Session 4

NoSQL Database Systems

Big Data Analytics Technology, MSc in Data Science,
Coventry University UK

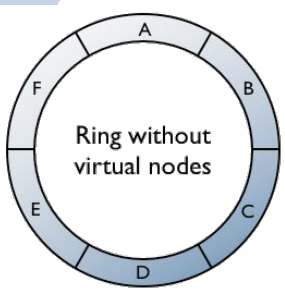
Miyuru Dayarathna

Cassandra Architecture

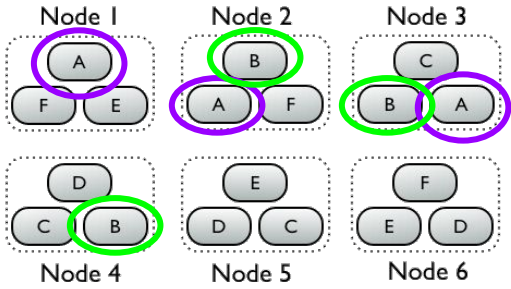
- Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure.
- Peer-to-peer distributed system
  - Assumption: System and hardware failures can and do occur
  - Coordinator : any node responsible for a particular client operation
- Key components:
  - **Virtual nodes** – assign data ownership to physical nodes
  - **Gossip** – exchanging information across the cluster
  - **Partitioner** – determines how to distribute the data across the nodes
  - **Replica placement strategy** – determines which nodes to place replicas on
- Cluster – stores data partitions of a Cassandra ring

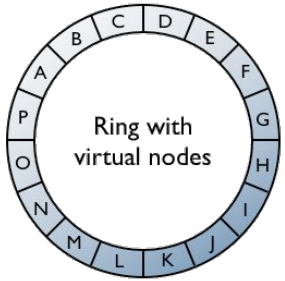
Cassandra Architecture - Virtual Nodes

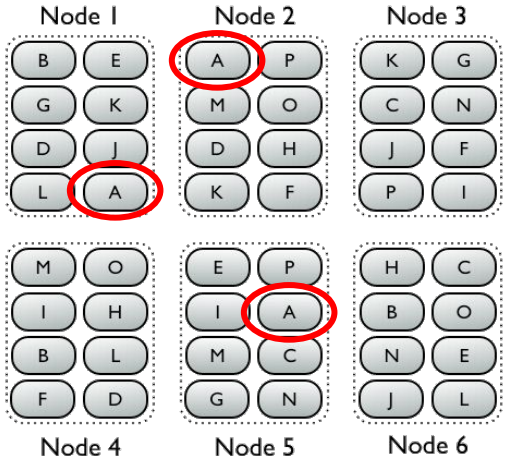
- Allows each node to own a large number of small partition ranges
  - Easier for adding/removing nodes - the small partition ranges are simply transferred
- Uses consistent hashing to distributed data



Example : Replication factor = 3







https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeDistribute.html

Cassandra Architecture

- The Cassandra architecture is very sophisticated and relies on the use of several different theoretical constructs (i.e., main pillars of Cassandra's construction).

- System Keyspace
- Peer-to-Peer
- Gossip and Failure Detection
- Anti-Entropy and Read Repair
- Memtables, SSTables, and Commit Logs
- Tombstones
- Hinted Handoff
- Compaction
- Bloom Filters
- Staged Event-Driven Architecture (SEDA)

Cassandra Architecture - System Keyspace

- Cassandra has an internal keyspace called **system** that it uses to store metadata about the cluster to aid in operations.

- This is similar to MSSQL Server's *master* and *tempdbs*
  - **master**: keeps information about disk space,usage, system settings, and general server installation notes

  - **tempdb**: used as a workspace to store intermediate results and perform general tasks

Cassandra Architecture - System Keyspace (Contd.)

- The system keyspace stores metadata for the local node as well as *hinted handoff* information
- The metadata includes
  - The node's token
  - The cluster name
  - Keyspace and schema definitions to support dynamic loading
  - Migration data
  - Whether or not the node is bootstrapped

Cassandra Architecture - System Keyspace (Contd.)

- The schema definitions are stored in two column families:
  - Schema column family : holds user keyspace and schema definitions
  - Migrations column family : records the changes made to a keyspace
- System keyspace cannot be modified

Caveats of Master/Slave setup

- In the traditional databases which can be deployed in multiple machines (E.g., MySQL, Google's Bigtable), some nodes are designated as masters and some are slaves.
- Masters and slaves have different roles in the overall cluster.
- The master acts as the authoritative source for data, and slaves synchronize their data to the master.
- Important ramification: Master can be single point of failure. In such a setup, master node can have far-reaching effects if it goes offline.

Cassandra Architecture - Peer-to-Peer

- Cassandra has a peer-to-peer distribution model where any given node is structurally identical to any other node.
- There is no **master** node that acts differently than a **slave** node.
- The aim of Cassandra's design is overall system availability and ease of scaling.
- The peer-to-peer design can improve general database availability, because while taking any given Cassandra node offline may have a potential impact on overall throughput, it is a graceful degradation that does not interrupt service.

Cassandra Architecture - Peer-to-Peer (Contd.)

- Because the behavior of each node is identical, in order to add a new server, you simply need to add it to the cluster.
- The new node will not immediately accept requests so that it has time to learn the topology of the ring and accept data that it may also be responsible for.
- After it does this, it can join the ring as a full member and begin accepting requests. This is largely automatic and requires minimal configuration. For this reason, the P2P
- design makes both scaling up and scaling down an easier task than in master/slave replication.

Cassandra Architecture - Gossip and Failure Detection

- Cassandra uses a gossip protocol for intra-ring communication so that each node can have state information about other nodes.
- Gossip process
  - Runs every second
  - Exchanges state messages with up to 3 other nodes in the cluster
  - Enables to detect failures
- Gossiped message
  - Information about a gossiping node + other nodes that it knows about
  - Acquired
    - Directly - By direct communication
    - Indirectly - Second hand, third hand, ..
  - Has a version
    - Older information is overwritten with the most current state

Cassandra Architecture - Partitioner

- Determines how data is distributed across the nodes
  - Including replicas
- Hash function for computing the token (hash) of a row key
- Types of partitioners:
  - **Murmur3Partitioner (default)** – uniformly distributes data across the cluster based on MurmurHash hash values
    - Non-cryptographic hash function
    - Values from -2^{63} to $+2^{63}$

Cassandra Architecture - Partitioner (Contd.)

Types of partitioners:

- **RandomPartitioner (default for previous versions)** - uniformly distributes data across the cluster based on MD5 hash values
  - Values is from 0 to 2^{127} -1
- **ByteOrderedPartitioner** – orders rows lexically by key bytes
  - "Hash" = hexadecimal representation of the leading character(s) in key
  - Allows ordered scans by primary key
  - Can have problems with load balancing

Cassandra Architecture - Replication

- All replicas are equally important
  - There is no primary or master replica
- When replication factor exceeds the number of nodes,writes are rejected
  - Reads are served as long as the desired consistency level can be met
- Replica placement strategies:
  - SimpleStrategy
  - NetworkTopologyStrategy

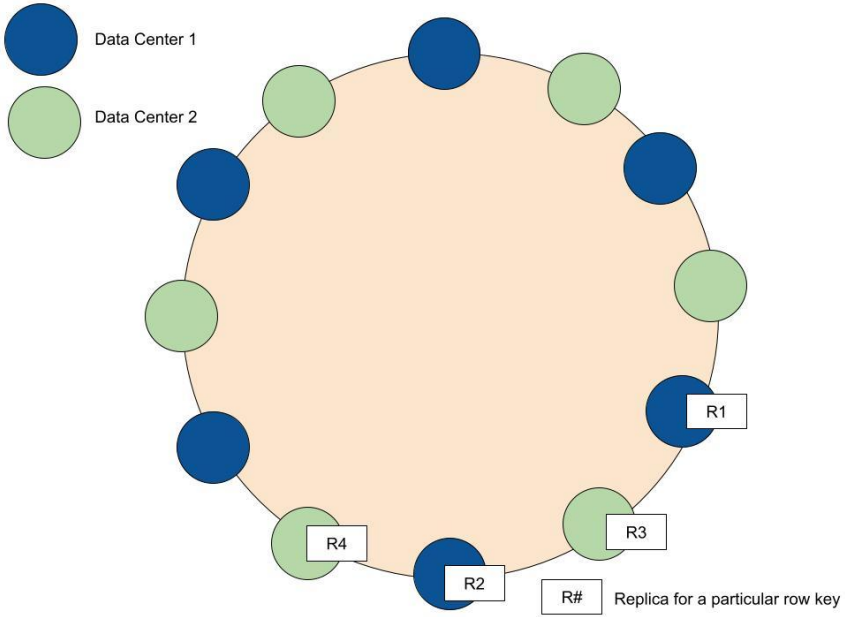
Cassandra Architecture - Replication (Contd.)

