Storage System Internals

Ankur Shrivastava (Pricing at GoJek)

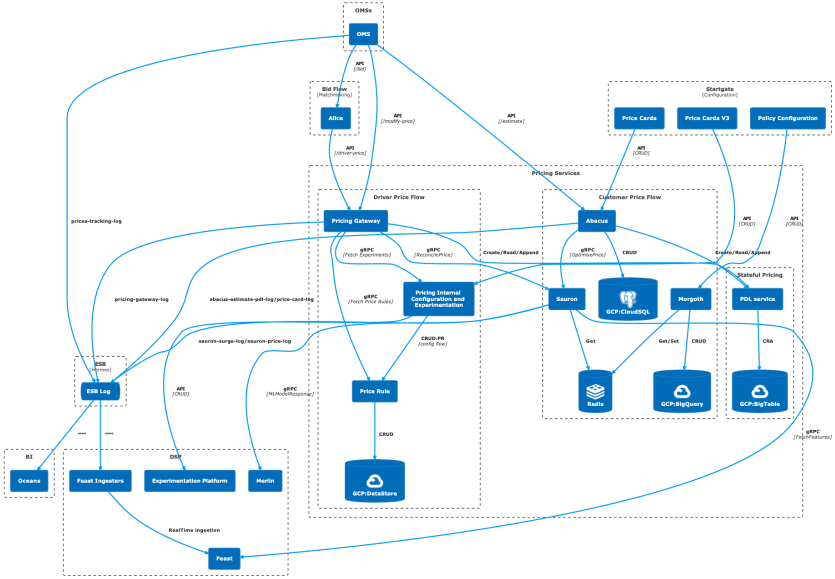
January 17, 2022

Who am I?

- Ankur Shrivastava
- Principal Software Engineer
- Team Lead of Pricing (GoJek)
- Worked at Skyscanner, Carousell, Flipkart, Amazon, Zynga, etc.

Context

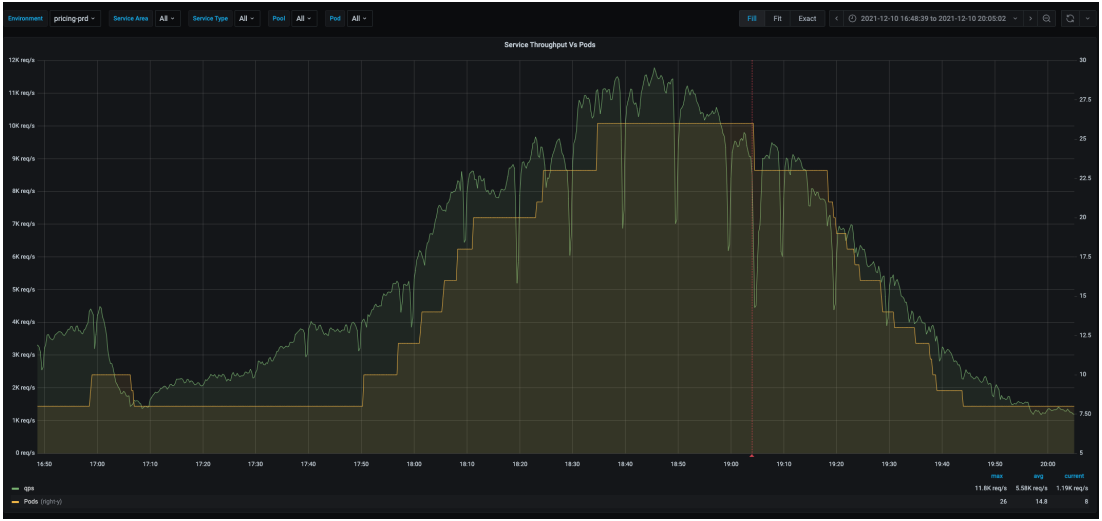
Pricing Architecture (Simplified)



Pricing Architecture

- Pricing stack is entirely on K8S (PDG Managed)
- We have no VMs (other than GKE)
- No state in K8S
- All services autoscale with traffic
- Deployments triggered by CI and take ∼5 mins max.
- All service to service calls over gRPC
- No Single Point of Failure (DB, Cache, LB, Pods have automated recovery)
- Read More at https://go-jek.atlassian.net/l/c/z0wxUr9R

Pricing Gateway Autoscaling



Dynamic Driver Pricing system processes 11.8k individual driver price requests per second (708,000 rpm) and can handle sudden 5x increase in traffic.

Zero downtime cluster/project movement

We can migrate K8S cluster / GCP Project
without any impact on latency.

New Stateful Pricing Service

Requirements

Based onhttps://tools.ietf.org/html/rfc2119

Must Have

- Read and Append API for PDL data exposed by Pricing
  - PDL can only be created by pricing
  - PDL components can never be updated only appended
- OMSs should communicate price changes back to pricing
- Ability to lookup PDL data from a given PDL ID

- Need to store Pricing PDL

New Stateful Pricing Service

- Get PDL (read)
  - Food: 300 qps * number of service types (=1) * number of payment types (=1) => 300qps
  - Transport: 350qps * number of service types (=1) * number of payment types (=2) => 700 qps
- Append PDL (write)
  - Food: 600 qps * number of service types (=1) * number of payment types (=5) => 3k qps
  - Transport: Need to be done (@Amit Kumar Rai - Jan 2022)
- Estimate API (write)
  - Food: 600 qps * number of service types (=1) * number of payment types (=1) => 600 qps
  - Transport: 350qps * number of service types (=10) * number of payment types (=2) => 7k qps

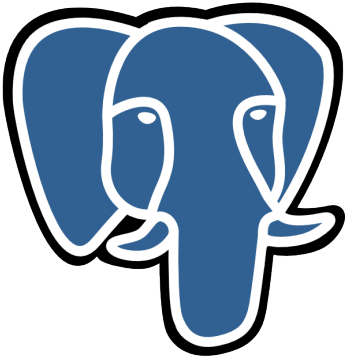
- Need to store Pricing PDL
- Writes $\Rightarrow$ 11,000+ per second
- Reads $\Rightarrow$ 1,000+ per second
- Writes $>$ Reads
  (by a factor of 11)

New Stateful Pricing Service

- Get PDL (read)
  - Food: 300 qps * number of service types (=1) * number of payment types (=1) => 300qps
  - Transport: 350qps * number of service types (=1) * number of payment types (=2) => 700 qps
- Append PDL (write)
  - Food: 600 qps * number of service types (=1) * number of payment types (=5) => 3k qps
  - Transport: Need to be done (@Amit Kumar Rai - Jan 2022)
- Estimate API (write)
  - Food: 600 qps * number of service types (=1) * number of payment types (=1) => 600 qps
  - Transport: 350qps * number of service types (=10) * number of payment types (=2) => 7k qps

- Need to store Pricing PDL
- Writes $\Rightarrow$ 11,000+ per second
- Reads $\Rightarrow$ 1,000+ per second
- Writes $>$ Reads
  (by a factor of 11)
- Ingestion: $\sim$400M keys per hr, $\sim$40GB data per hour
- QPS will continue to grow
- Each price will only be accessed $\sim$10 times over its lifetime
- We will never access this data after a few hours

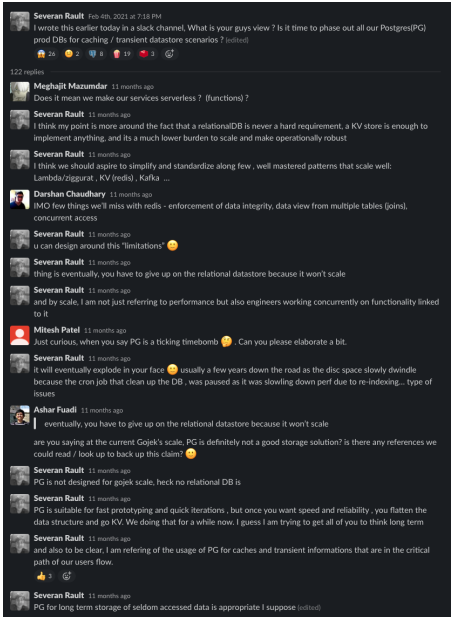
Stateful Pricing - Storage



PostgreSQL

- PostgreSQL+Redis for the win!

Stateful Pricing - Storage



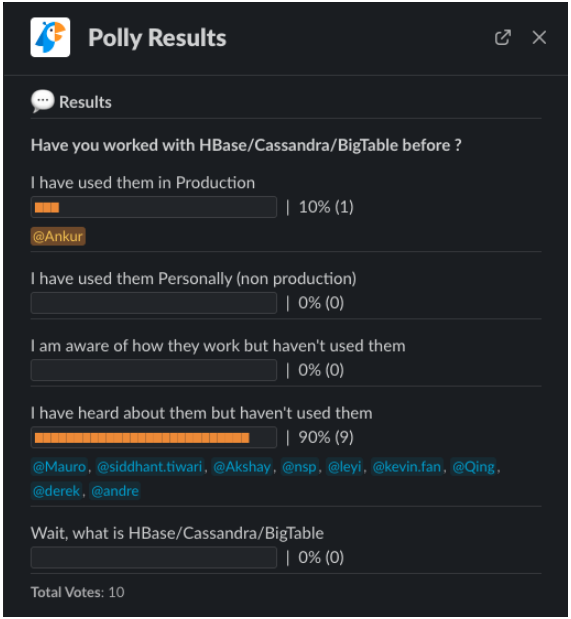
- ~~PostgreSQL+Redis for the win!~~

Stateful Pricing - Storage

```
$ bash misc/loadtest/grpc-write.sh
stateful-pricing-grpc-int.pricing.golabs.io
Summary:
  Count:   696028
  Total:   300.00 s
  Slowest: 659.14 ms
  Fastest: 8.95 ms
  Average: 20.99 ms
  Requests/sec:    2320.09
Response time histogram:
  8.953 [1]     |
  73.972 [690581]  |∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
  138.991 [4796]   |
  204.010 [333]    |
  269.028 [140]    |
  334.047 [71] |
  399.066 [44] |
  464.085 [0]  |
  529.104 [0]  |
  594.122 [0]  |
  659.141 [12] |
Latency distribution:
  10 % in 13.94 ms
  25 % in 15.55 ms
  50 % in 17.95 ms
  75 % in 21.85 ms
  90 % in 29.40 ms
  95 % in 40.77 ms
  99 % in 67.58 ms
Status code distribution:
  [OK]            695978 responses
  [Unavailable]   50 responses
Error distribution:
  [50]   rpc error: code = Unavailable desc = transport is closing
Cleaning up file based variables
Job succeeded
```

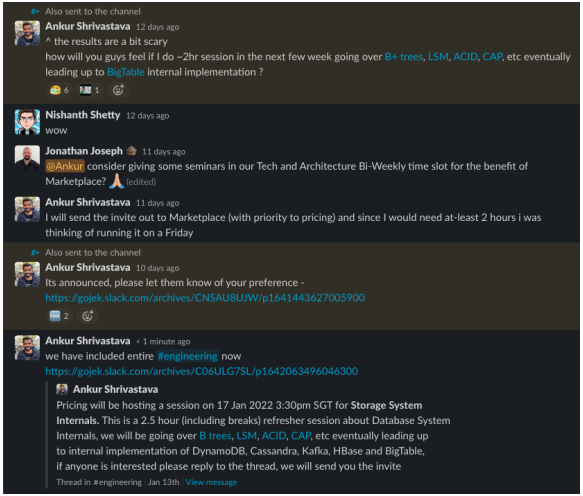
- ~~PostgreSQL+Redis for the win!~~
- BigTable might work better
- Lets build a service and load test (integration)

Stateful Pricing - Storage



- ~~PostgreSQL+Redis for the win!~~
- BigTable might work better
- Lets build a service and load test (integration)
- Lets ask if the rest of the team have used it

Stateful Pricing - Storage



- ~~PostgreSQL+Redis for the win!~~
- BigTable might work better
- Lets build a service and load test (integration)
- Lets ask if the rest of the team have used it
- That is a risk we need to mitigate

