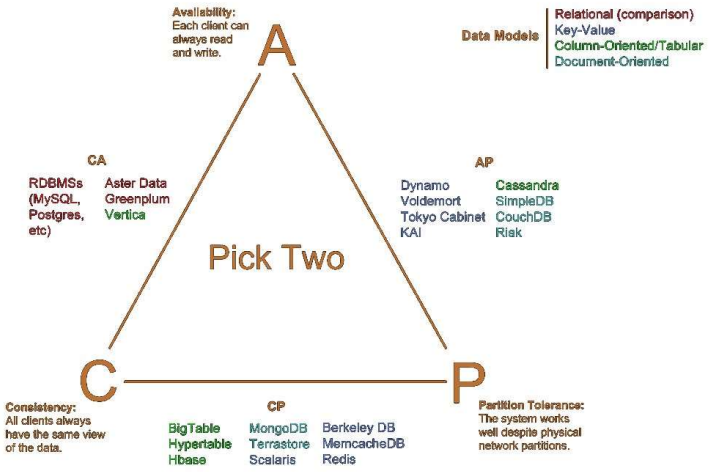
How did we get here?

- 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

No SQL!

Highly Parallelizable

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

SQL-like

Consistency

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

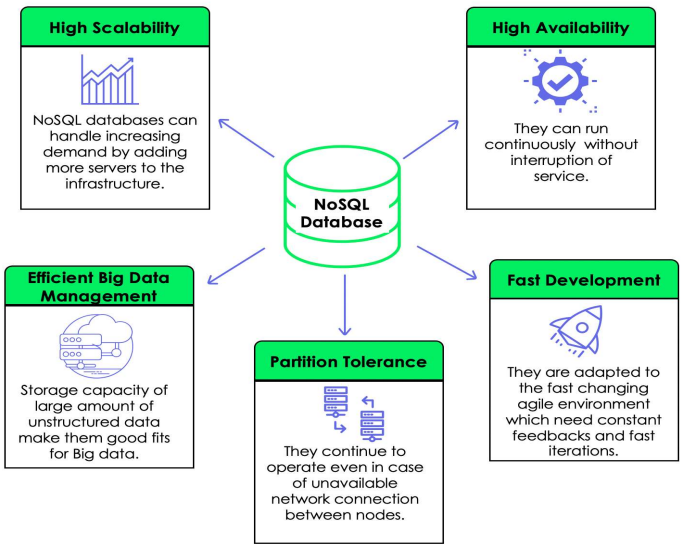
Why NoSQL?

Example #3: Wikipedia pages

Large collection of documents
Combination of structured and unstructured data

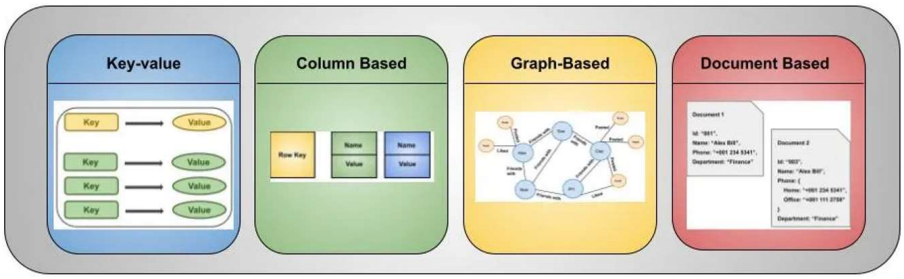
Task: Retrieve introductory paragraph of all pages about U.S. presidents before 2015

Key Features of NoSQL



4 Types of NoSQL DBs

- There are **4** basic types of NoSQL DBs.
- We will discuss the key-value, column based and document based DBs.
- Graph DBs will be discussed in next lecture.



7 Types of NoSQL DBs

- Another **3** types are added later

Popular NoSQL DBs

- Some of the most popular NoSQL DBs (all except DynamoDB are open-source)

Dynamo: Outline

- Background & motivation
- System internals
- 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.
- Mostafa Elhemali, et. al. "**Amazon DynamoDB**: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service", in *Proceedings of the 2022 USENIX Annual Technical Conference*, Carlsbad, CA, USA, July 2022.
- Joseph Idziorek, et. Al. "Distributed Transactions at Scale in **Amazon DynamoDB**", in *Proceedings of the 2023 USENIX Annual Technical Conference*, Boston, MA, USA, July 2023.

Amazon DynamoDB

DynamoDB
Core Components

amazon
web services™

Background

- Amazon's eCommerence 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
- Need high availability and SLA(Service Level Agreements) guarantee

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 rigid 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)

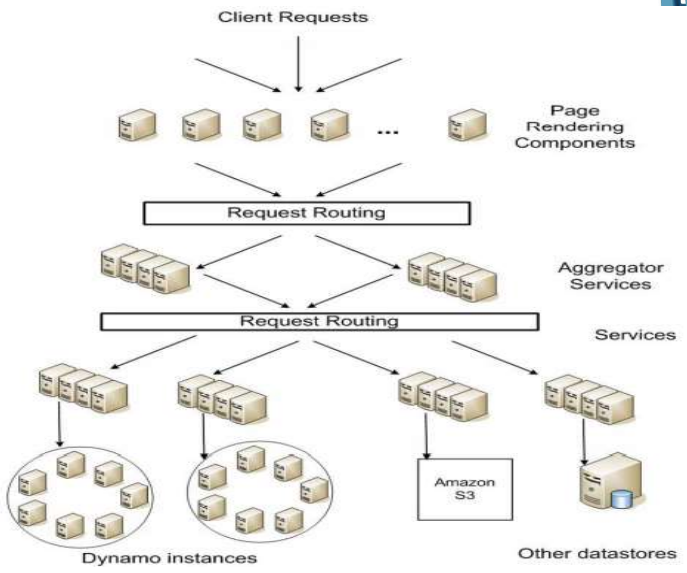
Service Level Agreements

- **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



Design Consideration

- Sacrifice strong consistency for availability
- Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".
- Other principles:
  - Incremental scalability.
  - Symmetry.
  - Decentralization.
  - Heterogeneity.

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

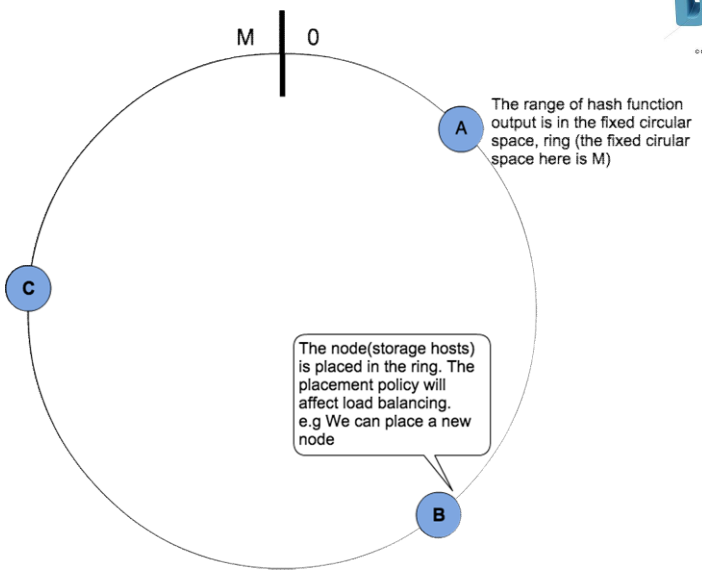
Implementation

- 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

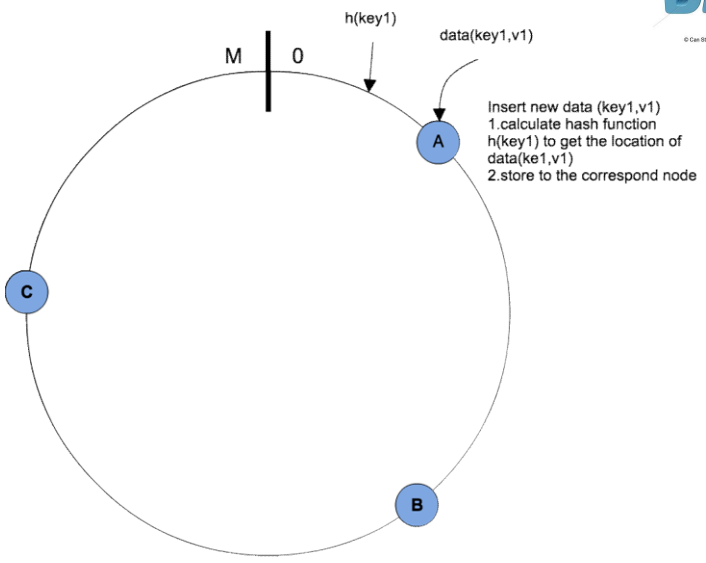
Basic Consistent Hashing

- 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

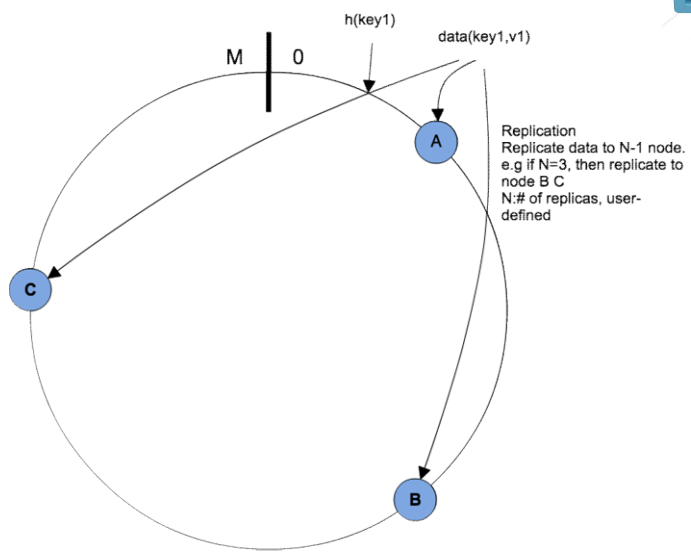
Partition: Consistent



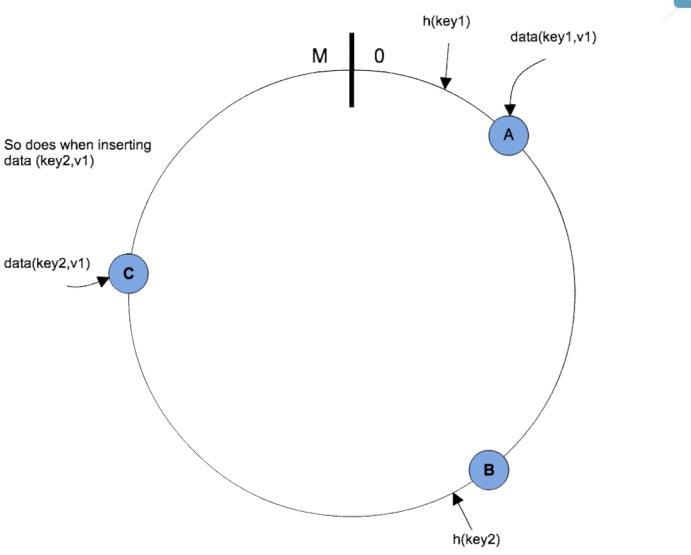
Insert New Data



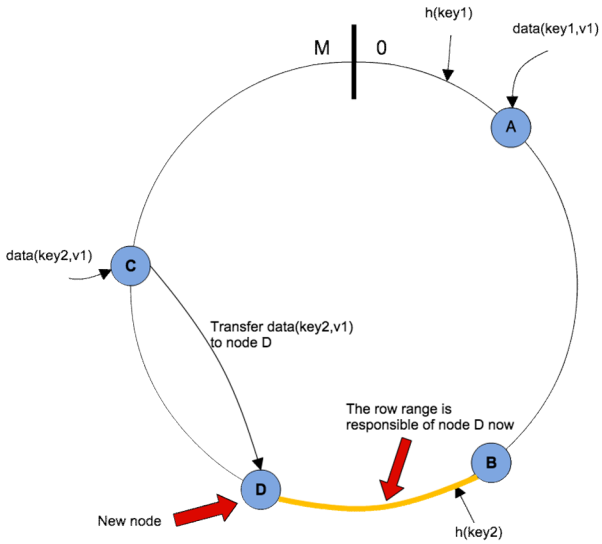
Insert new data: Replication



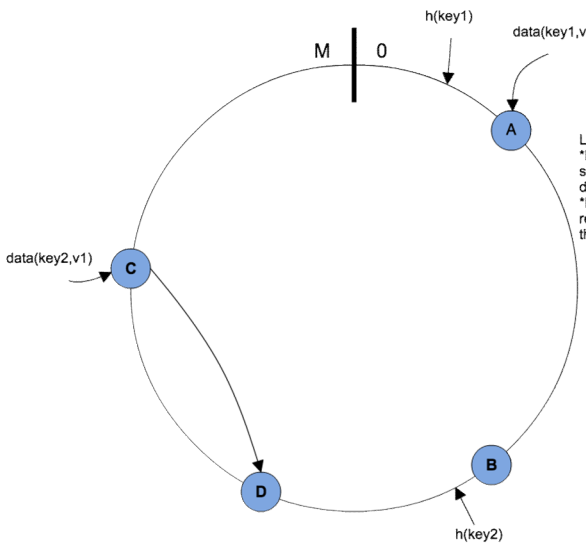
Insert New Data



Adding New Node



Load Balancing



Implementation

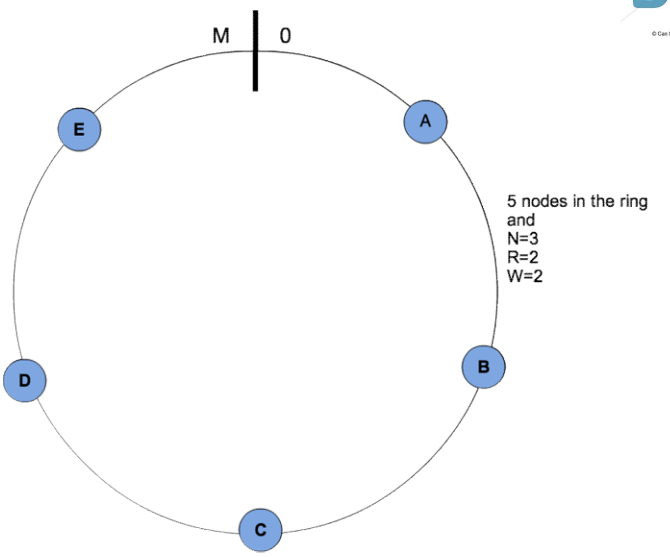
Handling temporary failures

- 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

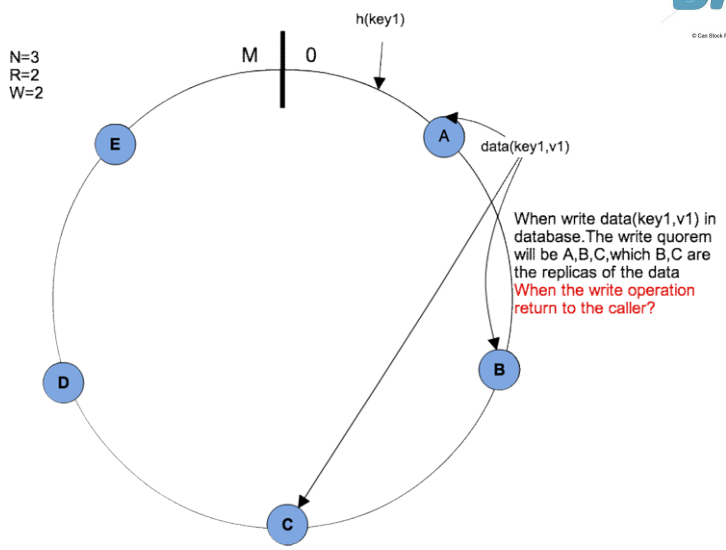
Sloppy Quorem

- R/W is the minimum number of nodes that must participate in a successful read/write operation.
- 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
- Setting R + W > N yields a quorum-like system.