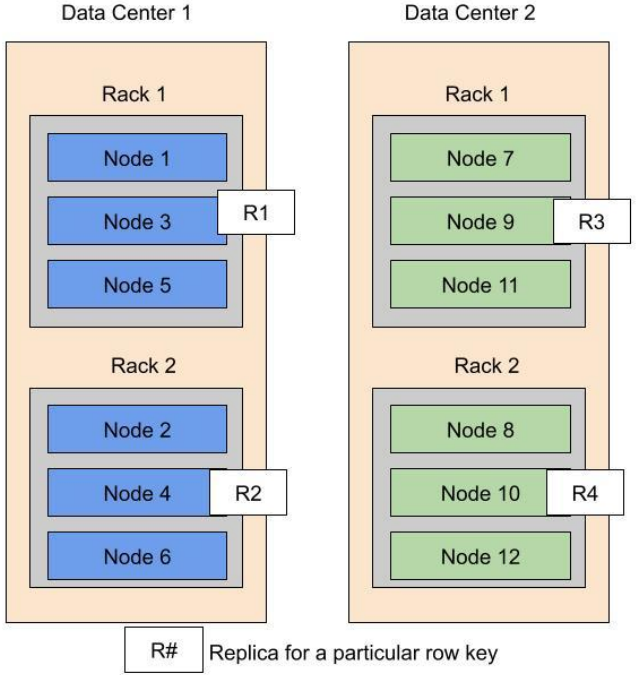
- SimpleStrategy
  - Places the first replica on a node determined by the partitioner
  - Additional replicas are placed on the next nodes clockwise in the ring
  - For a single data center only
    - Collection of related nodes, physical or virtual

Cassandra Architecture - Replication (Contd.)

- NetworkTopologyStrategy
  - Places replicas within a data center
    - We set number of replicas per a data center
      - The first replica is placed according to the partitioner
      - Additional replicas are placed by walking the ring clockwise until a node in a different rack is found
        - Motivation: nodes in the same rack often fail at the same
          - e.g., power, cooling, or network issue
      - If no such node exists, additional replicas are placed in different nodes in the same rack

Cassandra Architecture - Replication Example





Data centers = 2
Total replication factor = 4
(set per data center)

Replicas assigned to different racks

Cassandra Architecture - Replication (Contd.)

- How many replicas to configure in each data center?
  - Compromise between
    - Need for being able to satisfy reads locally
      - Without cross data-center latency
    - Failure scenarios
  - Most commonly: 2-3 replicas in each data center
  - Can be asymmetric (different replication factors for different data centers)

Cassandra Architecture - Snitches

- A snitch determines which data centers and racks nodes belong to.
- They inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into data centers and racks.
- The replication strategy places the replicas based on the information provided by the new snitch
- If we change snitches, we may need to perform additional steps because the snitch affects where replicas are placed.

Cassandra Architecture - Snitches (Contd.)

- All nodes must have exactly the same snitch configuration
- Various types:
  - **SimpleSnitch** – does not recognize data centers/racks
  - **RackInferringSnitch** – Racks and data centers are assumed to correspond to the 3rd and 2nd octet of the node's IP address
  - **PropertyFileSnitch** – uses a user-defined description of the network
  - **Dynamic snitching** – monitors performance of reads, chooses the best replica based on this history

Cassandra Architecture - Snitches (Contd.)

- Various types:
  - **GossipingPropertyFileSnitch** – Automatically updates all nodes using gossip when adding new nodes and is recommended for production.
  - **EC2Snitch** – Use the Ec2Snitch with Amazon EC2 in a single region.
  - **Ec2MultiRegionSnitch** – Use the Ec2MultiRegionSnitch for deployments on Amazon EC2 where the cluster spans multiple regions.
  - **GoogleCloudSnitch** – Use the GoogleCloudSnitch for Cassandra deployments on Google Cloud Platform across one or more regions.

  ...

Cassandra Architecture - Anti-Entropy and Read Repair

- Where you find gossip protocols, you will often find their counterpart, anti-entropy, which is also based on an epidemic theory of computing.
- Anti-entropy is the replica synchronization mechanism in Cassandra for ensuring that data on different nodes is updated to the newest version.
- Anti-entropy node repairs are important for every Cassandra cluster. Frequent data deletions and downed nodes are common causes of data inconsistency. Use anti-entropy repair for routine maintenance and when a cluster needs fixing by running the nodetool repair command.

https://docs.datastax.com/en/cassandra-oss/2.2/cassandra/operations/opsRepairNodesManualRepair.html

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- Cassandra accomplishes anti-entropy repair using Merkle trees, similar to Dynamo and Riak.
- Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version.
- Cassandra has two phases to the process:
  - Build a Merkle tree for each replica
  - Compare the Merkle trees to discover differences

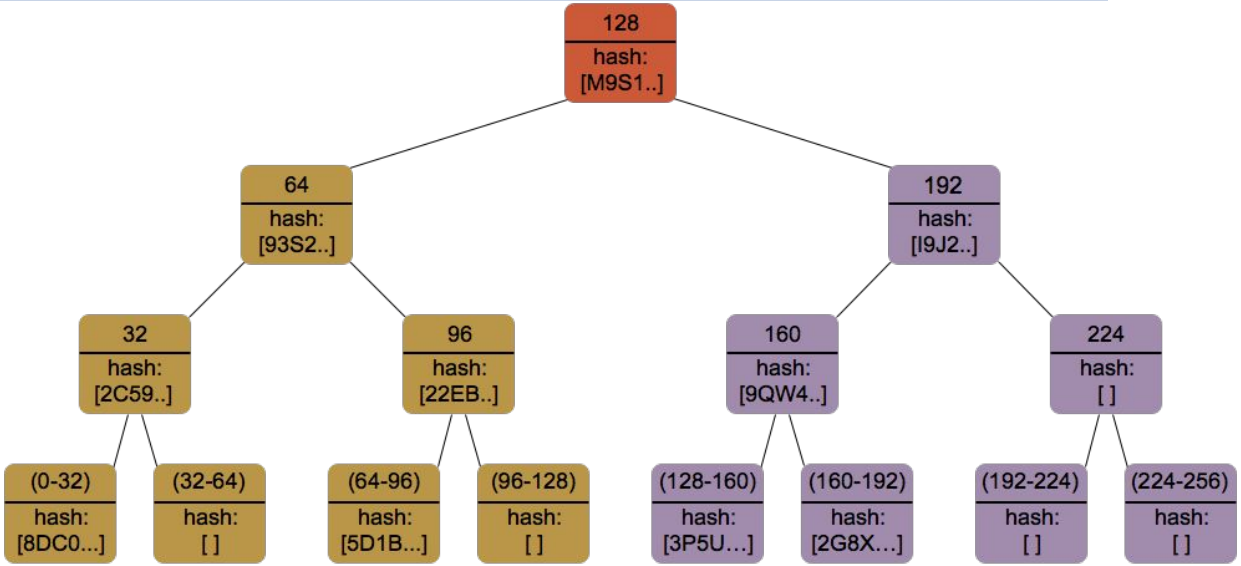
Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- Merkle trees are binary hash trees whose leaves are hashes of the individual key values.
- The leaf of a Cassandra Merkle tree is the hash of a row value.
- Each Parent node higher in the tree is a hash of its respective children.
- Because higher nodes in the Merkle tree represent data further down the tree, Casandra can check each branch independently without requiring the coordinator node to download the entire data set

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- For anti-entropy repair Cassandra uses a compact tree version with a depth of 15 (2^{15} = 32K leaf nodes).
- For example, a node containing a million partitions with one damaged partition, about 30 partitions are streamed, which is the number that fall into each of the leaves of the tree.
- Cassandra works with smaller Merkle trees because they require less storage memory and can be transferred more quickly to other nodes during the comparison process.

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)



Row key: jack	Row key: jill	Row key: terry	Row key: misty
Row token: 5	Row token: 7	Row token: 10	Row token: 20
hash: 8DC0…	hash: 5D1B…	hash: 3P5U…	hash: @G8X...

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- After the initiating node receives the Merkle trees from the participating peer nodes, the initiating node compares every tree to every other tree.
- If a difference is detected, the differing nodes exchange data for the conflicting range(s), and the new data is written to SSTables.
- The comparison begins with the top node of the Merkle tree. If no difference is detected, the process proceeds to the left child node and compares and then the right child node.

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- When a node is found to differ, inconsistent data exists for the range that pertains to that node.
- All data that corresponds to the leaves below that Merkle tree node will be replaced with new data
- For any given replica set, Cassandra performs validation compaction on only one replica at a time.

