Apache Cassandra

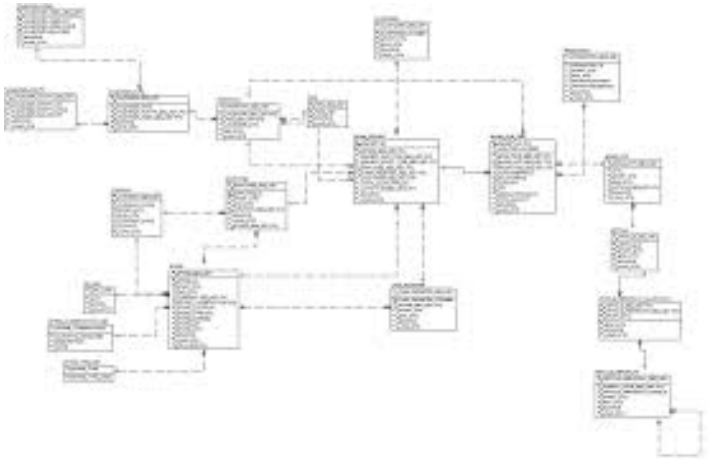
Overview

Relational Databases

- Transactions with ACID properties - Atomicity, Consistency, Isolation & Durability
- Adherence to Strong Schema of data being written/read
- Real time query management (in case of data size < 10 Tera bytes)
- Execution of complex queries involving join & group by clauses
- Support ad-hoc queries
- Not designed to work in modern distributed architecture
- Data are organized in the 3NF form - storage and memory efficient

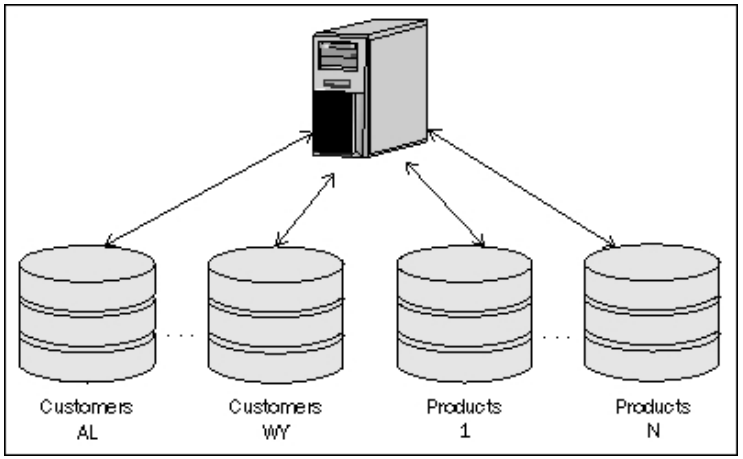
Third normal form does not scale

- Queries are unpredictable
- Users are impatient
- Data must be denormalized
- If data > memory, performance degrade
- Disk seeks are the worst



Problems with sharding

- Data is all over the place
- No more joins
- No more aggregations
- Denormalized all thing
- Querying secondary index requires hitting every shard
- Adding shards requires manual movement of data
- Manage Schema Changes



Challenges with high availability

- Master failover is messy
- Multi DC failover is harder
- Downtime is frequent
  - Change in database settings
  - Drive, power supply failures
  - OS updates

NoSQL Databases

A NoSQL (originally referring to "non SQL", "non relational" or "not only SQL") database provides a mechanism for storage and retrieval of data which is modeled in means other than the tabular relations used in relational databases.

Common goals of NoSQL databases

- Support data models, data definition languages (DDLs), and interfaces beyond the standard SQL available in popular relational databases.
- **Typically distributed systems without centralized control**.
- Emphasize horizontal scalability and high availability, in some cases at the cost of strong consistency and ACID semantics.
- Tend to support rapid development and deployment.
- Take flexible approaches to schema definition, in some cases not requiring any schema to be defined up front.
- Provide support for Big Data and analytics use cases.

Types of NoSQL Databases

- Key-value stores
- Column stores
- Document stores
- Graph databases
- Object databases
- XML databases

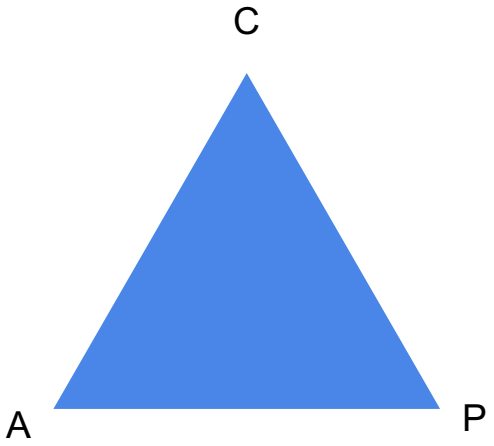
http://nosql-database.org.

CAP Theorem

What is CAP Theorem

- It is often used to understand the tradeoffs between different databases - Proposed by Eric Brewer in 2000

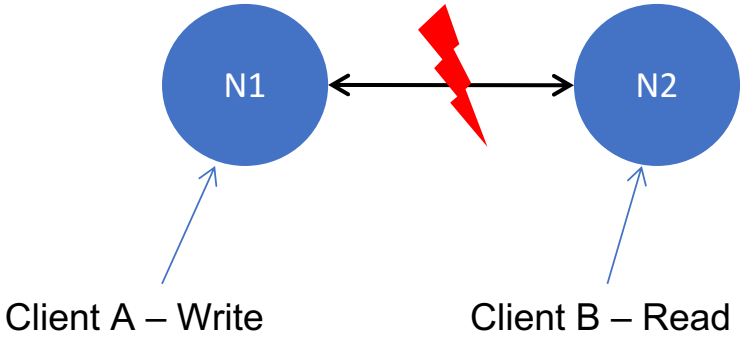
- **Consistency**: Every read receives the most recent write or an error
- **Availability**: Every request receives a (non-error) response – without guarantee that it contains the most recent write
- **Partition Tolerance**: The system continues to operate despite a part of the system is not communicated with the rest



Choose 2! You cannot get all.

CAP Theorem

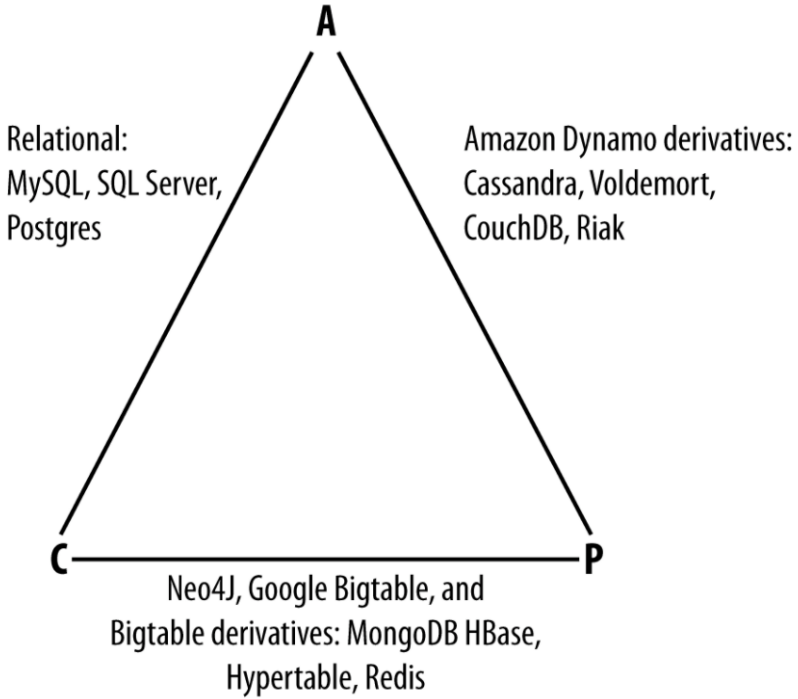
In the presence of a network partition, one has to choose between consistency and availability



	Response	Conclusion
Option 1	DB is down	DB is not partition tolerant
Option 2	N2 returns the local value	Available, sacrifices consistency
Option 3	N2 returns an error that other replicas are not reachable	Guarantees consistency, sacrificing the availability
Option 4 **	N2 waits for N1 to sync and then returns the response	Guarantees consistency and trade off latency

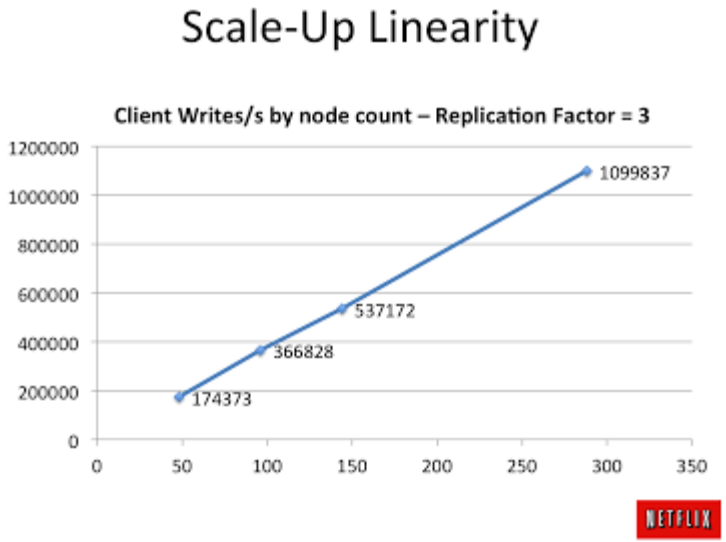
** This is an extensions to CAP known and known as PACELC theorem

What You Get


A
Relational:
MySQL, SQL Server,
Postgres
Amazon Dynamo derivatives:
Cassandra, Voldemort,
CouchDB, Riak
C
P
Neo4J, Google Bigtable, and
Bigtable derivatives: MongoDB HBase,
Hypertable, Redis

What is Apache Cassandra

- Fast Distributed Database
- Peer-to-Peer architecture
- Row oriented
- High Availability
- Linear Scalability
- Predictable Performance
- No Single Point of Failure
- Multi DC
- Commodity Hardware
- Easy to operationally manage
- Needs to redesign data mode to move RDBMS based application to Cassandra
- Decentralized



http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html

Cassandra transactions are atomic, isolated and durable

- **Atomicity**
  - In Cassandra, a write operation is atomic at the partition level, meaning the insertions or updates of two or more rows in the same partition are treated as one write operation. A delete operation is also atomic at the partition level.
- **Isolation**
  - Cassandra write and delete operations are performed with full row-level isolation. This means that a write to a row within a single partition on a single node is only visible to the client performing the operation – the operation is restricted to this scope until it is complete.
- **Durability**
  - All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success.

