Apache Cassandra : Architecture

Architecture

Architecture

This section describes the general architecture of Apache Cassandra.

- **Overview**
- **Dynamo**
- **Storage Engine**
- **Guarantees**
- **Snitches**

Overview

Overview

Apache Cassandra is an open-source, distributed NoSQL database. It implements a partitioned wide-column storage model with eventually consistent semantics.

Cassandra was initially designed at Facebook using a staged event-driven architecture (SEDA). This initial design implemented a combination of Amazon's Dynamo distributed storage and replication techniques and Google's Bigtable data and storage engine model. Dynamo and Bigtable were both developed to meet emerging requirements for scalable, reliable and highly available storage systems, but each had areas that could be improved.

Apache Cassandra was designed as a best-in-class combination of both systems to meet emerging largescale, both in data footprint and query volume, storage requirements. As applications began to require full global replication and always available low-latency reads and writes, a new kind of database model was required to meet these new requirements. Relational database systems at that time struggled to meet the the requirements.

Apache Cassandra was designed to meet these challenges with the following design objectives in mind:

- 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 and cluster growth
- Partitioned key-oriented queries
- Flexible schema

Features

Cassandra provides the Cassandra Query Language (**CQL**), an SQL-like language, to create, modify, and delete database schema, as well as access data. CQL allows users to organize data within a cluster of **Cassandra** nodes using:

- Keyspace: Defines how a dataset is replicated, per datacenter. Replication is the number of copies saved per cluster. Keyspaces contain tables.
- Table: Tables are composed of rows and columns. Columns define the typed schema for a single datum in a table. Tables are partitioned based on the columns provided in the partition key. **Cassandra** tables can flexibly add new columns to tables with zero downtime.
- Partition: Defines the mandatory part of the primary key all rows in **Cassandra** 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 primary key made up of the partition key and optionally additional 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 and aggregates
- Storage-attached indexing (SAI) for secondary indexes
- Local secondary indexes (2i)
- User-defined types, functions and aggregates
- Single-partition lightweight transactions with atomic compare and set semantics
- (Experimental) materialized views

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

- Cross-partition transactions
- Distributed joins
- Foreign keys or referential integrity.

Operating

Apache Cassandra configuration settings are configured in the `cassandra.yaml` file that can be edited by hand or with the aid of configuration management tools. Some settings can be manipulated live using an online interface, but others require a restart of the database to take effect.

Cassandra provides tools for managing a cluster. The `nodetool` command interacts with Cassandra's live control interface, allowing runtime manipulation of many settings from `cassandra.yaml`. The `auditlogviewer` is used to view the audit logs. The `fqltool` is used to view, replay and compare full query logs.

In addition, **Cassandra** supports out of the box atomic snapshot functionality, which presents a point in time (PIT) snapshot of Cassandra's data for easy integration with many backup tools. **Cassandra** also supports incremental backups where data can be backed up as it is written.

Dynamo

Dynamo

Apache Cassandra relies on a number of techniques from Amazon's Dynamo distributed storage key-value system. Each node in the Dynamo system has three main components:

- Request coordination over a partitioned dataset
- Ring membership and failure detection
- A local persistence (storage) engine

Cassandra primarily draws from the first two clustering components, while using a storage engine based on a Log Structured Merge Tree (LSM). In particular, Cassandra relies on Dynamo style:

- Dataset partitioning using consistent hashing
- Multi-master replication using versioned data and tunable consistency
- Distributed cluster membership and failure detection via a gossip protocol
- Incremental scale-out on commodity hardware

Cassandra was designed this way to meet large-scale (PiB+) business-critical storage requirements. In particular, as applications demanded full global replication of petabyte-scale datasets along with always available low-latency reads and writes, it became imperative to design a new kind of database model as the relational database systems of the time struggled to meet the new requirements of global-scale applications.

Dataset Partitioning: Consistent Hashing

Cassandra achieves horizontal scalability by partitioning all data stored in the system using a hash function. Each partition is replicated to multiple physical nodes, often across failure domains such as racks and even datacenters. As every replica can independently accept mutations to every key that it owns, every key must be versioned. Unlike in the original Dynamo paper where deterministic versions and vector clocks were used to reconcile concurrent updates to a key, Cassandra uses a simpler last-write-wins model where every mutation is timestamped (including deletes) and then the latest version of data is the "winning" value. Formally speaking, Cassandra uses a Last-Write-Wins Element-Set conflict-free replicated data type for each CQL row, or LWW-Element-Set CRDT, to resolve conflicting mutations on replica sets.