- Merkle tree building is quite resource intensive, stressing disk I/O and using memory.

Cassandra Architecture - Anti-Entropy and Read Repair - nodetool repair

- The nodetool repair command can be run on either a specified node or on all nodes if a node is not specified.
- The node that initiates the repair becomes the coordinator node for the operation.
- To build the Merkle trees, the coordinator node determines peer nodes with matching ranges of data. A major, or validation, compaction is triggered on the peer nodes.

Cassandra Architecture - Anti-Entropy and Read Repair - nodetool repair

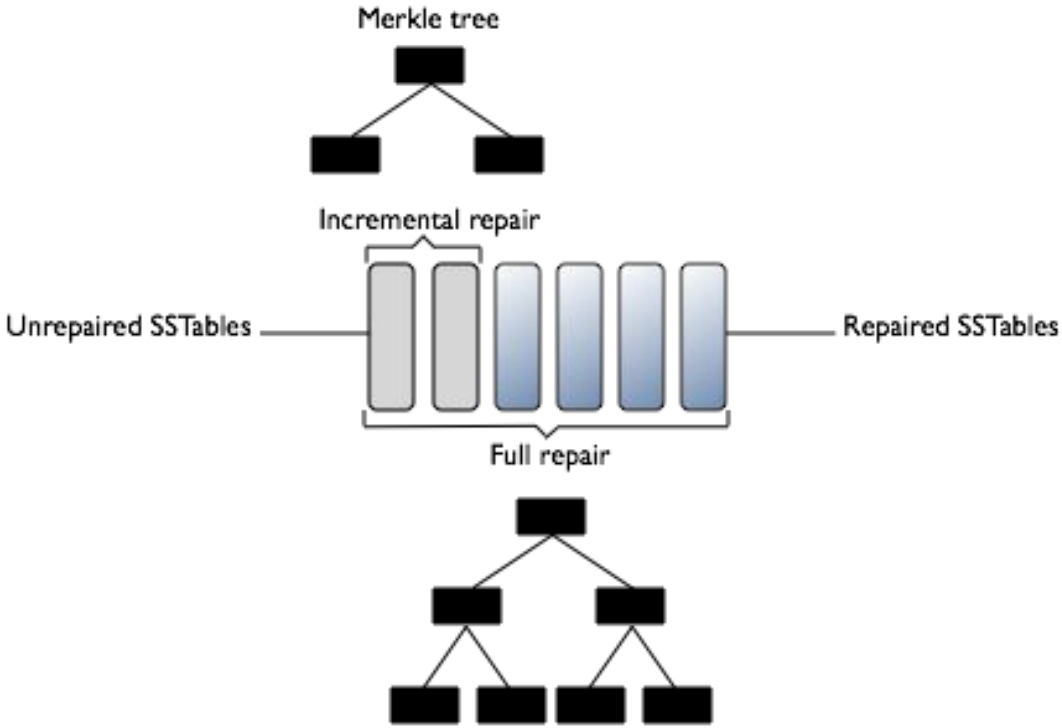
- The validation compaction reads and generates a hash for every row in the stored column families, adds the result to a Merkle tree, and returns the tree to the initiating node.
- Merkle trees use hashes of the data, because in general, hashes will be smaller than the data itself.

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- Full vs Incremental Repair
  - In the full repair of a node's data (like described above) Cassandra compares all SSTables for that node and makes necessary repairs
  - The default setting is incremental repair.
  - An incremental repair persists data that has already been repaired, and only builds Merkle trees for unrepaired SSTables.
  - This more efficient process depends on new metadata that marks the rows in an SSTable as repaired or unrepaired.

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

Merkle Trees of an Incremental Versus a Full Repair



Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- Reducing the size of the Merkle tree improves the performance of the incremental repair process, assuming repairs are run frequently.
- Incremental repairs work like full repairs, with an initiating node requesting Merkle trees from peer nodes with the same unrepaired data, and then comparing the Merkle trees to discover mismatches.
- Once the data has been reconciled and new SSTables built, the initiating node issues an anti-compaction command.
- Anti-compaction is the process of segregating repaired and unrepaired ranges into separate SSTables, unless the SSTable fits entirely within the repaired range.
- In the latter case, the SSTable metadata repairedAt is updated to reflect its repaired status.

Cassandra Architecture - Anti-Entropy and Read Repair (Contd.)

- Full repair is the default in Cassandra 2.1 and earlier.
- Incremental repair is the default for Cassandra 2.2 and later.
- In Cassandra 2.2 and later, when a full repair is run, SSTables are marked as repaired and anti-compacted.

Cassandra Architecture - Anti-Entropy and Read Repair - Parallel vs Sequential Repair

- Sequential repair takes action on one node after another.
- Parallel repair repairs all nodes with the same replica data at the same time.
- Sequential repair takes a snapshot of each replica. Snapshots are hardlinks to existing SSTables.
- They are immutable and require almost no disk space. The snapshots are active while the repair proceeds, then Cassandra deletes them.
- When the coordinator node finds discrepancies in the Merkle trees, the coordinator node makes required repairs from the snapshots

Cassandra Architecture - Anti-Entropy and Read Repair - Parallel vs Sequential Repair

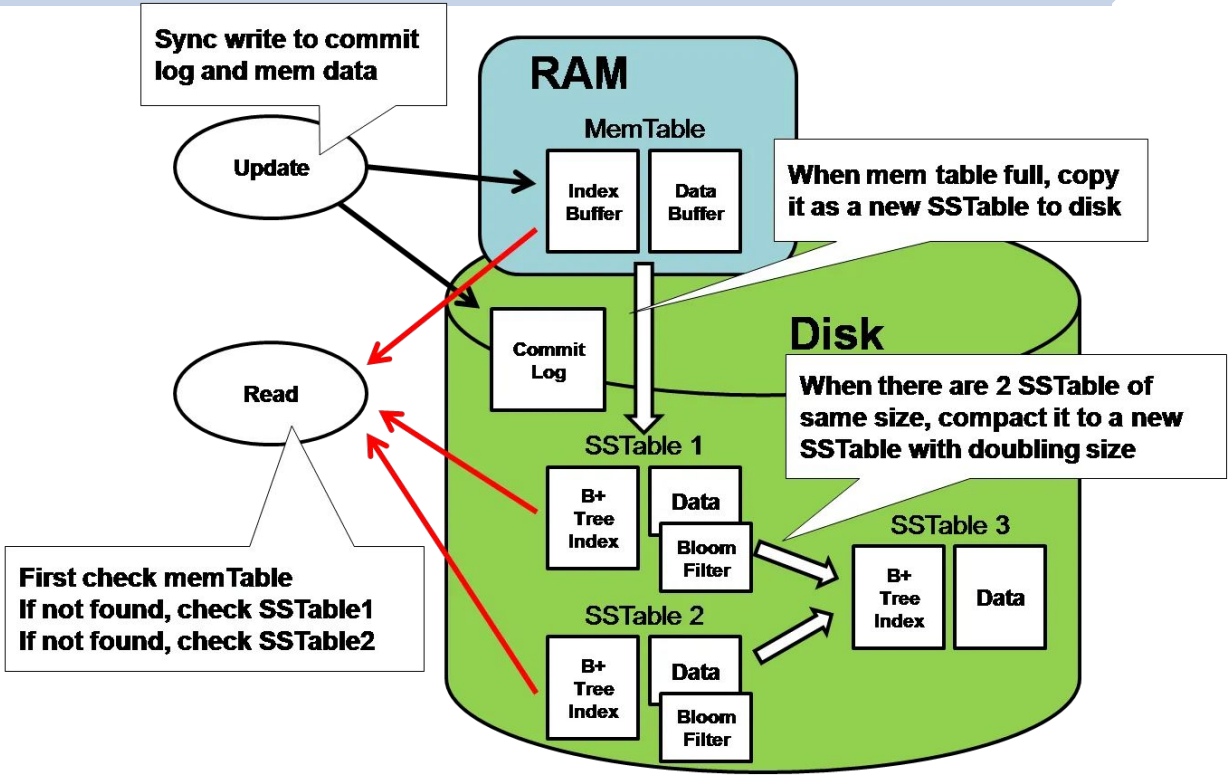
- Sequential repair is the default in Cassandra 2.1 and earlier.
- Parallel repair is the default for Cassandra 2.2 and later.

Memtables, SSTables, and Commit Logs

- When you perform a write operation, it's immediately written to the 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, to ensure that if a write operation does not make it to the in-memory store, it will still be possible to recover the data during failures.