Before we begin

- We will Simplify things
- Real world is more nuanced

Before we begin

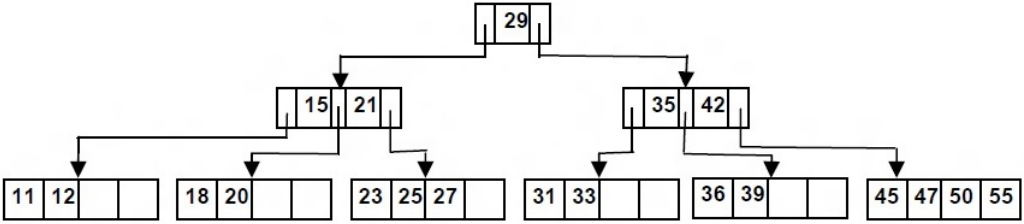
- We will Simplify things
- Real world is more nuanced
- After this talk you will be able to understand
  - How Databases differ from one another
  - Identify which DB works best for a given use case
  - How to identify possible bottlenecks

Before we begin

- We will Simplify things
- Real world is more nuanced
- After this talk you will be able to understand
  - How Databases differ from one another
  - Identify which DB works best for a given use case
  - How to identify possible bottlenecks
- You will still have to
  - Read the Manual/Documentation
  - Do your own load/failure tests

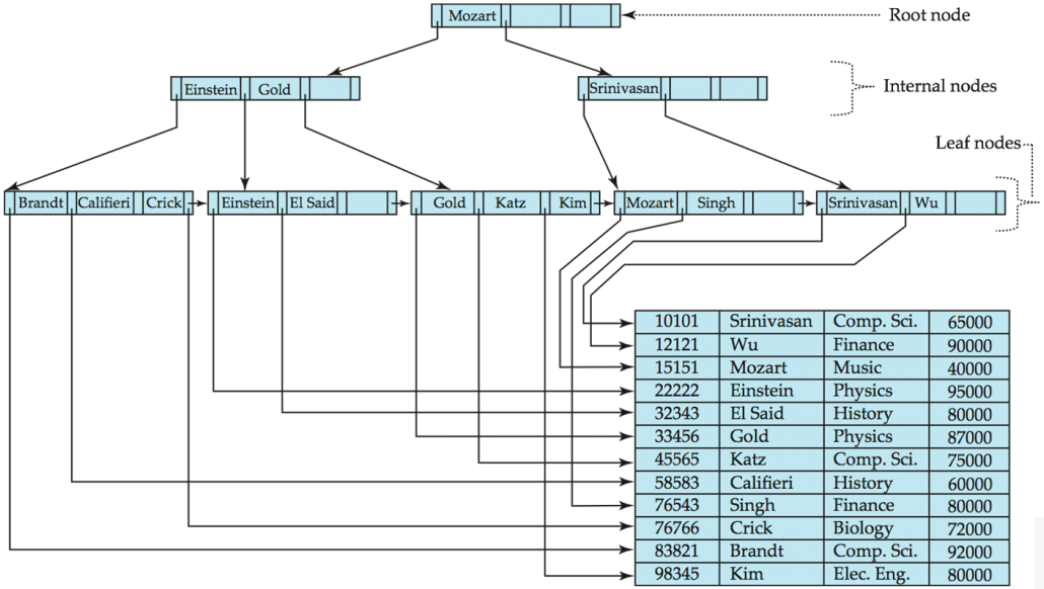
B−Tree

B−Trees

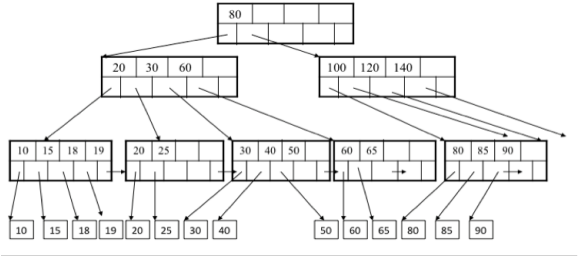


B-Tree of order 5

B+ Tree - index


Mozart
Root node
Einstein
Gold
Srinivasan
Internal nodes
Leaf nodes
Brandt
Califieri
Crick
Einstein
El Said
Gold
Katz
Kim
Mozart
Singh
Srinivasan
Wu
10101 Srinivasan Comp. Sci. 65000
12121 Wu Finance 90000
15151 Mozart Music 40000
22222 Einstein Physics 95000
32343 El Said History 80000
33456 Gold Physics 87000
45565 Katz Comp. Sci. 75000
58583 Califieri History 60000
76543 Singh Finance 80000
76766 Crick Biology 72000
83821 Brandt Comp. Sci. 92000
98345 Kim Elec. Eng. 80000

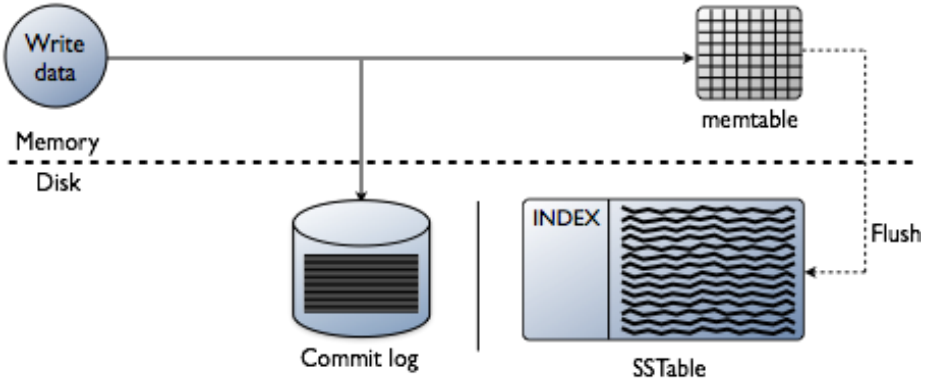
B−Trees



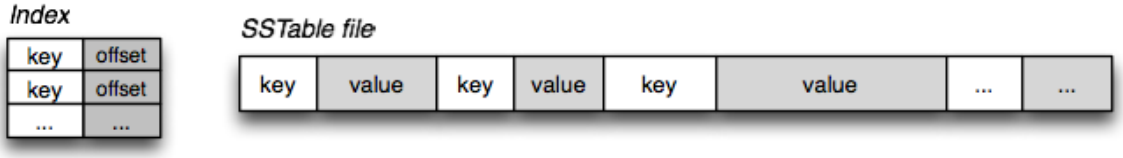
- B Tree variations ⇒ B+ Trees, B* Trees, etc
- Typical reads much higher than writes
- Keeps keys sorted for sequential traversing
- Uses a hierarchical index to minimize the number of disk reads
- Self-Balancing
- No additional maintenance step required
- No Mechanism to delete stale/expired data

Log Structured Merge (a.k.a LSM)

LSM - Write

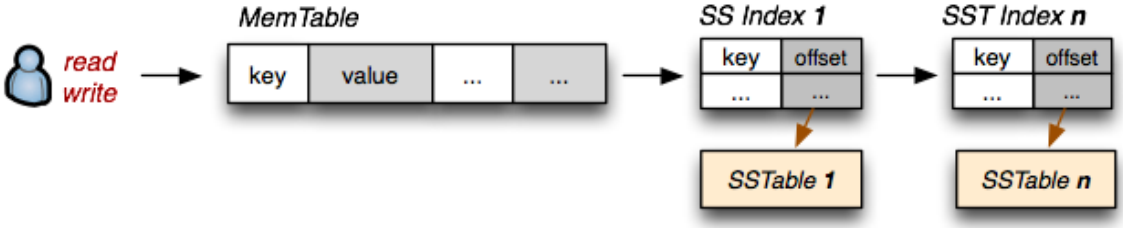

Write data
memtable
Memory
Disk
Flush
INDEX
Commit log
SSTable

LSM - SSTable



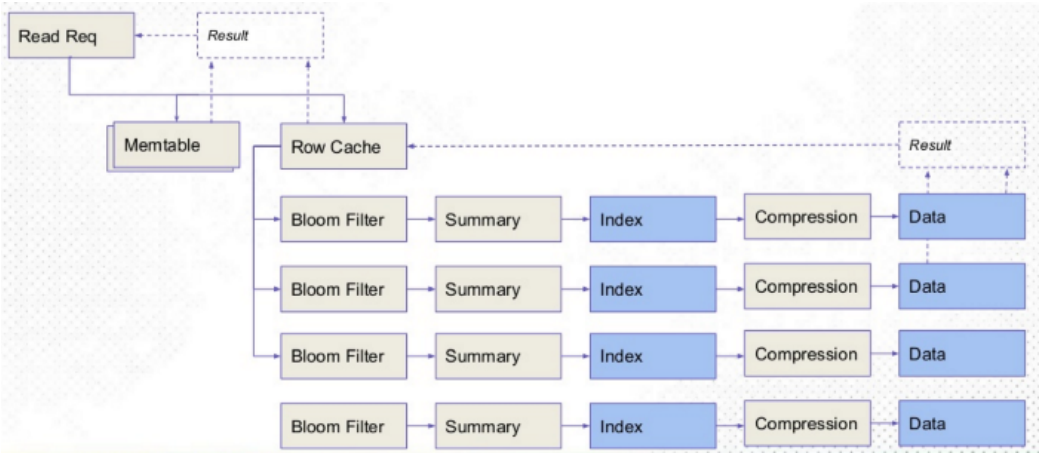
- Sorted String Table
- 2 sections - Index and Data
- SSTables also contain a Bloomfilter
- Append only storage
- Persisted on Disk

LSM - Memtable

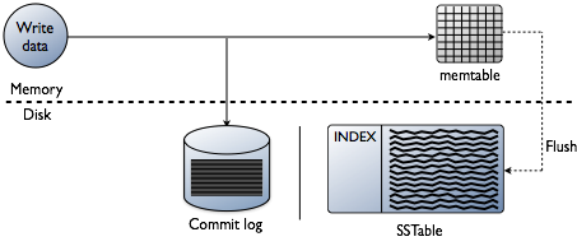


- Memtable helps speed up the requests
- Memtable is flushed to disk into SSTables (time/size)
- Allows Batching of writes to disk

LSM - Read

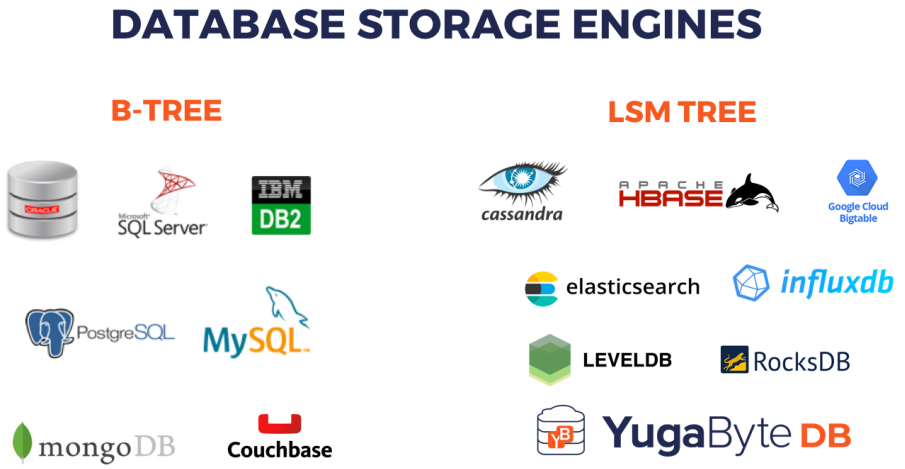

Read Req
Result
Memtable
Row Cache
Result
Bloom Filter
Summary
Index
Compression
Data
Bloom Filter
Summary
Index
Compression
Data
Bloom Filter
Summary
Index
Compression
Data
Bloom Filter
Summary
Index
Compression
Data

Log Structured Merge (LSM)

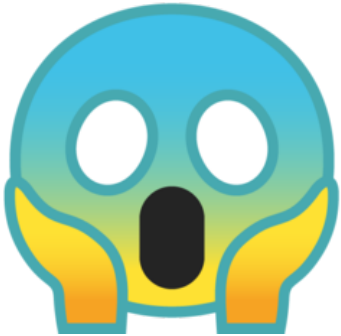


- Typical writes much higher than reads
- Deletion of values results in tombstones
- Updates across results in versions (by-design / memtable flush)
- Append only storage
- Extra maintenance required (SS table compaction)

What is used where?



Time for ACID Trip



ACID

ACID defines properties of a Transaction.

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Issues with Transaction

Primary focus is Isolation.

Issues with Transaction

Primary focus is Isolation.