Consistent Hashing using a Token Ring

Cassandra partitions data over storage nodes using a special form of hashing called consistent hashing. In naive data hashing, you typically allocate keys to buckets by taking a hash of the key modulo the number of buckets. For example, if you want to distribute data to 100 nodes using naive hashing you

might assign every node to a bucket between 0 and 100, hash the input key modulo 100, and store the data on the associated bucket. In this naive scheme, however, adding a single node might invalidate almost all of the mappings.

Cassandra instead maps every node to one or more tokens on a continuous hash ring, and defines ownership by hashing a key onto the ring and then "walking" the ring in one direction, similar to the Chord algorithm. The main difference of consistent hashing to naive data hashing is that when the number of nodes (buckets) to hash into changes, consistent hashing only has to move a small fraction of the keys.

For example, if we have an eight node cluster with evenly spaced tokens, and a replication factor (RF) of 3, then to find the owning nodes for a key we first hash that key to generate a token (which is just the hash of the key), and then we "walk" the ring in a clockwise fashion until we encounter three distinct nodes, at which point we have found all the replicas of that key. This example of an eight node cluster with gRF=3 can be visualized as follows:

[image] | *ring.png*

You can see that in a Dynamo-like system, ranges of keys, also known as **token ranges**, map to the same physical set of nodes. In this example, all keys that fall in the token range excluding token 1 and including token 2 (grange(t1, t2]) are stored on nodes 2, 3 and 4.

Multiple Tokens per Physical Node (vnodes)

Simple single-token consistent hashing works well if you have many physical nodes to spread data over, but with evenly spaced tokens and a small number of physical nodes, incremental scaling (adding just a few nodes of capacity) is difficult because there are no token selections for new nodes that can leave the ring balanced. Cassandra seeks to avoid token imbalance because uneven token ranges lead to uneven request load. For example, in the previous example there is no way to add a ninth token without causing imbalance; instead we would have to insert 8 tokens in the midpoints of the existing ranges.

The Dynamo paper advocates for the use of "virtual nodes" to solve this imbalance problem. Virtual nodes solve the problem by assigning multiple tokens in the token ring to each physical node. By allowing a single physical node to take multiple positions in the ring, we can make small clusters look larger and therefore even with a single physical node addition we can make it look like we added many more nodes, effectively taking many smaller pieces of data from more ring neighbors when we add even a single node.

Cassandra introduces some nomenclature to handle these concepts:

- **Token**: A single position on the Dynamo-style hash ring.
- **Endpoint**: A single physical IP and port on the network.
- **Host ID**: A unique identifier for a single "physical" node, usually present at one gEndpoint and containing one or more gTokens.

- **Virtual Node** (or **vnode**): A gToken on the hash ring owned by the same physical node, one with the same gHost ID.

The mapping of **Tokens** to **Endpoints** gives rise to the **Token Map** where Cassandra keeps track of what ring positions map to which physical endpoints. For example, in the following figure we can represent an eight node cluster using only four physical nodes by assigning two tokens to every node:

[image] | *vnodes.png*

Multiple tokens per physical node provide the following benefits:

1. When a new node is added it accepts approximately equal amounts of data from other nodes in the ring, resulting in equal distribution of data across the cluster.
2. When a node is decommissioned, it loses data roughly equally to other members of the ring, again keeping equal distribution of data across the cluster.
3. If a node becomes unavailable, query load (especially token-aware query load), is evenly distributed across many other nodes.

Multiple tokens, however, can also have disadvantages:

1. Every token introduces up to `2 * (RF - 1)` additional neighbors on the token ring, which means that there are more combinations of node failures where we lose availability for a portion of the token ring. The more tokens you have, the higher the probability of an outage.
2. Cluster-wide maintenance operations are often slowed. For example, as the number of tokens per node is increased, the number of discrete repair operations the cluster must do also increases.
3. Performance of operations that span token ranges could be affected.

Note that in Cassandra `2.x`, the only token allocation algorithm available was picking random tokens, which meant that to keep balance the default number of tokens per node had to be quite high, at `256`. This had the effect of coupling many physical endpoints together, increasing the risk of unavailability. That is why in `3.x +` a new deterministic token allocator was added which intelligently picks tokens such that the ring is optimally balanced while requiring a much lower number of tokens per physical node.