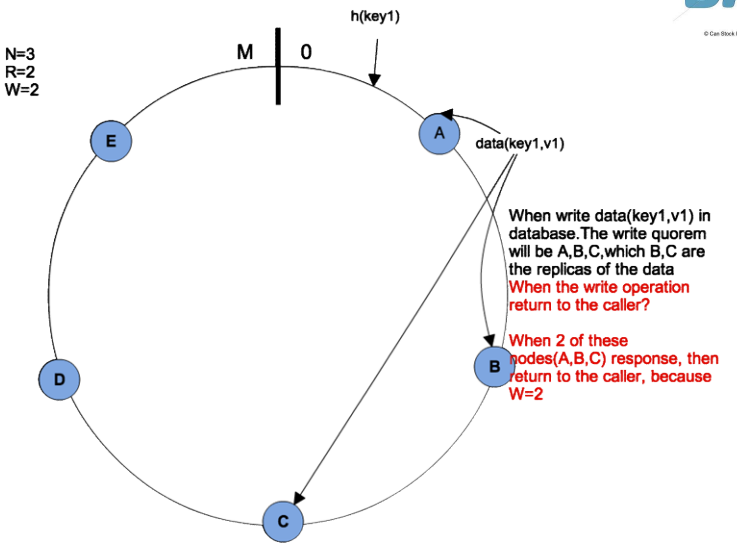
Sloppy Quorem



Sloppy Quorem: Write

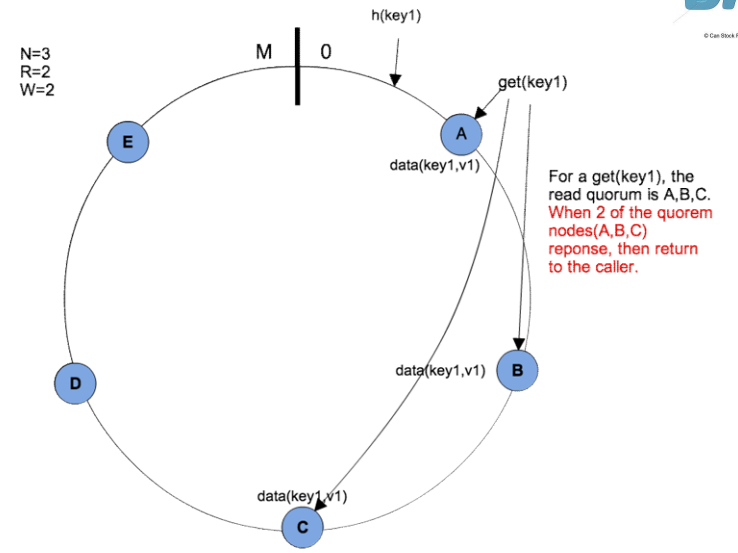


Sloppy Quorem: Write



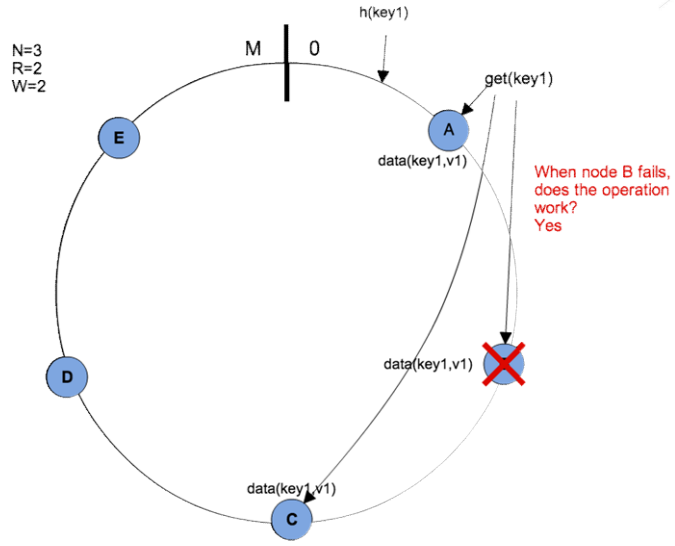
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems Structured Big Data 2 – NoSQL, NewSQL & Distributed SQL 51

Sloppy Quorem: Read

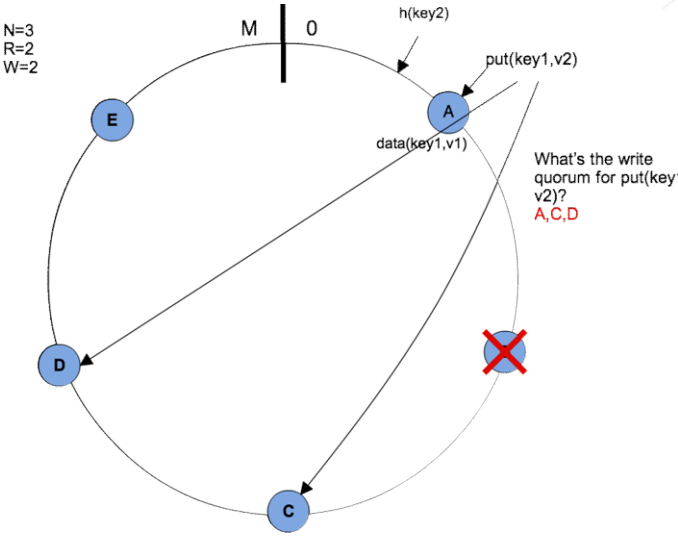


CSIE59830/CSIEM0410/AIIA50050 Big Data Systems Structured Big Data 2 – NoSQL, NewSQL & Distributed SQL 52

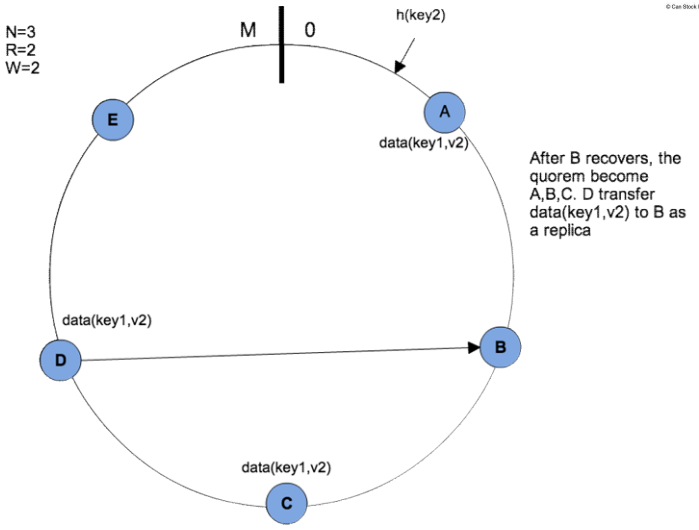
Sloppy Quorem: Read



Sloppy Quorum: write after B fails



Sloppy Quorum: After B Recover



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

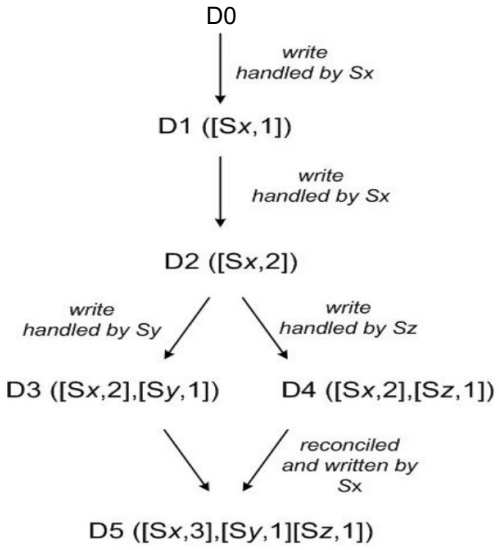
Data Versioning

- 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.

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.*

Data Versioning with Vector Clock



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?

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

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

Amazon DynamoDB

- A fully managed NoSQL cloud database service motivated by Dynamo.
- Multi-tenant architecture
- Boundless scale for tables
- Provide predictable performance
- Highly available (99.99 for regular table, 99.999 for global tables)
- Flexible use cases (doesn't require a particular data model or consistency model, no fixed schema, …)

DynamoDB Data Model

- A DynamoDB table is a collection of items.
- Each item is a collection of attributes.
- Primary key is specified at table creation time and contains a partition key or a partition and sort keys (a composite primary key).
- Item is stored/located based on hashing partition key followed by sort key (if present).
- A table can have secondary indexes.
- Support ACID transactions.

DynamoDB CRUD APIs

- The primary CRUD operations for reading/writing items in DynamoDB table:

Operation	Description
`PutItem`	Inserts a new item, or replaces an old item with a new item.
`UpdateItem`	Updates an existing item, or adds a new item to the table if it doesn't already exist.
`DeleteItem`	The DeleteItem operation deletes a single item from the table by the primary key.
`GetItem`	The GetItem operation returns a set of attributes for the item with the given primary key.

Table 1: DynamoDB CRUD APIs for items

DynamoDB Tables

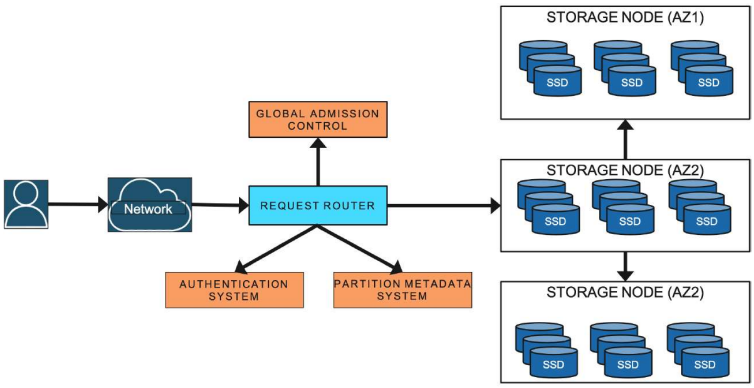
- A table is divided into multiple partitions.
- Each partition hosts a disjoint and contiguous part of the key-range.
- Each partition has multiple replicas across different Availability Zones and form a replication group.
- Use Multi-Paxos for leader election and consensus.
- Only the leader can serve write and strongly consistent read requests.
- Upon a write, the leader generates a write-ahead log record and sends it to its peers (other replicas).
- A write is acknowledged once a quorum of peers persists the log record to their write-ahead logs.

DynamDB Replication

- Supports strongly and eventually consistent reads.
- Any replica can serve eventually consistent reads.
- Leader extends leadership by a lease mechanism.
- On leader failure, any peer can start an election.
- A replication group consists of storage replicas to keep the write-ahead logs and the B-tree for the key-value data. (next page)
- Can also contains log replicas with recent write-ahead logs only. (next page)

DynamoDB Architecture

- DynamoDB consists of tens of microservices such as: metadata service, request routing service, the storage nodes, and the autoadmin service.



DynamoDB Services

- Metada service: stores routing info about tables, indexes, and replication groups.
- Request routing service: for authorizing, authenticating, and routing requests to servers.
- Request routers: look up routing info from the metada service for request routing.
- Autoadmin service: All resource creation, update, and data definition requests are routed to the autoadmin service.
- Storage service: storing customer data on storage nodes which host replicas of partitions.

DynamDB Replication

- Storage replica

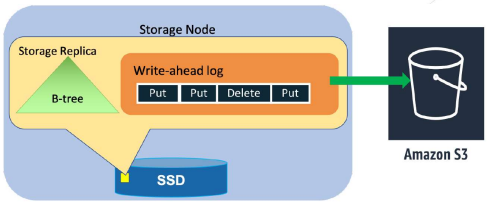


Figure 2: Storage replica on a storage node

- Log replica

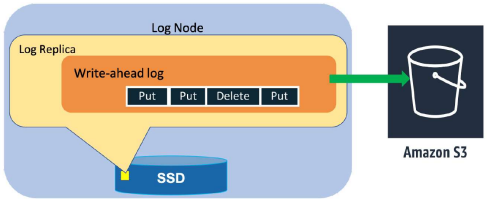


Figure 3: Log replica on a log node

DynamoDB Transasctions 0

- Distributed transactions with full ACID were added to Amazon DynamoDB using a timestamp ordering protocol.
- Still able to exploit the semantics of a key-value store to achieve low latency for both transactional and non-transactional operations.
- On Prime Day 2022, DynamoDB handles trillions (10^{12}) API calls with high availability, single-digit millisecond responses and 105.2 million requests/s
- High scalability, high availability, and predictable performance at scals.

DynamoDB Transasctions 1

- Transactions are submitted as single request and either succeed or fail w/o blocking.
- Transactions rely on a transaction coordinator while non-transaction operations bypass the two-phase coordination.
- Transactions update items in place. No multi-version. Read-only and read-write might conflict.
- Transactions do not acquire locks. An optimistic concurrency control scheme is used to avoid locking altogether.

DynamoDB Transasctions 2

- Transactions are serially ordered using timestamps. Timestamp ordering is extended to accommodate and exploit the semantics of a key-value store.

Operation	Description
`TransactGetItems`	Reads a set of items from a consistent snapshot and returns their values
`TransactWriteItems`	Performs a set of writes that include PutItem, UpdateItem, and DeleteItem operations and optionally a set of conditions
`CheckItem`	Checks that the latest value of an item matches the condition

Table 2: DynamoDB Transaction APIs

DynamoDB Transactions - Architecture

- The architecture for transaction processing is done by adding a fleet of Transaction Coordinators. Any one can take responsibility for any transaction.

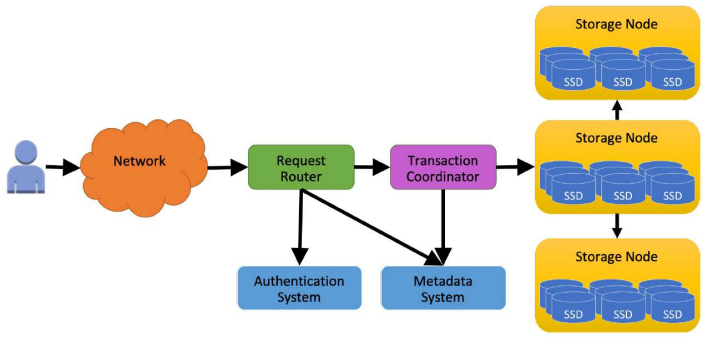


Figure 1: DynamoDB Transactions high-level architecture

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems Structured Big Data 2 – NoSQL, NewSQL & Distributed SQL 77

DynamoDB Transactions – Two-phase Protocol

- A two-phase protocol ensures that all of the writes within a transaction are performed atomically and in the proper order.

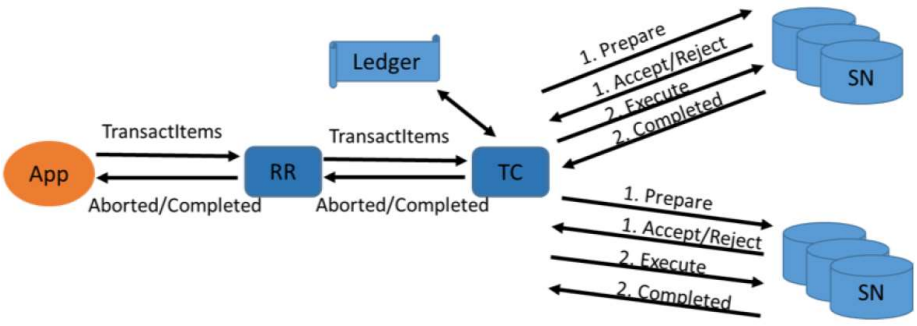


Figure 2: Two-phase protocol

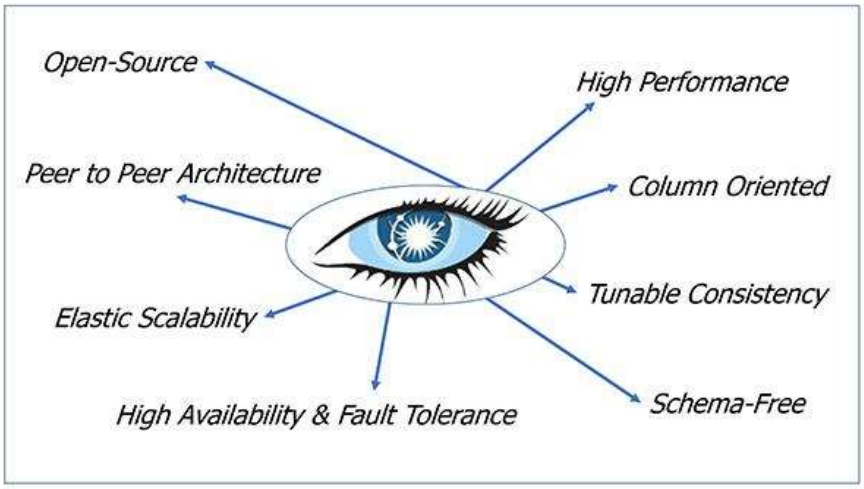
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems Structured Big Data 2 – NoSQL, NewSQL & Distributed SQL 78

Use Cases of DynamoDB

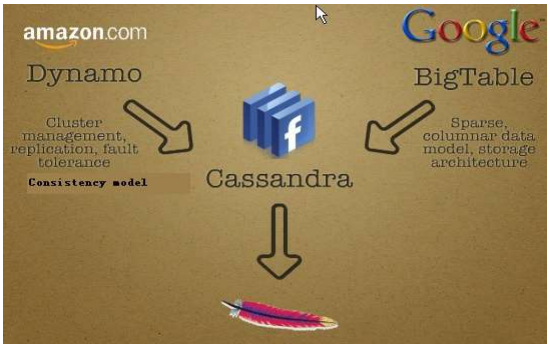
- **Duolilngo** – an online learning site with 31 billion data objects and ~ 18 million monthly users.
- **Major Leagure Baseball (MLB)** – MLB uses a combination of AWS components among which DynamoDB plays a key role.
- **Hess Corporation** – a well-known energy company. DynamoDB helps in separating potential buyers' data freom business systems.
- **GE Healthcare** – well-known for medical imaging equipment. Use DynamoDB to increase customer value.
- **NTT Docomo** – a popular mobile phone operating company. Use DynamoDB for voice recognition services and marketing data management.