- **Dirty Read** – Transaction reads data that's not committed.

Issues with Transaction

Primary focus is Isolation.

- **Dirty Read** – Transaction reads data that's not committed.
- **Non Repeatable Read** – Different values on multiple reads of same row.

Issues with Transaction

Primary focus is Isolation.

- **Dirty Read** – Transaction reads data that's not committed.
- **Non Repeatable Read** – Different values on multiple reads of same row.
- **Phantom Read** – Different number of rows returned when same query executed multiple times.

Isolation Levels

SQL defines 4 levels of Isolation.

Isolation Levels

SQL defines 4 levels of Isolation.

- **Read Uncommitted** – Transaction can read uncommitted data.
- **Read Committed** – Transaction only read committed data (Lock on current row).
- **Repeatable Read** – Transactions don't have *Non Repeatable Reads* (Locks on all referenced rows).
- **Serializable** – Concurrently executing Transaction appear to be serially executing (No defined order)

ACID – how does it come together?

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Let's put our CAP on!

CAP Theorem

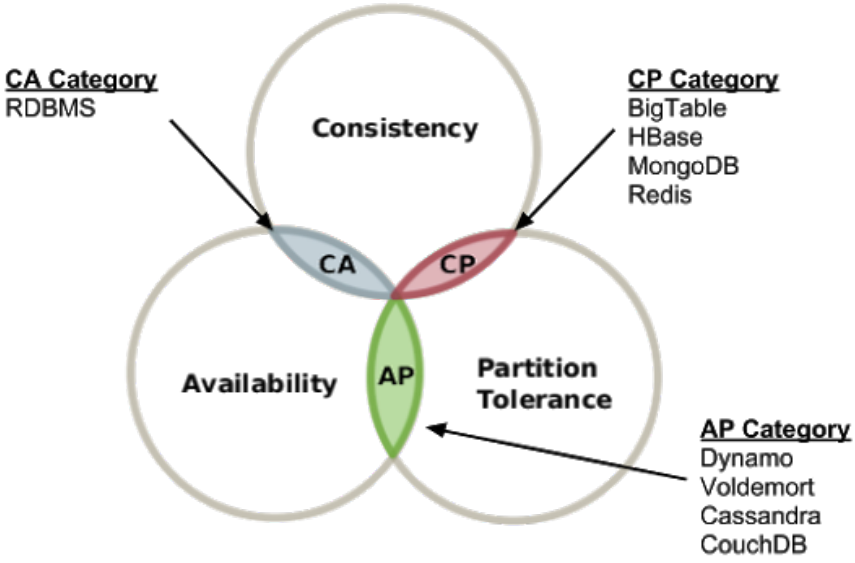
A **Distributed Database** can only provide two out of the following three guarantees.

CAP Theorem

A **Distributed Database** can only provide two out of the following three guarantees.

- **C**onsistency – Every read receives the most recent write or an error.
- **A**vailability – Every request receives a ($non - error$) response.
- **P**artition Tolerance – System continues to operate in case of network partition.

CAP Theorem


CA Category
RDBMS
Consistency
CP Category
BigTable
HBase
MongoDB
Redis
CA
CP
Availability
AP
Partition
Tolerance
AP Category
Dynamo
Voldemort
Cassandra
CouchDB

CAP Theorem Issues

- CAP Theorem is **only** valid for partition events.
- CAP Theorem does not define normal working.

CAP Theorem Issues

- CAP Theorem is **only** valid for partition events.
- CAP Theorem does not define normal working.
- PACELC Theorem goes one level extra.

PACELC Theorem

PACELC Theorem

- Extension of CAP Theorem.
- Describes normal case tradeoff between
  - Consistency
  - Latency

PACELC Theorem

- Extension of CAP Theorem.
- Describes normal case tradeoff between
  - Consistency
  - Latency
- **PAC** – When there is **P**artition a system is either **A**vailable or **C**onsistent.

PACELC Theorem

- Extension of CAP Theorem.
- Describes normal case tradeoff between
  - Consistency
  - Latency
- **PAC** – When there is **P**artition a system is either **A**vailable or **C**onsistent.
- **ELC** – **E**lse the system focuses on either **L**atency or **C**onsistency.

Where do modern DBs fit?

DDBS	P+A	P+C	E+L	E+C
DynamoDB	✓		✓[a]	
Cassandra	✓		✓[a]	
Cosmos DB		✓	✓[b]	
Couchbase		✓	✓	✓
Riak	✓		✓[a]	
VoltDB/H-Store		✓		✓
Megastore		✓		✓
BigTable/HBase		✓		✓
MySQL Cluster		✓		✓
MongoDB	✓			✓
PNUTS		✓	✓	
Hazelcast IMDG[8][6]	✓	✓	✓	✓
FaunaDB[9]		✓	✓	✓

↑ is default behaviour, some can be tuned

Sharding

Sharding

- Shard is partition of data.
- Each shard acts as the single source for this subset of data

Sharding

- Shard is partition of data.
- Each shard acts as the single source for this subset of data
- Each shard can be kept in different physical location.
- Allows databases to scale linearly.
- Multiple copies of same shard are normally kept on different servers.

Modulo Hashing

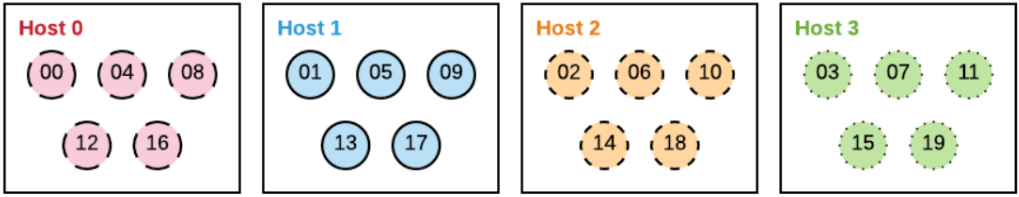
Modulo operation finds the remainder after division of one number by another

Modulo Hashing

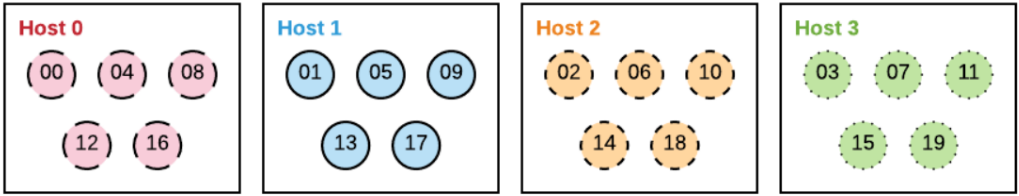
- Choose a hash function (Murmur3, xxHash)
- Hash the incoming key and take modulo
- Select the server based on that modulo

$$server := servers[hash(key)\%len(servers)]$$

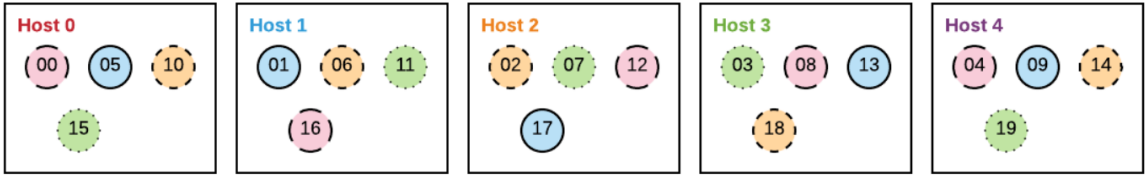
Problems with Modulo



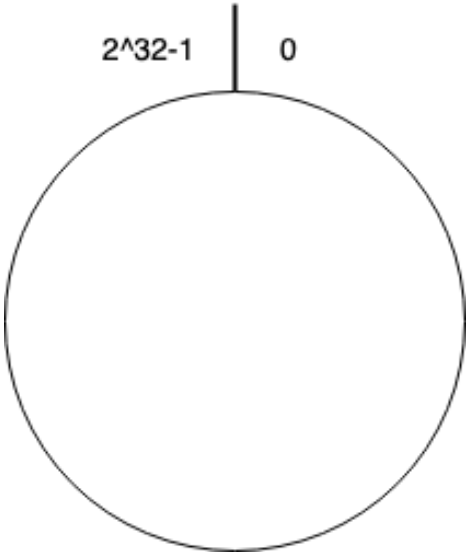
Problems with Modulo



On adding new server

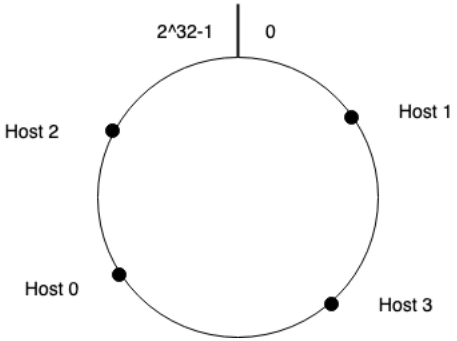


Consistent Hashing



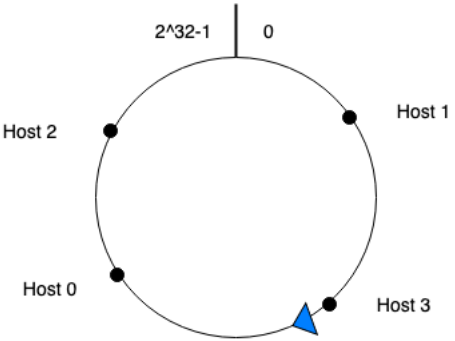
- Take key space from $0 \ldots 2^{32} - 1$
- Each point on circle becomes a shard (vbuckets)

Consistent Hashing



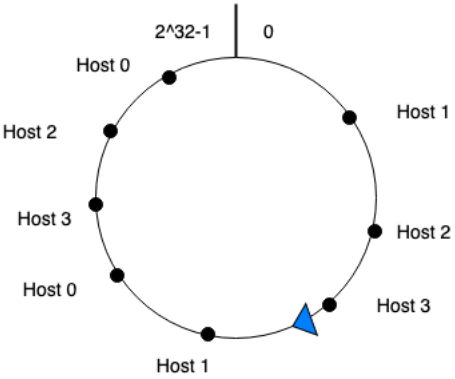
- Take key space from $0 \ldots 2^{32} - 1$
- Each point on circle becomes a shard (vbuckets)
- Hash and put each server on the circle.

Consistent Hashing



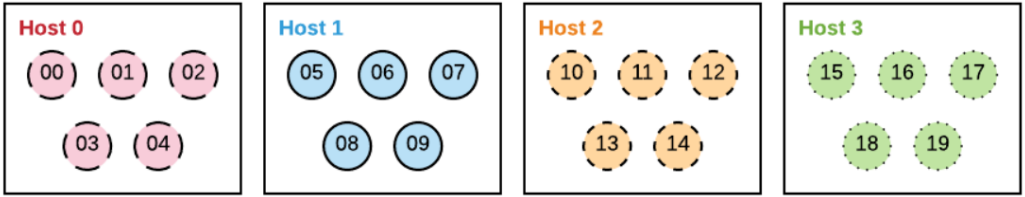
- Take key space from $0 \ldots 2^{32} - 1$
- Each point on circle becomes a shard (vbuckets)
- Hash and put each server on the circle.
- Host owns shard (vbuckets) starting from itself till the next host

Consistent Hashing

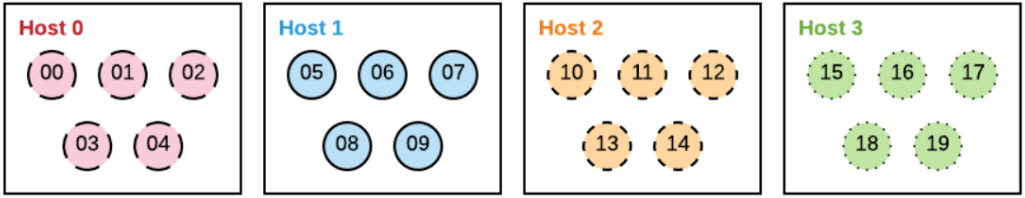


- Take key space from $0 \ldots 2^{32} - 1$
- Each point on circle becomes a shard (vbuckets)
- Hash and put each server on the circle.
- Host owns shard (vbuckets) starting from itself till the next host
- Do this multiple times