DB Ranking

Rank			DBMS	Database Model	Score		
Sep 2017	Aug 2017	Sep 2016			Sep 2017	Aug 2017	Sep 2016
1.	1.	1.	Oracle	Relational DBMS	1359.09	-8.78	-66.47
2.	2.	2.	MySQL	Relational DBMS	1312.61	-27.69	-41.41
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1212.54	-12.93	+0.99
4.	4.	4.	PostgreSQL	Relational DBMS	372.36	+2.60	+56.01
5.	5.	5.	MongoDB	Document store	332.73	+2.24	+16.74
6.	6.	6.	DB2	Relational DBMS	198.34	+0.87	+17.15
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	128.81	+1.78	+5.50
8.	8.	↓ 7.	Cassandra	Wide column store	126.20	-0.52	-4.29
9.	9.	↑ 10.	Redis	Key-value store	120.41	-1.49	+12.61
10.	10.	↑ 11.	Elasticsearch	Search engine	120.00	+2.35	+23.52

https://db-engines.com/en/ranking

Cassandra Use Cases

- Over 1000 companies use Cassandra today
- Accenture uses Cassandra for message
  - 2 billion messages / day
- Netflix use Cassandra for their catalog and playlist
  - 10 million transactions / sec
- EBay - personalization, messaging and fraud detection
  - Connects 100 millions customer with 400 million items and stored 250 TB data

Cassandra Performance

University of Toronto Study

	Nodes (no of transactions/sec)					
Load Type	**1**	**2**	**4**	**8**	**16**	**32**
Load Process	18,686.43	31,144.21	53,067.62	86,924.94	173,001.20	326,427.07
Read Mostly Workload	11,173.75	18,137.80	39,481.33	65,963.30	116,363.93	221,073.15
Balanced Read/Write	18,925.59	35,539.69	64,911.39	117,237.91	210,237.90	384,682.44
Read Modify Read	10,555.73	19,919.52	37,418.16	69,221.07	141,715.80	256,165.66
Mixed Operational and Analytical Workload	4,455.63	9,343.11	19,737.82	36,177.48	73,600.53	120,755.00
Insert Mostly	24,163.62	47,974.09	85,324.69	159,945.39	NA	NA

https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks

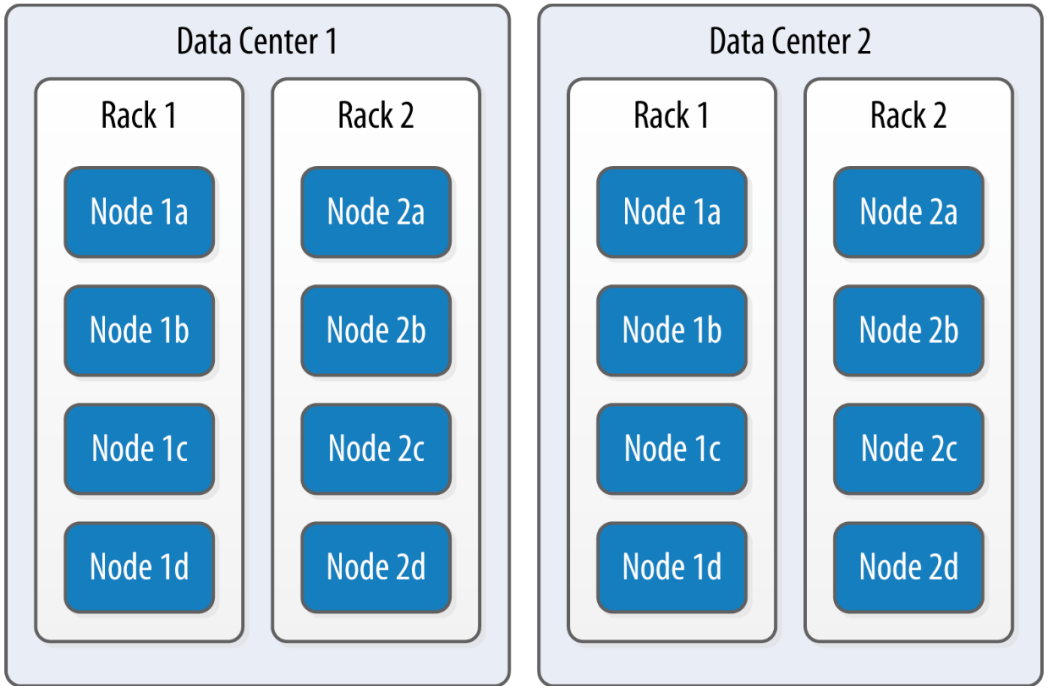
Cassandra Limitations

- Does not support query time join,
  - Cassandra recommends “join on write”
  - Data duplication can scale, but joins cannot
- Does not support group by queries
  - Supports a limited variation of sum, avg functions
  - Does not support order by on any ad-hoc columns
- Does not support where clause on ad-hoc columns
  - Cassandra recommends duplicate data based on the query requirements
- Secondary Index is supported but anti pattern
  - Recommends use materialized views
- 3NF Data modeling is antipattern in Cassandra
  - Recommends query first data model

Node

- Node is single server running Cassandra JVM process
- Node can handle typically 3k-5k transactions / secs / core
- Node can manage about 1-3 TB data
- Node generally have 8-32 GB of RAM
- Node is connected to the other nodes through 10 Gbps lines
- Supports both SSD or Rotational disks in JBOD architecture

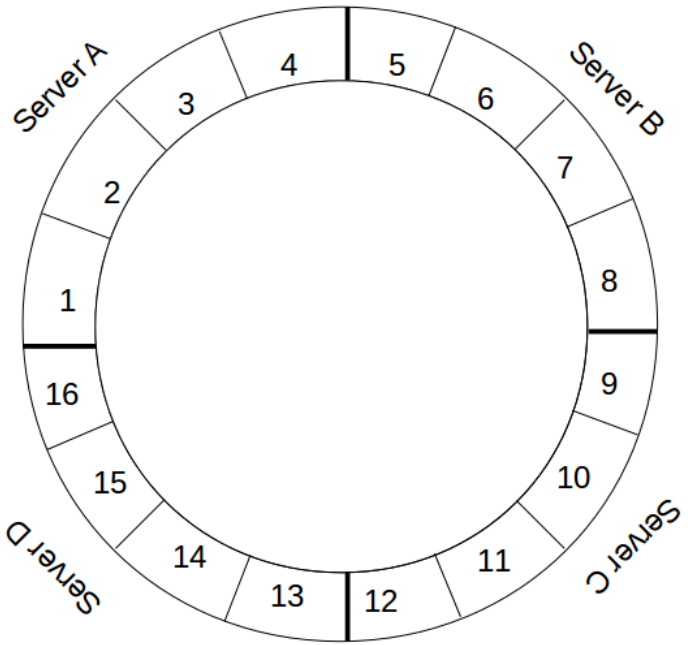
Data Centers and Racks



Data center rack

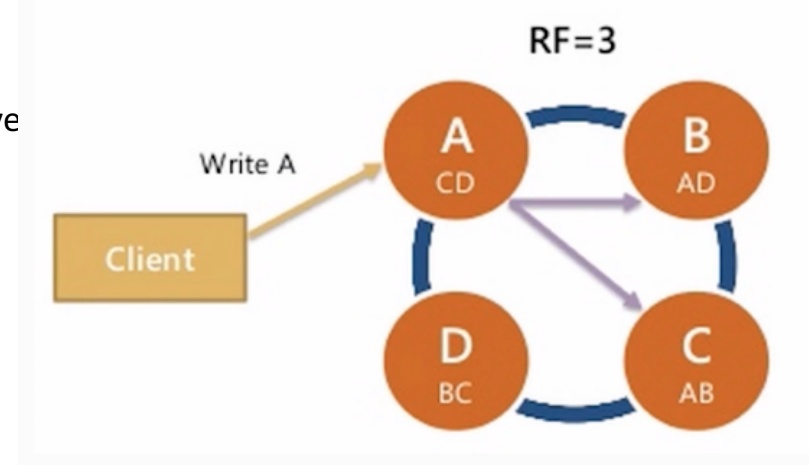
Hash Ring

- Cassandra Hash ring
- No config servers, zookeeper
- Data is partitioned around ring
- Data is replicated to RF = N servers
- All nodes hold data and can handle write and read queries
- Location of the data is determined by Partition Key



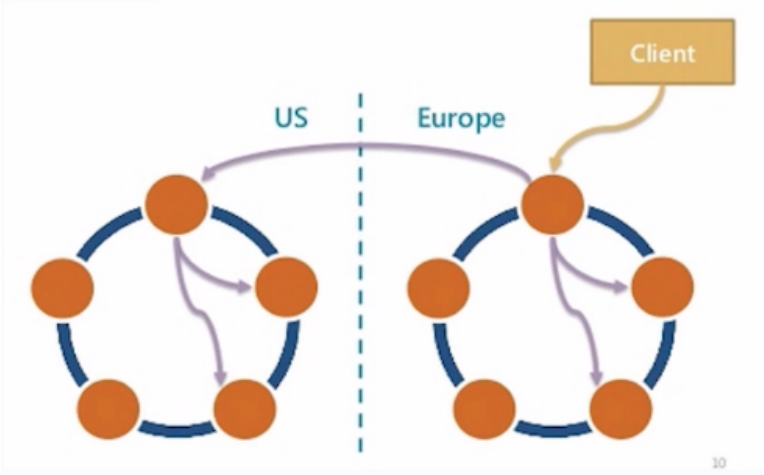
Replication

- Data is replicated automatically
- Specified by replication-factor
- If a machine is down, missing data is replaye