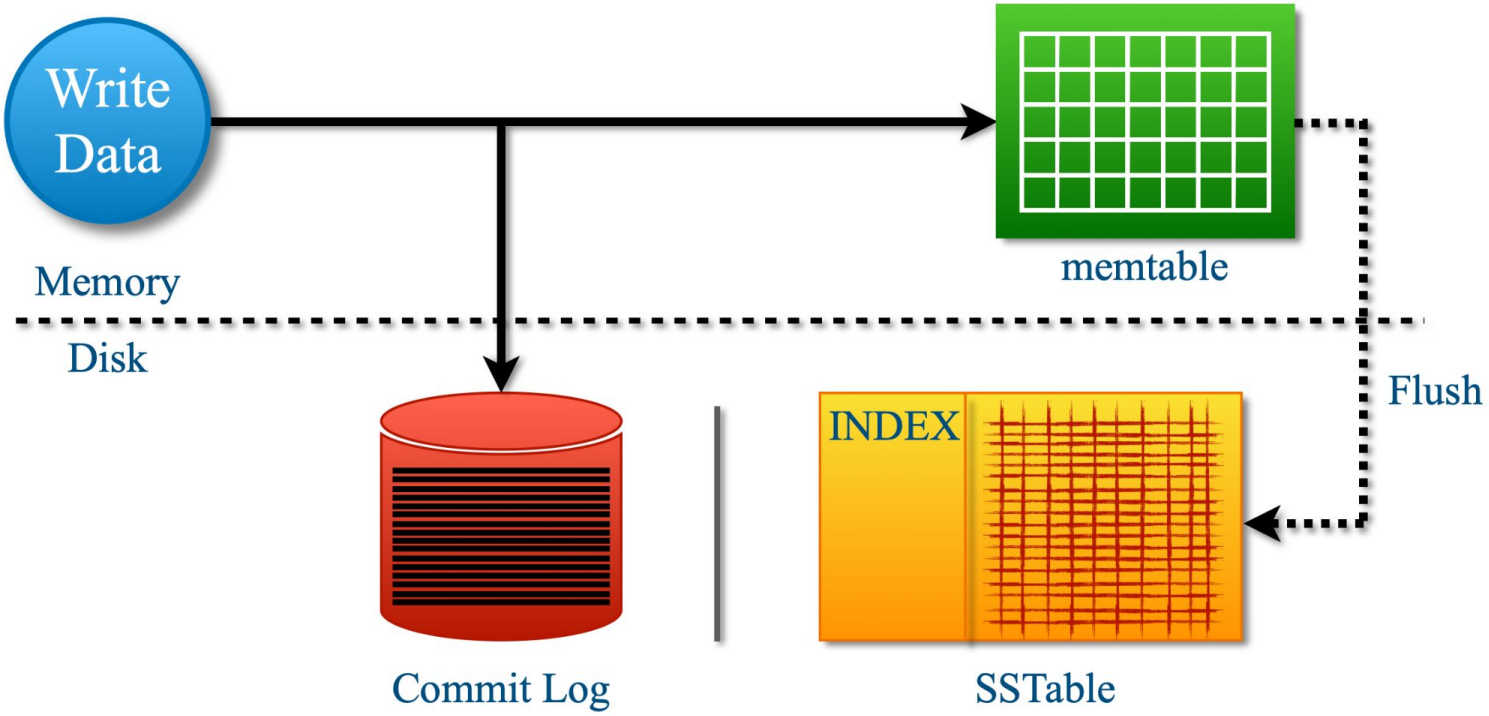
Memtables, SSTables, and Commit Logs



Memtables, SSTables, and Commit Logs

- After it's written to the commit log, the value is written to a memory (RAM)-resident data structure called the memtable.
- When the number of objects stored in the memtable reaches a threshold, the contents of the memtable are flushed to disk in a file called an SSTable.
- The flushing of data from memtable(RAM) to SSTables(Disk) is done using the configurable threshold or when the commit log threshold *commitlog_total_space_in_mb* is exceeded.
- A new memtable is then created. This flushing is a non-blocking operation.

Memtables, SSTables, and Commit Logs



https://www.cloudduggu.com/cassandra/read-and-write-operation/

Memtables, SSTables, and Commit Logs

- Multiple memtables may exist for a single column family, one current and the rest waiting to be flushed.
- They typically should not have to wait very long, as the node should flush them very quickly unless it is overloaded.
- The Data is written on the SSTables tables which are immutable which means when the memtable is flushed the data is not overwritten in SSTables despite a new file being created.
- The partitions are stored on multiple SSTables so that they can be easily searched.

Memtables, SSTables, and Commit Logs

- The SSTable is a concept borrowed from Google's Bigtable.
- Once a memtable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application.
- Despite the fact that SSTables are compacted, this compaction changes only their on-disk representation.
- It essentially performs the "merge" step of a mergesort into new files and removes the old files on success.

Memtables, SSTables, and Commit Logs

- Each SSTable also has an associated Bloom filter, which is used as an additional performance enhancer.
- All writes are sequential, which is the primary reason that writes perform so well in Cassandra.
- No reads or seeks of any kind are required for writing a value to Cassandra because all writes are append operations.
- This makes one key limitation on performance the speed of your disk.

Memtables, SSTables, and Commit Logs

- Compaction is intended to amortize the reorganization of data, but it uses sequential IO to do so. Therefore, the performance benefit is gained by splitting.
- The write operation is just an immediate append, and then compaction helps to organize for better future read performance.
- If Cassandra naively inserted values where they ultimately belonged, writing clients would pay for seeks up front.
- On reads, Cassandra will check the memtable first to find the value.

Cassandra Architecture - Hinted Handoff

- Lets take the scenario of a write request is sent to Cassandra, but the node where the write properly belongs to is not available due to network partition, hardware failure, or some other reason.
- In order to ensure general availability of the ring in such a situation, Cassandra implements a feature called hinted handoff.
- A *hint* can be considered as a little post-it note that contains the information from the write request.

Cassandra Architecture - Hinted Handoff (Contd.)

If the node where the write belongs has failed, the Cassandra node that receives the write will create a hint, which is a small reminder that says, “I have the write information that is intended for node B. I’m going to hang onto this write, and I’ll notice when node B comes back online; when it does, I’ll send it the write request.” That is, node A will “hand off” to node B the “hint” regarding the write.

Cassandra Architecture - Compaction

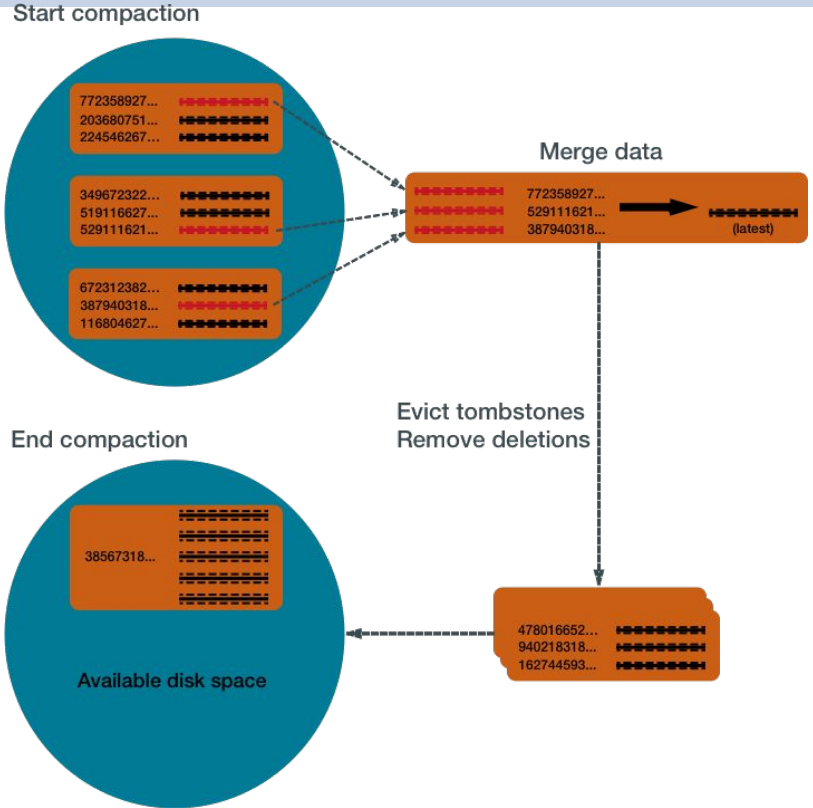
- A compaction operation in Cassandra is performed in order to merge SSTables.

- During compaction,
  - the data in SSTables is merged: the keys are merged
  - columns are combined
  - tombstones are discarded
  - a new index is created.

Cassandra Architecture - Compaction (Contd.)