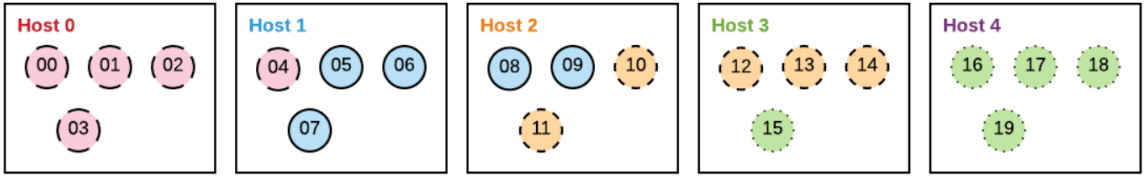
Using Consistent Hashing - ketama



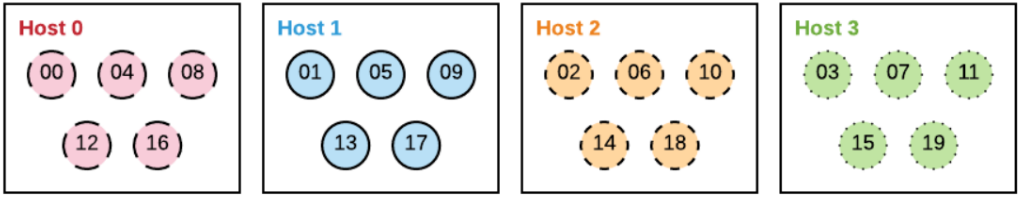
Using Consistent Hashing - ketama



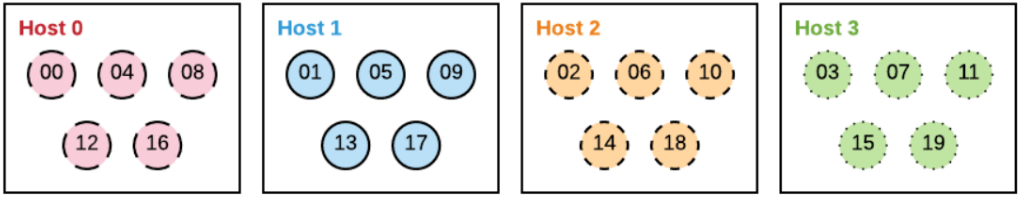
On adding new server



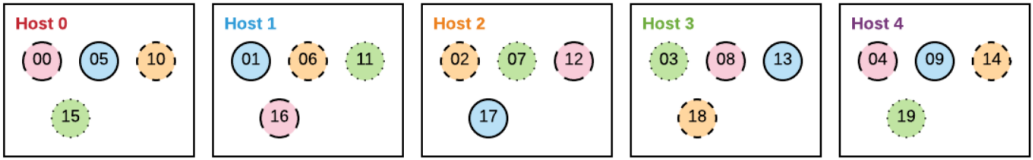
Initial ↓


Host 0
00
04
08
12
16
Host 1
01
05
09
13
17
Host 2
02
06
10
14
18
Host 3
03
07
11
15
19

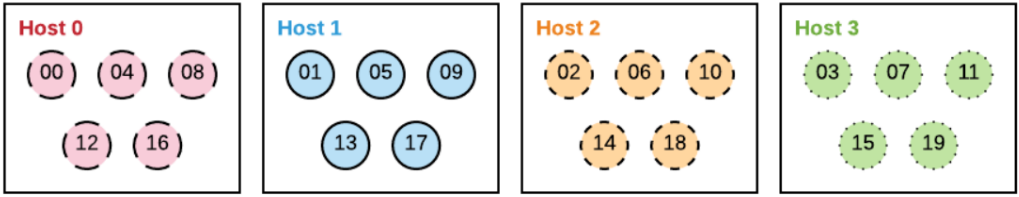
Initial ↓



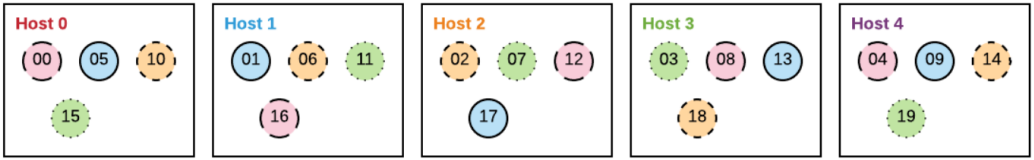
Modulo ↓



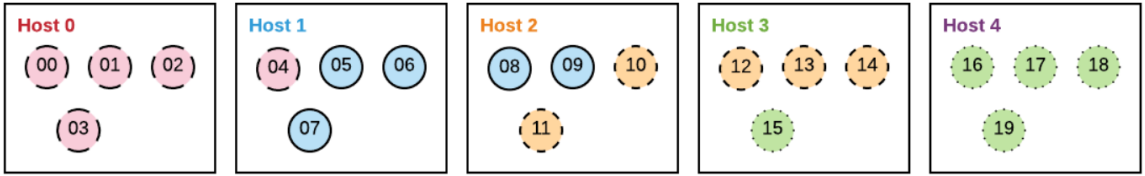
Initial ↓



Modulo ↓



Consistent (ketama) ↓

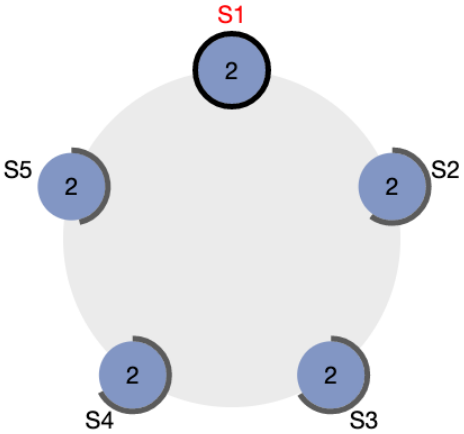


Quorums and Consensus Algorithms

Quorums

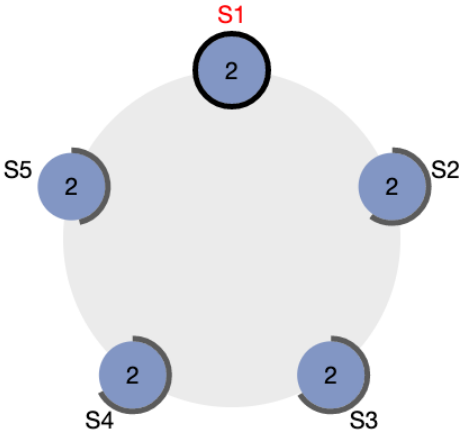
The minimum number of members of an assembly or society that must be present at any of its meetings to make the proceedings of that meeting valid.

Quorums



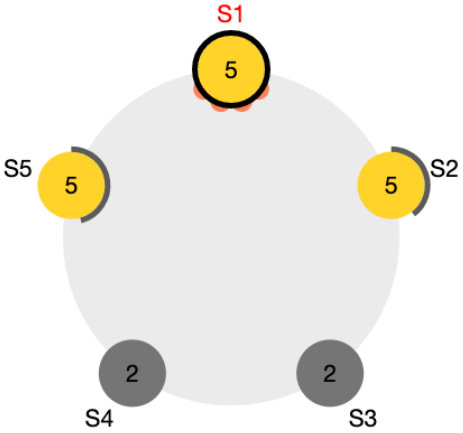
- Majority!

Quorums



- Majority!
- Defined as $\lceil \frac{n+1}{2} \rceil$ where n = number of nodes

Quorums



- Majority!
- Defined as $\lceil \frac{n+1}{2} \rceil$ where n = number of nodes
- Can work with failure of $n - \lceil \frac{n+1}{2} \rceil$ nodes

Consensus Algorithms

- Consensus is a fundamental problem in fault-tolerant distributed systems.
- Consensus involves multiple servers agreeing on values.

Consensus Algorithms

- Consensus is a fundamental problem in fault-tolerant distributed systems.
- Consensus involves multiple servers agreeing on values.
- Consensus Algorithms can help maintain Quorums.

Consensus Algorithms

- Consensus is a fundamental problem in fault-tolerant distributed systems.
- Consensus involves multiple servers agreeing on values.
- Consensus Algorithms can help maintain Quorums.
- Some of the most popular Consensus Algorithms are
  - Paxos
  - Raft

Consensus Algorithms

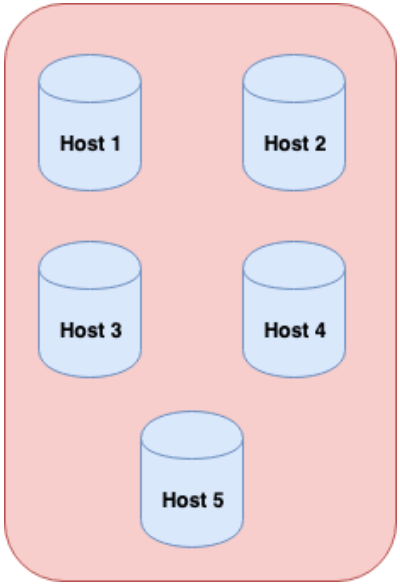
- Consensus is a fundamental problem in fault-tolerant distributed systems.
- Consensus involves multiple servers agreeing on values.
- Consensus Algorithms can help maintain Quorums.
- Some of the most popular Consensus Algorithms are
  - Paxos
  - Raft
- Zookeeper Atomic Broadcast Protocol (ZAB) another Consensus Algorithm but uses Two Phase commit.

How can nodes maintain data consistency without complex consensus algorithms?

How do multiple nodes maintain data consistency?

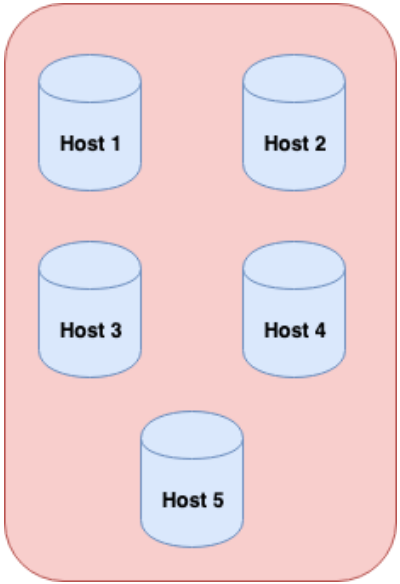
$$R + W > N$$

How do multiple nodes maintain data consistency?



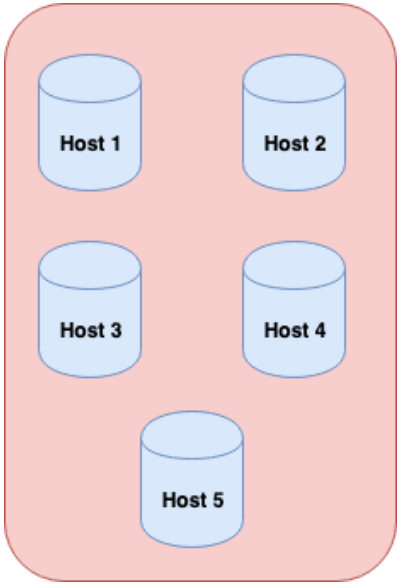
- Reads Ack + Writes Ack > Total Number of Replicas

How do multiple nodes maintain data consistency?



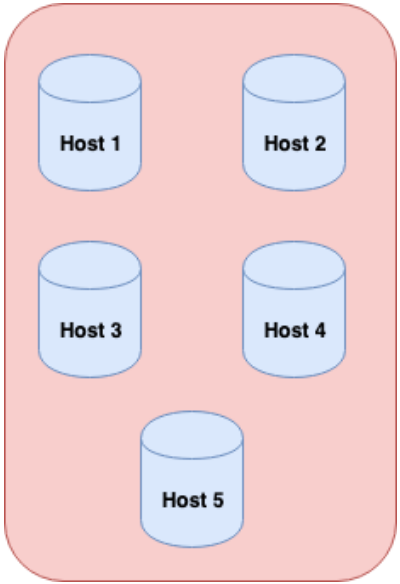
- Reads Ack + Writes Ack > Total Number of Replicas
- Lets walk through some example
- R:1 + W:5 > N:5

How do multiple nodes maintain data consistency?