Multi DC

- Typical usage: client writes to local DC and replicates async to other DC
- Replication factor per key space per data center
- Datacenters can be physical or logical



Purpose of multiple data centers
•High availability
•Data locality in term of users
•Data back up
•Workloads segregation – OLTP vs. OLAP

Partitioner

- A partitioner is a hash function for computing the token of a partition key
  - Generates 64 bit hash code
- Available partitioner are
  - Murmur3Partitioner (default)
  - RandomPartitioner
  - ByteOrderedPartitioner

Virtual Nodes

- Virtual node automates the token range assignment to nodes
- Based on the snitch information, Cassandra automatically assigns one or more token ranges to each node.
  - Default is 256 vnodes per node
  - Specified in num_tokens cassandra.yaml file
- Other benefits
  - Vnodes make it easier to maintain a cluster containing heterogeneous machines. Server with more horse power can be assigned more token ranges.
  - Vnodes speed up bootstraping, decommisioning and repairing processing by spreading the workload across the cluster

Gossip

- Each node initiates a gossip round every one seconds
- Picks the one to three nodes to gossip with
- Nodes can gossip with any other node in the cluster
- Probabilistically seed and downed nodes
- Nodes do not tack which nodes they gossiped with prior
- Reliably and efficiently spreads node metadata through the cluster
- Fault tolerant

What is Gossiped

- Cluster meta data - state of the node
- Heartbeat State
  - Generation
  - Version
  - Timestamp
- Application State
  - Status
  - Location - DC, Rack
  - Schema Version
  - Load - disk pressure
  - Severity - IO pressure

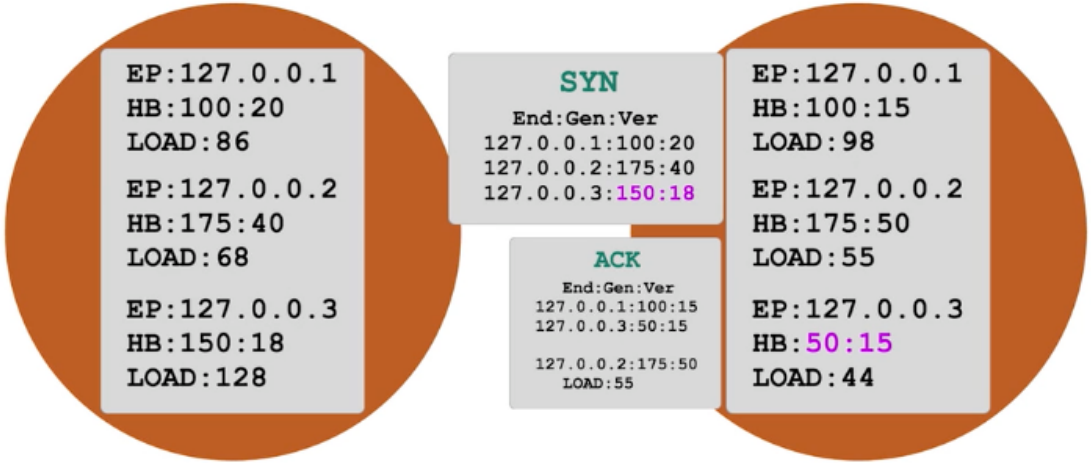
Gossip and Failure Detection

Each round of gossip requires three messages.

A. The gossip initiator sends its chosen friend a GossipDigestSynMessage.

B. When the friend receives this message, it returns a GossipDigestAckMessage.

C. When the initiator receives the ack message from the friend, it sends the friend a GossipDigestAck2Message to complete the round of gossip.



Snitches

- The job of a snitch is to determine relative host proximity for each node in a cluster, which is used to determine which nodes to read and write from.
- Snitches gather information about your network topology so that Cassandra can efficiently route requests. The snitch will figure out where nodes are in relation to other nodes
- Example of use of snitch:
  - To support the maximum speed for reads, Cassandra selects a single replica to query for the full object, and asks additional replicas for hash values in order to ensure the latest version of the requested data is returned. The role of the snitch is to help identify the replica that will return the fastest, and this is the replica which is queried for the full data.
- Snitches: SimpleSnitch, Dynamic Snitch, Amazon EC2, Google Cloud, and Apache Cloudstack

Simple Snitch

```
public class SimpleSnitch extends AbstractEndpointSnitch
{
   public String getRack(InetAddress endpoint)
   {
       return "rack1";
   }

   public String getDatacenter(InetAddress endpoint)
   {
       return "datacenter1";
   }
}
```

Property File Snitch

- Read datacenter and rack information for all nodes from a file
- You must keep files in sync with all nodes in the clusters
- cassandra-topolofy.properties
- Old school - do not use it

```
175.56.12.105=DC1:RAC1
175.50.13.200=DC1:RAC1
175.54.35.197=DC1:RAC2
175.54.35.152=DC1:RAC2

120.53.24.101=DC2:RAC1
120.55.16.200=DC2:RAC1
120.57.18.103=DC2:RAC2
120.57.18.177=DC2:RAC2
```

Gossip Property File Snitch

- Relives the pain of property file snitch
- Declare the current nodes DC/rack information in a file - cassandra-rackdc.properties
- You must set each individual nodes settings, but you do not have to copy settings as with property file snitch
- Gossip spreads the settings through the cluster

```
dc = DC1
rack=RAC1
```

Dynamic Snitch

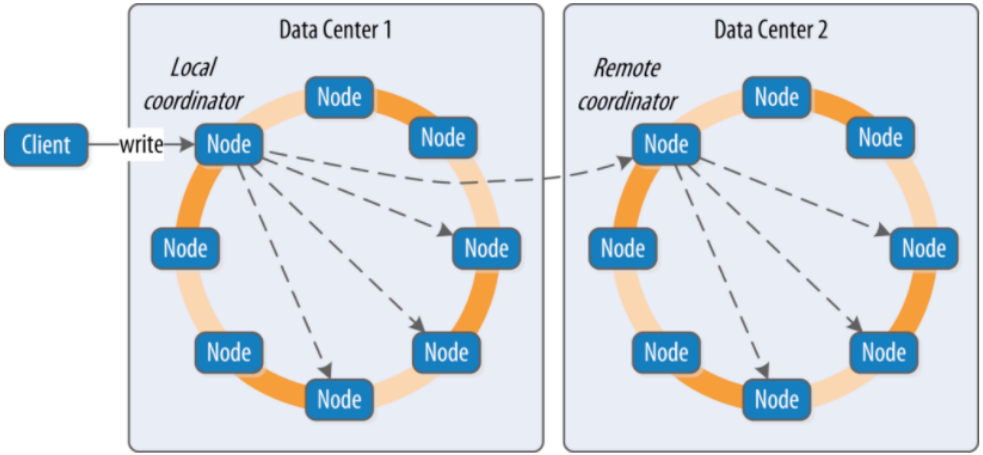
- Layered automatically with other snitches
- Maintains a pulse on each node’s performance
- Determines which node to query replicas from depending the node health
- Tuned on by default for all snitches

Replication Strategy

- Cassandra OOB support 2 strategies
  - SimpleStrategy and NetworkTopologyStrategy.
- The SimpleStrategy places replicas at consecutive nodes around the ring, starting with the node indicated by the partitioner.
- The NetworkTopologyStrategy allows you to specify a different replication factor for each data center. Within a data center, it allocates replicas to different racks in order to maximize availability.

Consistent Model

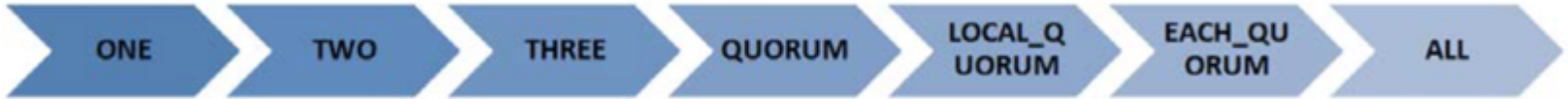
- Per query consistency
- ALL, Quorum, ONE
- How many replicas for query to respond OK



