Apache Cassandra

Distributed NoSQL database. It implements a partitioned wide column storage model with eventually consistent semantics. It has been designed to meet the following challeng objectives:-

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

Features: Cassandra provides CQL; allows users to organize data within a cluster of Cassandra nodes using

→ Keyspace: specifies the replication strategy for a dataset across different datacenters. Replication refers to the no. of copies stored within a cluster. Keyspaces serve as containers for tables.

→ Table: Comprised of rows & cols. Cols define the typed schema for a single datum in a table. Tables are

Date: / /

partitioned based on the cols. provided in the partition key. Cassandra can flexibly add new cols to tables with zero ~~ada~~ downtime.

→ Partition: Defines the mandatory part of the pk key all rows in Cass. must have to identify the node in a cluster where the row is stored. All performant queries supply the partition key in the query.

→ Row: Contains a collection of columns identified by a unique pk key made up of the partition key and optionally clustering keys.

→ Column: A single datum with a type which belongs to a row.

CQL supports numerous advanced features over a partitioned dataset such as:

→ Collection types including sets, maps and lists
→ User defined types, tuples, functions & aggregates
→ Storage-attached indexing (SAI) for secondary indexes
→ local secondary indexes (2i)
→ ~~Us~~ Single-partition lightweight transactions with atomic compare & set semantics.
→ (Experimental) materialized views

Cassandra explicitly chooses not to implement ops that require cross-partition coordination as they are typically slow & hard to provide highly available global semantics. For example, Cassandra does not provide:

→ Cross-partition transactions

→ Distributed joins

→ Foreign keys or referential integrity.

Dynamo

Cass. borrows from Amazon's Dynamo distributed storage key-value system. Each node in Dynamo has

→ Request coordination over a partitioned dataset }

→ Ring membership & failure detection. } +

→ A local persistence (storage) engine ← LSM tree

In particular, Cassandra relies on Dynamo style

→ Dataset partitioning using consistent hashing

→ Multi-master replication using versioned data & tunable consistency.

→ Distributed cluster mem. & failure detection via a gossip protocol

→ Incremental scale-out on commodity HW.

Partitioning: As every replica can independently accept

Mo Tu We Th Fr Sa Su

mutations to every key that it owns, every key must be versioned. Unlike Dynamo whose determinism versions & vector clocks were used to reconcile concurrent updates to a key, Cass. uses a simpler last-write-wins model. Formally, Cass. uses LWW Element set CRDT (Conflict replicated Data Type) for each CQL row to resolve conflicting mutations on replicas.

Cass. maps every node to one or more tokens on a continuous hash ring & defines ownership by hashing a key onto the ring and then walking the ring in one direction, similar to 'Chord' algorithm.

Simple single token consistent hashing works well if there are many physical nodes to spread over. But with evenly spaced tokens and a small no. of physical nodes, incremental scaling is difficult because there are no token selections for new nodes that can leave the ring balanced → uneven request load. Hence, use Virtual nodes → assign multiple tokens to each phy. node. Hence make small clusters look larger. However, this slows cluster-wide maintenance, increases probability of an outage (due to more combinations of node failure) and reduce performance of ops over token ranges. Cass. has a deterministic token allocator to intelligently pick tokens such that the ring is optimally balanced while requiring a much lower no. of tokens per phy. node.

Replication: Cass. supports pluggable replication strategies which determine which physical nodes act as replicas for a given token range. Every keyspace of data has its own replication strategy. Production → NetworkTopologyStrategy Testing → SimpleStrategy.

→ NetworkTopology: Recommended over SimpleStrategy in production; makes it easier to add new physical or virtual datacenters to the cluster later, if required.

In addition to allowing the replication factor to be specified individually by datacenter, it also attempts to choose replicas within a datacenter from different racks as specified by the Snitch.

→ SimpleStrategy: allows a single integer replication factor. Treats all nodes identically, ignoring any configured DCs or racks.

Data Versioning: Cass. uses mutation timestamp versioning to guarantee eventual consistency. Cass. correctness depends on their clocks → proper time sync. process such as NTP

Replica Sync: As replicas in Cassandra can accept mutations independently, it is possible for some replicas to have newer data than others. Cassandra has many best-effort techniques to drive convergence of replicas including Replica read repair in the read path and Hinted handoff in the write path.
These techniques are only best effort, however, and to guarantee eventual consistency, Cassandra implements anti entropy repair where replicas calculate hierarchical hash trees over their datasets called Merkle tree that can be compared across replicas to identify mis-matched data. Like Dynamo, Cass. supports full ~~read~~ repairs where replicas hash their entire dataset, create Merkle trees and send them to each other and sync. any ranges that dont match. Unlike Dynamo, Cass. also implements sub-range repairs & incremental repairs. Sub-range repair allows Cas. to increase the resolution of

the hash trees (potentially down to the single partition level) by creating a large no. of trees that span only a portion of the data range. Inc. repairs allows Cas. to only repair the partitions that have changed since the last repair.