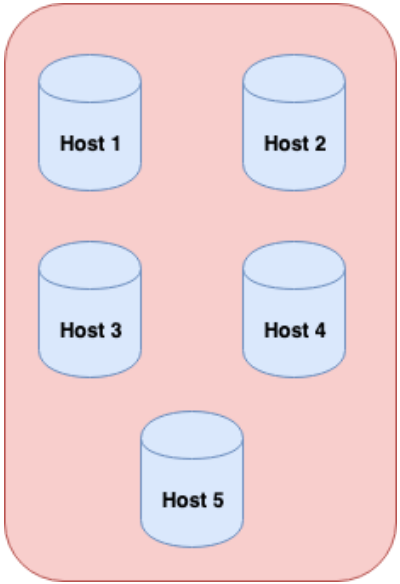
- Reads Ack + Writes Ack > Total Number of Replicas
- Lets walk through some example
- R:1 + W:5 > N:5
- R:5 + W:1 > N:5

How do multiple nodes maintain data consistency?



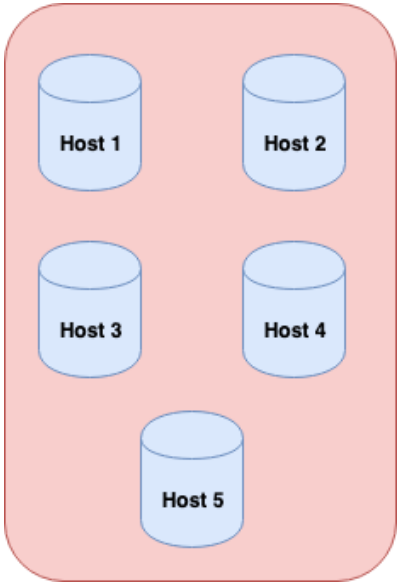
- Reads Ack + Writes Ack > Total Number of Replicas
- Lets walk through some example
- R:1 + W:5 > N:5
- R:5 + W:1 > N:5 (read repair)
- R:3 + W:3 > N:5

How do multiple nodes maintain data consistency?



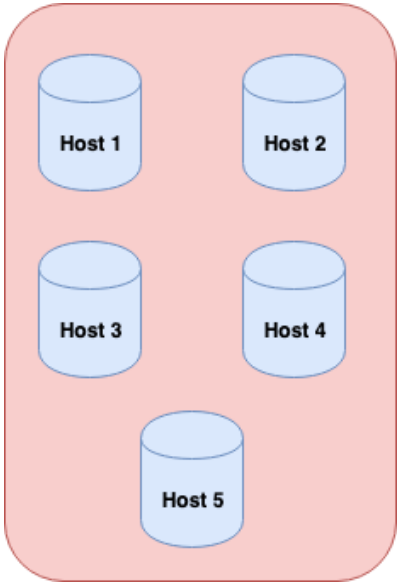
- Reads Ack + Writes Ack > Total Number of Replicas
- Lets walk through some example
- R:1 + W:5 > N:5
- R:5 + W:1 > N:5 (read repair)
- R:3 + W:3 > N:5
- R:2 + W:3 = N:5 ??

How do multiple nodes maintain data consistency?



- Reads Ack + Writes Ack > Total Number of Replicas
- Lets walk through some example
- R:1 + W:5 > N:5
- R:5 + W:1 > N:5 (read repair)
- R:3 + W:3 > N:5
- R:2 + W:3 = N:5 ??
  (eventual consistency)

How do multiple nodes maintain data consistency?



- Reads Ack + Writes Ack > Total Number of Replicas
- Lets walk through some example
- R:1 + W:5 > N:5
- R:5 + W:1 > N:5 (read repair)
- R:3 + W:3 > N:5
- R:2 + W:3 = N:5 ??
  (eventual consistency)
- R:2 + W:2 < N:5

How do multiple nodes maintain data consistency?

$$R + W > N \text{ (Strong)}$$
$$R + W \leq N \text{ (Eventual)}$$


cassandra

Cassandra

- LSM Based

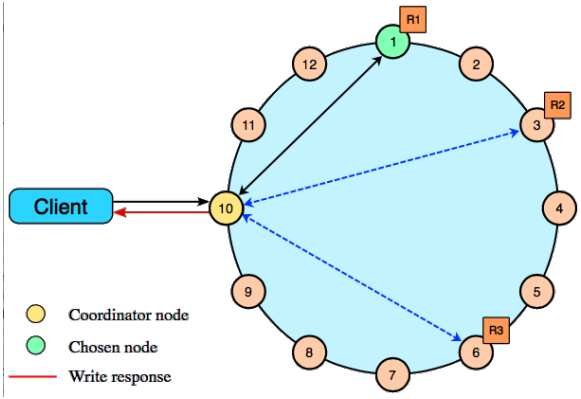
Cassandra

- LSM Based
- Tunable Consistency (request time)
  - ONE / LOCAL_ONE / TWO / THREE
  - QUORUM / LOCAL_QUROM / EACH_QUROM (only for writes)
  - ANY (only for writes), ALL ...

Cassandra

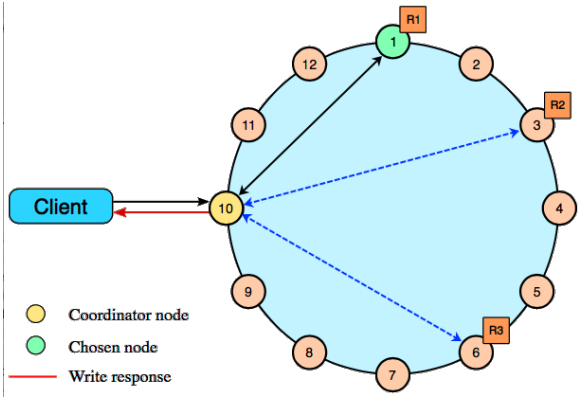
- LSM Based
- Tunable Consistency (request time)
  - ONE / LOCAL_ONE / TWO / THREE
  - QUORUM / LOCAL_QUROM / EACH_QUROM (only for writes)
  - ANY (only for writes), ALL ...
- Uses repair mechanism to fix inconsistency
  - hinted handoff (node is down)
  - read repair (fix data when reading)
  - anti-entropy node repair (manual/background repair)
- Consistent Hashing
- Built-in support for multi-master multi-DC clusters
- Gossip for cluster membership

Cassandra - ONE Write (W:1 N:3)



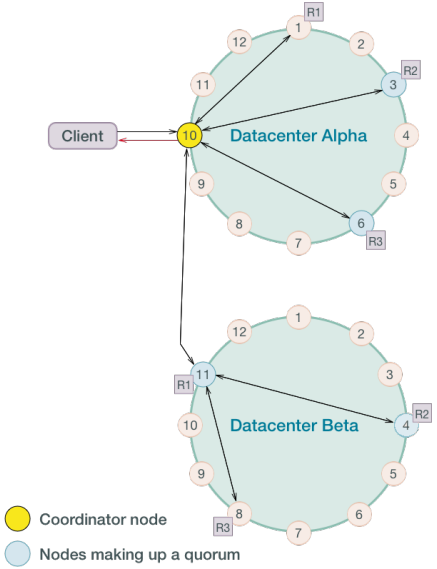
- Writes can go to any node
- Node receiving the request becomes Coordinator

Cassandra - ONE Write (W:1 N:3)



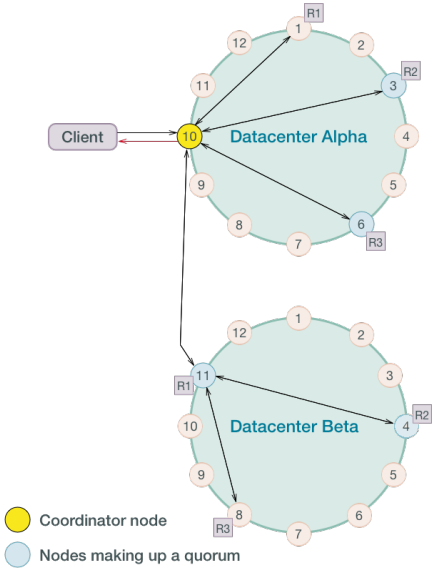
- Writes can go to any node
- Node receiving the request becomes Coordinator
- Coordinator writes to all replicas
- When one replica write is successful, user gets a success
- Num Acks depends on Consistency level
  - ONE / LOCAL_ONE / TWO / THREE
  - QUORUM / LOCAL_QUROM / EACH_QUROM (only for writes)
  - ANY (only for writes), ALL ...

Cassandra - Multi DC Write



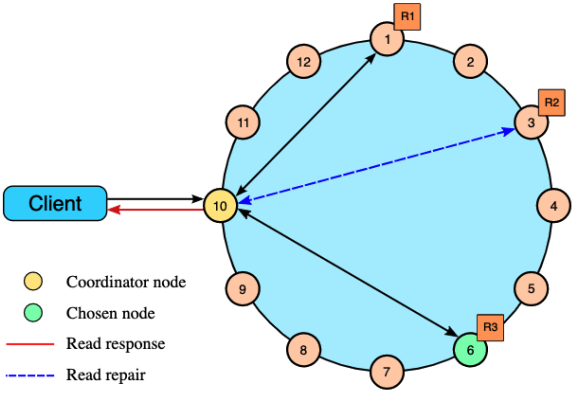
- Writes can go to any node
- Node receiving the request becomes Coordinator

Cassandra - Multi DC Write



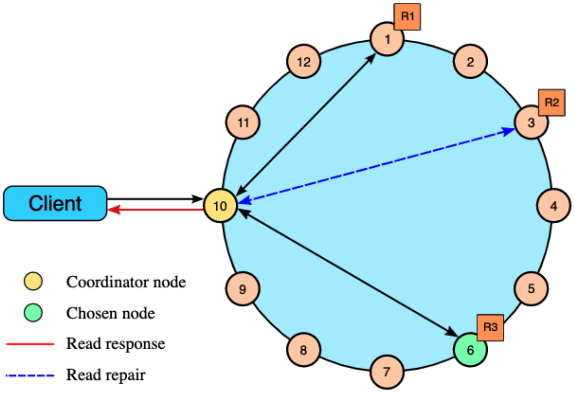
- Writes can go to any node
- Node receiving the request becomes Coordinator
- Coordinator writes to all replicas
- When one replica write is successful, user gets a success
- Num Acks depends on Consistency level
  - ONE / LOCAL_ONE / TWO / THREE
  - QUORUM / LOCAL_QUROM / EACH_QUROM (only for writes)
  - ANY (only for writes), ALL ...

Cassandra - QUORUM Read (R:2 N:3)



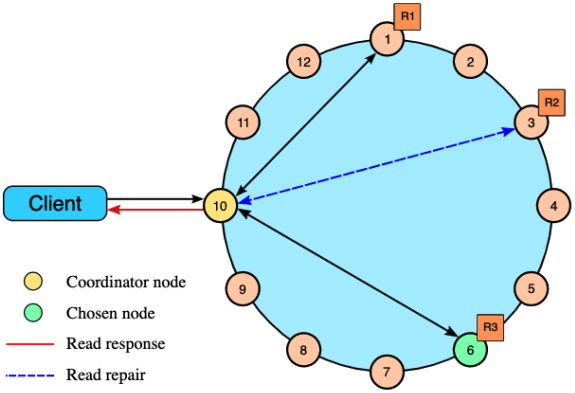
- Reads can go to any node
- Node receiving the request becomes Coordinator

Cassandra - QUORUM Read (R:2 N:3)



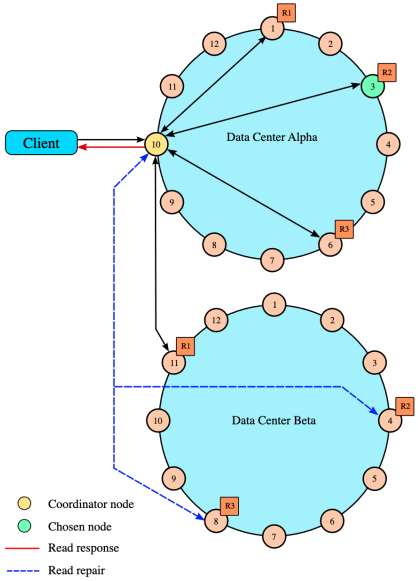
- Reads can go to any node
- Node receiving the request becomes Coordinator
- Coordinator reads from all replicas
- When Quorum (2) replicas return up to date values, user get a response

Cassandra - QUORUM Read (R:2 N:3)