Multi-master Replication: Versioned Data and Tunable Consistency

Cassandra replicates every partition of data to many nodes across the cluster to maintain high availability and durability. When a mutation occurs, the coordinator hashes the partition key to determine the token range the data belongs to and then replicates the mutation to the replicas of that data according to the `Replication Strategy`.

All replication strategies have the notion of a **replication factor** (`RF`), which indicates to Cassandra

how many copies of the partition should exist. For example with a `RF=3` keyspace, the data will be written to three distinct **replicas**. Replicas are always chosen such that they are distinct physical nodes which is achieved by skipping virtual nodes if needed. Replication strategies may also choose to skip nodes present in the same failure domain such as racks or datacenters so that Cassandra clusters can tolerate failures of whole racks and even datacenters of nodes.

Replication Strategy

Cassandra 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. All production deployments should use the `NetworkTopologyStrategy` while the `SimpleStrategy` replication strategy is useful only for testing clusters where you do not yet know the datacenter layout of the cluster.

`NetworkTopologyStrategy`

`NetworkTopologyStrategy` requires a specified replication factor for each datacenter in the cluster. Even if your cluster only uses a single datacenter, `NetworkTopologyStrategy` is recommended over `SimpleStrategy` to make 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, `NetworkTopologyStrategy` also attempts to choose replicas within a datacenter from different racks as specified by the `Snitch`. If the number of racks is greater than or equal to the replication factor for the datacenter, each replica is guaranteed to be chosen from a different rack. Otherwise, each rack will hold at least one replica, but some racks may hold more than one. Note that this rack-aware behavior has some potentially surprising implications. For example, if there are not an even number of nodes in each rack, the data load on the smallest rack may be much higher. Similarly, if a single node is bootstrapped into a brand new rack, it will be considered a replica for the entire ring. For this reason, many operators choose to configure all nodes in a single availability zone or similar failure domain as a single "rack".

`SimpleStrategy`

`SimpleStrategy` allows a single integer `replication_factor` to be defined. This determines the number of nodes that should contain a copy of each row. For example, if `replication_factor` is 3, then three different nodes should store a copy of each row.

`SimpleStrategy` treats all nodes identically, ignoring any configured datacenters or racks. To determine the replicas for a token range, Cassandra iterates through the tokens in the ring, starting with the token range of interest. For each token, it checks whether the owning node has been added to the set of replicas, and if it has not, it is added to the set. This process continues until `replication_factor` distinct nodes have been added to the set of replicas.

Transient Replication

Transient replication is an experimental feature in Cassandra {40_version} not present in the original Dynamo paper. This feature allows configuration of a subset of replicas to replicate only data that hasn't been incrementally repaired. This configuration decouples data redundancy from availability. For instance, if you have a keyspace replicated at RF=3, and alter it to RF=5 with two transient replicas, you go from tolerating one failed replica to tolerating two, without corresponding increase in storage usage. Now, three nodes will replicate all the data for a given token range, and the other two will only replicate data that hasn't been incrementally repaired.

To use transient replication, first enable the option in `cassandra.yaml`. Once enabled, both `SimpleStrategy` and `NetworkTopologyStrategy` can be configured to transiently replicate data. Configure it by specifying replication factor as `<total_replicas>/<transient_replicas` Both `SimpleStrategy` and `NetworkTopologyStrategy` support configuring transient replication.

Transiently replicated keyspaces only support tables created with `read_repair` set to `NONE`; monotonic reads are not currently supported. You also can't use `LWT`, logged batches, or counters in {40_version}. You will possibly never be able to use materialized views with transiently replicated keyspaces and probably never be able to use secondary indices with them.

Transient replication is an experimental feature that is not ready for production use. The expected audience is experienced users of Cassandra capable of fully validating a deployment of their particular application. That means you have the experience to check that operations like reads, writes, decommission, remove, rebuild, repair, and replace all work with your queries, data, configuration, operational practices, and availability requirements.

Anticipated additional features in `4.next` are support for monotonic reads with transient replication, as well as LWT, logged batches, and counters.