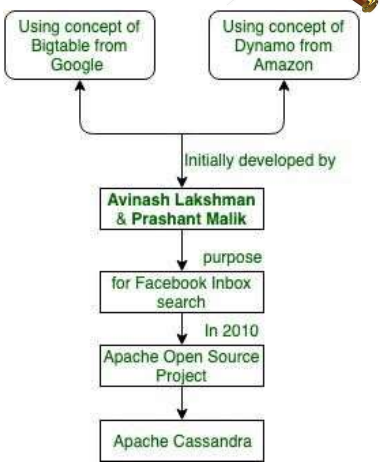
Apache Cassandra

Apache Cassandra



Best of Both Worlds





Proven

- The Facebook stores 150TB of data on 150 nodes
- Used at Twitter, Rackspace, Mahalo, Reddit, Cloudkick, Cisco, Digg, SimpleGeo, Ooyala, OpenX, others.
- At **Netflix**, Cassandra runs 30 million ops/s on its most active single cluster and 98% of streaming data is stored on Cassandra.
- **Apple** runs 160,000+ Cassandra instances with thousands of clusters.

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
- Latest version: Apache Cassandra 5.0 (2023-11-04)

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

Features of Cassandra

- Big data ready
- Easy data distribution
- Flexible data storage
- Elastic scalability with fast linear-scale performance
- Good read-write performance with fast writes
- Highest availability with always on architecture
- Transaction support
- Self-healing and automation
- Geographical distribution
- Platform agnostic
- Vendor independent

Cassandra and CAP

- Cassandra is usually described as an "AP" system under the CAP theorem.
- Cassandra is configurably consistent : You can set the Consistency Level you need and tune it to be more AP or CP according to your use case.

Design Goals 1

- 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

Design Goals 2

- Full multi-master database replication
- Global availability at low latency
- Scaling out on commodity hardware
- Linear throughput increase with each additional processor
- Online load balancing and cluster growth
- Partitioned key-oriented queries
- Flexible schema

Data Model

- The whole cluster contains several keyspaces
- Keyspace (Database) – Typically, a cluster has one keyspace per application
- Data is stored as a multi dimensional map indexed by key (row key)
- Column Families (Tables) – Contains several simple columns or super columns
- Super Column – Consists of several columns
- Column – Described by **name**, **value**, **timestamp**
- Row is a unit of replication.
- Column is a unit of storage.

Keyspace

- Keyspace is the outermost container for data.
- Basic attributes of a Keyspace:
  - Replication facto
  - Replica placement strategy
  - Column families
- A keysapce contains one or more column families.
- A column family contains a collection of rows.
- Each row contains ordered columns.

Creating Keyspace

- The syntax of creating a Keyspace

```
CREATE KEYSPACE Keyspace name
WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 3};
```

- Replica placement strategies
  - Simple strategy (rack-unaware strategy)
  - Ole network topology strategy (rack-aware strategy)
  - Network topology strategy (rack-aware strategy)

Column Family

- *column_family : column*

keyA	column1	column2	column3
keyC	column1	column7	column11

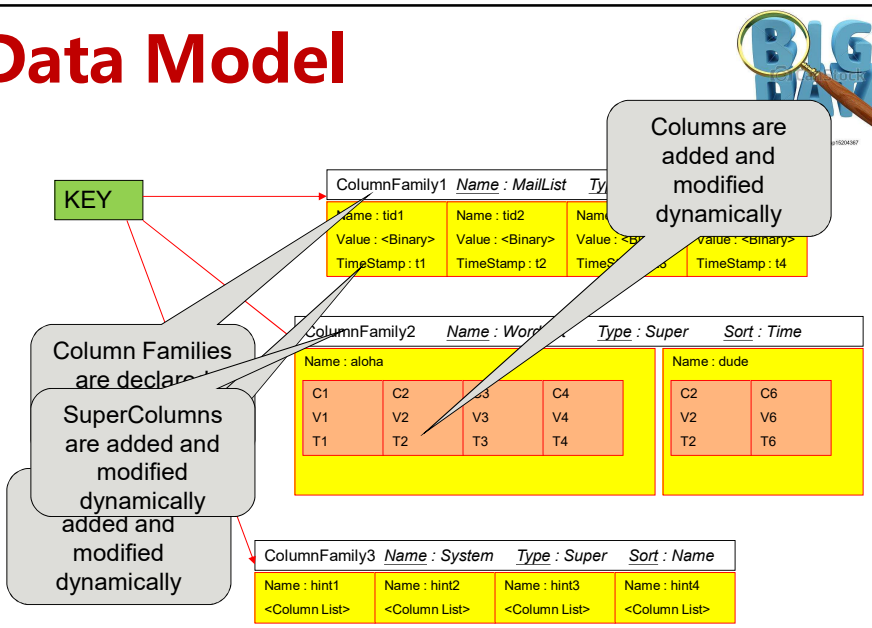
Column
Byte[] Name
Byte[] Value
I64 timestamp

Super column family

- *column_family : super_column : column*

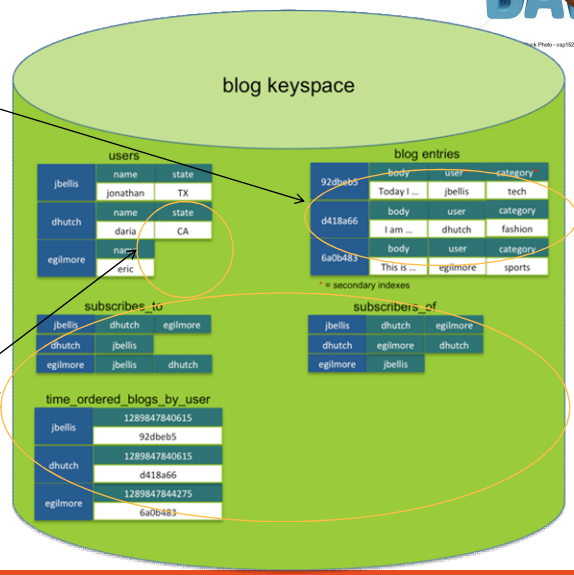
keyF	**Super1**			**Super2**		
	column	column	column	column	column	column
keyJ	**Super1**			**Super5**		
	column	column	column	column	column	column

Data Model



Data Model Example

- **Column Families**:
  - Like SQL tables
  - but may be unstructured (client-specified)
  - Can have index tables
- "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 indicates how recent and in-sync all replicas of a row of data are.
- 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

Strong Consistency Levels

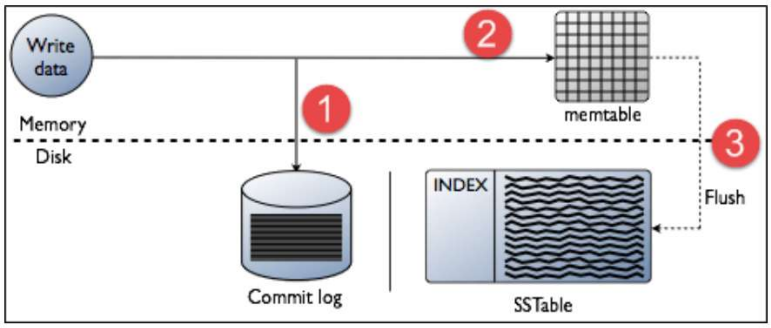
- **Write CL = QUORUM and Read CL = QUORUM**
  - If *RF* = 3, *W = QUORUM* or *LOCAL_QUORUM, R = QUORUM* or *LOCAL_QUORUM*, then *W (2) + R (2) > RF (3)*
- **Write CL = ALL and Read CL = ONE**
  - If *RF* = 3, *W = ALL, R = ONE*, then *W (3) + R (1) > RF (3)*
- Consistency level can be configured based on application needs.

Write Operations

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

Write Op in Cassandra

- A replica node responds with success if data is written successfully to commit log and memTable.



Write Operation in Cassandra

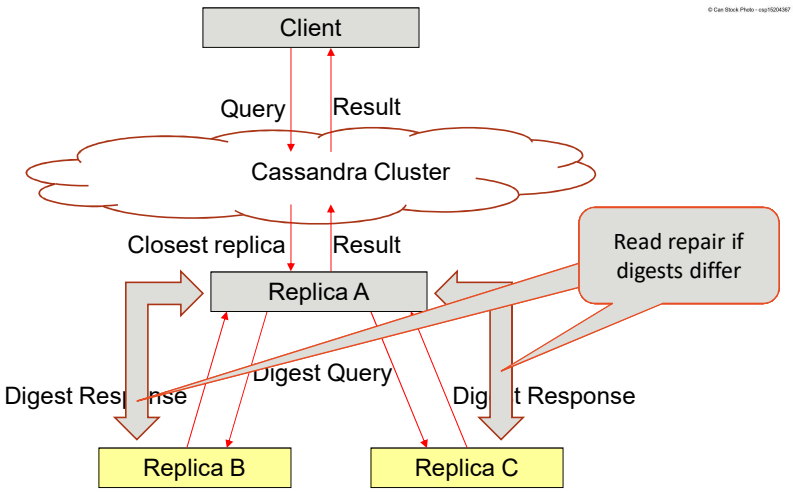
Write Properties

- 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 Operations

- Three types of read requests (next slide)
  - Direct request
  - Digest request
  - Read repair request
- Coordinator sends direct request to one replica. Then sends digest requests to # replicas specified by the consistency level.
- After that, sends digest requests to all remaining replicas.
- A background read repair request is send to each outdated replica.

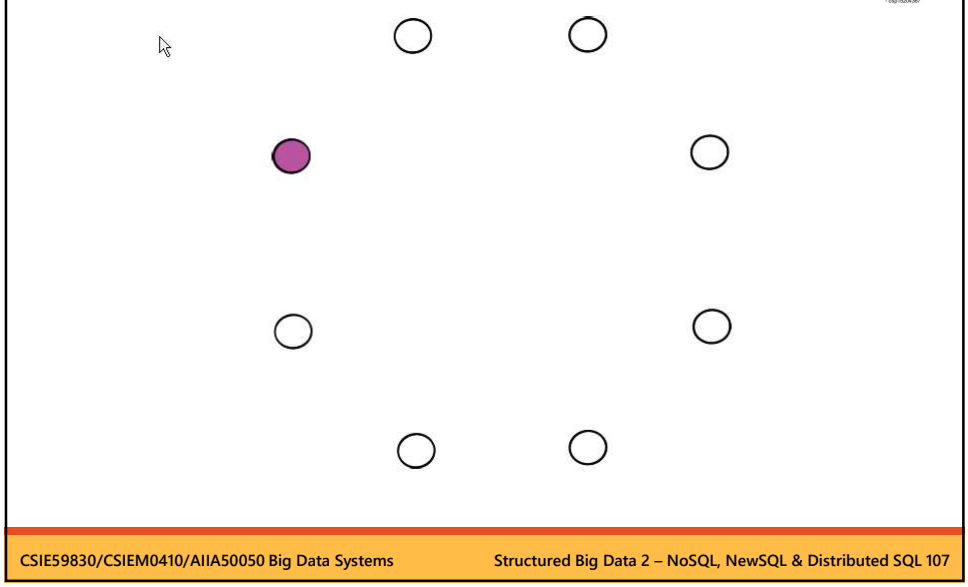
Read



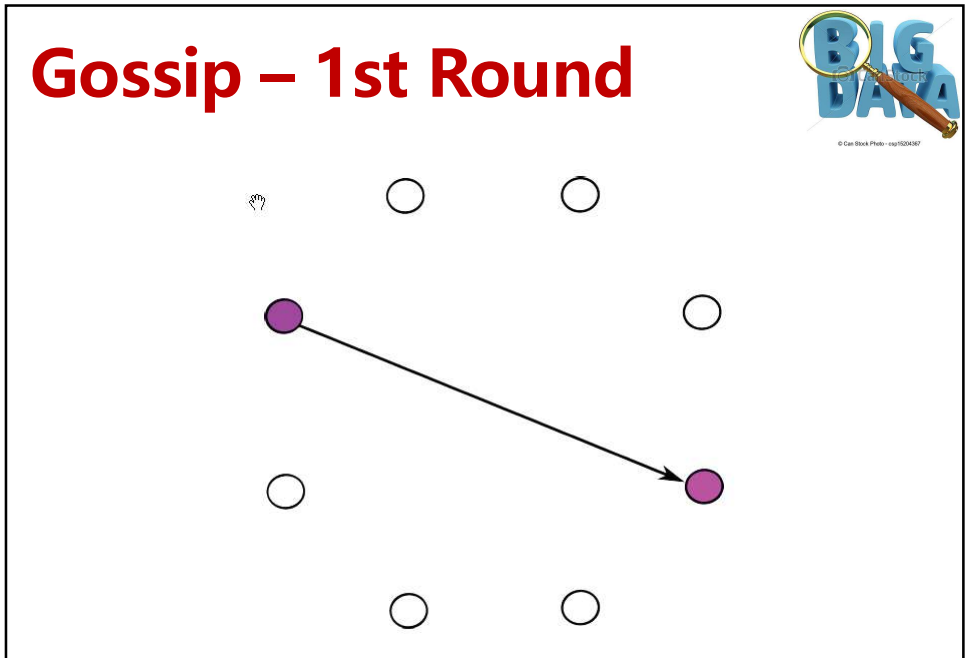
Gossip Protocols

- 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.