Consistency Levels

Write Consistency Levels - in the order of write latency period

Read Consistency Levels - in the order of write latency period



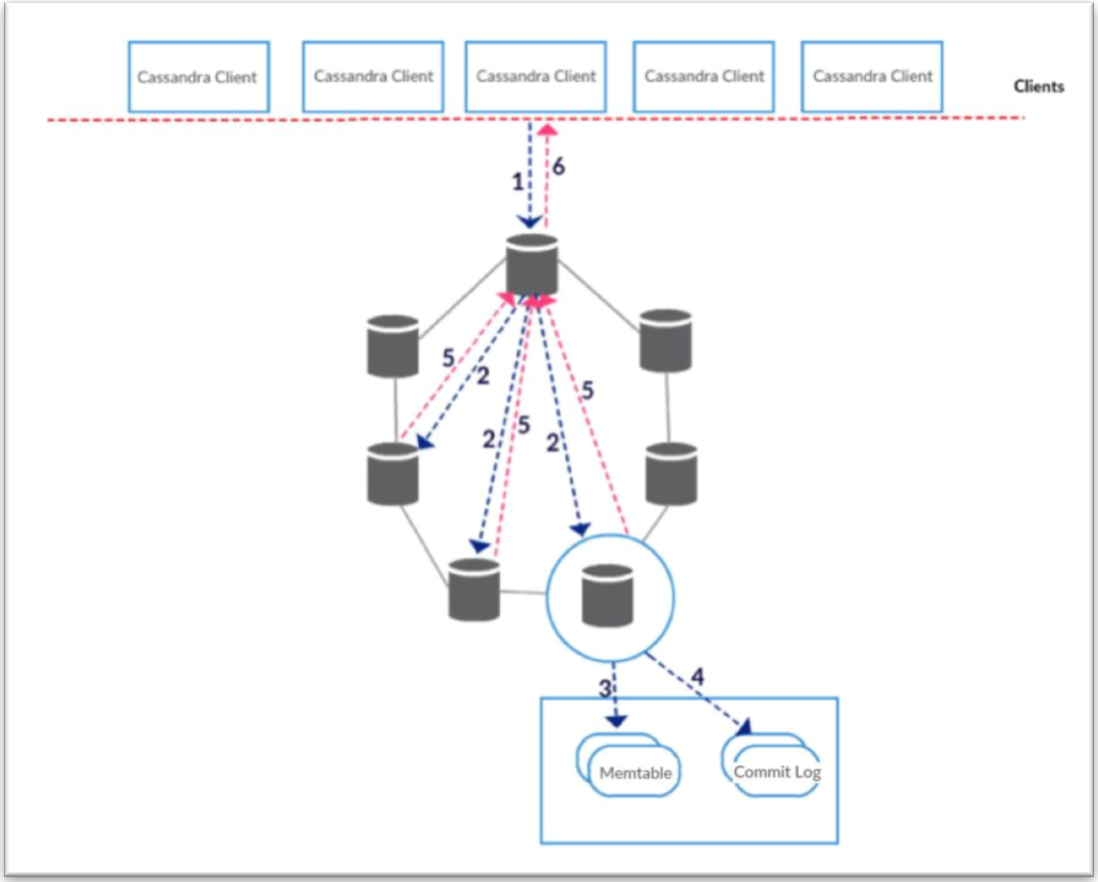
- As you increase consistency latency grows as well
- You can set consistency level at transaction level - just before each select, insert etc.
- Choose the consistency level according to the table you are handling
- ANY or ALL are probably not good choice.

Guaranteed Strong Consistency

- Write all, read one
- Write all, read quorum
- Write quorum, read all
- Write one, read all
- Write quorum, read quorum

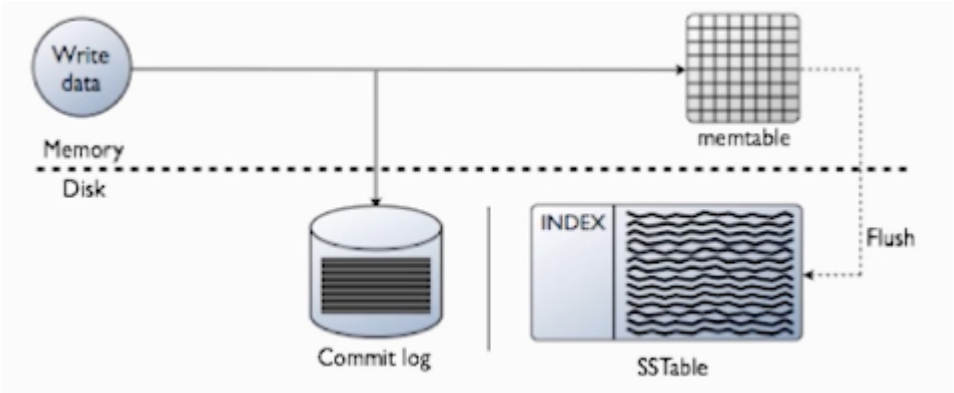
Write Consistency Levels + Read Consistency Level > Replication factor

Write Operation



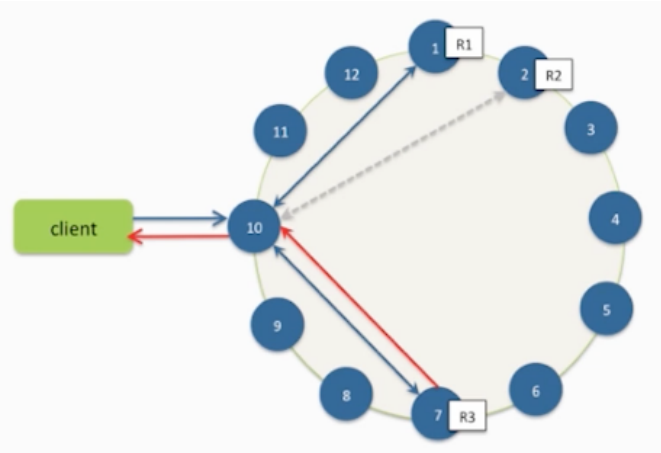
Write Path

- Writes are written to any node in the cluster … called coordinator node for that particular request.
- Writes are written to commit logs then to memtable
- Every write include timestamp
- Memtable flushed to the disk periodically asynchronously without locking (sstable)
- New memtable is created in memory
- Deletes are special write case, called tombstone



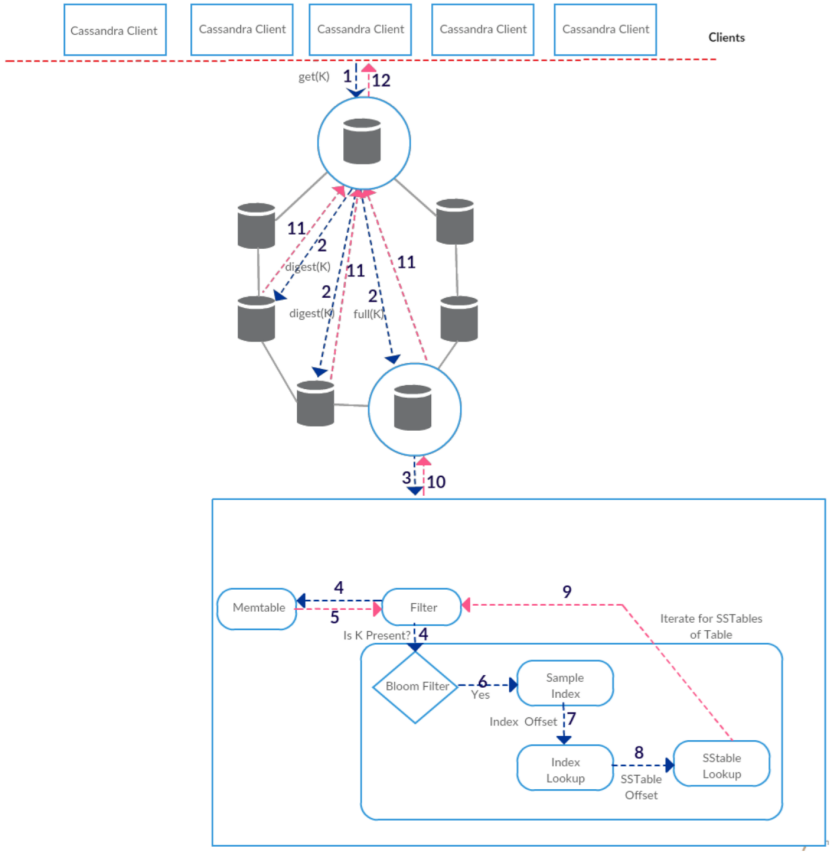
Read Path

- A query can land to any server act as coordinator
- Contacts nodes with the requested key
- On each node data is pulled from SSTables and merged
- Consistency < ALL performs read repair in background (read_repair_chance)
- Choice disk (SSD vs HDD) has significant impact on read performance
- Compaction has significant impact on performance
- The fastest node returns data … other node that contains the replica returns digest
- Uses conflict resolution protocol called - last write wins.