Data Versioning

Cassandra uses mutation timestamp versioning to guarantee eventual consistency of data. Specifically all mutations that enter the system do so with a timestamp provided either from a client clock or, absent a client-provided timestamp, from the coordinator node's clock. Updates resolve according to the conflict resolution rule of last write wins. Cassandra's correctness does depend on these clocks, so make sure a proper time synchronization process is running such as NTP.

Cassandra applies separate mutation timestamps to every column of every row within a CQL partition. Rows are guaranteed to be unique by primary key, and each column in a row resolves concurrent mutations according to last-write-wins conflict resolution. This means that updates to different primary keys within a partition can actually resolve without conflict! Furthermore the CQL collection types such as maps and sets use this same conflict-free mechanism, meaning that concurrent updates to maps and sets are guaranteed to resolve as well.

Replica Synchronization

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 <read-repair>` in the read path and `Hinted handoff <hints>` in the write path.

These techniques are only best-effort, however, and to guarantee eventual consistency Cassandra implements `anti-entropy repair <repair>` where replicas calculate hierarchical hash trees over their datasets called Merkle trees that can then be compared across replicas to identify mismatched data. Like the original Dynamo paper Cassandra supports full repairs where replicas hash their entire dataset, create Merkle trees, send them to each other and sync any ranges that don't match.

Unlike the original Dynamo paper, Cassandra also implements sub-range repair and incremental repair. Sub-range repair allows Cassandra to increase the resolution of the hash trees (potentially down to the single partition level) by creating a larger number of trees that span only a portion of the data range. Incremental repair allows Cassandra to only repair the partitions that have changed since the last repair.

Tunable Consistency

Cassandra supports a per-operation tradeoff between consistency and availability through **Consistency Levels**. Cassandra's consistency levels are a version of Dynamo's `R + W > N` consistency mechanism where operators could configure the number of nodes that must participate in reads (`R`) and writes (`W`) to be larger than the replication factor (`N`). In Cassandra, you instead choose from a menu of common consistency levels which allow the operator to pick `R` and `W` behavior without knowing the replication factor. 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 consistency levels are available:

`ONE`

: Only a single replica must respond.

`TWO`

: Two replicas must respond.

`THREE`

: Three replicas must respond.

`QUORUM`

: A majority (n/2 + 1) of the replicas must respond.

`ALL`

All of the replicas must respond.

`LOCAL_QUORUM`

A majority of the replicas in the local datacenter (whichever datacenter the coordinator is in) must respond.

`EACH_QUORUM`

A majority of the replicas in each datacenter must respond.

`LOCAL_ONE`

Only a single replica must respond. In a multi-datacenter cluster, this also guarantees that read requests are not sent to replicas in a remote datacenter.

`ANY`

A single replica may respond, or the coordinator may store a hint. If a hint is stored, the coordinator will later attempt to replay the hint and deliver the mutation to the replicas. This consistency level is only accepted for write operations.

Write operations **are always sent to all replicas**, regardless of consistency level. The consistency level simply controls how many responses the coordinator waits for before responding to the client.

For read operations, the coordinator generally only issues read commands to enough replicas to satisfy the consistency level. The one exception to this is when speculative retry may issue a redundant read request to an extra replica if the original replicas have not responded within a specified time window.

Picking Consistency Levels

It is common to pick read and write consistency levels such that the replica sets overlap, resulting in all acknowledged writes being visible to subsequent reads. This is typically expressed in the same terms Dynamo does, in that `W + R > RF`, where `W` is the write consistency level, `R` is the read consistency level, and `RF` is the replication factor. For example, if `RF = 3`, a `QUORUM` request will require responses from at least `2/3` replicas. If `QUORUM` is used for both writes and reads, at least one of the replicas is guaranteed to participate in *both* the write and the read request, which in turn guarantees that the quorums will overlap and the write will be visible to the read.

In a multi-datacenter environment, `LOCAL_QUORUM` can be used to provide a weaker but still useful guarantee: reads are guaranteed to see the latest write from within the same datacenter. This is often sufficient as clients homed to a single datacenter will read their own writes.

If this type of strong consistency isn't required, lower consistency levels like `LOCAL_ONE` or `ONE` may be used to improve throughput, latency, and availability. With replication spanning multiple datacenters, `LOCAL_ONE` is typically less available than `ONE` but is faster as a rule. Indeed `ONE` will succeed if a single replica is available in any datacenter.