- Compaction is the process of freeing up space by merging large accumulated datafiles.
- This is roughly analogous to rebuilding a table in the relational world.
- But as Stu Hood points out, the primary difference in Cassandra is that it is intended as a transparent operation that is amortized across the life of the server.

Cassandra Architecture - Compaction (Contd.)



- Compaction works on a collection of SSTables.
- From these SSTables, compaction collects all versions of each unique row and assembles one complete row, using the most up-to-date version (by timestamp) of each of the row's columns.

Cassandra Architecture - Compaction (Contd.)

- The merge process is performant, because rows are sorted by partition key within each SSTable, and the merge process does not use random I/O.
- The new versions of each row is written to a new SSTable.
- The old versions, along with any rows that are ready for deletion, are left in the old SSTables, and are deleted as soon as pending reads are completed.
- Compaction causes a temporary spike in disk space usage and disk I/O while old and new SSTables co-exist. As it completes, compaction frees up disk space occupied by old SSTables.
- It improves read performance by incrementally replacing old SSTables with compacted SSTables.

Cassandra Architecture - Compaction (Contd.)

- There are a bounded number of SSTables to inspect to find the column data for a given key.
- If a key is frequently mutated, it's very likely that the mutations will all end up in flushed SSTables.
- Compacting them prevents the database from having to perform a seek to pull the data from each SSTable.
- Cassandra can read data directly from the new SSTable even before it finishes writing, instead of waiting for the entire compaction process to finish.

Cassandra Architecture - Compaction (Contd.)

- As Cassandra processes writes and reads, it replaces the old SSTables with new SSTables in the page cache.
- The process of caching the new SSTable, while directing reads away from the old one, is incremental — it does not cause a the dramatic cache miss.
- Cassandra provides predictable high performance even under heavy load.

Cassandra Architecture - Compaction Strategies

- Cassandra supports different compaction strategies, which control,
  - How SSTables are chosen for compaction
  - How the compacted rows are sorted into new SSTables.
- Each strategy has its own strengths

Cassandra Architecture - Compaction Strategies

- **SizeTieredCompactionStrategy (STCS)**
  - Recommended for write-intensive workloads.
- **LeveledCompactionStrategy (LCS)**
  - Recommended for read-intensive workloads.
- **TimeWindowCompactionStrategy (TWCS)**
  - Recommended for time series and expiring TTL workloads.
- **DateTieredCompactionStrategy (DTCS)**

https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlHowDataMaintain.html#dmlHowDataMaintain__dml_which_compaction_strategy_is_best

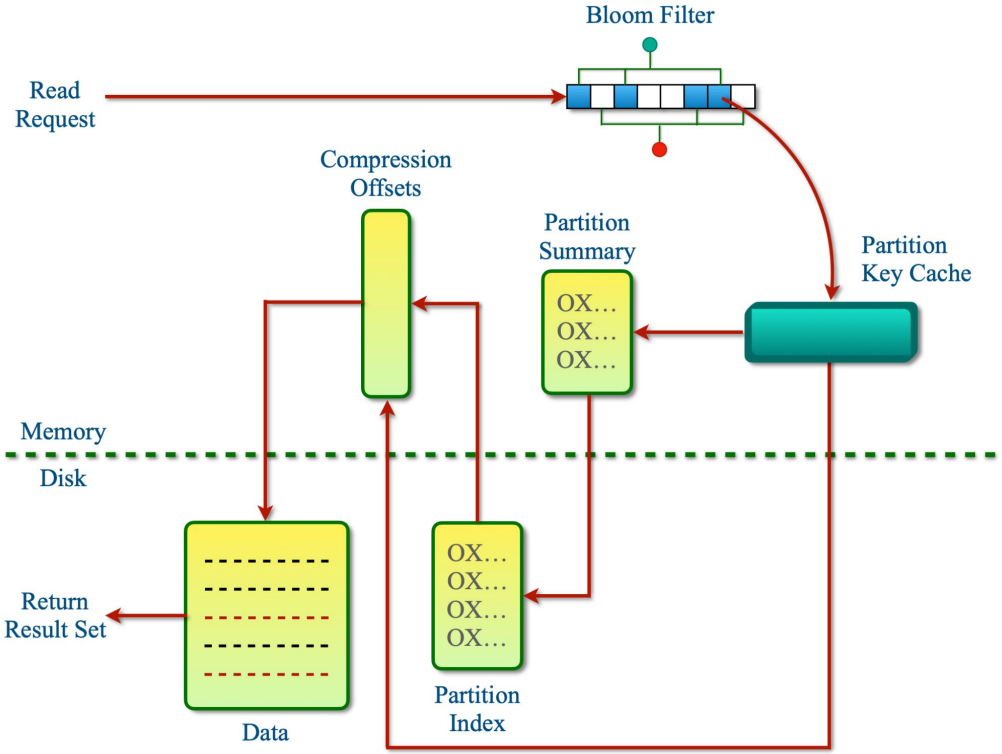
Cassandra Architecture - Bloom Filters

- Bloom filters are used as a performance booster. They are named for their inventor, Burton Bloom.
- Bloom filters are very fast, nondeterministic algorithms for testing whether an element is a member of a set.
- They are non deterministic because it is possible to get a false-positive read from a Bloom filter, but not a false-negative.

Cassandra Architecture - Bloom Filters (Contd.)

- 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. 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 disk access on key lookups. Disk access is typically much slower than memory access. Therefore, in a way, a Bloom filter is a special kind of cache.
- When a query is performed, the Bloom filter is checked first before accessing disk.

Cassandra Architecture - Bloom Filters (Contd.)



https://www.cloudduggu.com/cassandra/read-and-write-operation/

Cassandra Architecture - Bloom Filters (Contd.)

- Because false-negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly doesn’t; but if the filter thinks that the element is in the set, the disk is accessed to make sure.

Cassandra - Data read operation

- The Cassandra Read operation goes through different stages to find out exact data starting from the data present in the Memtable(RAM) till the data present in the SSTable(DISK) files.
- First, the Read request will be made from the Client
- The request data will be checked in the memtable(RAM). If the requested data is present then data will be read from memtable(RAM) and merged with SSTables(DISK) files to send final data to the client.

Cassandra - Data read operation

- If the row cache is enabled then it will be checked to find the data.
- The Bloom Filters are loaded in the Heap memory that will be checked to find out the SSTables file that can store the requested partition data.
- Since Bloom Filters works on probabilistic function and can return false positives.
- In some cases Bloom Filters does not return the SSTable file then Cassandra further checks in the partition key cache.

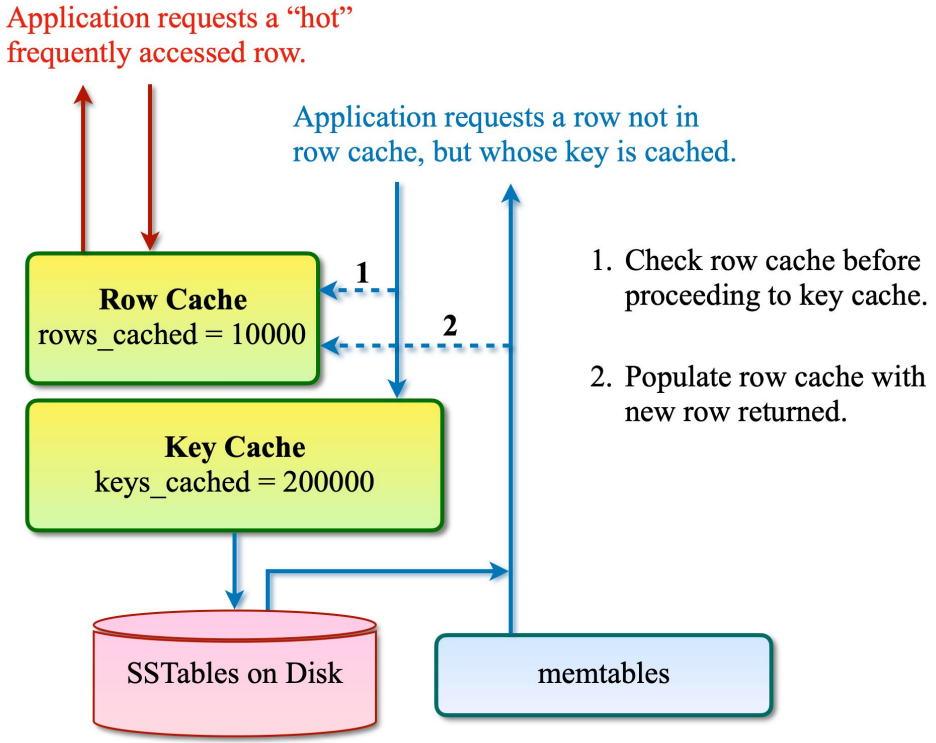
Cassandra - Data read operation (Contd.)