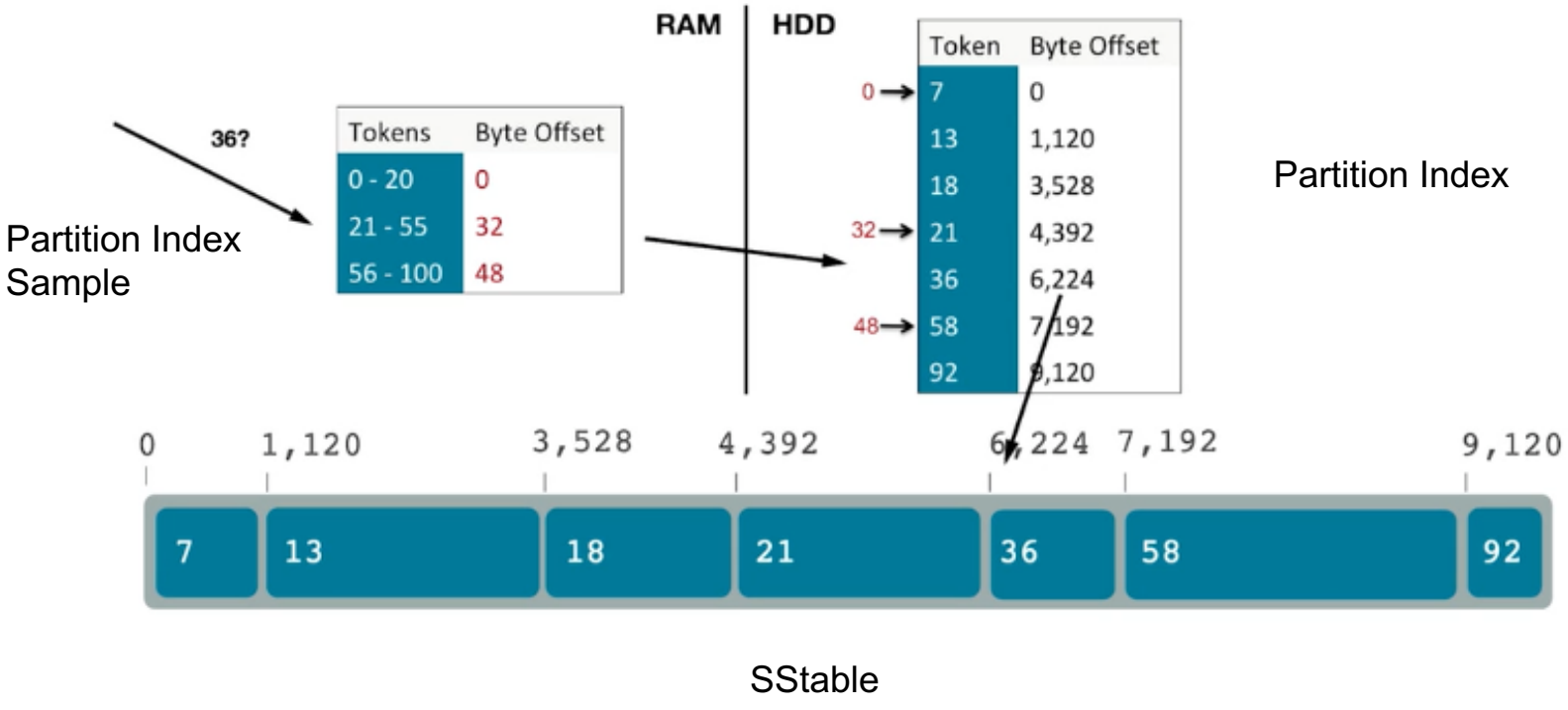


Client can send request either in native binary or Thrift protocol

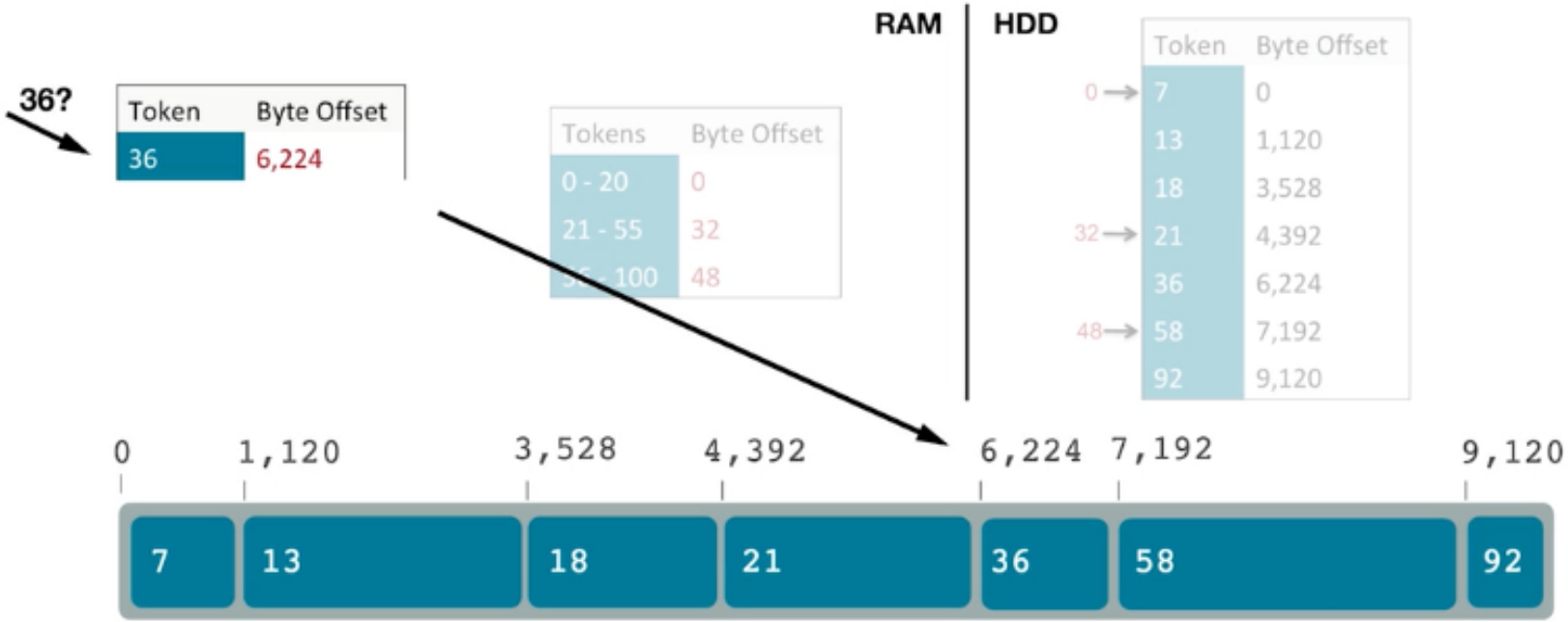
Read Operation



Readiing an SSTable

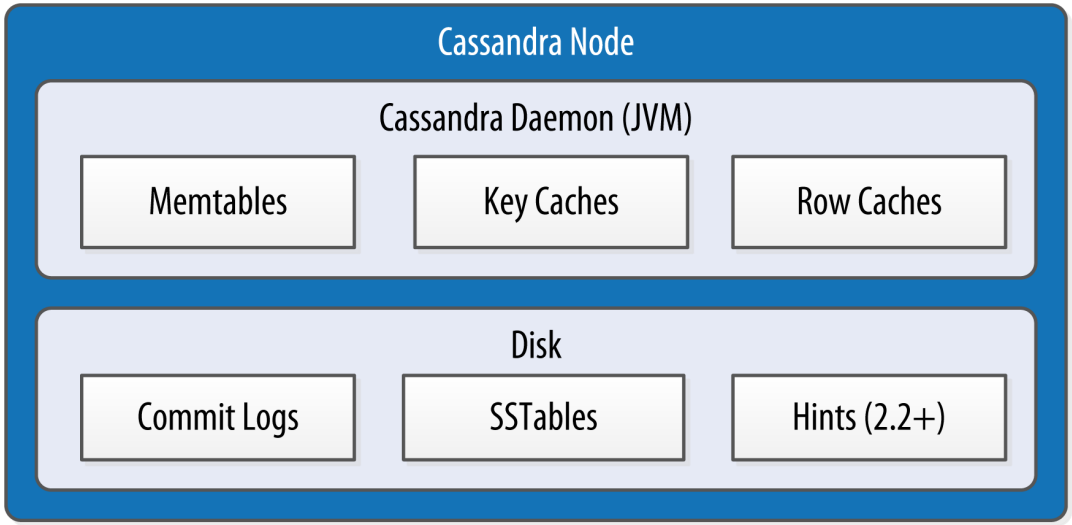


Reading SSTable using Index Cache



You can turn on index caching at table properties

Data Structures



Commit Logs

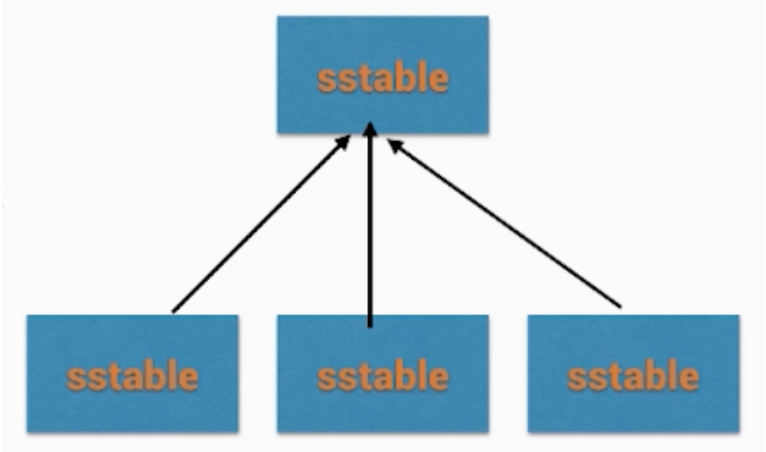
- When you perform a write operation, it’s immediately written to a commit log.
- The commit log is a crash-recovery mechanism that supports Cassandra’s durability goals.
- A write will not count as successful until it’s written to the commit log
- Following unexpected shutdown, next time you start the node, the commit log gets replayed.
- That’s the only time the commit log is read; clients never read from it.
- Commit log common at the server level

Memtable

- After it’s written to the commit log, the value is written to a memory-resident data structure called the memtable.
- Each memtable contains data for a specific table. In early implementations of Cassandra, memtables were stored on the JVM heap, but improvements starting with the 2.1 release have moved the majority of memtable data to native memory. This makes Cassandra less susceptible to fluctuations in performance due to Java garbage collection.

What is SSTable

- Immutable data file for row storage
- Every cell value includes - write time stamp collected at client side
- Partition is spread across multiple SSTables
- Same column can be used in multiple SSTables
- Merged through compaction, only latest timestamp is kept
- Deletes are written as tombstones
- Easy backups!
- SSTables Contain
    - data
    - index and bloom filter
    - checksum
    - statistics



SS Tables

File Name	Description
CompressionInfo.db	compression metadata information that includes uncompressed data length, chuck size, and a list of the chunk offsets
Data.db	Data file stores the base data of SSTable which contains the set of rows and their columns. For each row, it stores the row key, data size, column names bloom filter, columns index, row level tombstone information, column count, and the list of columns. The columns are stored in sorted order by their names.
Statistics.db	Statistics file contains metadata for a SSTable. The metadata includes histograms for estimated row size and estimated column count. It also includes the partitioner used for distributing the key, the ratio of compressed data to uncompressed data and the list of SSTable generation numbers from which this SSTable is compacted. If a SSTable is created from Memtable flush then the list of ancestor generation numbers will be empty.
Index.db	Index file contains the SSTable Index which maps row keys to their respective offsets in the Data file. Row keys are stored in sorted order based on their tokens. Each row key is associated with an index entry which includes the position in the Data file where its data is stored. New versions of SSTable (version "ia" and above), promoted additional row level information from Data file to the index entry to improve performance for wide rows. A row's columns index, and its tombstone information are also included in its index entry.
Summary.db	Summary file contains the index summary and index boundaries of the SSTable index. The index summary is calculated from SSTable index. It samples row indexes that are index_interval (Default index_interval is 128) apart with their respective positions in the index file. Index boundaries include the start and end row keys in the SSTable index.
Filter.db	Stores bloom filter data
Digest.crc32	Store CRC code for the Data.db
TOC.txt	TOC.txt contains the list of components for the SSTable.