Distributed Cluster Membership and Failure Detection

The replication protocols and dataset partitioning rely on knowing which nodes are alive and dead in the cluster so that write and read operations can be optimally routed. In Cassandra liveness information is shared in a distributed fashion through a failure detection mechanism based on a gossip protocol.

Gossip

Gossip is how Cassandra propagates basic cluster bootstrapping information such as endpoint membership and internode network protocol versions. In Cassandra's gossip system, nodes exchange state information not only about themselves but also about other nodes they know about. This information is versioned with a vector clock of `(generation, version)` tuples, where the generation is a monotonic timestamp and version is a logical clock that increments roughly every second. These logical clocks allow Cassandra gossip to ignore old versions of cluster state just by inspecting the logical clocks presented with gossip messages.

Every node in the Cassandra cluster runs the gossip task independently and periodically. Every second, every node in the cluster:

1. Updates the local node's heartbeat state (the version) and constructs the node's local view of the cluster gossip endpoint state.
2. Picks a random other node in the cluster to exchange gossip endpoint state with.
3. Probabilistically attempts to gossip with any unreachable nodes (if one exists)
4. 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, and the only difference between seed and non-seed nodes is that seed nodes are allowed to bootstrap into the ring without seeing any other seed nodes. Furthermore, once a cluster is bootstrapped, seed nodes become hotspots for gossip due to step 4 above.

As non-seed nodes must be able to contact at least one seed node in order to bootstrap into the cluster, it is common to include multiple seed nodes, often one for each rack or datacenter. Seed nodes are often chosen using existing off-the-shelf service discovery mechanisms.

NOTE	Nodes do not have to agree on the seed nodes, and indeed once a cluster is bootstrapped, newly launched nodes can be configured to use any existing nodes as seeds. The only advantage to picking the same nodes as seeds is that it increases their usefulness as gossip hotspots.

Currently, gossip also propagates token metadata and schema *version* information. This information

forms the control plane for scheduling data movements and schema pulls. For example, if a node sees a mismatch in schema version in gossip state, it will schedule a schema sync task with the other nodes. As token information propagates via gossip it is also the control plane for teaching nodes which endpoints own what data.

Ring Membership and Failure Detection

Gossip forms the basis of ring membership, but the **failure detector** ultimately makes decisions about if nodes are `UP` or `DOWN`. Every node in Cassandra runs a variant of the Phi Accrual Failure Detector, in which every node is constantly making an independent decision of if their peer nodes are available or not. This decision is primarily based on received heartbeat state. For example, if a node does not see an increasing heartbeat from a node for a certain amount of time, the failure detector "convicts" that node, at which point Cassandra will stop routing reads to it (writes will typically be written to hints). If/when the node starts heartbeating again, Cassandra will try to reach out and connect, and if it can open communication channels it will mark that node as available.

NOTE	`UP` and `DOWN` state are local node decisions and are not propagated with gossip. Heartbeat state is propagated with gossip, but nodes will not consider each other as `UP` until they can successfully message each other over an actual network channel.

Cassandra will never remove a node from gossip state without explicit instruction from an operator via a decommission operation or a new node bootstrapping with a `replace_address_first_boot` option. This choice is intentional to allow Cassandra nodes to temporarily fail without causing data to needlessly re-balance. This also helps to prevent simultaneous range movements, where multiple replicas of a token range are moving at the same time, which can violate monotonic consistency and can even cause data loss.

Incremental Scale-out on Commodity Hardware

Cassandra scales-out to meet the requirements of growth in data size and request rates. Scaling-out means adding additional nodes to the ring, and every additional node brings linear improvements in compute and storage. In contrast, scaling-up implies adding more capacity to the existing database nodes. Cassandra is also capable of scale-up, and in certain environments it may be preferable depending on the deployment. Cassandra gives operators the flexibility to choose either scale-out or scale-up.

One key aspect of Dynamo that Cassandra follows is to attempt to run on commodity hardware, and many engineering choices are made under this assumption. For example, Cassandra assumes nodes can fail at any time, auto-tunes to make the best use of CPU and memory resources available and makes heavy use of advanced compression and caching techniques to get the most storage out of limited memory and storage capabilities.

Simple Query Model

Cassandra, like Dynamo, chooses not to provide cross-partition transactions that are common in SQL Relational Database Management Systems (RDBMS). This both gives the programmer a simpler read and write API, and allows Cassandra to more easily scale horizontally since multi-partition transactions spanning multiple nodes are notoriously difficult to implement and typically very latent.