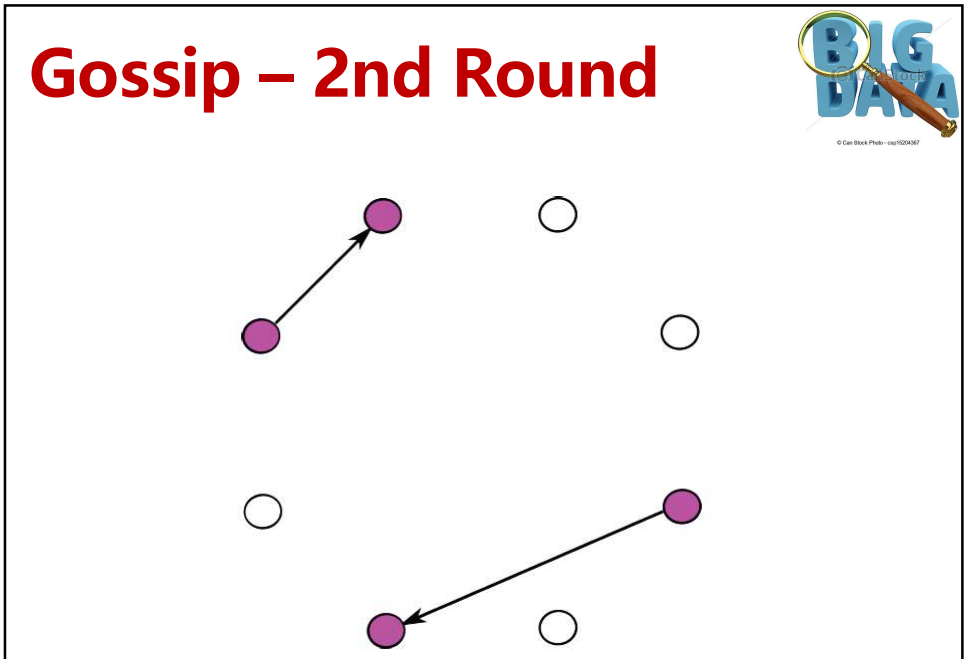
Gossip - Initial State



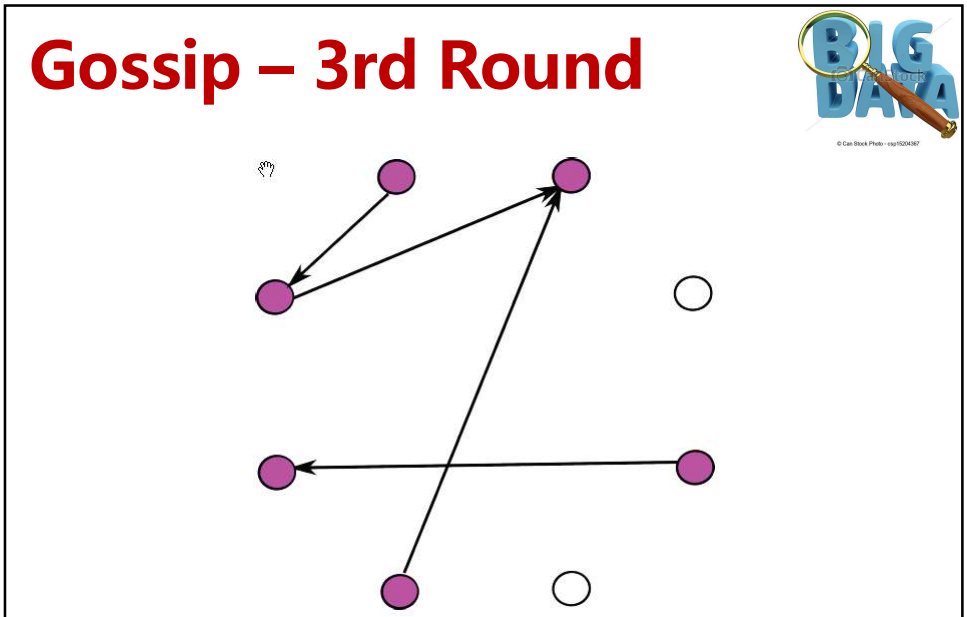
Gossip – 1st Round



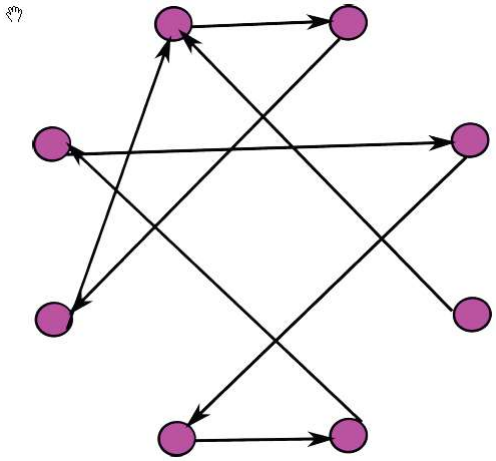
Gossip – 2nd Round



Gossip – 3rd Round

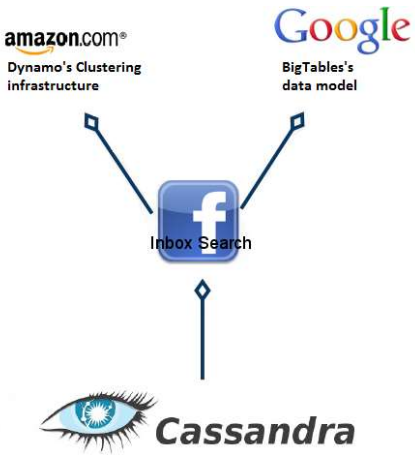


Gossip – 4th Round



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?

- At 2010, FB chose HBase instead of Cassandra for their new Real-Time Messaging System to store 135+ billion messages a month.
- Cassandra's eventual consistency model
  - Wasn't a good match for their 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(filesystem of 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
- But Cassandra has been improved significantly over the years.

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)

Dynamo vs Bigtable vs Cassandra

Table II : Comparison of Dynamo, Bigtable and Cassandra

	Dynamo	Bigtable	Cassandra
Data model	Key-value, row store	Column store	Column store similar to Bigtable
API	Single tuple	Single tuple and range	Single tuple and range
Data partition	Random	Ordered	Random and ordered
Optimized for	Writes	Writes	Writes
Consistency	Eventual	Atomic	Tunable consistency level
Multiple versions	Version	Time stamp	Time stamp
Persistency	Local and pluggable	Replicated and distributed file system	Replicated and distributed file system
Architecture	Decentralized	Hierarchical	Decentralized
Concurrency control	Multi version concurrency control	Locks and time stamps	Multi version concurrency control
Client library	Yes	Yes	Yes
Data storage	Plug-in	Google File system	Disk
Replication	Asynchronous	Synchronous & asynchronous	Asynchronous

Yahoo PNUTS/Sherpa

- Yahoo's hosted data serving platform
- Motivation: designed for web apps
  - Complex data management
    - ex: comments
  - Response time is very important
  - Focus on scalability and availability
    - Downtime ⇔ Money loss
- Key-Value based data model, schema free
- Geographically distributed
- Relaxed consistency model

Examples

- User database
  - Different users can be mastered in different data centers
  - User himself wants to see the update immediately with low latency, while others might not.
- Social Applications
  - Others don't have to immediately know you're in a relationship with someone
- Session Data
  - Only needs to be preserved locally, no need to be synced world-wide
  - Even if corrupted, just re-login :P

Geographical Distribution

- Web audience is worldwide
- Benefit for response time
  - Think about CDN(Content Distribution Network)
- Higher availability
  - Catastrophic crash on a single data center
- Consistency is sacrificed

Sherpa Overview

- Sherpa is a suite of data services:
  - PNUTS: Data serving platform
  - YMB: message delivery service
  - YDOT (ordered table), YDHT (hash table): sort and hash files to organize globally
- Distributed NoSQL key-value store
- At 2015/06, handles 1M queries/second
- We will focus on PNUTS

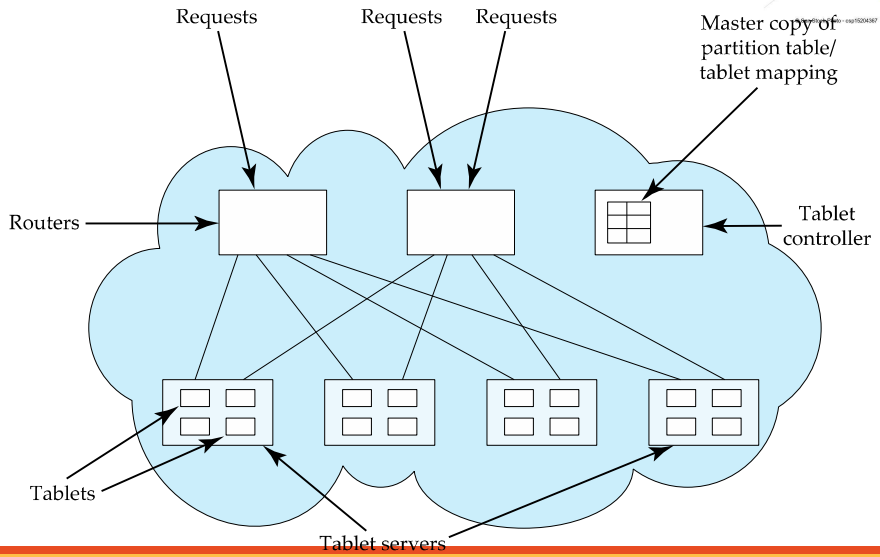
What is PNUTS?

- **PNUTS**: Platform for Nimble(**靈巧的, 敏捷的**) Universal Table Storage
- Goals/Requirements
  - Scalability, Scalability, **Scalability**
  - **Low latency**
  - **Availability** (…or else, $--)
  - A certain degree of **Consistency**

System Architecture Overview

- Servers are split into regions
  - Think of different data centers
  - Wide-area replication
- Data hashed by key and splited into tablets
- Tablets splits/migrates when it's too big
- A tablet controller/tablet router would determine which server to ask
  - Simple API
- Regions sync via *Yahoo! Message Broker*
  - A reliable message delivery service
  - Supports publish/subscribe

PNUTS Data Storage Architecture



Relaxed Consistency

- *Timeline consistency* supported
  - It's guaranteed that updates are performed in order, although results might not be seen immediately
  - The model lies between *serializable* and *eventual consistency*
- Transactions are not that important here
- But the model should be simple to web developers

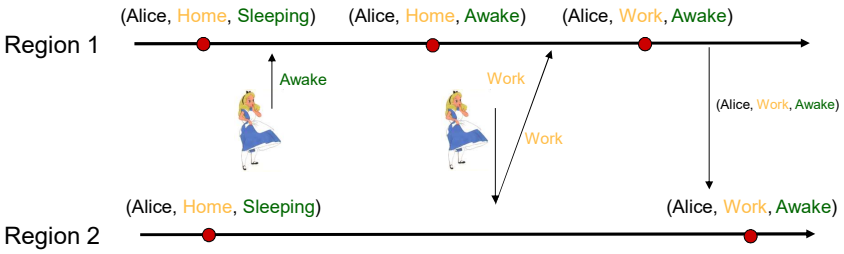
Timeline Consistency

- The consistency is at record (row) level
- Data is replicated to regions, but only one region would become the ***master*** of a record
- Updates are commited if published to YMB
  - On master: directly publish it
  - On non-master: forward to master and publish
- Low latency: no need to wait to sync to all replicas
  - Important for web applications

Timeline Consistency Example

Transactions:

- Alice changes status from "Sleeping" to "Awake"
- Alice changes location from "Home" to "Work"



No replica should see record as (Alice, Work, Sleeping)

Eventual Consistency

- Timeline consistency comes at a price
  - Writes not originating in record master region forward to master and have longer latency
  - When master region down, record is unavailable for write
- PNUT added eventual consistency mode
  - On conflict, latest write per field wins
  - Target customers
    - Those that externally guarantee no conflicts
    - Those that understand/can cope

APIs

- Different levels
  - `read-any`
  - `read-critical(required_version)`
  - `read-latest`
  - `write`
  - `test-and-set-write(required_version)`
- Users got to choose between response time or consistency

PNUTS in Production

- Over 100 Yahoo! applications/platforms on PNUTS
  - Movies, Travel, Answers
  - Over 450 tables, 50K tablets
  - (Sherpa hosts over 2,000 tables, 1 trillion records, 2015/06)
- Growth, past 18 months
  - 10s to 1000s of storage servers
  - Less than 5 data centers to over 15
- Should be more by now!

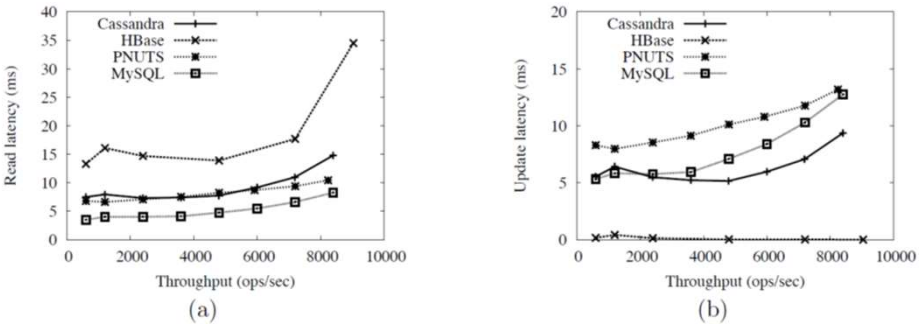
Comparison using YCSB

- Taken from 'Benchmarking Cloud Serving Systems with YCSB' by Brain F Cooper et all.
- YCSB is Yahoo Cloud Server Benchmarking framework.
- Comparison between Cassandra, HBase, PNUTS, and Shared MySQL.
- Cassandra and Hbase have higher read latencies on a read heavy workload than PNUTS and MySQL, and lower update latencies on a write heavy workload.

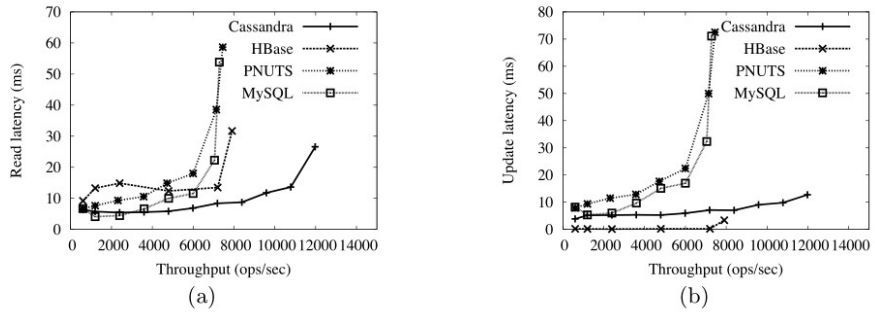
Comparison using YCSB

- PNUTS and Cassandra scaled well as the number of servers and workload increased proportionally.
- Cassandra, HBase and PNUTS were able to grow elastically while the workload was executing.
- HBase's performance was more erratic as the system scaled.

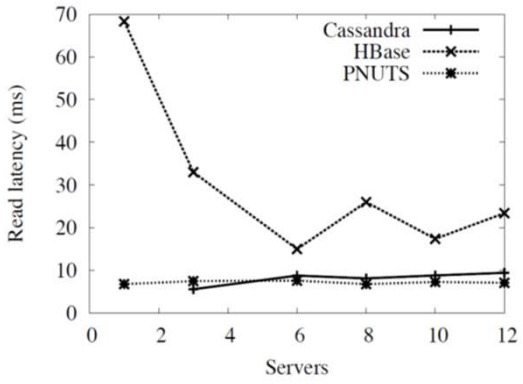
Comparison – Read Heavy



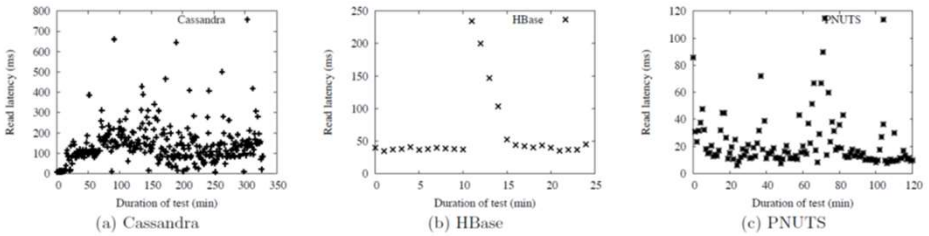
Comparison – Update Heavy



Comparison – Cluster Size



Comparison – Elastic Speedup



- Start with 2 servers
- Add more servers, one at a time.
- Until 6 servers

MongoDB

What is MongoDB

- Originally developed by 10gen (Founded in 2007)
- A document-oriented, NoSQL database
  - Hash-based, schema-less database
  - No Data Definition
  - Can store hashes with any keys and values
  - Keys are a basic data type (stored as strings)
  - Document Identifiers (_id) will be created for each document
  - Application tracks the schema and mapping
  - Uses BSON format (ased on JSON – B stands for Binary)
- Written in C++
- Supports APIs (drivers) in many computer languages: JavaScript, Python, Ruby, Perl, Java, Java Scala, C#, C++, Haskell, Erlang
- Latest release: MongoDB 7.0 (2023)

Popularity of MongoDB

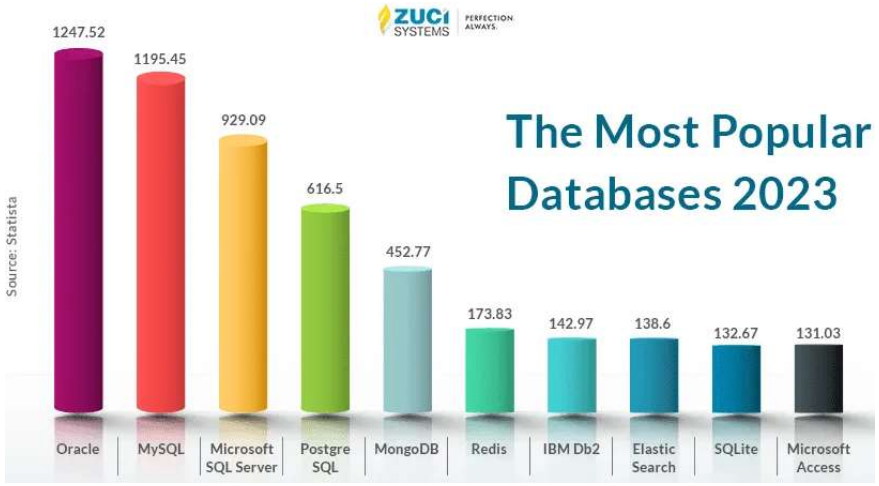
- MongoDB is very popular in recent years.
- It is the only NoSQL in the top 5 DB engine ranking

417 systems in ranking, December 2023

Rank			DBMS	Database Model	Score		
Dec 2023	Nov 2023	Dec 2022			Dec 2023	Nov 2023	Dec 2022
1.	1.	1.	Oracle	Relational, Multi-model	1257.41	-19.62	+7.10
2.	2.	2.	MySQL	Relational, Multi-model	1126.64	+11.40	-72.76
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	903.83	-7.59	-20.52
4.	4.	4.	PostgreSQL	Relational, Multi-model	650.90	+14.05	+32.93
5.	5.	5.	MongoDB	Document, Multi-model	419.15	-9.40	-50.18
6.	6.	6.	Redis	Key-value, Multi-model	158.35	-1.66	-24.22
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	137.75	-1.87	-7.18
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	134.60	-1.40	-12.02
9.	↑ 10.	9.	Microsoft Access	Relational	121.75	-2.74	-12.08
10.	↑ 11.	↑ 11.	Snowflake	Relational	119.88	-1.12	+5.11

Most Popular DBs

- Another statistics



Key MongoDB Features

- Document-oriented storage
- Schema free
- Full index support
- Replication & high availability
- Auto-sharding
- Easy and efficient querying
- Fast in-place updates
- Map/Reduce integration

MongoDB

- mongoDB = "Hu**mongo**us **DB**"
- Open-source (MongoDB Community)
- Commercial version needs Licence fee.
- "High performance, high availability"
- Automatic scaling
- CP system on CAP

Data Model

- Document-Based (max doc size is 16 MB)
- Documents are in BSON format, consisting of field-value pairs
- Each document stored in a collection
- Collections
  - Have index set in common
  - Like tables of relational DB's.
  - Documents do not have to have uniform structure

Data Model

- A MongoDB instance may have zero or more databases
- A database may have zero or more 'collections'.
- A collection may have zero or more 'documents'.
- A document may have one or more 'fields'.
- MongoDB 'Indexes' function much like their RDBMS counterparts.