Cache

- Cassandra provides three forms of caching:
  - The key cache stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk. The key cache is stored on the JVM heap.
  - The row cache caches entire rows and can greatly speed up read access for frequently accessed rows, at the cost of more memory usage. The row cache is stored in off-heap memory.
  - The counter cache was added in the 2.1 release to improve counter performance by reducing lock contention for the most frequently accessed counters.
- By default, key and counter caching are enabled, while row caching is disabled, as it requires more memory.
- Cassandra saves its caches to disk periodically in order to warm them up more quickly on a node restart.

Hinted Handoff

- When a write request is sent to Cassandra, but a replica node where the write properly belongs is not available, Cassandra implements a feature called *hinted handoff* to ensure general availability of the ring in such a situation.
- A *hint* is like a little Post-it note that contains the information from the write request.
- You can control how fast hints will send off to the node (hinted_handoff_throttle_in_kb) once the node is up and what is max amount of hints a node would stored (max_hint_window_in_ms).

Tombstones

- When you execute a delete operation, the data is not immediately deleted. Instead, it's treated as an update operation that places a tombstone on the value. A tombstone is a deletion marker that is required to suppress older data in SSTables until compaction can run.
- Garbage Collection Grace Seconds - is the amount of time that the server will wait to garbage-collect a tombstone. [Default: 864,000 seconds, or 10 days]
- The purpose of this delay is to give a node that is unavailable time to recover; if a node is down longer than this value, then it is treated as failed and replaced.

Bloom Filters

- Bloom filters are used to boost the performance of reads.
- Bloom filters are very fast, non-deterministic algorithms for testing whether an element is a member of a set.
- Bloom filters work by mapping the values in a data set into a bit array and condensing a larger data set into a digest string using a hash function. The digest, by definition, uses a much smaller amount of memory than the original data would.
- The filters are stored in memory and are used to improve performance by reducing the need for disk access on key lookups.
- A Bloom filter is a special kind of cache. When a query is performed, the Bloom filter is checked first before accessing disk.

Compaction

- Compaction is the process of freeing up space by merging large accumulated datafiles.
- SSTables are immutable, which helps Cassandra achieve such high write speeds.
- Periodic compaction of these SSTables is important in order to support fast read performance and clean out stale data values.
- A compaction operation in Cassandra is performed in order to merge SSTables.
- During compaction, the data in SSTables is merged: the keys are merged, columns are combined, tombstones are discarded, and a new index is created.

Compaction Types

- SizeTieredCompactionStrategy (STCS) is the default compaction strategy and is recommended for write-intensive tables
- LeveledCompactionStrategy (LCS) is recommended for read-intensive tables
- DateTieredCompactionStrategy (DTCS), which is intended for time series or otherwise date-based data.

Compaction Strategy

SizeTieredCompactionStrategy (default)

- This strategy triggers a minor compaction whenever there are a number of similar sized SSTables on disk
- Using this strategy causes bursts in I/O activity while a compaction is in process, followed by longer and longer lulls in compaction activity as SSTable files grow larger in size.
- Good option for frequent writes

LeveledCompactionStrategy

- Creates SSTables of a fixed, relatively small size (5 MB by default) that are grouped into levels.
- Each level (L0, L1, L2 and so on) is 10 times as large as the previous.
- Disk I/O is more uniform and predictable as SSTables are continuously being compacted into progressively larger levels.
- Good option in read heavy, low updates scenarios

Read Repair Chance

- Specifies the probability with which read repairs should be invoked on non-quorum reads. The value must be between 0 and 1.
- Performed when read is at a consistency level less than ALL
- Request reads only a subset of the replicas
- We cannot be sure replicas are in sync
- Generally you are safe, but no guarantees
- Response sent immediately when consistency level is met
- Read repair is done asynchronously in the background
- 0.10 is default

Nodetool Repair

- Syncs all data in cluster
- Run to synchronize a failed node coming back online
- Run on nodes not read from very often
- Expensive
  - Grows with amount of data of in the cluster
- Use with cluster servicing high writes/deletes
- Last line of defense
- Run as maintenance once in ~ 10 days

Anti-Entropy, Repair, and Merkle Trees

- Cassandra uses an anti-entropy protocol, which is a type of gossip protocol for repairing replicated data. Anti-entropy protocols work by comparing replicas of data and reconciling differences observed between the replicas.
- Anti-entropy is a manually initiated repair process
- Running nodetool repair causes Cassandra to execute a major compaction

Data Types

Type	Details
text	● UTF8 encoded string ● varchar is same as text
int	● Signed ● 32 bits
timestamp	● Date and time ● 64 bit integer ● Epoch - stores numbers of seconds since Jan 1, 1970 midnight
UUID and TIMEUUID	● Universal Unique Identifier - 128 bit value ● Generated via uuid() ● TIMEUUID embeds a TIMESTAMP value and is sortable ● Generated via now()

Cassandra Row Structure

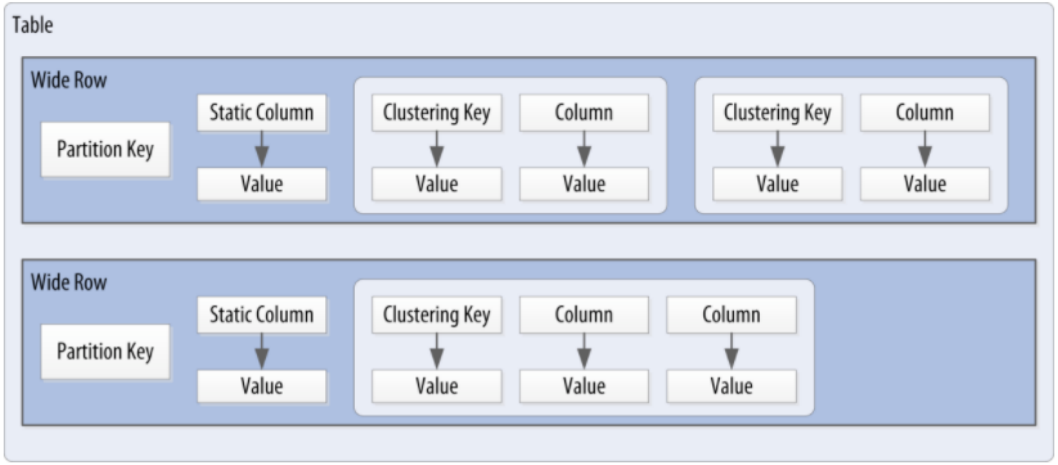
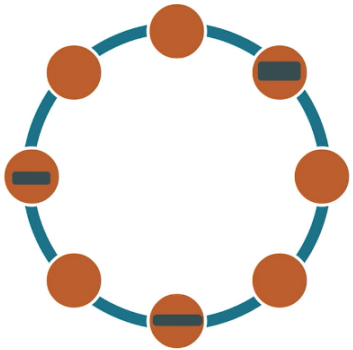


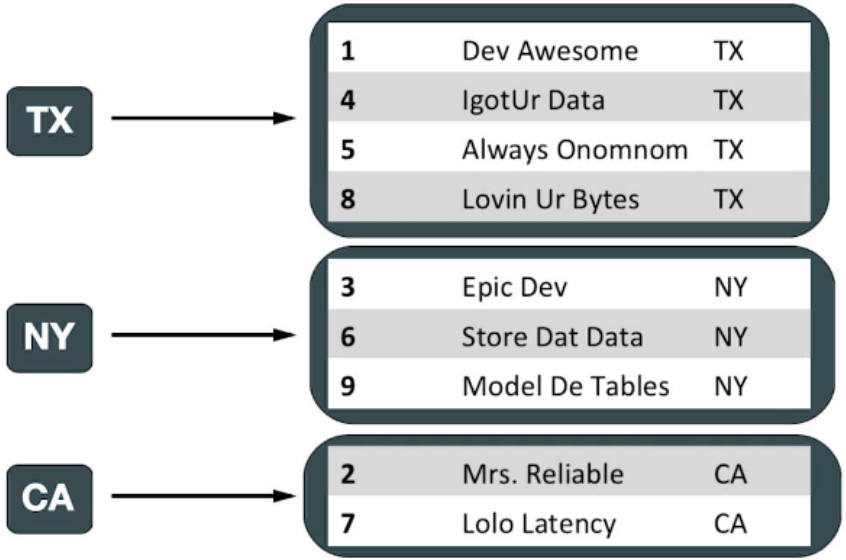
Table Partition

Partitions

- Partition is a grouping
- First value aka **partition key** in the primary key is the partition key. In the example, state is the partition key.