- The Partition Key Cache is used to store the partition index in heap memory and the partition index of data will be searched in that.
- If the Partition Key is present in the Partition Key Cache then Cassandra will go to compression offset to find the Disk that has the data.
- If the Partition Key is not present in the Partition Key Cache then the partition summary is searched to find user-requested data.

Cassandra - Data read operation (Contd.)

- Partition Index is used to store the Partition key of the data that will be used in the Compression offset map to find out the exact location of the Disk which has stored the data.
- The Compression offset map is used to hold the exact location of data. It uses the Partition key to locate that.
- Once the Compression offset map indicates the location where data is stored the further process is to fetch the data and share it with the user.

Cassandra - Data read operation (Contd.)



The flow and steps of the read operation in Cassandra when the data will be present in the Row Cache(the frequent access data are stored here).

https://www.cloudduggu.com/cassandra/read-and-write-operation/

Cassandra - Structure of SSTable

Cassandra creates the following structure from each of the SSTable.

- **Data (Data.db)**
  - Cassandra stores the data in the data.db file of SSTable.
- **Primary Index (Index.db)**
  - This structure is used to store the index of the row key that points to the data which are stored in the data file.
- **Bloom filter (Filter.db)**
  - It is a memory stored structure that is used to check the data is present in the memtable before going to the SSTables.

Cassandra - Structure of SSTable (Contd.)

- **Compression Information (CompressionInfo.db)**
  - This file is used to store the detail of uncompressed data length, chunk offsets, and other important details concerning compression.
- **Statistics (Statistics.db)**
  - This file is used to store the Statistical metadata information of the data stored in the SSTable.
- **SSTable Index Summary (SUMMARY.db)**
  - It is used to store the SSTable partition Index Summary and some portions in the memory.

Cassandra Architecture - Tombstones

- Soft Delete - In the relational world, instead of actually executing a delete SQL statement, the application will issue an update statement which changes a value in a column called something like "deleted".

- 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.

Cassandra Architecture - Tombstones (Contd.)

- There's a related setting called Garbage Collection Grace Seconds.
- This is the amount of time that the server will wait to garbage-collect a tombstone.
- By default, it's set to 864,000 seconds, the equivalent of 10 days. Cassandra keeps track of tombstone age, and once a tombstone is older than GCGraceSeconds, it will be garbage-collected.
- 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.

Staged Event-Driven Architecture (SEDA)

- Cassandra implements a Staged Event-Driven Architecture (SEDA).
- SEDA is a general architecture for highly concurrent Internet services, originally proposed in a 2001 paper : "*SEDA: An Architecture for Well-Conditioned, Scalable Internet Services" by Matt Welsh, David Culler, and Eric Brewer*
- In a typical application, a single unit of work is often performed within the confines of a single thread.
- A write operation, for example, will start and end within the same thread.

Staged Event-Driven Architecture (SEDA) (Contd.)

- However, Cassandra's concurrency model is different and is based on SEDA
- A single operation may start with one thread, which then hands off the work to another thread, which may hand it off to other threads.
- But it’s not up to the current thread to hand off the work to another thread.
- Instead, work is subdivided into what are called stages, and the thread pool (really, a java.util.concurrent.ExecutorService) associated with the stage determines execution.

Staged Event-Driven Architecture (SEDA) (Contd.)

- A stage is a basic unit of work, and a single operation may internally state-transition from one stage to the next. Because each stage can be handled by a different thread pool, Cassandra experiences a massive performance improvement.

- This SEDA design also means that Cassandra is better able to manage its own resources internally because different operations might require disk IO, or they might be CPU-bound, or they might be network operations, and so on, therefore, the pools can manage their work according to the availability of these resources.

Staged Event-Driven Architecture (SEDA) (Contd.)

- Operations which are represented as stages in Cassandra
  - Read
  - Mutation
  - Gossip
  - Response
  - Anti-Entropy
  - Load Balance
  - Migration
  - Streaming

Key-Value Store

- A key-value store, or key-value database is a simple database that uses an associative array (think of a map or dictionary) as the fundamental data model where each key is associated with one and only one value in a collection.
- In each key-value pair the key is represented by an arbitrary string such as a filename, URI or hash.
- The value can be any kind of data like an image, user preference file or document.
- The value is stored as a blob requiring no upfront data modeling or schema definition.

Key-Value Store (Contd.)

- The storage of the value as a blob removes the need to index the data to improve performance.
- However, you cannot filter or control what's returned from a request based on the value because the value is opaque.
- In general, key-value stores have no query language.
- They provide a way to store, retrieve and update data using simple get, put and delete commands; the path to retrieve data is a direct request to the object in memory or on disk.
- The simplicity of this model makes a key-value store fast, easy to use, scalable, portable and flexible.

Key-Value Store - Scalability and reliability

- Key-value stores scale out by implementing partitioning (storing data on more than one node), replication and auto recovery.
- They can scale up by maintaining the database in RAM and minimize the effects of ACID guarantees (a guarantee that committed transactions persist somewhere) by avoiding locks, latches and low-overhead server calls.

Key-Value Store - Use cases

Key-value stores handle size well and are good at processing a constant stream of read/write operations with low latency making them perfect for:

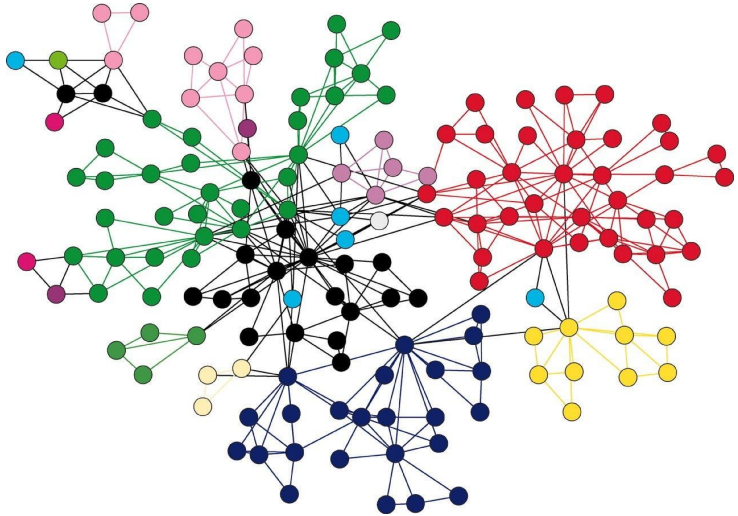
- Session management at high scale
- User preference and profile stores
- Product recommendations; latest items viewed on a retailer website drive future customer product recommendations
- Ad servicing; customer shopping habits result in customized ads, coupons, etc. for each customer in real-time
- Can effectively work as a cache for heavily accessed but rarely updated data

Some popular Key-Value Stores

- Aerospike
- Apache Cassandra
- Berkeley DB
- Couchbase Server
- Redis
- Riak
- Amazon SimpleDB
- Tokyo Cabinet
- Microsoft Dynomite
- Amazon Dynamo
- Project Voldemort

Graph Databases

- Instead of storing data in tables or columns, graph databases use three basic constructs to represent data:
  - nodes
  - edges
  - properties.



Graph Databases (Contd.)

Graph databases differ from other non-relational offerings such as key-value stores in that they represent the edges as first-class citizens, and not just the nodes

In graph databases, the relationships are given equal status with the nodes, as the relationship between the nodes is considered central to certain use cases.

Popular Graph Databases

- Neo4j
- Dex
- HypergraphDB
- Infogrid
- VertexDB
- FlockDB
- TigerGraph

Benchmarking NoSQL Systems - YCSB

- It is hard to compare performance of NoSQL systems
  - **Different Data Models**
    - BigTable model in Cassandra and HBase
    - Simple hashtable model in Voldemort
    - Document model in CouchDB
  - **Different Design Decisions**
    - Read or write optimized
    - Synchronous or asynchronous replication
    - Latency or durability: sync writes to disk at write time or not

The Yahoo! Cloud Serving Benchmark (YCSB)

- The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source database benchmarking suite and a critical analytical component of cloud-based database management system (DBMS) evaluation.