MongonDB vs RDBMS

MongoDB	RDBMS
Collection	Table/View
Document	Tuple/Row
Field	Column
PK: _id Field	PK: Any Attribute(s)
Reference	Foreign Key
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

JSON

- "JavaScript Object Notation"
- Easy for humans to write/read, easy for computers to parse/generate
- Objects can be nested
- Built on
  - name/value pairs
  - Ordered list of values

BSON

- "Binary JSON"
- Binary-encoded serialization of JSON-like docs
- Zero or more key/value pairs are stored as a single entity
- Each entry consists of a field name, a data type, and a value
- Also allows "referencing"
- Embedded structure reduces need for joins
- Large elements in a BSON document are prefixed with a length field to facilitate scanning
- **Goals**: Lightweight, traversable, efficient (decoding and encoding)

Why JSON as Doc Struc?

- JSON document has simple structure and very easy to understand the content.
- JSON is smaller, faster and lightweight compared to XML.
- For data delivery between servers and browsers, JSON is a better choice
- Easy in parsing, processing, validating in all languages with rich set of tools
- JSON can be mapped more easily into object oriented system.

JSON Format

- Data is in name/value pairs
- A name/value pair consists of a field name followed by a colon, followed by a value:
  - Example: “name”: “R2-D2”
- Data is separated by commas
  - Example: “name”: “R2-D2”, race : “Droid”
- Curly braces { } hold objects
  - Example: {“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}
- An array is stored in brackets [ ]
  - Example: [{“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}, {“name”: “Yoda”, affiliation: “rebels”}]

A JSON Document Example

```
{
  "_id" : 1,
  "name" : { "first" : "John", "last" : "Backus" },
  "contribs" : [ "Fortran", "ALGOL", "BCNF", "FP" ],
  "awards" : [
    { "award" : "W.W. McDowell Award",
      "year" : 1967,
      "by" : "IEEE Computer Society“
    }, {
      "award" : "Draper Prize",
      "year" : 1993,
      "by" : "National Academy of Engineering“
    }
  ]
}
```

JSON vs XML

XML	JSON
It is a markup language.	It is a way of representing objects.
This is more verbose than JSON.	This format uses less words.
It is used to describe the structured data.	It is used to describe unstructured data which include arrays.
JavaScript functions like *eval*(), *parse*() doesn't work here.	When *eval* method is applied to JSON it returns the described object.

Example:
<car>
<company>Volkswagen</company>
<name>Vento</name>
<price>800000</price> </car>
	Example:
	{
	"company": Volkswagen,
	"name": "Vento",
	"price": 800000
	}

Processing JSON vs XML

Using XML

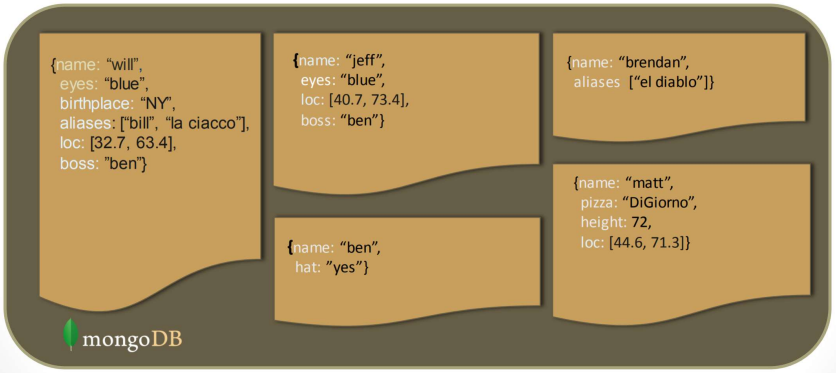
1. Fetch an XML document from web server.
2. Use the XML DOM to loop through the document.
3. Extract values and store in variables.
4. It also involves type conversions.

Using JSON

1. Fetch a JSON string.
2. Parse the JSON string using eval() or parse() JavaScript functions.

Schema Free

- MongoDB does not need any pre-defined schema
- Every document could have different data
  - Addresses NULL data fields



Index Functionality

- B+ tree indexes
- An index is automatically created on the **_id** field (the PK)
- Users can create other indexes to improve query performance or to enforce Unique values for a particular field
- Supports single field index as well as Compound index
  - Like SQL order of the fields in a compound index matters
  - If an array field is indexed, MongoDB creates separate index entries for every element of the array
- Sparse property of an index ensures that the index only contain entries for documents that have the indexed field. (so ignore records that do not have the field defined)
- If an index is both unique and sparse – then the system will reject records that have a duplicate key value but allow records that do not have the indexed field defined

Getting Started

- MongoDB can be installed on Linux, Windows and macOS.
- Can also use MongoDB Atlas online.
- You should install at least the Linux version to work with other Big Data tools we've discussed so far.
- Go to https://docs.mongodb.com/manual/installation/
- Follow the Installation Tutorials, you should have your MongoDB running in minutes.

Using MongoDB

- To use MongoDB, you can start the mongod (the primary daemon) first. Then on a terminal window, type mongo for the mongodb shell.
- You can also use MongoDB Atlas online service.
- Use PyMongo if you intend to work with MongoDB from Python.
- You can create RDDs from MongoDB collections using **pymongo-spark** library(PyMongo+PySpark).
- Can also use the MongoDB Spark Connector package(`mongo-spark-connector`)

CRUD Operations

- **C**reate
  - db.<collection>.insertOne(<document>)
  - db.<collection>.insertMany([<doc1>, <doc2>, …])
  - db.<collection>.save( )
  - db.<collection>.update(, , { upsert: true })
- **R**ead
  - db.<collection>.find(,)
  - db.<collection>.findOne(,)
- **U**pdate
  - db.<collection>.update(, ,)
- **D**elete
  - db.<collection>.remove(,)

The <collection> specifies the collection or the 'table' to store the document.

The insert Method

- To insert data into MongoDB collection, use **insert** or **save**.
- The **insertOne()** and **insertMany()** are for inserting one or many docs.

"db.COLLECTION_NAME.insertOne(document)"

```
db.StudentRecord.insertMany([
  {
    "Name": "Tom",
    "Age": 30,
    "Role": "Student",
    "University": "CU",
  },
  {
    "Name": "Sam",
    "Age": 22,
    "Role": "Student",
    "University": "OU",
  }
])
```

The find() Method

- To query data from MongoDB collection, you can use the **find()** method.
- The basic syntax: **db.COLLECTION_NAME.find()**
  - Returns a cursor, which is iterated over shell to display first 20 results.
  - Add .**limit**(<number>) to limit results
  - db.<collection>.**findOne**() to get one document
- To display the results in a formatted way, you can use **pretty()** method.

  db.StudentRecord.find().pretty()

Querying with find()

- db.<collection>.**find**(

  { <field1>:<value1>, <field2>:<value2> })

  is like

```
SELECT *
FROM <table>
WHERE <field1> = <value1> AND <field2> = <value2>;
```

Querying with find()

- db.<collection>.**find**({ **$or**:
  [{ <field>:<value1> }, { <field>:<value2> }]
  })

  Is like

  SELECT *
  FROM <table>
  WHERE <field> = <value1> **OR** <field> = <value2>;

The update() Method

- db.<collection>.**update**(
  {<field1>:<value1>}, //all docs with field = value
  { **$set**: {<field2>:<value2>} }, //set field to value
  { **multi**: true, //update multiple docs
  upsert: true } //if no doc match, insert new doc
)

The remove() Method

- **remove()** method is used to remove a document from the collection. It accepts two parameters: deletion criteria and justOne flag.
- **deletion criteria** – (Optional) deletion criteria on documents to be removed.
- **justOne** – (Optional) if set to true or 1, then remove only one document.
- Syntax:

db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)

remove() Examples

- Remove based on DELETION_CRITERIA

  db.StudentRecord.remove({"Name": "Tom"})
- Remove Only One: Removes first record

  db.StudentRecord.remove(DELETION_CRITERIA,1)
- Remove all Records

  db.StudentRecord.remove()

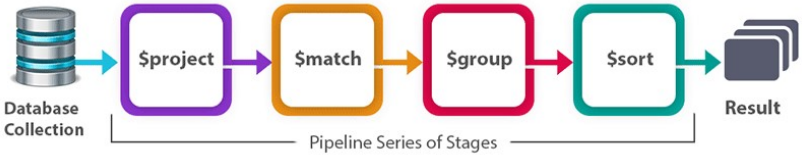
CRUD Isolation

- By default, all writes are atomic only on the level of a single document.
- This means that, by default, all writes can be interleaved with other operations.
- You can isolate writes on an **unsharded** collection by adding $isolated:1 in the query area:

```
db.<collection>.remove( { <field>:<value>,
                          $isolated: 1 } )
```


MongoDB Aggregations

- The MongoDB aggregation framework provides powerful mechanisms for document processing such as the aggregation pipeline.
- An **aggregation pipeline** consists of stages:
  - Each stage performs an operation on the i/p docs.
  - The docs that are o/p from a stage are passed on as the i/p of the next stage.
  - A pipeline can return results for groups of docs.



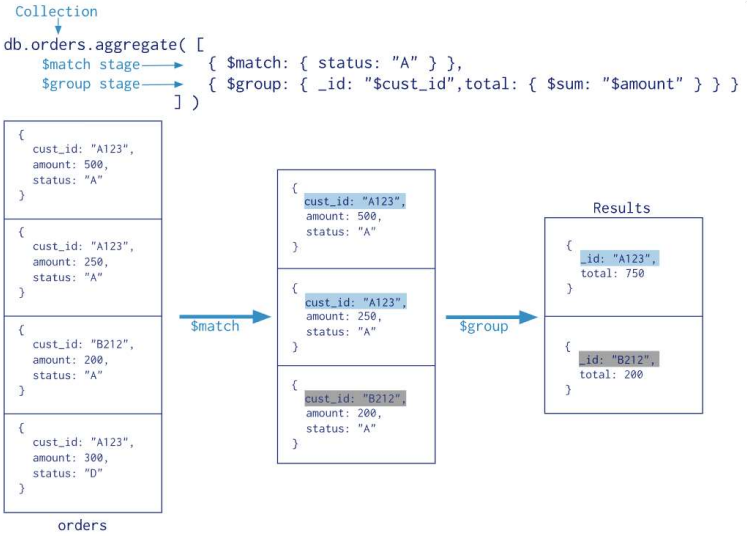
Aggregation Operators

- Each stage begins with a stage operators such as:
  - **$match:** Matching(filtering) the i/p documents.
  - **$project:** Pick a subset of a collection's fields.
  - **$group**: Classify documents according to value.
  - **$sort:** Sort the documents based on value.
  - **$skip:** Skip doex and pass the remaining.
  - **$limit:** Limit the first n docs to pass.
  - **$unwind:** Deconstruct an array field to return documents for each element.
  - **$out:** Output and add new documents to a collection.
  - **Expression** makes reference to the field's name.

Aggregation Accumulators

- Various aggregation accumulators can be in the group stage:
  - **Sum:** Sums the numeric values for docs in the group.
  - **Count:** Totals the number of documents.
  - **Avg:** Determines the average of all given values across all documents.
  - **Min:** The mininum value from all the documents.
  - **Max:** The maximum value from all the documents.
  - **First:** Retrieves the first document from the group.
  - **Last:** Retrieves the last document from the group.

Example of Aggregation

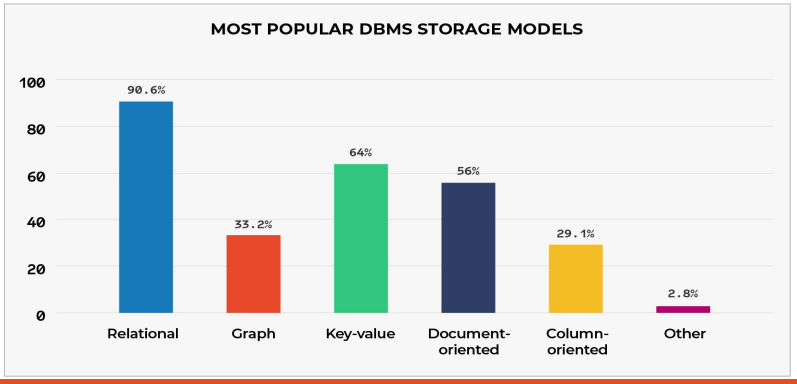


Aggregation Resources

- Can also use single purpose aggregation methods to aggregate docs from a single collection.
- For more information:
  - Introduction to the MongoDB Aggregation Framework (https://www.mongodb.com/developer/products/mongodb/introduction-aggregation-framework/)
  - MongoDB Manual - Aggregation Pipeline (https://www.mongodb.com/docs/manual/core/aggregation-pipeline/)
  - Aggregation in MongoDB (https://www.geeksforgeeks.org/aggregation-in-mongodb/)

NoSQL Conclusion

- In 2021 Dzone Data Persistence Trend Report, relational DBs are still the most popular.
- Combined, NoSQL DBs are more popular.



NoSQL 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

NewSQL Databases

Why NewSQL

- Traditional RDBMS guarantee ACID, support SQL, transactions, … but not scalable!
- NoSQL DBs scale well, flexible schema, flexible consistency, … but no guarantee and powerful query language.
- Can we have the best of both worlds?!

What do We Need?

- New DBMSs that can scale across multiple machines natively and provide ACID guarantees.
- DBMS that delivers the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads.

We want NewSQL !!

Stonebraker's Definition

- Michael Ralph Stonebraker (renowned scholar in DB, received the 2014 Turing Award) summarized the key features of NewSQL:
  - SQL as the primary interface.
  - ACID support for transactions
  - Non-locking concurrency control.
  - High per-node performance.
  - Parallel, shared-nothing architecture.
- Michael Stonebraker. New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps. *Communications of the ACM*, June, 2011.