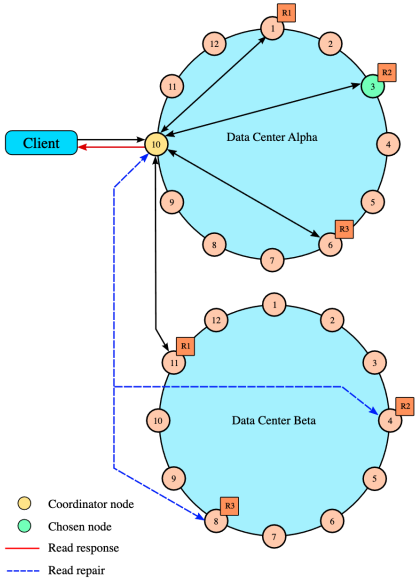
- Reads can go to any node
- Node receiving the request becomes Coordinator
- Coordinator reads from all replicas
- When Quorum (2) replicas return up to date values, user get a response
- When replicas have different versions of the row, the replica with the most recent version will return the requested data
- a read repair may be initiated for the disagreeing replicas
- Last Write Wins (timestamp)

Cassandra - Multi-DC QUORUM Read (R:4 N:6)



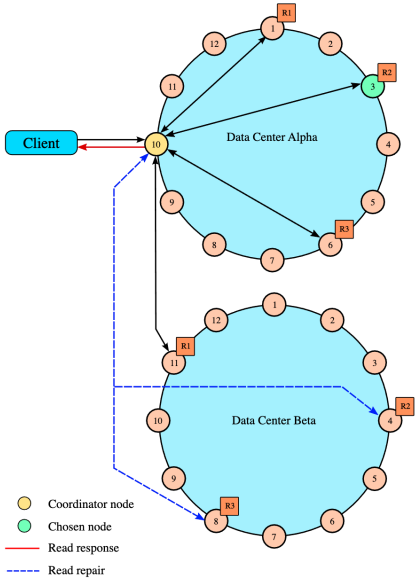
- Reads can go to nodes in any DC
- Node receiving the request becomes Coordinator

Cassandra - Multi-DC QUORUM Read (R:4 N:6)



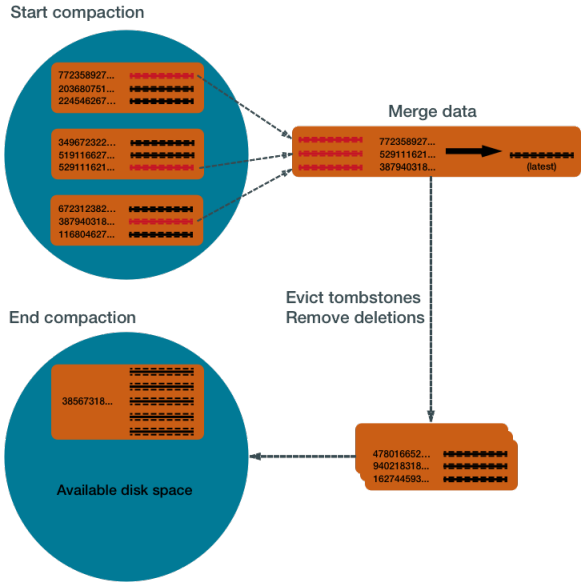
- Reads can go to nodes in any DC
- Node receiving the request becomes Coordinator
- Coordinator reads from all replicas across DC
- When Quorum (4 out of 6 total) replicas return up to date values, user get a response

Cassandra - Multi-DC QUORUM Read (R:4 N:6)



- Reads can go to nodes in any DC
- Node receiving the request becomes Coordinator
- Coordinator reads from all replicas across DC
- When Quorum (4 out of 6 total) replicas return up to date values, user get a response
- When replicas have different versions of the row, the replica with the most recent version will return the requested data
- a read repair may be initiated for the disagreeing replicas
- Last Write Wins (timestamp)

Cassandra - Compaction



- Deletes / Updates can result in multiple versions of data in SSTables
- SSTables are compacted based on defined strategy
  - Size-Tiered Compaction (STCS)
  - Date-Tiered Compaction(DTCS)
  - Time-Window Compaction(TWCS)
  - Leveled Compaction (LCS)
- Compaction takes care of clean up as well (TTL)

Cassandra

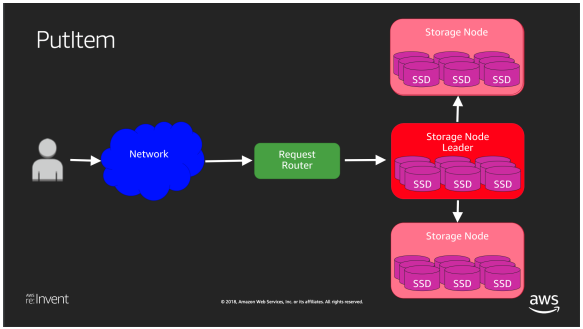
For more please refer to the Excellent Cassandra Documentation!
https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlIntro.html

DynamoDB

amazon web services™

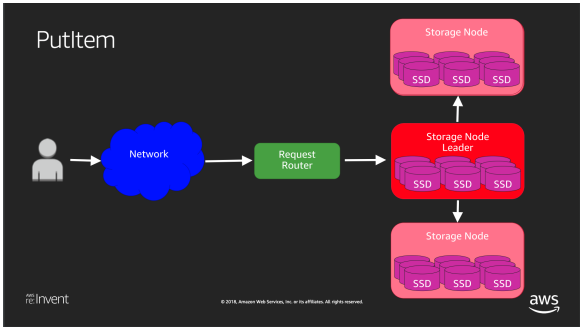
gojek

DynamoDB – Writes



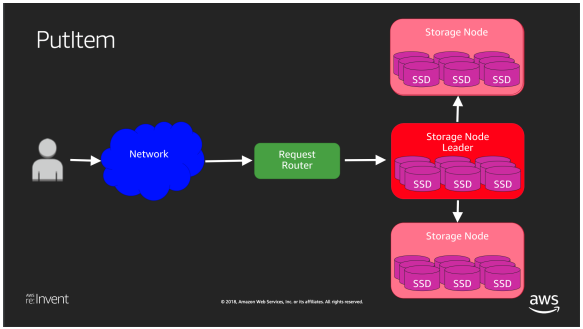
- B-tree based structure with replication log.

DynamoDB – Writes



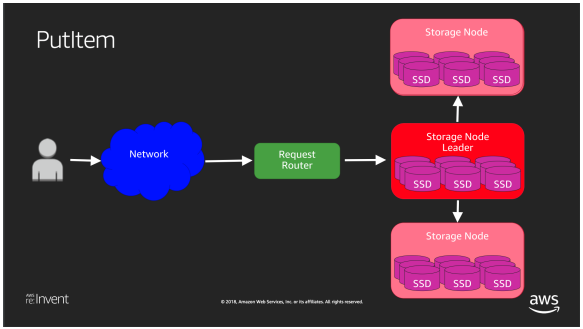
- B-tree based structure with replication log.
- Requests flow through a request router

DynamoDB – Writes



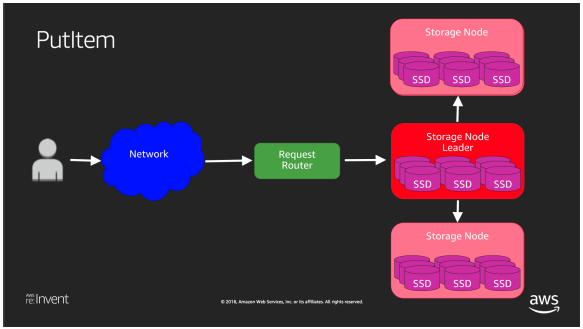
- B-tree based structure with replication log.
- Requests flow through a request router
- Consistent Hashing for partitioning

DynamoDB – Writes



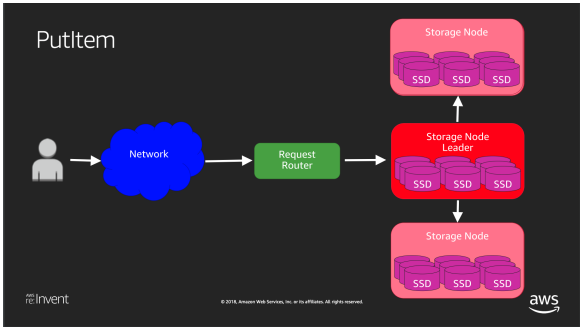
- B-tree based structure with replication log.
- Requests flow through a request router
- Consistent Hashing for partitioning
- Paxos for Leader Election.

DynamoDB – Writes



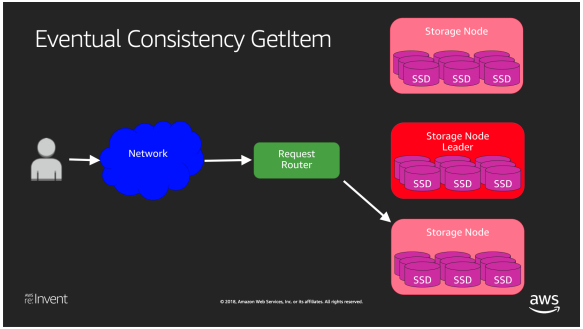
- B-tree based structure with replication log.
- Requests flow through a request router
- Consistent Hashing for partitioning
- Paxos for Leader Election.
- Num Acks of 2
- Replica Count of 3
- Replicas across AZ

DynamoDB – Writes



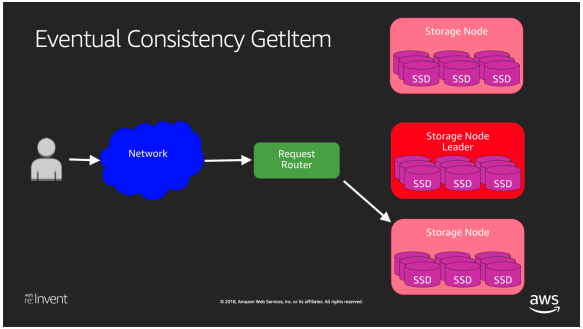
- B-tree based structure with replication log.
- Requests flow through a request router
- Consistent Hashing for partitioning
- Paxos for Leader Election.
- Num Acks of 2
- Replica Count of 3
- Replicas across AZ
- HeartBeat for peer failure detection.

DynamoDB – Reads



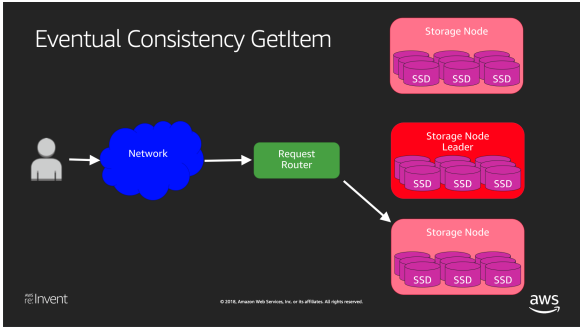
- Consistent Hashing for partitioning

DynamoDB – Reads



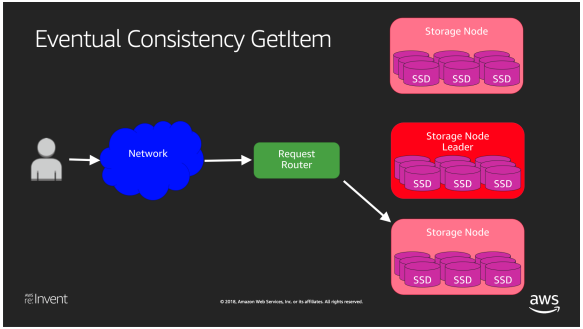
- Consistent Hashing for partitioning
- Read from single replica.