Instead, Cassandra chooses to offer fast, consistent, latency at any scale for single partition operations, allowing retrieval of entire partitions or only subsets of partitions based on primary key filters. Furthermore, Cassandra does support single partition compare and swap functionality via the lightweight transaction CQL API.

Simple Interface for Storing Records

Cassandra, in a slight departure from Dynamo, chooses a storage interface that is more sophisticated than "simple key-value" stores but significantly less complex than SQL relational data models. Cassandra presents a wide-column store interface, where partitions of data contain multiple rows, each of which contains a flexible set of individually typed columns. Every row is uniquely identified by the partition key and one or more clustering keys, and every row can have as many columns as needed.

This allows users to flexibly add new columns to existing datasets as new requirements surface. Schema changes involve only metadata changes and run fully concurrently with live workloads. Therefore, users can safely add columns to existing Cassandra databases while remaining confident that query performance will not degrade.

Storage Engine

Storage Engine

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending in with a write of data to disk:

- Logging data in the commit log
- Writing data to the memtable
- Flushing data from the memtable
- Storing data on disk in SSTables

Logging writes to commit logs

When a write occurs, **Cassandra** writes the data to a local append-only (commit log on disk. This action provides **configurable durability** by logging every write made to a **Cassandra** node. If an unexpected shutdown occurs, the commit log provides permanent durable writes of the data. On startup, any mutations in the commit log will be applied to (memtables. The commit log is shared among tables.

All mutations are write-optimized on storage in commit log segments, reducing the number of seeks needed to write to disk. Commit log segments are limited by the `commitlog_segment_size` option. Once the defined size is reached, a new commit log segment is created. Commit log segments can be archived, deleted, or recycled once all the data is flushed to (SSTables. Commit log segments are truncated when **Cassandra** has written data older than a certain point to the SSTables. Running `nodetool drain` before stopping **Cassandra** will write everything in the memtables to SSTables and remove the need to sync with the commit logs on startup.

- `commitlog_segment_size`: The default size is 32MiB, which is almost always fine, but if you are archiving commitlog segments (see commitlog_archiving.properties), then you probably want a finer granularity of archiving; 8 or 16 MiB is reasonable. `commitlog_segment_size` also determines the default value of `max_mutation_size` in `cassandra.yaml`. By default, `max_mutation_size` is a half the size of `commitlog_segment_size`.

NOTE	If `max_mutation_size` is set explicitly then `commitlog_segment_size` must be set to at least twice the size of `max_mutation_size`.

- `commitlog_sync`: may be either *periodic* or *batch*.
  - `batch`: In batch mode, **Cassandra** won't acknowledge writes until the commit log has been fsynced to disk.
  - `periodic`: In periodic mode, writes are immediately acknowledged, and the commit log is simply synced every "commitlog_sync_period" milliseconds.
    - `commitlog_sync_period`: Time to wait between "periodic" fsyncs *Default Value:* 10000ms

Default Value: batch

NOTE	In the event of an unexpected shutdown, **Cassandra** can lose up to the sync period or more if the sync is delayed. If using `batch` mode, it is recommended to store commit logs in a separate, dedicated device.

- `commitlog_directory`: This option is commented out by default. When running on magnetic HDD, this should be a separate spindle than the data directories. If not set, the default directory is `$CASSANDRA_HOME/data/commitlog`.

Default Value: `/var/lib/cassandra/commitlog`

- `commitlog_compression`: Compression to apply to the commitlog. If omitted, the commit log will be written uncompressed. LZ4, Snappy,Deflate and Zstd compressors are supported.

Default Value: (complex option):

```
#   - class_name: LZ4Compressor
#     parameters:
```

- `commitlog_total_space`: Total space to use for commit logs on disk. This option is commented out by default. If space gets above this value, **Cassandra** will flush every dirty table in the oldest segment and remove it. So a small total commit log space will tend to cause more flush activity on less-active tables. The default value is the smallest between 8192 and 1/4 of the total space of the commitlog volume.

Default Value: 8192MiB

Memtables

When a write occurs, **Cassandra** also writes the data to a memtable. Memtables are in-memory structures where **Cassandra** buffers writes. In general, there is one active memtable per table. The memtable is a write-back cache of data partitions that **Cassandra** looks up by key. Memtables may be stored entirely on-heap or partially off-heap, depending on `memtable_allocation_type`.