- It allows users to comparatively measure how various modern SQL and NoSQL DBMS perform simple database operations on generated datasets.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears (Yahoo! Research), SoCC'2010

YCSB Goals

- Build a standard benchmark for cloud serving systems
  - Evaluate different systems on different workloads
    - A package of standard workloads
    - A workload generator
- Evaluate both performance and scalability
  - Future direction: availability and replication

YCSB Client

- Define the dataset and load it into the database
  - Data records
  - A set of operations
- Execute the operations while measuring performance

YCSB Workloads

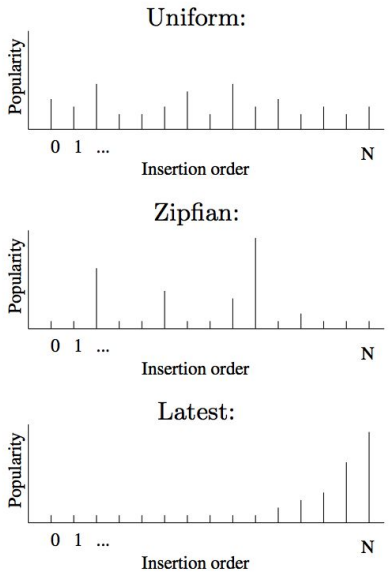
- A table of data records
  - Each with fields.
  - Each record is identified by a primary key.
  - The values of each record is a random string of length .
  - E.g. 1000 byte records, =10, =100.

- Basic operations
  - Insert, update, read, scan

YCSB Workloads (Contd.)

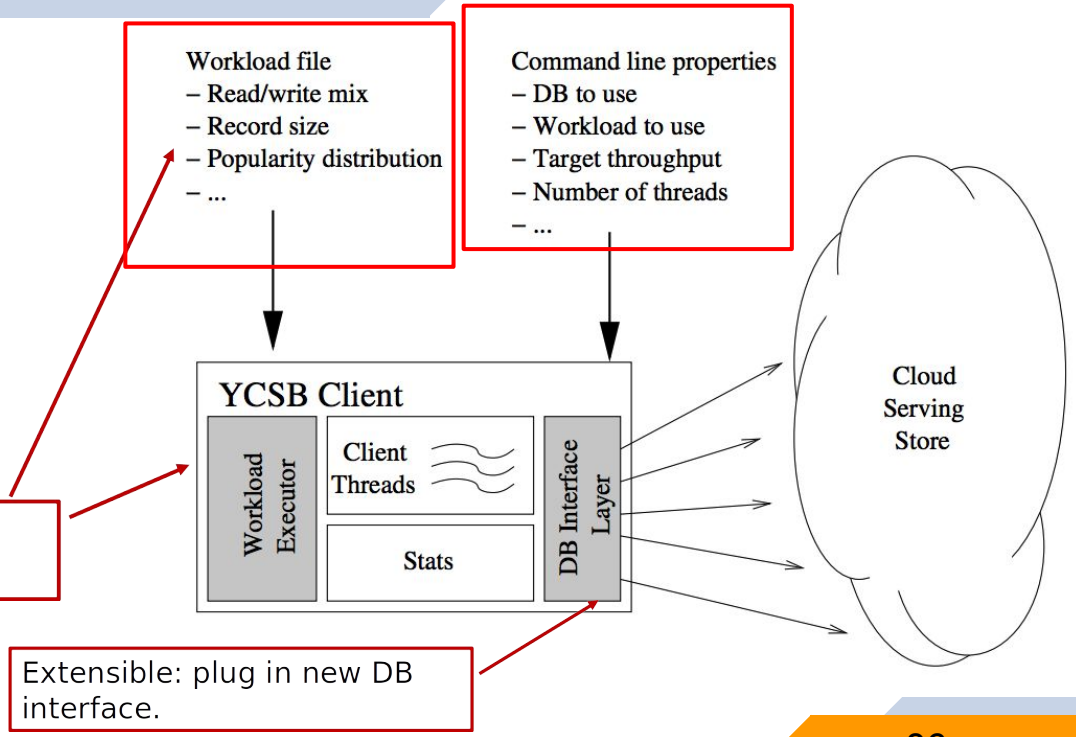
- One workload is a mix of basic operations with choices of
  - Which operations to perform
  - Which record to read or write
  - How many records to scan

- Decisions are governed by random distributions
  - Uniform
  - Zipfian
  - Latest
  - Multinomial



YCSB Client Architecture

- Java program
- Client threads
  - Load the database
  - Execute the workload
  - Measure the latency and achieved throughput
- Extensibility



YCSB - Experimental Setup

- Servers
  - Six storage servers
  - YCSB client runs on a separate server.
- Tested systems
  - Cassandra, HBase, PNUTS, and Shared MySQL.
- Database
  - 120 million 1KB records = 20 GB per sever

YCSB - Experimental Setup (Contd.)

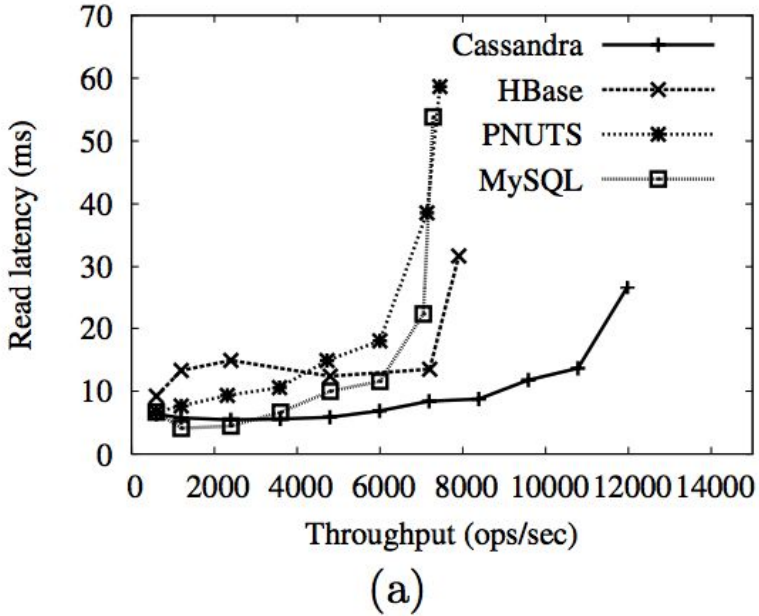
YCSB provides a package of standard workloads.

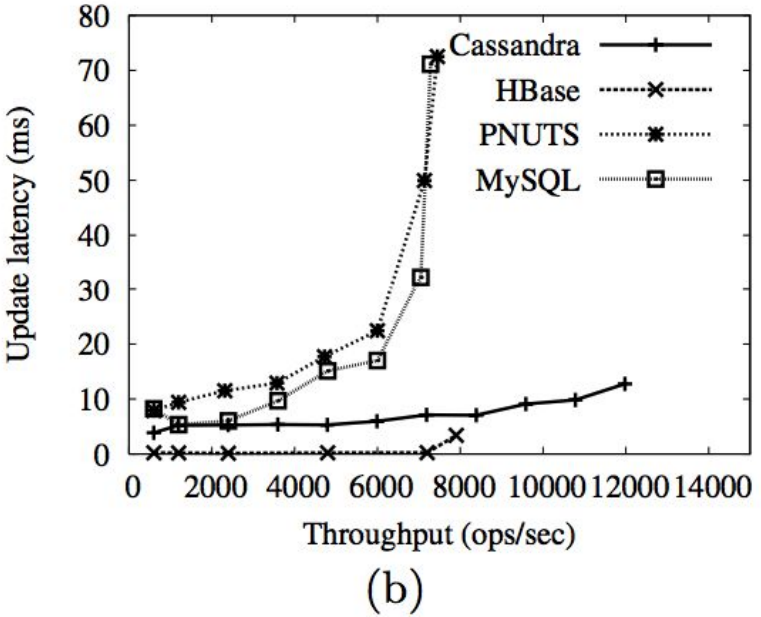
Workload	Operations	Record selection	Application example
A—Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B—Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C—Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D—Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E—Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform*	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

*Workload E uses the Zipfian distribution to choose the first key in the range, and the Uniform distribution to choose the number of records to scan.

YCSB - Workload A: update heavy

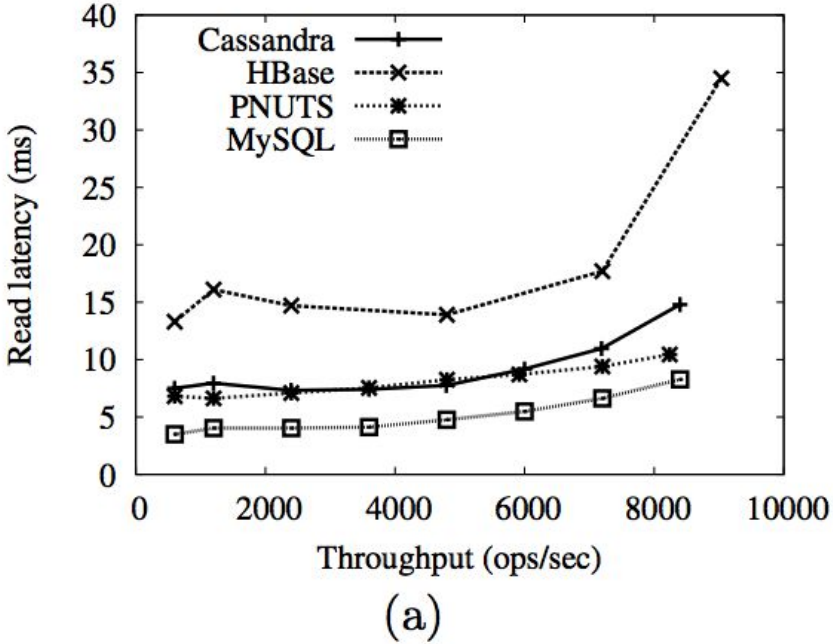
50% reads and 50% updates.