<u>Tunable Consistency</u> : Cas. supports a per-operation tradeoff between consistency & availability through consistency levels, a version of Dynamo's R+W > N. Generally, writes will be visible to subsequent reads when the read consistency level contains enough nodes to guarantee a quorum intersection with the write consistency level.

The following CLs are available:

→ ONE: Only a single replica must respond

→ TWO: 2 reps must respond

→ THREE: 3 reps —

→ QUORUM : A majority (n/2 + 1) of the reps must respond

→ ALL : All reps must respond

→ LOCAL_QUORUM: A maj of reps in the local DC (whichever DC the coordinator is in) must respond

→ EACH_QUORUM : A maj of reps in each DC must respond

→ LOCAL_ONE: Only a single rep. must respond. In a multi-DC cluster, this also guarantees that read requests are not sent to reps in a remote DC.

→ ANY: A single rep. may respond, or the coordinator may store a hint. If a hint is stored, the coordinator will later attempt to replay the hint & deliver the mutation to the replicas. The consistency level is only accepted for write ops.

Write ops are always sent to all reps. CL controls how many responses the coordinator waits for before responding to the client. For read ops, the coordinator generally only issues read commands to enough replicas to satisfy the CL. Exception: retry.

Gossip: Nodes exchange state info about themselves and the nodes they know about → version with a vector clock of (generation, version) tuples,
generation → monotonic timestamp
version → logical clock that increments roughly every sec.
these logical clocks allows Cass. gossip to ignore old versions of cluster state just by inspecting the logical

clocks presented with gossip messages.
Every node in Cass. runs gossip independently & periodically.
Every second, every node in the cluster
→ Updates the local node's heartbeat state (the version)
and constructs the node's local view of the cluster gossip
endpoint state.
→ Picks a random other node in the cluster to exchange
gossip endpoint state with.
→ Probabilistically attempts to gossip with any unreachable
nodes (if one exists)
→ Gossips with a seed node if that didn't happen in step 2.
When an operator first bootstraps a Cassandra cluster,
they designate certain nodes as seed nodes. Any node can
be a seed node. → Bootstrap into the ring without seeing
any other seed nodes. After bootstrap, due to step 4,
seed nodes become gossip hotspots.
Gossip also propagates token metadata & schema
version info. This info forms the control plane for
scheduling data movements & schema pulls. For eg, if a
node sees a mismatch in schema version in gossip state,
it will schedule a schema sync task with the other node.

As token info. propagates via gossip it is also the control plane for teaching nodes which endpoints own what data.

Gossip forms the basis of ring membership but the failure detector ultimately makes decisions about node health (UP/DOWN). Every node in Cassandra runs a variant of the Phi Accrual Failure Detector.

Storage Engine: Optimized for high performance, write-oriented workloads. Based on Log Structured Merge (LSM) trees, which utilize an append-only approach instead of B-trees. Creates ~~E~~ a write path free of read lookups & bottlenecks.

Tradeoffs in terms of read performance & write amplification → caused by compaction, rewrite several SSTables multiple times. To enhance read ops, Cass. uses Bloom filters.

The core storage engine consists of memtables for in-memory data & immutable SSTables on disk. Data in SSTables is stored sorted to enable efficient

merge sort during compaction. Additionally, a WAL referred to as the commit log, ensures resiliency for crash & transaction recovery.
Sequence of steps in write path
→ logging data in the commit log
→ writing data to the memtable
→ Flushing data from the memtable
→ Storing data on disk in sstables.

Data must be read & written in a sequential order.
Paxos is used to implement lightweight transactions → handle concurrent ops using linearizable consistency. It ensures transaction isolation with compare & set transaction (CAS). With CAS, replica data is compared & data that is found to be out of date is set to the most consistent value. Reads with linearizable consistency allows reading the current state of the data, which may possibly be uncommitted, without making a new addition or update.

High Availability! - Fault tolerance. Detect failure → gossip
Durability :- Replicas.