NoSQL vs NewSQL

NoSQL

- New breed of non--relational database products
- Rejection of fixed table schema and join operations
- Designed to meet scalability requirements of distributed architectures
- And/or schema-less data management requirements

NewSQL

- New breed of relational database products
- Retain SQL and ACID
- Designed to meet scalability requirements of distributed architectures
- Or improve performance so horizontal scalability is no longer a necessity

NoSQL vs NewSQL

NoSQL

- BigTable: data mapped by rwo key, column key and time stamp
- Key-value stores: store keys and associated values
- Document store: stores all data as a single document
- Graph database: use nodes, properties and edges to store data and the relationships between entities

NewSQL

- MySQL storage engines: scale-up and scale-out
- Transparent sharding: reduce manual effort required to scale
- Appliances: take advantage of improved hardware performance, solid state drives
- New databases: designed specifically for scale-out

VoltDB
(Volt Active Data)

Scaling Traditional OLTP Databases

- Sharding improves performance but introduces…
  - Management complexity
    - disjointed backup/recovery and replication
    - manual effort to re-partition data
  - Application complexity
    - shard awareness
    - cross partition joins
    - cross partition transactions
  - And, each shard still suffers from traditional OLTP performance limitations
- If you can shard, your application is probably great in VoltDB.

VoltDB Overview

- "OLTP Through the Looking Glass"
  http://cs-www.cs.yale.edu/homes/dna/papers/oltpperf-sigmod08.pdf
- VoltDB avoids the overhead of traditional databases
  - K-safety for fault tolerance
    - **no logging**
  - In memory operation for maximum throughput
    - **no buffer management**
  - Partitions operate autonomously and single-threaded
    - **no latching** or **locking**
- Built to horizontally scale

VoltDB vs Others

	VoltDB	NoSQL	Traditional RDBMS
Scale-out architecture	✓	✓	
Built-in high availability	✓	✓	
Multi-master replication	✓	✓	
Built-in durability	✓		✓
ACID compliant	✓		✓
SQL data language	✓		✓
Cross-partition joins	Automatic	In app code	In app code
Cost at scale	$	$$$	$$$$

(From VoltDB Technical Overview)

VoltDB Architecture

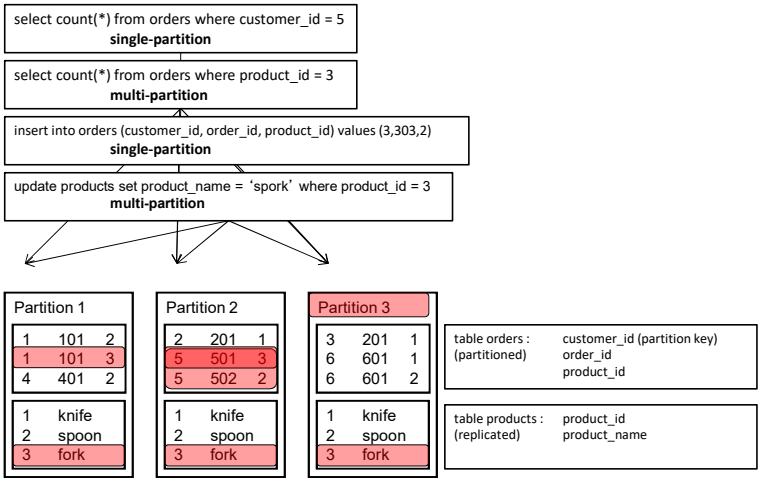
- In-memory storage to maximize throughput, avoiding costly disk accesses
- Serializing all data access, thus avoiding overheads such as locking, latching and buffer management
- Performance and scale through horizontal partitioning
- High availability through synchronous, multi-master replication (in VoltDB parlance, “K-safety”)
- Durability through an innovative combination of database snapshots and command logs that store recoverable state information on persistent devices (i.e., spinning disks and/or SSDs)

VoltDB Partitions 1/4

- 1 partition per physical CPU core
  - Each physical server has multiple VoltDB partitions
- Data - Two types of tables
  - Partitioned
    - Single column serves as partitioning key
    - Rows are spread across all VoltDB partitions by partition column
    - Transactional data (high frequency of modification)
  - Replicated
    - All rows exist within all VoltDB partitions
    - Relatively static data (low frequency of modification)
- Code - Two types of work – both ACID
  - Single-Partition
    - All insert/update/delete operations within single partition
    - Majority of transactional workload
  - Multi-Partition
    - CRUD against partitioned tables across multiple partitions
    - Insert/update/delete on replicated tables

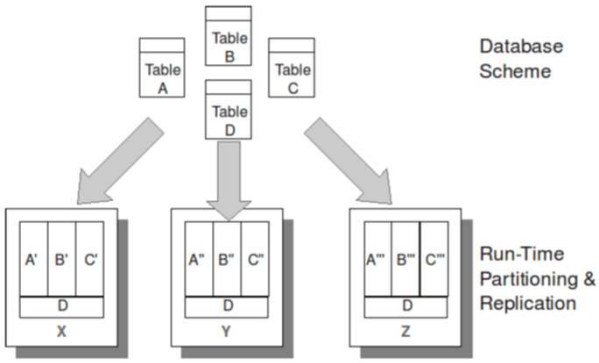
VoltDB Partitions 2/4

- **Single-partition vs. Multi-partition**



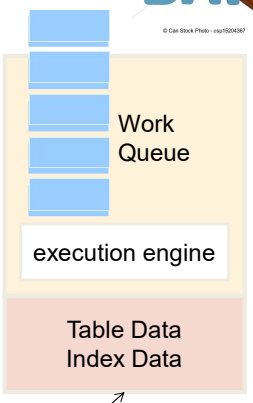
VoltDB Partitions 3/4

- For large tables(A, B, C), partitioning is appropriate.
- Replication of small, read-mostly tables (D) improves performance.



VoltDB Partitions (4/4)

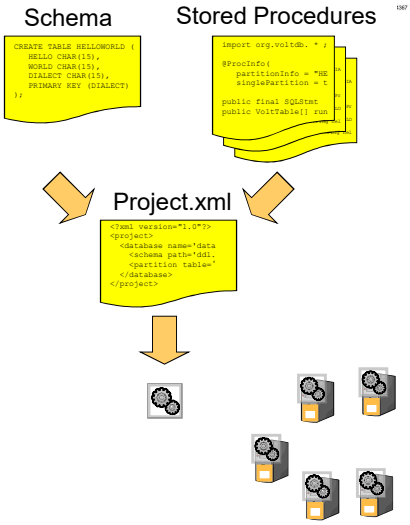
- Looking inside a VoltDB partition…
  - Each partition contains data and an execution engine.
  - The execution engine contains a queue for transaction requests.
  - Requests are executed sequentially (single threaded).



- Complete copy of all replicated tables
- Portion of rows (about 1/partitions) of all partitioned tables

VoltCompiler

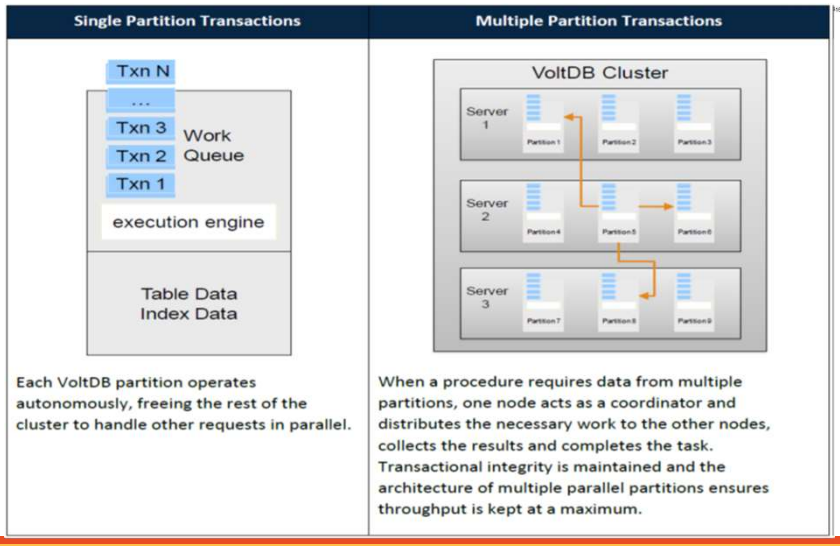
- The database is constructed from
  - The schema (DDL)
  - The work load (Java stored procedures)
  - The Project (users, groups, partitioning)
- VoltCompiler creates application catalog
  - Copy to servers along with 1 .jar and 1 .so
  - Start servers



VoltDB Transactions

- All access to VoltDB is via Java stored procedures (Java + SQL)
- A single invocation of a stored procedure is a transaction (committed on success)
- Limits round trips between DBMS and application
- High performance client applications communicate asynchronously with VoltDB

Single/Multiple Partition Transactions

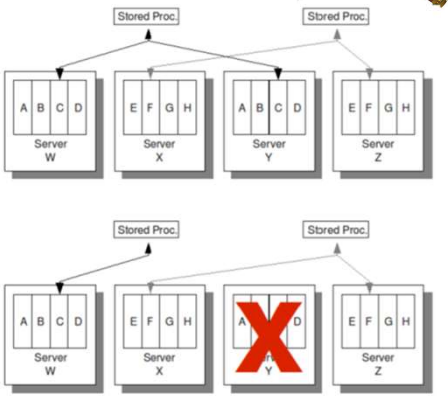


Cluster and Durability

- Scalability
  - Increase RAM in servers to add capacity
  - Add servers to increase performance / capacity
  - Consistently measuring 90% of single-node performance increase per additional node
- High availability
  - K-safety for redundancy(next slide)
- Snapshots
  - Scheduled, continuous, on demand
- Spooling to data warehouse
- Disaster Recovery/WAN replication (Future)
  - Asynchronous replication

K-safety

- A synchronous multi-master replication strategy for fault tolerance.
- When a DB is configured for K-safety, VoltDB automatically replicates partitions so that the DB can withstand the loss of "K" nodes.



Network Fault Detection

- When network partition occurs, may have "split brain" scenario if K-safety works on both parts.
- VoltDB automatically detects net faults, evaluates and assigns all work to the "surviving" sub cluster.
- Snapshots data in the "orphaned" sub cluster and performs orderly shutdown of those nodes.
- Once net repaired, orphaned nodes can be reintroduced to the cluster using Live Node Rejoin. (next slide)

Live Node Rejoin

- Nodes can be reintroduced to the cluster via the "rejoin" operation.
- On rejoin, the node retrieves data(partitions) from its sibling nodes.
- Siblings continue to serve during rejoining.
- Once the rejoined node catches up, it returns to normal operation.
- The cluster regains its full K-safety and performance status.

Asynchronous Comm

- Client applications communicate asynchronously with VoltDB
  - Stored procedure invocations are placed "on the wire"
  - Responses are pulled from the server
  - Allows a single client application to generate > 100K TPS
  - The client library can simulate synchronous if needed

Traditional

```
salary := get_salary(employee_id);
```

VoltDB

```
callProcedure(asyncCallback, "get_salary", employee_id);
```

Transaction Control

- VoltDB does not support client-side transaction control
- Client applications cannot:
  - insert into t_colors (color_name) values ('purple');
  - rollback;
- Stored procedures commit if successful, rollback if failed
- Client code in stored procedure can call for rollback

Interfacing with VoltDB

- Client applications interface with VoltDB via stored procedures
  - Java stored procedures – Java and SQL
  - No ODBC/JDBC

Lack of Concurrency

- Single-threaded execution within partitions (single-partition) or across partitions (multi-partition)
- No need to worry about locking/dead-locks
  - great for "inventory" type applications
    - checking inventory levels
    - creating line items for customers
- Because of this, transactions execute in microseconds.
- However, single-threaded comes at a price
  - Other transactions wait for running transaction to complete
  - Don't do anything crazy in a Stored Procedure (request web page, send email)
  - Useful for OLTP, not OLAP

Throughput vs. Latency

- VoltDB is built for throughput over latency
- Latency measured in mid single-digits in a properly sized cluster
- Do not estimate latency too optimistically. Many causes of latency should be considered:
  - Other applications
  - Frequent snapshots
  - I/O contention
  - JVM statistics collection
  - Hardware power saving options

SQL Support

- SELECT, INSERT (using values), UPDATE, and DELETE
- Aggregate SQL supports AVG, COUNT, MAX, MIN, SUM
- Materialized views using COUNT and SUM
- Hash and Tree Indexes
- SQL functions and functionality will be added over time, for now use Java
- Execution plan for all SQL is created at compile time and available for analysis

SQL in Stored Procedures

- SQL can be parameterized, but not dynamic

  "select * from foo where bar = ?;" (YES)

  "select * from **?** where bar = ?;" (NO)

Connecting to the Cluster

- Clients connect to one or more nodes in the VoltDB cluster, transactions are forwarded to the correct node.
  - Clients are not aware of partitioning strategy
  - In the future, may send back data in the response indicating if the transaction was sent to the correct node.

Schema Changes

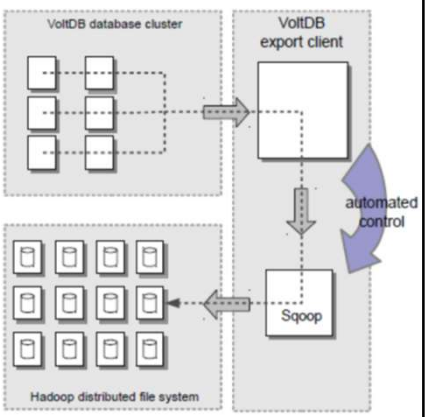
- Traditional OLTP
  - add table…
  - alter table…
- VoltDB
  - modify schema and stored procedures
  - build catalog
  - deploy catalog
- Add/drop users, stored procedures
- Add/drop tables
- Add/drop column, …

Table/Index Storage

- VoltDB is entirely in-memory
- Cluster must collectively have enough RAM to hold all tables/indexes (k + 1 copies)
- Even data distribution is important

Hadoop Integration

- Combination of VoltDB and Hadoop offer the flexibility to handle a continuum of "fast" and "deep" data applications.
- VoltDB's export client automates the process of exporting data from VoltDB to Hadoop.
- VoltDB can integrate directly with HDFS or through Hadoop's Sqoop import technology.



Apache Sqoop is a tool for transferring bulk data between Hadoop and RDBMS.

Libraries Support

- Many VoltDB-provided and community-developed libraries for application development.

VoltDB-Provided Libraries		Community-Developed Libraries
Java C++ .NET (C#)	Python PHP HTTP/JSON	Erlang Ruby Node.js

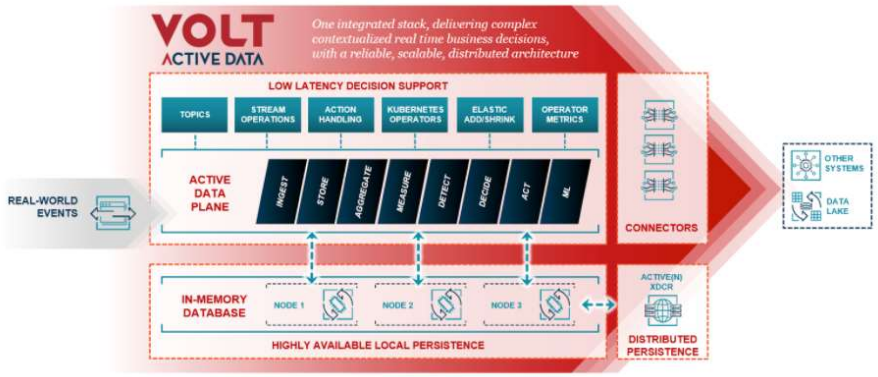
- The VoltDB Community Edition is open source.

Volt Active Data

- As the VoltDB evolved into a data platform (not just a DB), the name is changed to Volt Active Data.
- ACTIVE = Engaged in action; Being in a state of progress or motion; Having the power of quick motion.
- 'Active Data' means that it's not just passively reacting to events, but responding to them dynamically with integration and analysis of related data to serve core business functions.

Volt Active Data Snapshot

- A snapshot of the Volt Active Data platform:



- The work happens in the Active Data Plane (next slide)

Volt Active Data Plane 1

- Ingest: Read messages off the network, from wireline API and a C++/Java client or off Kafka.
- Store: Store data if we need to. Unlike a DB, which is “Always Store And Sometimes Process”, Volt is “Always Process And Sometimes Store”.
- Aggregate: Aggregate data on the fly to provide accurate running totals without having to scan or process any data at query time.
- Measure: Take incoming events and measure them in some non-trivial way for decision aid.

Volt Active Data Plane 2

- Detect: Detect anomalies. To spot when something hasn't happened as expected, as well as being driven by incoming events.
- Decide: Determine what to do next.
- Act: The action to take once a decision has been made. E.g. updating stored data and/or sending multiple messages.
- Machine learning: Work with any deterministic ML decision engine as long as it's in Java and can be instantiated inside a class.