PRIMARY KEY((state), id)



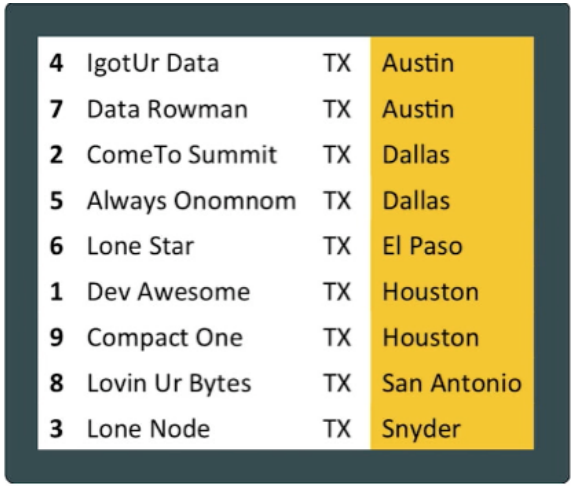
Cluster Columns

- Clustering column help sort the data within each partition

PRIMARY KEY ((state), id)

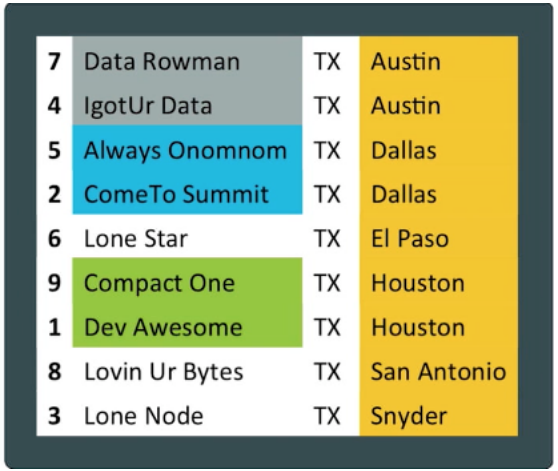
1	Dev Awesome	TX	Houston
2	ComeTo Summit	TX	Dallas
3	Lone Node	TX	Snyder
4	IgotUr Data	TX	Austin
5	Always Onomnom	TX	Dallas
6	Lone Star	TX	El Paso
7	Data Rowman	TX	Austin
8	Lovin Ur Bytes	TX	San Antonio
9	Compact One	TX	Houston

PRIMARY KEY ((state), city, id)



4	IgotUr Data	TX	Austin
7	Data Rowman	TX	Austin
2	ComeTo Summit	TX	Dallas
5	Always Onomnom	TX	Dallas
6	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
9	Compact One	TX	Houston
8	Lovin Ur Bytes	TX	San Antonio
3	Lone Node	TX	Snyder

PRIMARY KEY ((state), city, name
id)



7	Data Rowman	TX	Austin
4	IgotUr Data	TX	Austin
5	Always Onomnom	TX	Dallas
2	ComeTo Summit	TX	Dallas
6	Lone Star	TX	El Paso
9	Compact One	TX	Houston
1	Dev Awesome	TX	Houston
8	Lovin Ur Bytes	TX	San Antonio
3	Lone Node	TX	Snyder

Querying Clustering Columns

- Must provide partition key
- Clustering column can follow thereafter
- You can perform either equality or range on clustering columns
- All equality comparisons must come before inequality comparisons
- Since data is sorted on disk, range searches are a binary search followed by a sequential read

Allow Filtering

- ALLOW FILTERING relaxes the querying on partition key constraint
  - `SELECT * FROM users`
- You can then query on just clustering columns
- Causes Cassandra to scan all partitions in the table
- Do not use it
  - Unless you really have to
  - Best on small data sets
  - Consider use of Spark!

Operations

Topics

- Adding Nodes
- Bootstrapping a Node into a cluster
- Removing a Node
- Replace a Downed Node
- Run a Repair Operation
- Dividing SSTables with sstableslits
- Create snapshot
- Implement Multiple Data Centers
- Best Practices for Cluster Sizing
- Using Cassandra-stress Tool

Adding Node

- Data capacity problem
  - You data has outgrown the node's hardware capacity
- Reached Traffic Capacity
  - You application needs more rapid response with less latency
- Need more operational headroom
  - For node repair, compaction, and other resource intensive operation

Bootstrapping

- Simple process that brings a node up to speed
- The amount of time it takes to bootstrap depends on the amount of data
  - Can be a long running process
- Node announces itself to ring using seed node

The Bootstrapping Process

- Calculate range(s) of new node, notify ring of these pending ranges
- Calculate the node(s) that currently own these ranges and will no longer own once the bootstrap completes
- Stream the data from these nodes to the bootstrapping node (monitor with nodetool netstatus)
- Join the new node to the ring to it can serve traffic

Nodetool Cleanup

- Makes sure no data in left behind in SSTables that are outside the allocated token ranges
  - It essentially copies the SSTables to new SSTables keeping only valid data
- Compaction process will clean up
- Not essentials - it is more a deterministic process

```
$ bin/nodetool [options] cleanup  -- <keyspace> (<table>)
```

Use flags to specify:

```
  - h [host] [IP Address]
  - p port
  -pw password
  -u username
```

Replace a Downed Node

- First find the ip address of the down node using `nodetool status` command
- In the node, open the cassandra-env.sh
- Swap in the IP address of the dead node as replace_address value in the JVM option. This will eliminate bootstrapping of the new node.
- Use `nodetool removenode` to remove the dead node
- Use force option if necessary `nodetool assassinate`
- You can monitor the process using `nodetool netstats` command

What if the node was also a seed node?

- Need to add to list of seeds in cassandra.yaml
- Cassandra will not allow seed node to auto-bootstrap
- You will have to run repair on new seed node to do so
- Add a new node making the necessary changes to the cassandra.yaml file
- Specify new seed node in cassandra.yaml
- Start Cassandra on new seed node
- Run `nodetool repair` on the new seed node to manually bootstrap
- Remove the old seed node using `nodetool removednode` with the Host ID of the downed node
- Run `nodetool cleanup` on previously existing nodes

Repair Operation

- Repair is a deliberate process to cope with cluster entropy … a consistency check
- Entropy can arise from the node that were down longer than hint-window, dropped mutations or other causes
- A repair operates on all of the nodes in the replica set by default
- Ensures that all replicas have identical copies of a given partition
- Consists of two phases
  - Build Merkle tree of the data per partition
  - Replicas then compare the differences between their trees and stream the differences to each other as needed

A Merkle Tree Exchange

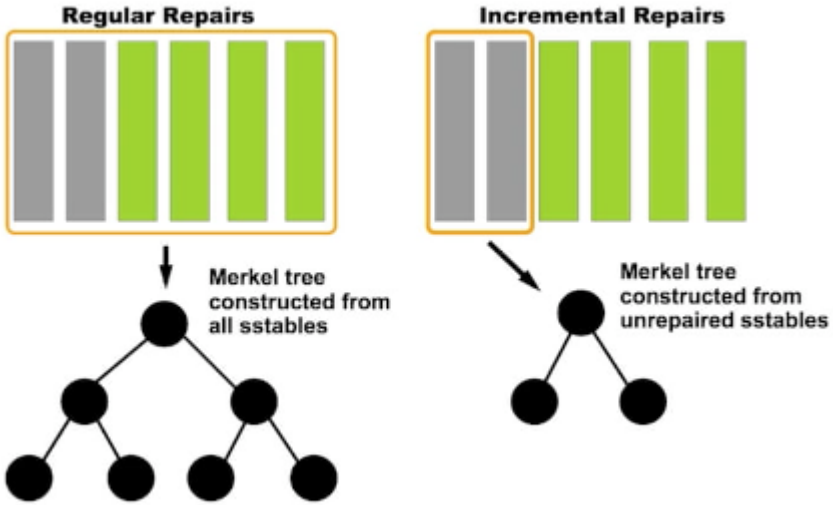
- Starts with root of the tree (a list of one hash value)
- The origin sends the list of hashes at the current level
- The destination diffs the list of hashes against its own, then requests subtrees that are different
- If there are no differences, the request can terminate
- Repeat steps 2 and 3 until leaf nodes are reached
- The origin sends the values of the keys in the resulting set

Why Repair is Necessary?

- A node's data can get inconsistent over time (Repair is just a maintenance action in this case)
- If a node goes down for some time, it misses writes and will need to catch up
- Sometimes it is best to repair a node:
  - If the node has been down longer than the length specified in max_hint_window_in_ms, the node is out of sync
  - Depending on the amount of data, might be faster to repair
- Sometimes it is better to bring the node back as a new node and bootstrap
  - If there is a significant amount of data, might be faster just to bring in a new node and stream data just to that node

What are Incremental Repairs

- To avoid the need for constant tree construction, incremental repairs have been introduced
  - Idea is to persist already repaired data and only calculate merkle tree for sstables that been created since
  - This allows the repair process to stay performant and lightweight



What are incremental repairs?

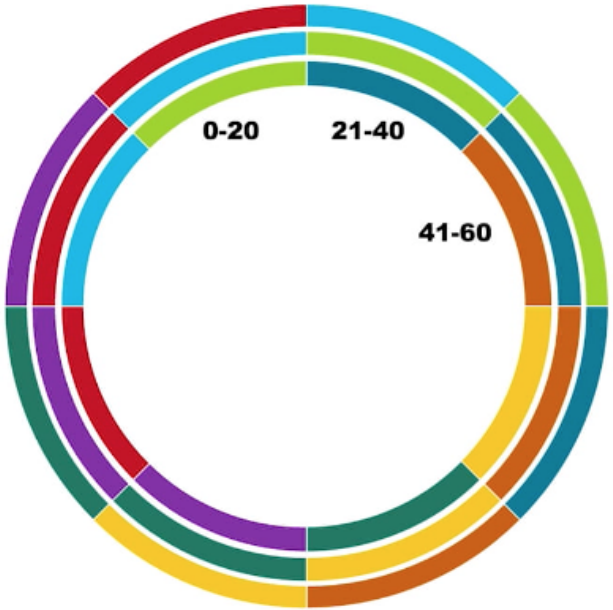
- Incremental repairs begin the repair leader sending out a prepare message to the peers
- Each node builds a Merkle tree from the unrepaired sstables
  - This can be distinguished by the repairedAt field in the each sstable's metadata
- Once the leader receives a merkle tree from each node, it compares the tree and issues streaming requests
  - This is just as in the classic repair case
- Finally the leader issues an anti-compaction command
  - Anti-compaction is the process of segregating repaired and unrepaired ranges into separate sstables
  - Repaired sstables are written with a new repairedAt field denoting the time of repair

Best Practices for Repair

- Run repair weekly
- Run repair on a small number of nodes at a time
- Schedule for low usage hours
- Run repair on a partition or subrange of a partition

Repair Type - Primary and Secondary Ranges

- Primary Range (inside ring) - first node that data is stored on based on partition key
- Secondary Range (outside rings) - additional replicas of the same data
- What are the implications of repair?



SSTableSplit - Why?

- You did a nodetool compaction
- Maybe you used SizeTieredCompactionStrategy for a major compaction
- This would result in an excessively large SSTable
- Good idea to split the table
- Using the size tiered compaction, we may have gotten some really large files over time
- May find yourself with a 200GB file you need to split up
- It is an anti-compaction in a way

Usage: sstablesplit

- Stop Cassandra Node

```
$ sudo service cassandra stop
$ sstablesplit [options] filename [filename]*
```

Options:

`--debug` Displays stack traces

`--no-snapshot` Do not snapshot the SSTables before splitting

`--size` Maximum size in MB for output SSTables (default: 50)

`--verbose` Verbose output

Snapshot

Why Backup?