DynamoDB – Reads



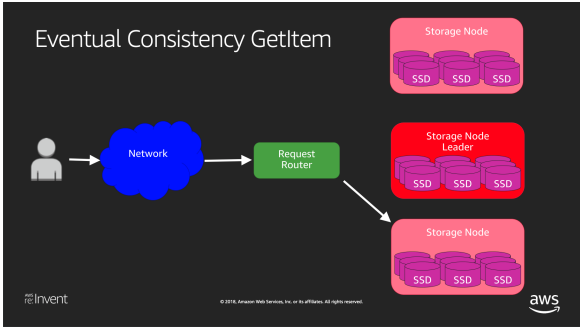
- Consistent Hashing for partitioning
- Read from single replica.
- $R = 1$, $W = 2$, $N = 3$
- $R + W == N$ (eventual consistency)

DynamoDB – Reads



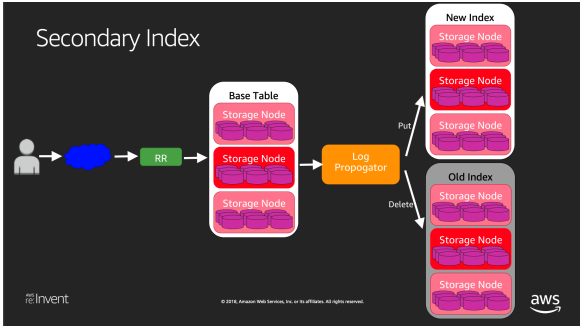
- Consistent Hashing for partitioning
- Read from single replica.
- $R = 1$, $W = 2$, $N = 3$
- $R + W == N$ (eventual consistency)
- Reads by default are eventually consistent (flag available in API – ConsistentRead)

DynamoDB – Reads



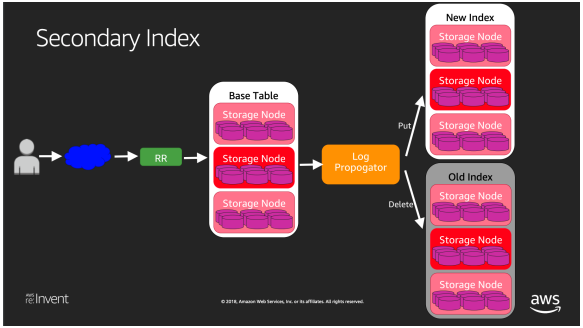
- Consistent Hashing for partitioning
- Read from single replica.
- $R = 1$, $W = 2$, $N = 3$
- $R + W == N$ (eventual consistency)
- Reads by default are eventually consistent (flag available in API – ConsistentRead)
- In case of conflicts use Vector Clocks (Last Write Wins / Custom)

DynamoDB – Secondary Index



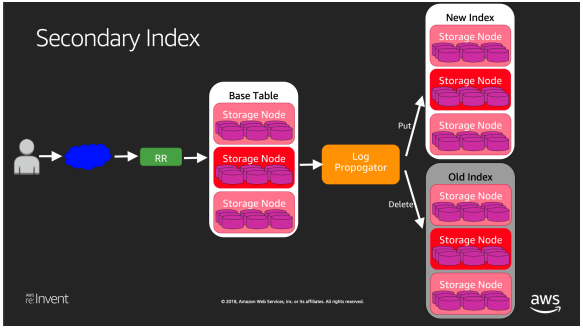
- Independent Table.

DynamoDB – Secondary Index



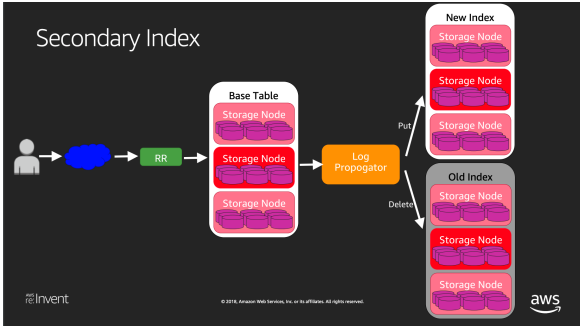
- Independent Table.
- Consistent Hashing for partitioning

DynamoDB – Secondary Index



- Independent Table.
- Consistent Hashing for partitioning
- Build from replication log.

DynamoDB – Secondary Index



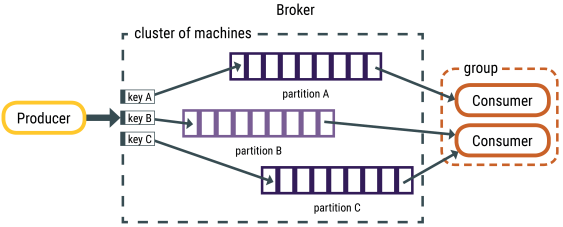
- Independent Table.
- Consistent Hashing for partitioning
- Build from replication log.
- Write Amplification (Specially for updates)

DynamoDB

For more please refer to **DAT321** talk from AWS re:Invent!
https://www.youtube.com/watch?v=yvBR71D0nAQ

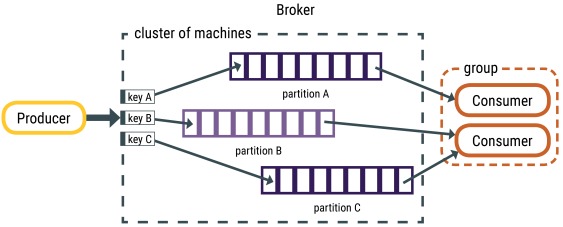

kafka

Kafka



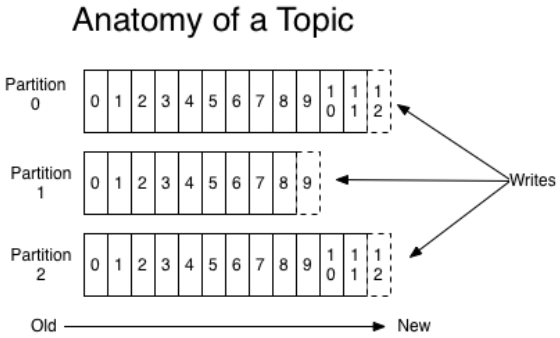
- Uses Replicated Logs
- Configurable replica count (topic creation / updatable)
- Configuration partition (topic creation / updatable)
- Configurable write Acknowledge (producer configuration)

Kafka



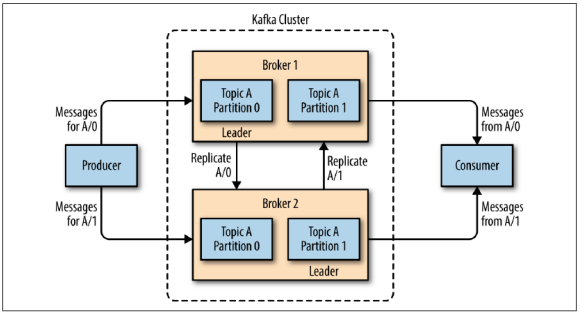
- Uses Replicated Logs
- Configurable replica count (topic creation / updatable)
- Configuration partition (topic creation / updatable)
- Configurable write Acknowledge (producer configuration)
- Typically topic has a retention value (TTL)
- We can store values indefinitely by using Log Compaction.

Kafka - Topics



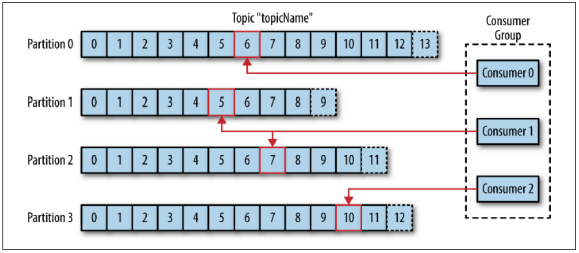
- Each Topic (Queue) comprises of multiple Partitions

Kafka - Topics



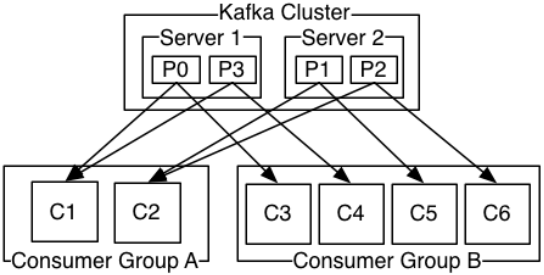
- Each Topic (Queue) comprises of multiple Partitions
- Each Broker is responsible for multiple partitions.
- Each partition has 1 leader
- Leader election (Zookeeper/raft quorum)
- All reads/writes go to the leader
- Non leader replicate data from leader

Kafka - Topics



- Number of Consumers $\leq$ Number of Partitions
- Ordering guarantees apply only within the partition

Kafka - Topics



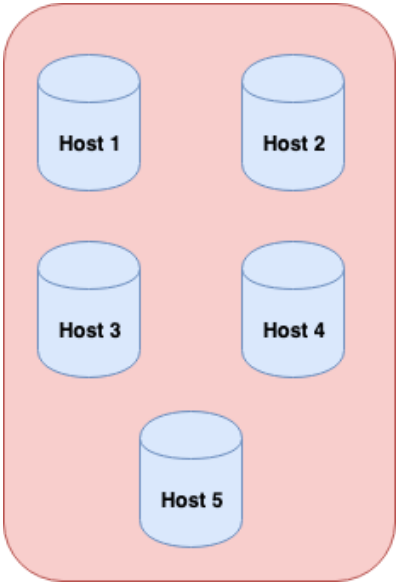
- Number of Consumers $\leq$ Number of Partitions
- Ordering guarantees apply only within the partition
- Multiple consumer groups can read same topic

Kafka

For more please refer to the Excellent Kafka Documentation!
https://kafka.apache.org/documentation/#design

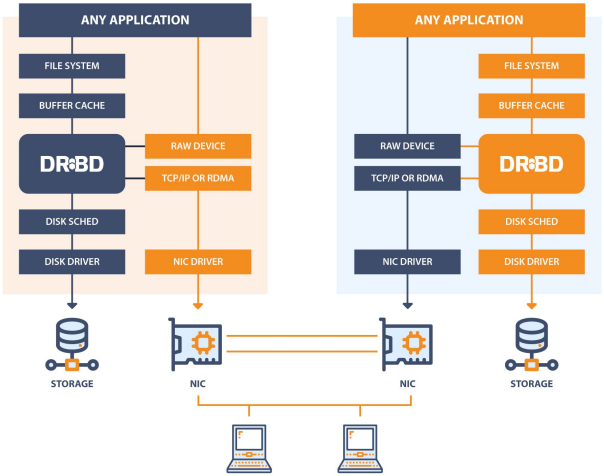
Is there another way multiple nodes can maintain data consistency?

Replicated Storage



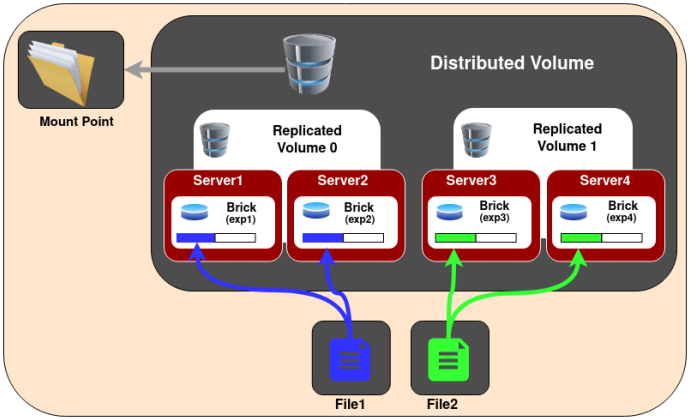
- Replication of writes is needed for data consistency
- Can we replicate stored data instead ?

