Chapter 6: Architecture

Link to chapter on Safari

Let's break this chapter into four categories of increasing granularity:

1. Node layout
2. Data layout
3. Data structures
4. Algorithms

Node Layout

Data Centers and Racks

A data center is a logical grouping of nodes in the same physical location; a rack is a logical grouping of nodes in the same physical rack in a DC. Cassandra starts with `DC1` and `RAC1` by default. Metadata on infrastructure is managed via gossip.

Gossip protocols

Cassandra's `Gossiper` class maintains metadata state on a Cassandra cluster. The protocol is, once per second:

1. Origin sends a `syn` message
2. Receiver sends `ack` to origin
3. Origin sends `ack2` to receiver

Cassandra's gossip protocol has to detect dead nodes. It does this with **Phi Accrual Failure Detection**. `Phi` is a measure of suspicion - how likely is it that this node is down? This value is pegged to network volatility, so it's much more flexible than traditional heartbeat-based monitoring.

Snitches

Snitches are a class meant to discover the locality and topography of nodes, so that requests can be more effectively routed. If a read comes into a node with a quorum of `N`, Cassandra uses information gathered by snitches to figure out the nearest nodes for routing.

Snitches usually implement an interface for a specific cloud deployment (Amazon EC2, Google Cloud). They can also be wrapped (decorator pattern?) with a `DynamicEndpointSnitch` class that layers on dynamic performance information (latency, availability) to the basic topographical information. That way, nearby underperformers can be slowly excluded from optimal routing, based on a modified version of the Phi failure deetction described above.

Data Layout

Rings and Tokens

Cassandra nodes are laid out in a ring, where each node is responsible for a range of `tokens`, where a single token is a 64-bit integer ID. Incoming data contains a partition key - that key is hashed into a token, and then assigned to to the node responsible for the range of that token.

Virtual Nodes

This method of `node <=> token` mapping was the norm until 1.2 - at that point, the idea of `virtual nodes` was introduced.

The process is simple:

1. Break up the entire token range into small ranges.
2. Assign a given `N` of these ranges (so, tokens) to each physical machine.
3. That's it! Note now one node isn't responsible for one contiguous range, but many smaller, non-continguous ranges. This helps load and infrastructural tasks.

Partitioners

Given a partition key column, we need to generate a token for placing the containing row into the Cassandra ring. We do this with a `partitioner`. The default since 1.2 is the "Murmur3Partitioner", based on Murmer hashing.

Replication Strategies

A given node in the ring can be a replica for a given range of data. The replication strategy (implemented in code via the Strategy pattern) is behind the *AbstractReplicationStrategy* interface. For example, *SimpleStrategy* places the replicated data in consecutive nodes in the ring (starting from the node that owns the token for the given partition key).

Consistency Levels

Say you have a replication of `N = 3`. This is specified at the keyspace level. Now, for RW access, you can specify a `consistency level` at the query / client level to designate how to get a piece of data from the owning key range / node and its replicas.

For example, `QUORUM` says for a set of replicas `N`, you need acknowledgement from `N / 2 + 1` replicas for a given read / write before the result is sent back to the client. Other available consistency levels include `ONE`, `TWO`, `ALL`.

Strong consistency is considered to be `R + W > N`, where `R` is read replicas used, `W` is write replicas used, and `N` is the replication factor.

Queries and Coordinator Nodes

All of the above data layout information comes together for querying clients in their read / write paths. When a RW operation hits the cluster, it can choose any node to serve as the *coordinator node*. The coordinator node

then figures out which node owns the token range for the request's partition key, and routes it on to that node and all needed replicas.

Data Structures

Now that we understand the layout of data in Cassandra, and how requests are mapped onto that layout, let's talk about the data structures and algorithms Cassandra uses to maximize performance and availability.

Memtables, SSTables, and Commit Logs

Let's say now a given write request has been routed to the appropriate owning node. The three general steps for persisting a write request to a single node go as follows:

1. Write request immediately to **commit log** with flush flag = 1
2. Add write to in-memory, sorted **memtable**
3. Flush memtables periodically to persisted, immutable **SSTable**

SSTables are sorted thanks to the memtable, which is sorted via some typical B-Tree like algorithm. SSTables also can be compressed and use Bloom filters to help read performance.

This process shows why writes in Cassandra are fast - it's just an append operation with an asynchronous flush! Reads are slower, since a given key has to be looked up through all existing SSTables (or in the memtable first, if the data hasn't been flushed yet).