Volt Active Data & Open Source

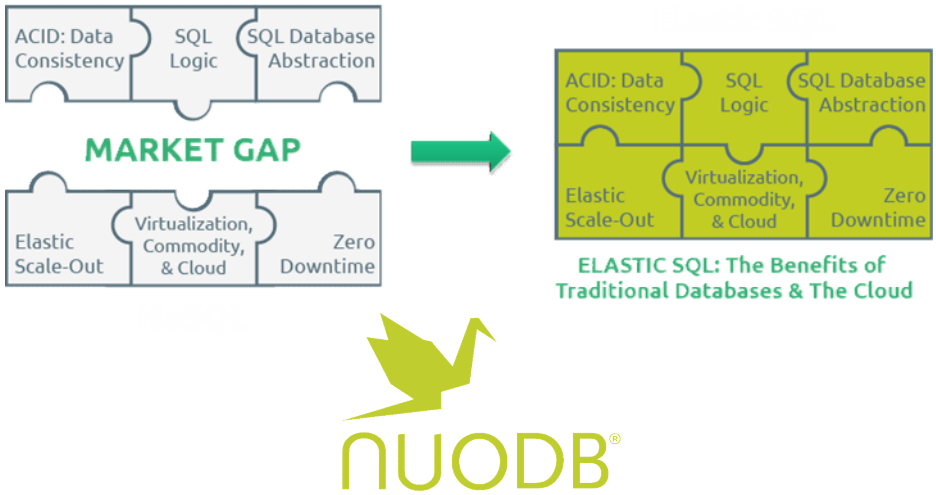
- Volt Active Data offers the fully open source, AGPL3-licensed Community Edition through GitHub (https://github.com/voltdb/voltdb/)
- The open source version is frozen at 2023.
- Trials of the enterprise edition are available at the Volt Active Data website (https://www.voltactivedata.com)

NuoDB

The Elastic SQL Database

- NuoDB is a NewSQL database which claims to be elastic, ANSI-SQL compliant w/o sharding or strongly-consistent replication.
- Combine the scale-out simplicity, elasticity, and continuous availability of cloud applications
- with the transactional consistency and durability of traditional databases.
- A relational database architected for the cloud.

The Missing Puzzle



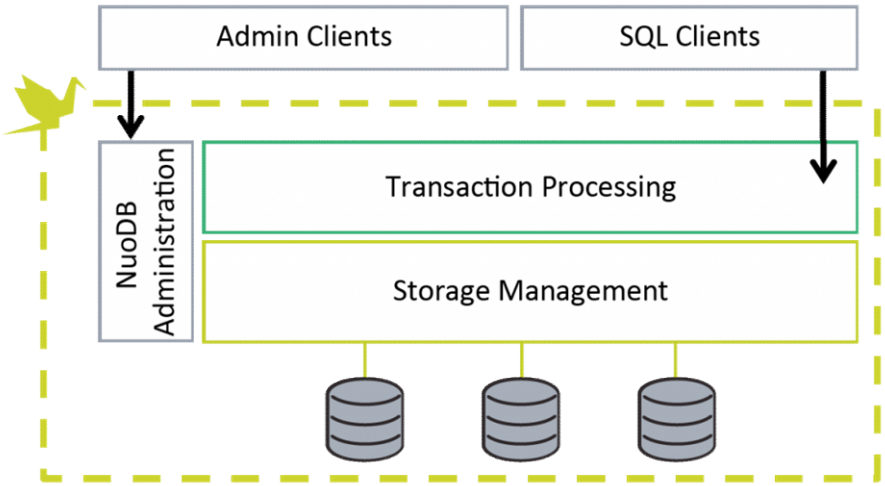
NuoDB takes the best from both sides

- Designed to be always on, always available, and still always consistent!
- **P2P architecture** ensures that database services can be natively distributed across multiple nodes, data centers, and even clouds
- without the complexity, expense, and additional software that traditional relational databases require.
- The Community Edition is open source with limited functionalities.

NuoDB Architecture 1

- NuoDB splits the traditional monolithic database process into two independent services: a transactional processing service and a storage management service. (next slide)
- Both services are scaled separately and handle failure independently.
- It also has an administration component for administrative functions.
- Splitting the transactional and storage processing services is key to making a relational system scale.

NuoDB Architecture 2



NuoDB Architecture 3

- Transaction service is responsible for **Atomicity**, **Consistency**, and **Isolation** in running transactions.
  - Has no visibility into how data is being made durable
  - A purely in-memory tier
  - An always-active, always consistent, on-demand cache
- Storage management service ensures **Durability**.
  - Responsible for making data durable on commit
  - Provide access to data when there's a miss in the transactional cache.
  - Through a set of peer-to-peer coordination messages

NuoDB Architecture 4

- Services are processes on an arbitrary # hosts.
- A single executable is running in one of two modes: as a Transaction Engine (TE) or a Storage Manager (SM).
- All processes are peers, with no single coordinator or point of failure.
- No special configuration required at any hosts.
- All processes (SMs and TEs) communicate through a simple peer-to-peer coordination protocol.

TEs and SMs

- TEs accept SQL client connections, parsing and running SQL queries against cached data.
- On a local cache miss, a TE can get the data from any of its peers (another TE or an SM).
- Makes bootstrapping, on-demand scale-out, and live migration very easy.
- Starting and scaling a DB is simple: by choosing how many processes to run, where, and in which roles.

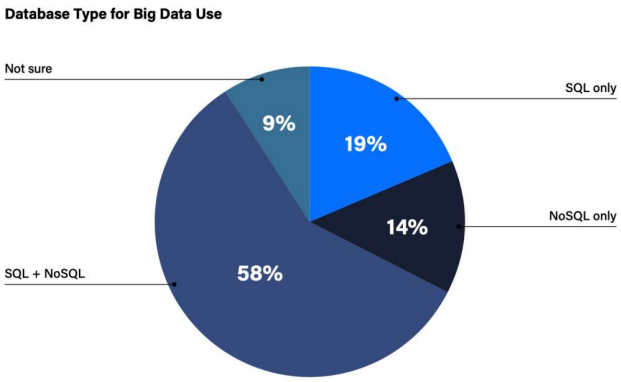
NuoDB References

- NuoDB Docs (https://doc.nuodb.com/nuodb/latest/introduction-to-nuodb/)
- Quick Dive into NuoDB Architecture (https://blog.3ds.com/topics/company-news/quick-dive-into-nuodb-architecture/)
- NuoDB: Distributed SQL Database (https://www.3ds.com/nuodb-distributed-sql-database)

Distributed SQL

SQL vs. NoSQL

- The debate of "SQL vs. NoSQL" for the past decade
- The truth is that companies are using both



SQL, NoSQL, NewSQL all fall short

- Traditional SQL DBs are not scalable, not elastic.
- NoSQL DBs (HBase, Cassandra, MongoDB, …) are scalable and elastic but sacrificing the best of SQL.
- NewSQL DBs (VoltDB, NuoDB, ClustrixDB, …) try to bring the best of both but fall short.
- What we REALLY need is complete SQL/transactions (compatibility) yet highly elastic/scalable, globally distributed, externally-consistent and fault tolerant.
- We want SQL from a distributed system POV, i.e. we want "distributed SQL" !!

Origin of Distributed SQL

- Google again was among the first to recognize that and develop Spanner.
- James C. Corbett, et al. "Spanner: Google's Globally-Distributed Database". *Proceedings of OSDI 2012*.
- Spanner demonstrated a new way of looking at databases, one that was rooted in distributed systems and global scale.
- Marks the beginning of the next evolution of DB: Distributed SQL.

What is Distributed SQL?

- A truly **Distributed SQL database** must support:
  1. Scalability: seamlessly scale wrt computing, data, users, storage, geo regions w/o operational complexity.
  2. Consistency: transactional consistency and isolation in **global scale**.
  3. Resiliency: always-on and fast recovery (~0 down time)
  4. Geo-replication: allow distribution of data throughout widely dispersed geo env.
  5. SQL: speak SQL, of course.
  6. Data locality: geo-partition of data based on locality
  7. Multi-cloud: can work with data/services on multi-cloud environments.

Foundational Requirements

- Just like all database systems, a Distributed SQL DB also has foundational requirements:
  1. Administration: friendly admin tools
  2. Optimization: insight into performance for different types/levels of optimization
  3. Security: AAA capabilities of **authentication**, **authorization**, and **accountability**
  4. Integration: integrate well with existing applications and ETL tools
- Must meets ALL the requirements in prev/current slide to be trusted for mission-critical applications.

Evaluating NewSQL using Distributed SQL Criteria

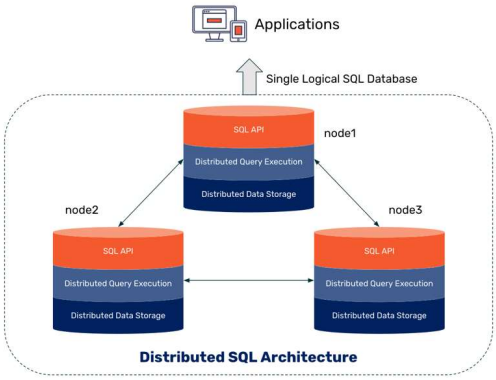
- Are NewSQL DBs good enough?

Evaluating NewSQL Databases Using Distributed SQL Criteria

	Vitess	Citus	VoltDB	NuoDB	ClustrixDB
1.0 Release	2011 (YouTube internal release)	2012	2010	2013	2013
SQL & Transactions (Compatibility)	MySQL (No Distributed ACID Transactions & Cross-Shard JOINs Are Discouraged)	PostgreSQL (No Distributed Transactions and Cross-Shard JOINs)	Proprietary SQL (No Foreign Keys)	Proprietary SQL (No Serializable Isolation)	MySQL (No Serializable Isolation)
Native Failover/Repair	✗	✗	✓	✓	✓
Horizontal Write Scalability	✓	✓	✓	✓	✓
Geographic Data Distribution	✓ Auto-Sharded with "Jurisdiction Awareness"; No Global Consistency & Native Region Failover	✗	✗ Multi-Master with LWW Conflict Resolution, No Global Consistency, Not Available in OSS	✗ No Native Sharding & Replication	✗
High Performance	✓	✗ Single Coordinator Node	✗ Writes Need Commit at All Replicas	✗	✗
Kubernetes-Native & Cloud Neutral	✓	✗	✗	✗	✗
Open Source	✓ Apache 2.0	✓ AGPL 3.0	✓ AGPL 3.0	✗	✗

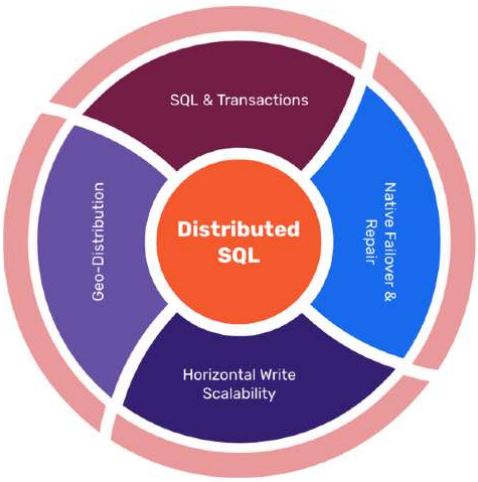
Distributed SQL Arch

- Should be like a single logical SQL DB with 3 layers:
  1. SQL API
  2. Distributed Query Execution
  3. Distributed Data Storag
- Support
  - Strongly consistent replication
  - Distributed ACID trans



Key Benefits of Distr SQL

- Developer agility with SQL & transactions
- Ultra resilience with native failover/repair
- Scale on-demand with horizontal write scalability
- Low user latency with geographic data distribution



Google Spanner

Papers

- James C. Corbett, et al. “Spanner: Google’s Globally-Distributed Database”. *Proceedings of OSDI 2012*. (the original paper)
- James C. Corbett, et al. “Spanner: Google’s Globally-Distributed Database”. *ACM Transactions on Computer Systems*, Vol. 31, No. 3, Article 8, August 2013. (the journal version, more details)
- Doug Judd. “Spanner under the hood: Understanding strict serializability and external consistency”. Google Cloud Blog. Apr 8, 2023. (https://cloud.google.com/blog/products/databases/strict-serializability-and-external-consistency-in-spanner)

What is Spanner?

- Scalable, multi-version, globally distributed, synchronously-replicated database.
- Support:
  - General-purpose transactions (ACID)
  - SQL-based query language
  - Schematized tables
  - Semi-relational data model
- Running in production
  - Storage for Google’s ad data
  - Replaced a sharded MySQL database

Motivations

- Bigtable is used in many projects.
- Consistently receiving complaints about Bigtable in **complex**, **evolving schemas**, requiring **strong consistency** on **wide-area replication**.
- Evolving Bigtable into a multi-version DB with schematized semi-relational tables.
- Data version is automatically timestamped with its commit time and garbage collected when expired.
- Can read data at specified timestamps.

Spanner Overview

- Feature: Lock-free distributed read transactions
- Property: External consistency of distributed transactions
  - First system at global scale
- Implementation: Integration of concurrency control, replication, and 2PC
  - Correctness and performance
- Enabling technology: TrueTime
  - Interval-based global time

Main Features

- Shards data across many machines in data centers all over the world.
- Replication for global availability, geographic locality, and auto clients failover btw replicas.
- Auto data reshard and auto data migration (even across datacenters) for load balancing.
- Scale up to millions of machines, hundreds of data centers, and trillions of database rows.
- First of its kind.

Replication Management

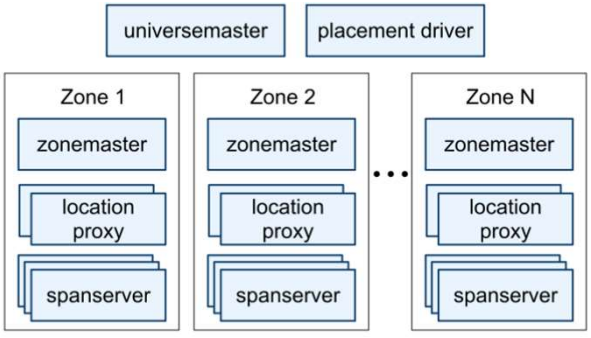
- Can configure dynamically at fine grain:
  - Placement of data on different datacenters
  - How far from users (control read latency)
  - How far btw replicas (control write latency)
  - How many replicas (control durability, availability, and read performance)
- Dynamic and transparent data movement btw datacenters for load balancing.
- Provide externally consistent reads/writes and globally consistent reads. (more on this later)

Global Timestamps

- Spanner assigns global commit timestamps to transactions that reflect serialization order.
- External consistency: if T_2 calls *Commit()* after T_1's call to *Commit()* has returned, then T_2 commit timestamp > T_1 commit timestamp.
- Guarantee at global scale. (i.e. transactions may be distributed across the globe)
- Enabled by a new TrueTime API that exposes clock uncertainty so that bounds can be determined to provide timestamp guarantees.

Spanner Architecture 1

- A Spanner deployment is called a universe organized as a set of zones.
- Zones are units of deployment analogous to that of Bigtable servers, as well as the locations for data replication.

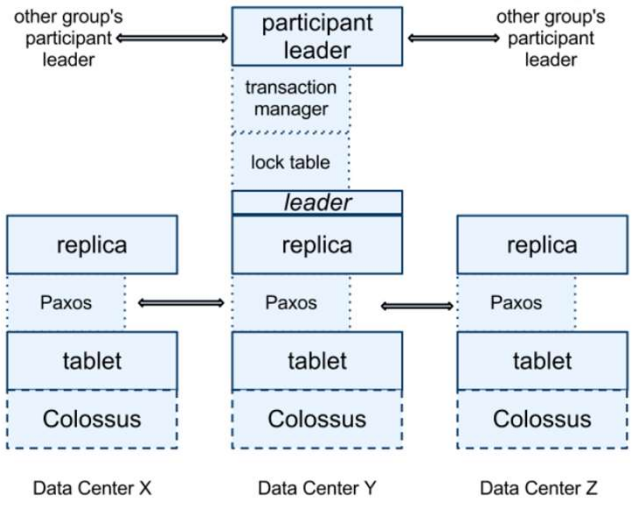


Spanner Architecture 2

- A zone has one zonemaster and 100s to 1000s spanservers.
- Zonemaster assigns data to spanservers.
- Spanservers serve data to clients.
- Per-zone location proxes are used by clients to find the spanservers of the target data.
- The universe master is a console for status display and interactive debugging.
- The placement driver handles auto movement of data across zones for replication and load balancing

Spanner Software Stack

- The Spanner software stack supports the replication and distributed transactions.