- We do not backup like traditional databases where we copy out all the data
- It is a distributed system; every server or node has a portion of the data
- SSTables are immutable, which is great! Makes them easy to back up.

Why Snapshots?

- Snapshot represents a state at a given time
- Snapshots create hard link to the file systems as opposed to copying data
- Therefore very fast!
- Represents the state of the data files at a particular point of time
- Can consist of single table, single keyspace or multiple keyspace

What is a Snapshot

- Represents the state of the data files at a particular point in time
- Snapshot directory is created (this has pointers)
- Then you can either leave it there or copy it offline to an NFS mount or copy S3

Incremental Backup

- Incremental backups create a hard link to every SSTable upon flush
  - User must manually delete them after creating a new snapshot
- Incremental backups are disabled by default
  - Configured in cassandra.yaml file setting increatemental_backups to true
- Need a snapshot before taking incremental backups
- Snapshot information is stored in a snapshots directory under each table directory
  - Snapshot need only be stored once offsite

Location of Snapshot and Incremental Backups

- Snapshots and incremental backups are stored in on each Cassandra node
  - Vulnerable to hardware failures
- Commonly, the files are copied to off-node location
  - Open source programs like tablesnap is useful for backing up to S3
  - Scripts can be used to automate backup files to another machine: cron + bash script, rsync etc

Auto Snapshot

- Critical safety factor
- If enabled, before a table is truncated or tables/keyspace is dropped, a snapshot is taken
- A configuration in cassandra.yaml, default is true
- Recommended to keep it true

How to take Snapshot?

```
$ bin/nodetool [options] snapshot (-cf <table> | -t
 <tag> --keyspace)
```

We can specify to take a snapshot of

- One more keyspaces
- A table specified to backup data

Clean up Snapshots

- `$ bin/nodetool clearsnapshot` command removed snapshots
- Same options as nodetool command
- Specify the snapshot file and keyspace
- Not specifying a snapshot name removes all snapshots
- Remember to remove old snapshots before taking new ones - previous snapshots are not automatically deleted
- To clear snapshots on all nodes at once, use parallel ssh utility - pssh, clusterssh

How to restore?

- Most common method is to delete current data files and copy the snapshot and incremental files to the appropriate data directories
  - If using incremental backups, copy the contents of the backups directory to each table directory
  - Table schema must already be present in order to use this command
  - Restart and repair the node after the file copying is done
- Another method is to use `sstableloader`
  - Great if you are loading it into a different size cluster
  - Must be careful about its use as it can add significant load to cluster while loading

Cluster Wide Backup And Restore

- OpsCenter
- SSH Programs - pssh, clusterssh
- Honorable mention - tablesnap and tablestore
  - For Cassandra backup to AWS S3
- Recovery

Multi Data Center Implementation

- Node - the virtual or physical host of a single Cassandra instance
- Rack - a logical grouping of physical related nodes
- Data Center - a logical grouping of a set racks
- Enables geographically aware read and write request routing
- Each node belongs to one rack in one data center
- The identity of each node’s rack and data center may be configured in its conf/cassandra-rackdc.properties

Adding a Secondary Data Center

- Ensures continuous availability of your data and application
- Live backup
- Improved performance
  - Lower latency by serving near from geo location
- Analytics
  - One DC is dedicated for transactional load and one for analytics

How cluster operate between data centers

- A data center is a grouping of nodes configured together for replication purposes
- Data replicates across data centers automatically and transparently
- Consistency level can be specified at LOCAL level for read/write operations
- Consistency level can also be specified as EACH
  - EACH meaning - each data center has its own quorum

Implementing Multi Data Center Cluster

- Use NetWorkTopologyStrategy rather than SimpleStrategy
- Use LOCAL_* consistency level for read/write operations to limit latency
- If possible, define one rack for entire cluster
- Specify the snitch

Cluster Sizing

Factors

- Application data model and use case
- Volume of data over time
- Velocity of data
- Replication Factor (RF)

1. Data Model and Use Cases
- What is the read/write bias?
- What are the SLAs
- Do I need to tier out data? Hot / cold

2. Volume of Data
- Estimate the volume based on your application

- Data per node = RF * Avg amount of data per row x number of rows / # nodes

3. Velocity
- How many writes per second

- How many reads per second

- Volume of data per operation

Cluster Size - Testing Limits

- Use cassandra-stress to simulate workload
- Test product level servers
- Monitor to find limits
  - Disk is the first thing to manage
  - CPU is second

Cluster Sizing - An Example

- Data growth 100G data / day
- Writes per second - 150 K
- Reads per second - 100 k

What additional information do we need?

- Replication Factor of 3
- Multiple Data Center
- 25ms read latency 95th percentile
    - At max packet size 50k writes / sec
    - At max packet size 40k reads / sec
    - 4TB per node to allow for compaction overhead

Cluster Sizing - Based on Volume

- (100G / day) x (30 days) x (6 months) = 18 TB
- (18 TB data) x (RF 3) = 54 TB
- 54 TB total data / (4TB max per node) = 14 nodes
- (14 nodes) x (2 Data Center) = 28 nodes

Cluster Sizing - Sizing for Velocity

- (150 K writes/ sec load) / (50k writes/sec per node) = 3 nodes
- Volume capacity covers the write capacity

Cluster Sizing - Future Capacity

- Validate your assumptions often
- Monitor for changes over time
- Plan for increasing cluster size before you need it
- Be ready to draw down if needed

Cassandra Stress Tool

- Java based utility built in Cassandra
- Used for basic benchmarking and load testing a Cassandra cluster
  - Quickly determine how a schema performs
  - Understand how your database scale
  - Optimize your data model and settings
  - Determine production capacity

Cassandra Stress

YAML file is configuration file for stress tool. It is split into few sections

- DDL - for defining your schema
- Column distributions - for defining the shape and size of each column globally and within each partition
- Insert Distributions - for defining how the data is written during the stress test
- DML - for defining how the data is queried during the stress test

Cassandra Stress

DDL

- Define the keyspace and table information
- If the schema is not already define, stress tool will define it when it runs
- If schema is already defined, then stress tool needs know only keyspace and table names

Cassandra Stress YAML - Example

```
### DML ###

# Keyspace Name
keyspace: stresscql

# The CQL for creating a keyspace (optional if it already exists)
keyspace_definition: |
  CREATE KEYSPACE stresscql WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};

# Table name
table: blogposts

# The CQL for creating a table you wish to stress (optional if it already exists)
table_definition: |
  CREATE TABLE blogposts (
        domain text,
        published_date timeuuid,
        url text,
        author text,
        title text,
        body text,
        PRIMARY KEY(domain, published_date)
  ) WITH CLUSTERING ORDER BY (published_date DESC);
    AND compaction = { 'class':'LeveledCompactionStrategy' }
    AND comment='A table to hold blog posts'
```

Stress Test: Column Distribution

- The columnspec section describes the different distribution for use for each column
- Columnspec section describes the different distributions to use for each column. These distributions model the size of the data in column, the number of unique values and the clustering of them within a given partition
- These distributions are used to auto generate data that looks like what you would see in reality

Stress Test: Column Distribution

EXP(min..max) — An exponential distribution over the range [min..max]
EXTREME(min..max,shape) — An extreme value (Weibull) distribution over the range [min..max]
GAUSSIAN(min..max,stdvrng) — A gaussian/normal distribution, where mean=(min+max)/2, and stdev is (mean-min)/stdvrng
GAUSSIAN(min..max,mean,stdev) — A gaussian/normal distribution, with explicitly defined mean and stdev
UNIFORM(min..max) — A uniform distribution over the range [min, max]
FIXED(val) — A fixed distribution, always returning the same value

Stress test: other column specs

Size distribution — Defines the distribution of sizes for text, blob, set and list types (default of UNIFORM(4..8))
Population distribution — Defines the distribution of unique values for the column values (default of UNIFORM(1..100B))
Cluster distribution — Defines the distribution for the number of clustering prefixes within a given partition (default of FIXED(1))

Stress Test: column spec - example

```
### Column Distribution Specifications ###

columnspec:
  - name: domain
    size: gaussian(5..100)       #domain names are relatively short
    population: uniform(1..10M)  #10M possible domains to pick from

  - name: published_date
    cluster: fixed(1000)         #under each domain we will have max 1000 posts

  - name: url
    size: uniform(30..300)

  - name: title                  #titles shouldn't go beyond 200 chars
    size: gaussian(10..200)

  - name: author
    size: uniform(5..20)         #author names should be short

  - name: body
    size: gaussian(100..5000)    #the body of the blog post can be long
```

Stress Test: insert distribution

- The insert section lets you specify how data is inserted during stress.
- For each insert operation you can specify the following distributions/ratios:
  - Partition distribution — The number of partitions to update per batch (default FIXED(1))
  - select distribution ratio — The ratio of rows each partition should insert as a proportion of the total possible rows for the partition (as defined by the clustering distribution columns). default FIXED(1)/1

default

```
### Batch Ratio Distribution Specifications ###

insert:
  partitions: fixed(1)           # Our partition key is the domain so only insert one per batch

  select:    fixed(1)/1000       # We have 1000 posts per domain so 1/1000 will allow 1 post per batch

  batchtype: UNLOGGED            # Unlogged batches

```

Stress Tool: DML

- You can specify any CQL query on the table by naming them under the 'queries' section.
- The 'fields' field specifies if the bind variables should be picked from the same row or across all rows in the partition

```
58
59  #
60  # A list of queries you wish to run against the schema
61  #
62  queries:
63     singlepost:
64        cql: select * from blogposts where domain = ? LIMIT 1
65        fields: samerow
66     timeline:
67        cql: select url, title, published_date from blogposts where domain = ? LIMIT 10
68        fields: samerow
```