Note that memtables used to be stored on the JVM heap, but are now in native memory for Cassandra - good for avoiding GC overhead!

Bloom filters

Bloom filters are a probabilistic data structure meant to aid in performance in SSTable partition key seeks. They check for record existence and a set, and will quickly answer either "maybe yes" or "definitely no". Behind the scenes, this is just clever bit mapping and hashing, sorta akin to a HyperLogLog's cleverness on cardinality estimates. For each SSTable on a node, an in-memory bloom filter will be accessed first to see if disk access on that SSTable can be skipped ("definitely no" for the given partition key).

Caching

1. *Key cache* - a map of (partition key -> row index), which helps find a record in SSTables faster.
2. *Row cache* - stores whole rows off heap. Great for very frequently accessed records.
3. *Counter cache* - reduces lock contention for frequently accessed counters.

These caches are periodically saved to disk to reduce warming periods on bounces. Row caching is disabled by default.

Algorithms

How does Cassandra manage consistency and performance in its distributed, leaderless setup? CAP is a problem,

right? Let's study a few approaches Cassandra takes to distributed consistency and performance.

Hinted Handoff

If a node goes down, a coordinator node can create a *hint* that says to hold on to the write for now, and then deliver the write to an owning node once that node is back online. This allows for some wiggle room in the durability vs. performance trade off, but NOT for consistency. So if your replication level is `N = 3`, and you have two nodes that persist and one instance of a hinted handoff for the last one, that WON'T be considered persisted at a consistency level of `THREE`.

Lightweight Transactions and Paxos

Cassandra uses Paxos to implement "lightweight" transactions. The coordinator is used as a Paxos leader, and the set of R/W operations in the transaction will be committed with linearizable consistency at all replicas. Cassandra actually adds two additional phases to the standard "prepare + commit" Paxos semantics:

1. Prepare/Promise
2. Read/Results
3. Propose/Accept
4. Commit/Ack

Note - lightweight transactions are limited to a single partition. The Paxos state (sequence numbers, last accepted proposal) is stored per partition.

Tombstones

Tombstones refer to how Cassandra performs a "soft delete" - instead of synchronously actually removing data from underlying SSTables / memtables, the row is marked for deleting and then actually removed during compaction. The removal time is also determined by the *Garbage Collection Grace Seconds* configuration - a tombstoned record will only be deleted after the `GCGS` time period has elapsed for a tombstoned record.

Compaction

Compaction is the process by which SSTables are merged to help read performance - keys are merged, columns combined, tombstones removed, and new indexes created (yielding the `Data`, `Index`, and `Filter` files of each SSTable). This is a transparent, background process. This is good to reduce seek time - if a key is frequently mutated, do you really want disk access on every SSTable?

There are several compaction strategies:

- `SizeTieredCompactionStrategy` - the default. Organizes SSTables into tiers - once a tier hits a certain number of SSTables, they're compacted. Good for write-intensive tables.
- `LeveledCompactionStrategy` - all SSTables are the same size and organized into progressively larger levels. A given key is guaranteed to only appear once per level. Good for read-intensive workflows that need stable latencies.
- `DateTieredCompactionStrategy` - a newer strategy that organizes and compacts SSTables by their Cassandra write time. Very good for access patterns that are biased towards recent data.

Anti-Entropy, Repair, and Merkle Trees

Anti-entropy protocols are gossip protocols used by Cassandra to repair data inconsistency between replicas. There are two kinds:

- `read repair` - at read time, Cassandra detects whether there are deviations from the latest data between replicas used for quorum, and will repair that discrepency at its replica during the read.
- `anti-entropy repair` - when `nodetool repair` is invoked and a major SSTable compaction begins, the node in question will initiate a "TreeRequest/TreeReponse conversation" involving the exchange of Merkle trees (a binary tree where every parent node is a hash of its children) to detect deviation.

Staged Event-Driven Architecture (SEDA)

The following functions are represented as individual stages (i.e. an event queue, a handler, and a thread pool) in Cassandra's SEDA-esque architecture:

```
Read (local reads), Mutation (local writes), Gossip, Request/response (interactions with other nodes), Anti-entropy (nodetool repair), Read repair, Migration (making schema changes), Hinted handoff
```

Not going to spend too much time on SEDA - see other notes for more on it.

Cassandra has moved away from pure SEDA over time, as has its originator (see article here) - sometimes batching is best!