Spanner Software Stack

- Each spanserver handles 100~1000 tablets.
- A tablet is a bag of mappings:

  (key:string, timestamp:int64) → string
- Spanner is more like a multi-version DB than a key-value store.
- Tablet data is stored as files in Colossus (GFS2).
- A Paxos state machine for each tablet to store the metadata and log for that tablet.

Replica Management

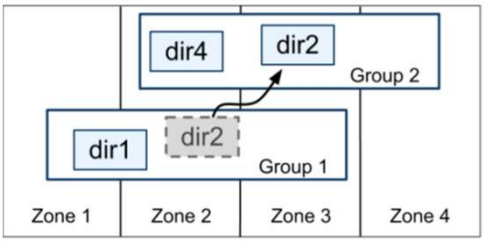
- Paxos state machines implement a consistently replicated bag of mappings.
- Writes initiate the Paxos protocol at the leader.
- Reads access state directly from tablet at any replica that is sufficiently up-to-date.
- The set of replicas is called a Paxos group.
- A lock table is maintained on each replica leader for two-phase locking in concurrency control.

Distributed Transactions

- A transaction manager on each leader is for distributed transaction management.
- The transaction manager implements a participant leader; the other replicas are participant slaves.
- When a transaction involves multiple Paxos groups, group leaders coordinate to perform two-phase commit (2PC) with one being the coordinator leader and others the coordinator slaves.

Directory and Placement

- A directory (not file directory, more like bucket) is a set of contiguous keys that share a common prefix.
- Applications can control locality by choosing keys.
- A directory is the unit of data placement. When a movement is required, data is moved directory-by-directory.



Replication Control

- A directory is also the smallest unit for replica placement configuration.
- **Administrators** control:
  - number/types of replicas
  - Geographic placement of replicas
- **Applications** control:
  - How data is replicated
- Spanner supports synchronous replication across datacenters.

Data Model

- Spanner offers applications with:
  1. A data model on schematized semi-relational tables.
  2. A query language
  3. General purpose transactions.
- An application creates one or more databases.
- Each database can have unlimited number of schematized tables.
- Tables have rows, columns, and versioned values.
- An SQL-like query language.

Data Model

- Every table has a set of one or more primary-key columns which form the name of a row.
- A table is a mapping from the primary-key columns to the non-primary-key columns.
- Database is partitioned into hierarchies of tables specified via INTERLEAVE IN declarations. (next)
- The top is a directory table.
- Each row in the directory table with key *K* and rows in descendant tables start with *K* forms a directory.

Schema and Layout

- Directories are formed by the interleaving of tables.
- Albums(2,1) represents the row from the Albums table for uid 2, aid 1.

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

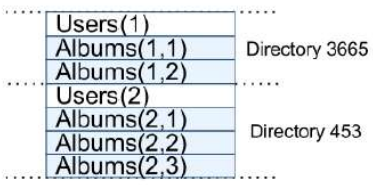


Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by INTERLEAVE IN.

TrueTime API

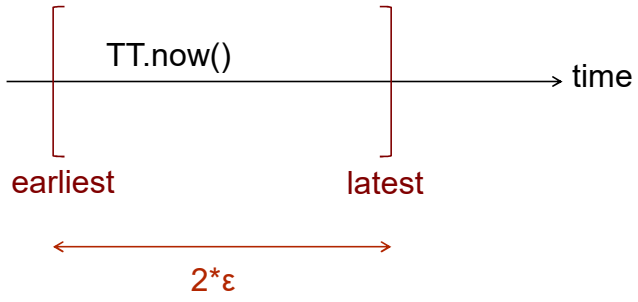
- The key enabler of the synchronous replication and distributed transactions.
- Represents time as a *TTinterval*, an interval with bounded time uncertainty.

Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(t)	true if t has definitely passed
TT.before(t)	true if t has definitely not arrived

Table 1: TrueTime API. The argument t is of type *TTstamp*.

TrueTime is Global

- TrueTime can be considered as "global wall-clock time" with bounded uncertainty.



TrueTime API

- Endpoints of a *TTinterval* are of type *TTstamp*.
- *TT.now()* method returns a *TTinterval* that is guaranteed to contain the absolute time during which *TT.now()* was invoked.
- *TT.after()* and *TT.before()* methods are convenience wrappers around *TT.now()*.
- Let the absolute time of an event *e* be $t_{abs}(e)$. TrueTime guarantees that for an invocation tt = *TT.now()*, $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$, where e_{now} is the invocation event.

TrueTime Implementation

- TrueTime is implemented by a set of time master machines per datacenter and a time slave daemon per machine.
- The underlying time references are GPS and atomic clocks.
- Masters have GPS receivers. remaining masters (Armageddon masters) are equipped with atomic clocks.

Supported Operations

- Spanner supports several types of transactions.

Operation	Timestamp Discussion	Concurrency Control	Replica Required
Read-Write Transaction	§ 4.1.2	pessimistic	leader
Read-Only Transaction	§ 4.1.4	lock-free	leader for timestamp; any for read, subject to § 4.1.3
Snapshot Read, client-provided timestamp	—	lock-free	any, subject to § 4.1.3
Snapshot Read, client-provided bound	§ 4.1.3	lock-free	any, subject to § 4.1.3

Table 2: Types of reads and writes in Spanner, and how they compare.

Operation Benchmarks

- Microbenchmarks on Spanner demonstrate the performance of different operations.

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

Scalability of Spanner

- Two-phase commit is crucial for concurrency control.
- Spanner scales well in two-phase commit.

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

Spanner in F1

- F1 was the Google's advertising backend which was originally based on MySQL.
- Spanner has been successfully evaluated on F1.

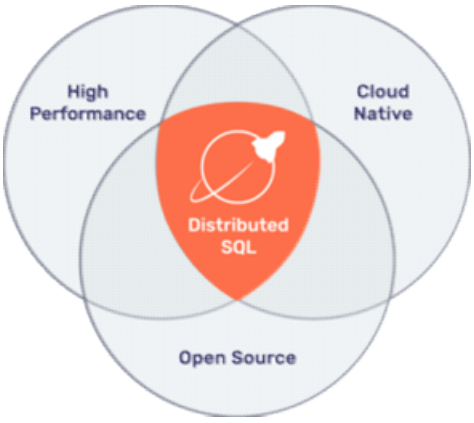
operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

YogabyteDB

YogabyteDB

- Another distributed SQL database:
  - Low total cost of ownership with high performance
  - Cloud neutral with Kubernetes native
  - High release velocity with open source



Design Goals of YogabyteDB

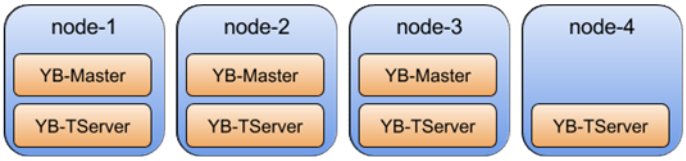
- Consistency
  - CP database (in terms of CAP theorem)
  - Single-rwo linearizability
  - Multi-row ACID transactions
- Query APIs
  - YSQL: a fully-relational SQL API
  - YCQL: a semi-relational SQL API rooted at Cassandra
- Performance
  - High write throughput, client concurrency, data density, and growing event data use-cases
- Geo-distributed deployments
- Cloud native architecture
  - Run on commodity hardware
  - Kubernetes ready
  - Open souce

Key Concepts

- A YugabyteDB universe is a group of nodes (VMs, physical machines, or containers) that collectively function as a distributed database.
- A *universe* consists of one or more *namespaces*. Each of these namespaces can contain one or more user *tables*.
- YugabyteDB automatically shards, replicates and load balances these tables across the nodes in the universe.
- A namespace is referred to as a database in YSQL and a keyspace in YCQL.

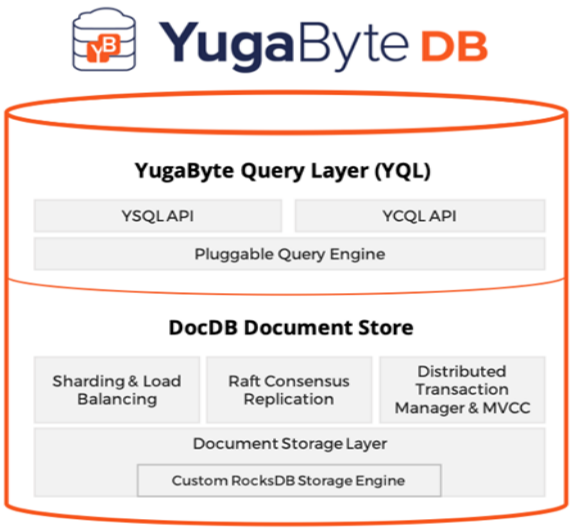
Component Services

- A universe comprises of two sets of servers, YB-TServer and YB-Master.
- The **YB-TServer** (*YugabyteDB Tablet Server*) is for hosting/serving user data and handle user queries.
- The **YB-Master** (*YugabyteDB Master Server*) is for keeping metadata, coordinating system-wide operations (create/alter/drop tables…) and initiating maintenance operations (load balancing…).



YogabyteDB Architecture

- YugabyteDB architecture follows a layered design.



CockroachDB

Scalable SQL with Global OLTP

- CockroachDB is a scalable SQL DBMS with global OLTP, high availability and strong consistency.
- Resilient to disasters through replication and automatic recovery.
- Rebecca Taft, et. al. “CockroachDB: The Resilient Geo-Distributed SQL Database”, *ACM SIGMOD 2020*, Portland, OR, USA.

Basic Concepts 1

- Cluster: a CockroachDB deployment, can have many nodes.
- Node: An individual machine running CockroachDB.
- Range: Data are stored as a giant sorted map of key-value pairs. This keyspace is divided into "ranges" (contiguous chunks of keys).
- Replica: Each range is replicated (3 by default) and stored on different node.

Basic Concepts 2

- Lease-holder: One replica in each range holds the "range lease" and is referred as the "leaseholder" for coordinating all read/write requests for the range.
- Raft Leader: One replica in each range is the "leader" for write requests to ensure majority with Raft consensus protocol. (almost always the same as the leaseholder)
- Raft Log: Each range has a time-ordered log of writes that all replicas have agreed on.

Basic Concepts 3

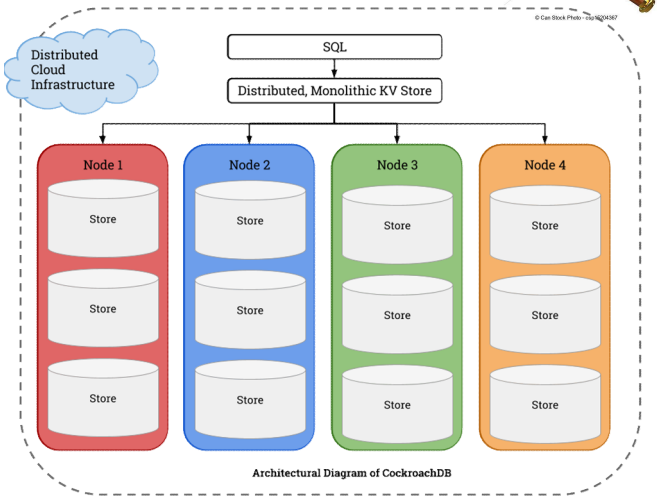
- CockroachDB uses both ACID and CAP theorem based consistency.
- When a range receives a write, a **quorum** of nodes containing replicas acknowledge the write.
- When a write doesn't achieve consensus, forward progress halts.

Relication in CockroachDB

- Two types of replication: synchronous and asynchronous.
- Synchronous replication requires all writes to propagate to a quorum before being considered committed.
- Asynchronous replication only requires a single node to receive the write to be considered committed. Updates will eventually be propogate to all other replcas.

CockroachDB Architecture

- CockroachDB is structured in layers.
- Higher layers treats lower layers as black boxes.
- Lower layers are unaware of the higher ones.

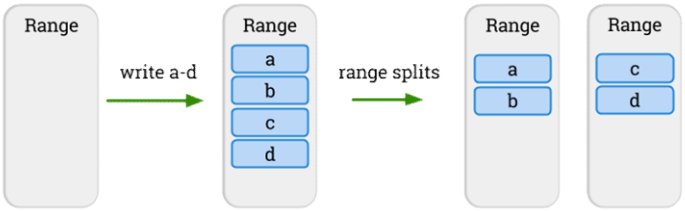


Layered Architecture

- SQL Layer: highest layer with familiar relational concepts API similar to Postgres.
- Distributed Key-Value Store: a distributed KV store as a monolithic sorted map.
- Distributed Transactions: implemented on top of other layers
- Nodes: physical/virtual machines or containers that contain stores.
- Store: Each node contains one or more stores managed with RocksDB.
- Range: Every store contains ranges. Each range covers a contiguous segment of the key space.

Horizontal Scaling

- CockroachDB starts off with a single, empty range encompassing the entire key space.
- When data in the rage grows to a threshold size (64MB), it is split into two ranges.
- Newly split ranges are rebalanced to stores with more capacity available



Range Distribution & Rebalancing in CockroachDB

Ranges in Stores

- Ranges are replicated using the Raft consensus protocol (default=replicated 3 ways).

Distributed Transactions

- Transactions are set of operations satisfying ACID semantics with strong consistency.
- CockroachDB provides distributed transactions using multi-version concurrency control (MVCC).
- CockroachDB provides snapshot isolation (SI) and serializable snapshot isolation (SSI), allowing externally consistent, lock-free reads and writes, both from a historical snapshot timestamp and from the current wall clock time.

SQL Support

- CockroachDB's SQL can encode, store, and retrieve the SQL table data and indexes.
- The SQL grammar supported is a derivative of PostgreSQL.
- The implementation leverages the distributed transactions and strong consistency provided by the monolithic sorted key-value map.

Deployment and Management

- Support simple deployment that fits well with the container model.
- Nodes are symmetric and self-organize.
- A single binary (easily put into a container) runs on each node of the cluster and export the local stores for accepting new writes and rebalances.
- Self-organizes by using a gossip network.
- The gossip network continually balances itself.
- CockroachDB can run on a laptop, corporate cluster or private cloud, as well as on any public cloud infrastructure.

Comparing Distr SQL DBs

By YogabyteDB

Comparing Distributed SQL Databases

	Amazon Aurora	Google Cloud Spanner	PingCap's TiDB	CockroachDB	YugabyteDB
SQL & Transactions Compatibility	PostgreSQL & MySQL (Full Compatibility)	Proprietary SQL (No Foreign Keys)	MySQL (No Foreign Keys & Serializable Isolation)	PostgreSQL (No Partial Indexes, Stored Procedures & Triggers)	PostgreSQL (Full Compatibility)
Native Failover/Repair	✓	✓	✓	✓	✓
Horizontal Write Scalability	✓ (Multi-Master)	✓ (Auto-Sharded)	✓ (Auto-Sharded)	✓ (Auto-Sharded)	✓ (Auto-Sharded)
Geographic Data Distribution	✓ (Only a Single Region Can Take Writes)	✓ (Global Clock Sync for Consistency)	✓ (Single Region Timestamp Generator Leads to High Latency)	✓ (Global Clock Sync for Consistency)	✓ (Global Clock Sync for Consistency)
High Performance	✗ (Read Replicas Cannot Process Writes)	✗ (Not Optimized for High Volume Ingest)	✓ (High Latency for Multi-Region Clusters)	✗ (Not Optimized for High Volume Ingest)	✓ (Optimized for High Volume Ingest)
Cloud Neutral w/ Kubernetes Native	✗ (Proprietary to AWS)	✗ (Proprietary to Google Cloud)	✓	✓	✓
Open Source	✗	✗	✓ (Apache 2.0)	✗ (BSL 1.0 is not Open Source)	✓ (Apache 2.0)

Distributed SQL Market

- The distributed SQL is quickly getting ground.

DISTRIBUTED SQL MARKET

	Monolithic (Vertical Write Scaling)	NewSQL (Horizontal Write Scaling, Single-Shard ACID & Single Region)	Globally Distributed SQL (Horizontal Write Scaling, Multi-Shard ACID)	
			Single Region*	Multi-Region
Open Source	PostgreSQL, MySQL	citusdata, Vitess	TiDB	YugaByte DB
Proprietary	Amazon Aurora	Clustrix, NUODB		Multi-Region: Google Cloud Spanner

LESS POWERFUL ⟷ MORE POWERFUL