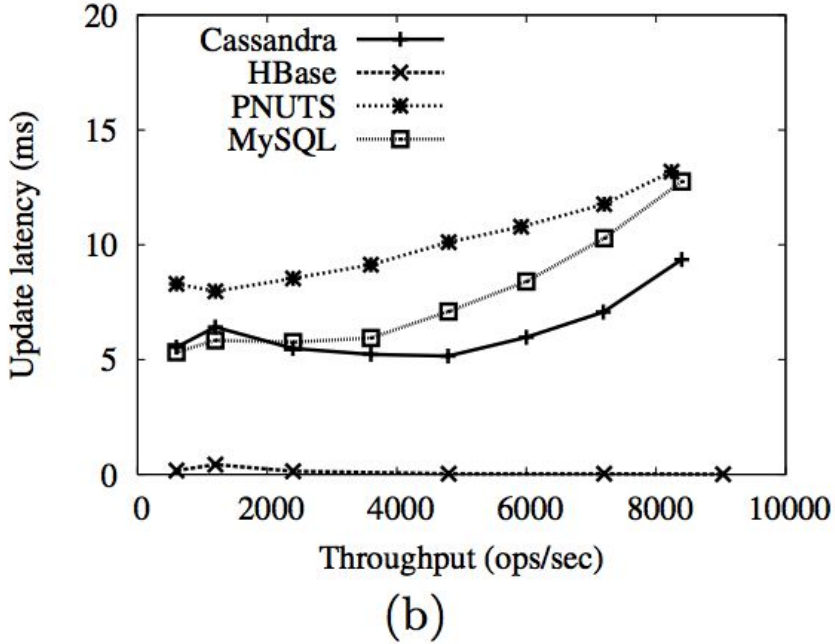




YCSB - Workload B: Read heavy

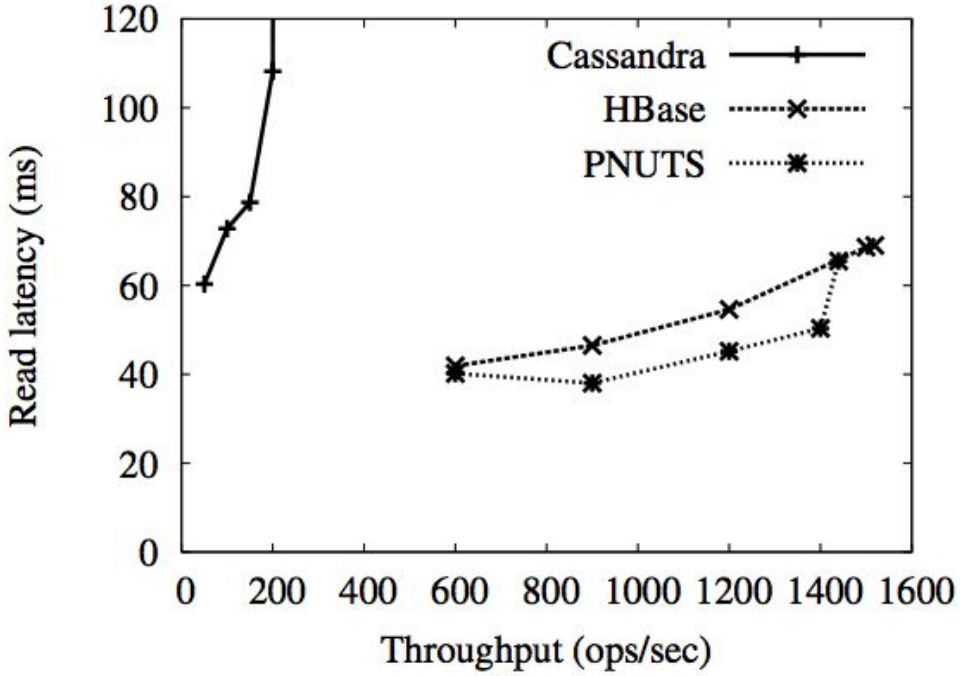
95% reads and 5% updates.





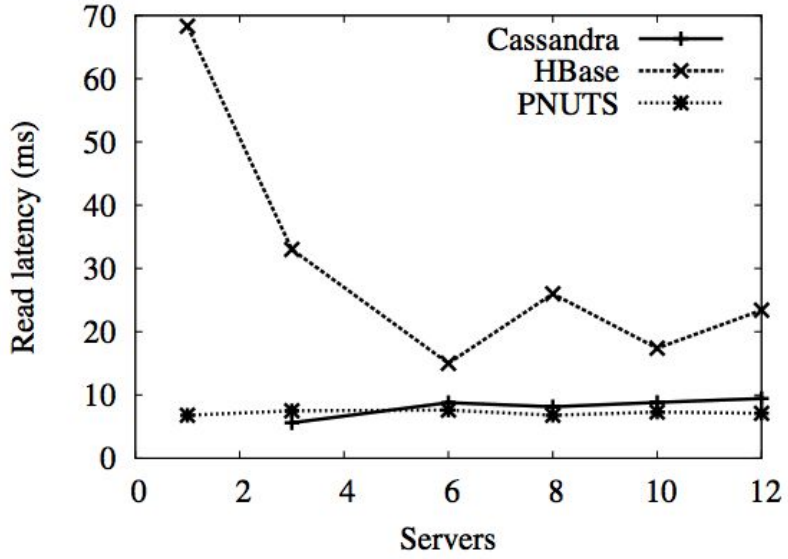
YCSB - Workload E: Short ranges

Scans of 1-100 records of size 1KB



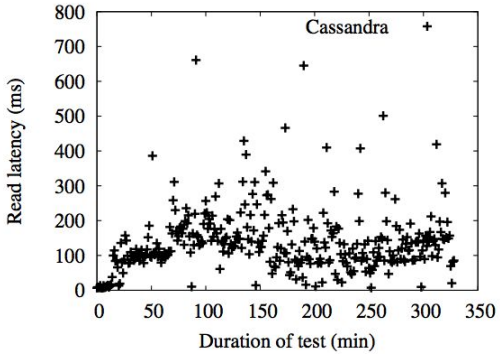
YCSB - Scalability (1)

Scaleup: Vary the number of storage severs from 2 to 12 while varying the data size and request rate proportionally.

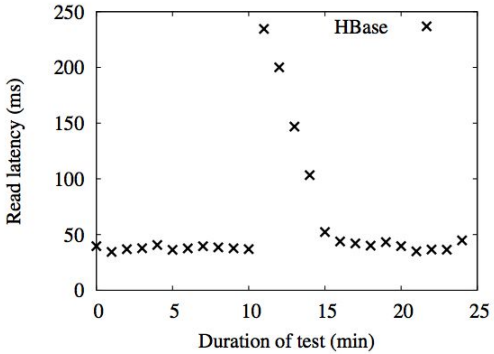


YCSB - Scalability (2)

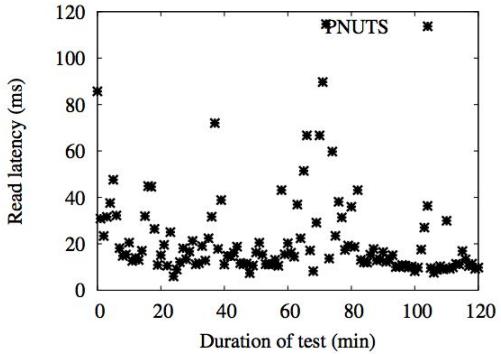
Elastic speedup: run a read-heavy workload on 2 servers; add a 3rd server, then 4th, then 5th, then 6th.



(a) Cassandra



(b) HBase



(c) PNUTS

NoSQL - Pros

- No predefined schemas
- Handle Bigdata
- Handle semi structured data
- Cheaper to manage
- Scale out/horizontal scaling

NoSQL - Downsides

- Data normalization
- Relational data
- Lacks well established query language such as SQL
- Lacks ACID Compliance
- Lacks data integrity

Thank you!