Replicated Storage



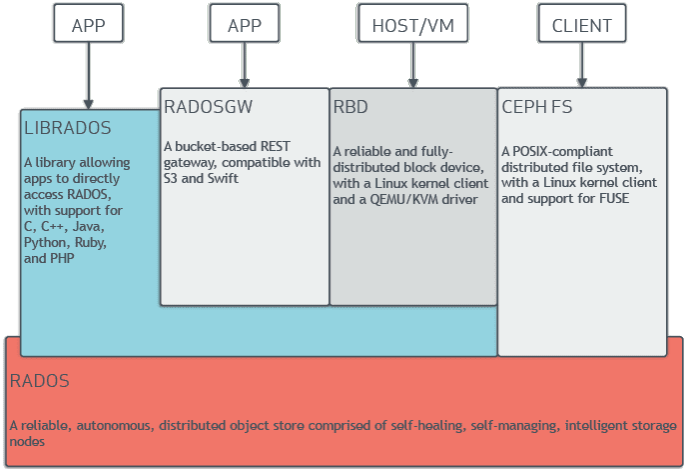
- Replication of writes is needed for data consistency
- Can we replicate stored data instead ?
- DRBD

Replicated Storage



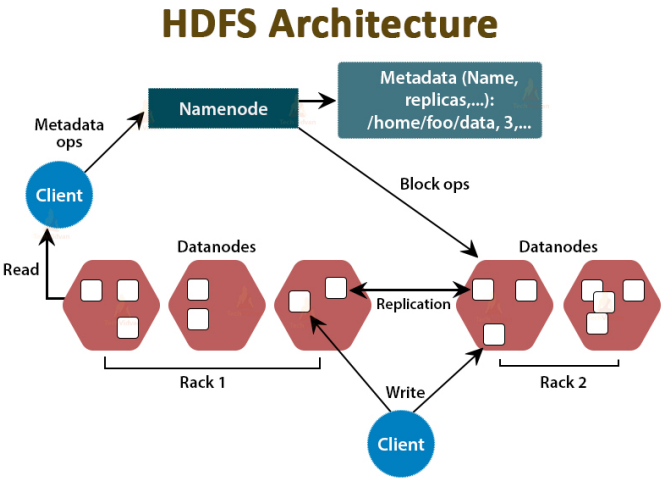
- Replication of writes is needed for data consistency
- Can we replicate stored data instead ?
- DRBD
- GlusterFS

Replicated Storage



- Replication of writes is needed for data consistency
- Can we replicate stored data instead ?
- DRBD
- GlusterFS
- Ceph.io

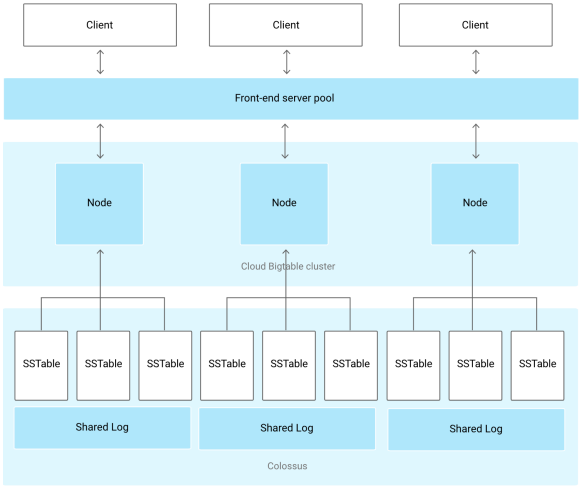
Replicated Storage



- Replication of writes is needed for data consistency
- Can we replicate stored data instead ?
- DRBD
- GlusterFS
- Ceph.io
- HDFS

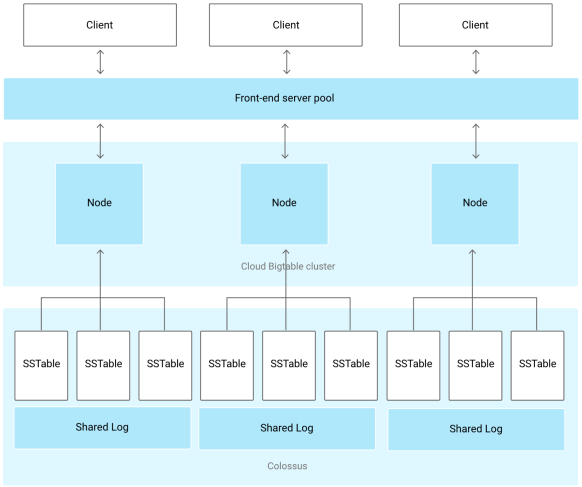
BigTable

BigTable



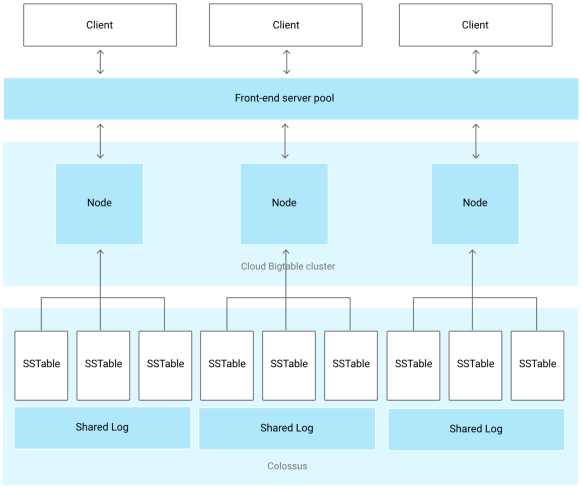
- LSM Based
- Uses dedicated request routers

BigTable



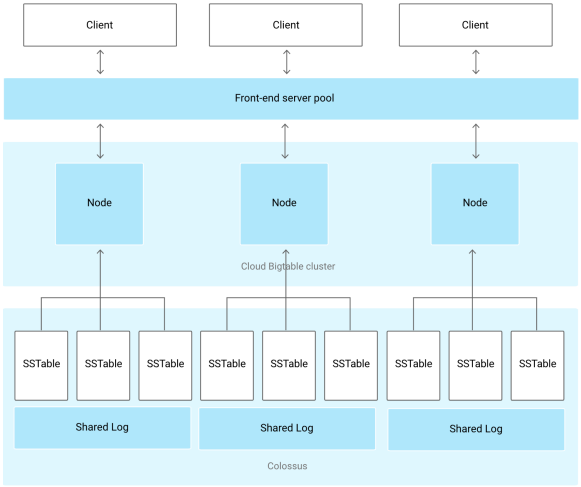
- LSM Based
- Uses dedicated request routers
- Key ranges are partitioned across nodes (Tablet)
- Each Node is responsible for multiple Tablets
- No replica Nodes/Tablets (strong consistency)

BigTable



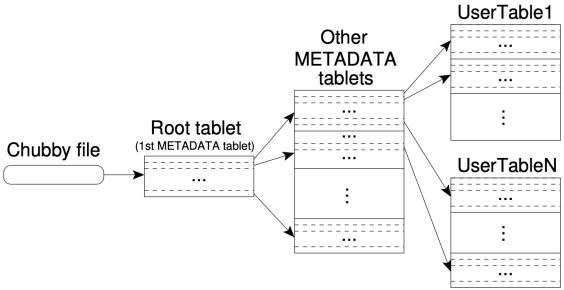
- LSM Based
- Uses dedicated request routers
- Key ranges are partitioned across nodes (Tablet)
- Each Node is responsible for multiple Tablets
- No replica Nodes/Tablets (strong consistency)
- Clusters in single AZ
- multi-cluster can be AZ across regions (multi-master)
- Uses Google's Internal distributed file system - Colossus (replaces GFS)

BigTable - Master



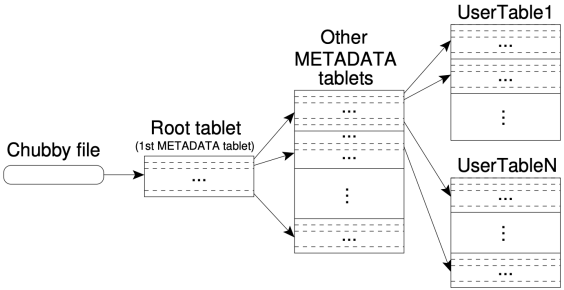
- Bigtable master manages nodes/tablets (does not serve traffic)
- BigTable Master is responsible for
  - Tablet assignment to server
  - detecting the addition and expiration of tablet
  - balancing tablet-server load
  - garbage collection of files
  - table and column family creations

BigTable - Tablets



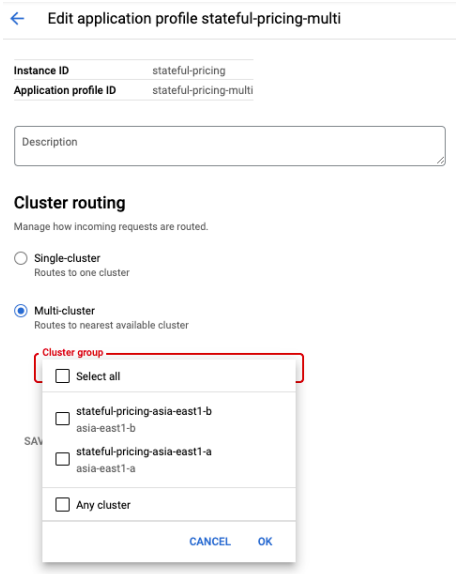
- Multiple METADATA tablets (B+ Tree style)
- Tablets contain associated row range (e.g. aardvark. . .apple)
- Multiple tablets may map to a single SSTable

BigTable - Tablets



- Multiple METADATA tablets (B+ Tree style)
- Tablets contain associated row range (e.g. aardvark. . .apple)
- Multiple tablets may map to a single SSTable
- Initially, each table consists of just one tablet.
- As a table grows, it is split into multiple tablets
- Each tablet is 100-200 MB in size by default.
- tablets can be pre-split at table creation

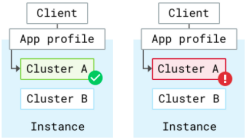
BigTable - App Profile



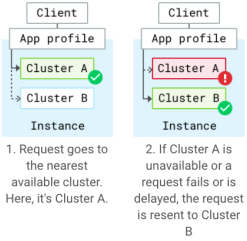
- Application profile defines how incoming requests are routed to clusters

BigTable - App Profile

Single-cluster



Multi-cluster



- Application profile defines how incoming requests are routed to clusters
- Single-cluster profiles always route to the selected cluster
- Multi-cluster profiles can route to any available cluster

BigTable - App Profile

Routing comparison

Single-cluster routing	Multi-cluster routing
Routes to one cluster	Routes to nearest available cluster
⚠ Manual failover	Automatic failover
Helps isolate CPU-intensive loads	Higher availability
Read-your-writes consistency	⚠ No read-your-writes consistency
Single-row transactions supported	⚠ No single-row transactions

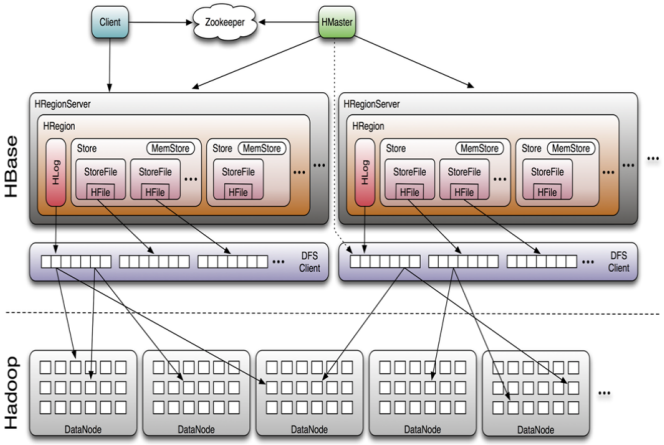
- Application profile defines how incoming requests are routed to clusters
- Single-cluster profiles always route to the selected cluster
- Multi-cluster profiles can route to any available cluster
- Single-cluster profiles unlock strong consistency
- Multi-cluster profiles have higher availability

BigTable

Read the BigTable Paper at https://research.google/pubs/pub27898/
You can also read the BigTable Documentation!
https://cloud.google.com/bigtable/docs/overview

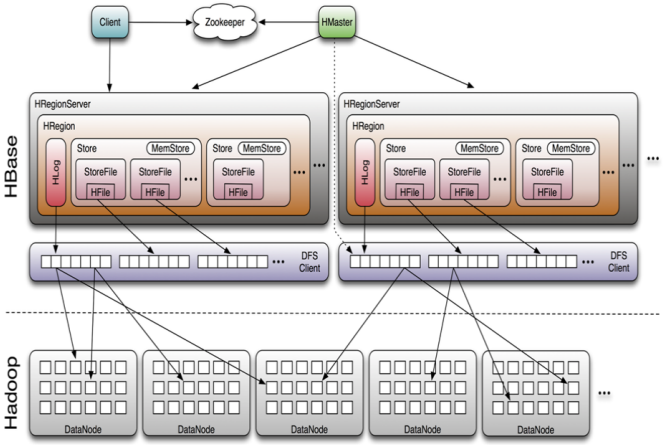
APACHE
HBASE

HBase



- Inspired by BigTable Paper
- LSM Based
- Uses HDFS for storage
- Open Source

HBase



- Inspired by BigTable Paper
- LSM Based
- Uses HDFS for storage
- Open Source
- HMaster - similar to BigTable master node
- RegionServer - similar to BigTable node, responsible for multiple regions
- Region - similar to BigTable tablet

HBase

For more please refer to the Excellent HBase Documentation!
https://hbase.apache.org/book.html

Q&A

Pricing is hiring L2/L3/L4
Internal Mobility and Referrals