The memtable stores writes in sorted order until reaching a configurable limit. When the limit is reached, memtables are flushed onto disk and become immutable **SSTables**. Flushing can be triggered in several ways:

- The memory usage of the memtables exceeds the configured threshold (see `memtable_cleanup_threshold`)
- The `commit log` approaches its maximum size, and forces memtable flushes in order to allow commit log segments to be freed.

When a triggering event occurs, the memtable is put in a queue that is flushed to disk. Flushing writes the data to disk, in the memtable-sorted order. A partition index is also created on the disk that maps the tokens to a location on disk.

The queue can be configured with either the `memtable_heap_space` or `memtable_offheap_space` setting in the `cassandra.yaml` file. If the data to be flushed exceeds the `memtable_cleanup_threshold`, **Cassandra** blocks writes until the next flush succeeds. You can manually flush a table using `nodetool flush` or `nodetool drain` (flushes memtables without listening for connections to other nodes). To reduce the commit log replay time, the recommended best practice is to flush the memtable before you restart the nodes. If a node stops working, replaying the commit log restores writes to the memtable that were there before it stopped.

Data in the commit log is purged after its corresponding data in the memtable is flushed to an SSTable on disk.

SSTables

SSTables are the immutable data files that **Cassandra** uses for persisting data on disk. SSTables are maintained per table. SSTables are immutable, and never written to again after the memtable is flushed. Thus, a partition is typically stored across multiple SSTable files, as data is added or modified.

Each SSTable is comprised of multiple components stored in separate files:

`Data.db`

The actual data, i.e. the contents of rows.

`Partitions.db`

The partition index file maps unique prefixes of decorated partition keys to data file locations, or, in the case of wide partitions indexed in the row index file, to locations in the row index file.

`Rows.db`

The row index file only contains entries for partitions that contain more than one row and are bigger than one index block. For all such partitions, it stores a copy of the partition key, a partition header, and an index of row block separators, which map each row key into the first block where any content with equal or higher row key can be found.

`Index.db`

An index from partition keys to positions in the `Data.db` file. For wide partitions, this may also include an index to rows within a partition.

`Summary.db`

A sampling of (by default) every 128th entry in the `Index.db` file.

`Filter.db`

A Bloom Filter of the partition keys in the SSTable.

`CompressionInfo.db`

Metadata about the offsets and lengths of compression chunks in the `Data.db` file.

`Statistics.db`

Stores metadata about the SSTable, including information about timestamps, tombstones, clustering keys, compaction, repair, compression, TTLs, and more.

`Digest.crc32`

A CRC-32 digest of the `Data.db` file.

`TOC.txt`

A plain text list of the component files for the SSTable.

`SAI*.db`

Index information for Storage-Attached indexes. Only present if SAI is enabled for the table.

NOTE	Note that the `Index.db` file type is replaced by `Partitions.db` and `Rows.db`. This change is a consequence of the inclusion of Big Trie indexes in Cassandra (CEP-25).

Within the `Data.db` file, rows are organized by partition. These partitions are sorted in token order (i.e. by a hash of the partition key when the default partitioner, `Murmur3Partition`, is used). Within a partition, rows are stored in the order of their clustering keys.

SSTables can be optionally compressed using block-based compression.

As SSTables are flushed to disk from `memtables` or are streamed from other nodes, **Cassandra** triggers compactions which combine multiple SSTables into one. Once the new SSTable has been written, the old SSTables can be removed.

SSTable Versions

From (BigFormat#BigVersion.

The version numbers, to date are:

Version 0

- b (0.7.0): added version to sstable filenames
- c (0.7.0): bloom filter component computes hashes over raw key bytes instead of strings
- d (0.7.0): row size in data component becomes a long instead of int
- e (0.7.0): stores undecorated keys in data and index components
- f (0.7.0): switched bloom filter implementations in data component

- g (0.8): tracks flushed-at context in metadata component

Version 1

- h (1.0): tracks max client timestamp in metadata component
- hb (1.0.3): records compression ration in metadata component
- hc (1.0.4): records partitioner in metadata component
- hd (1.0.10): includes row tombstones in maxtimestamp
- he (1.1.3): includes ancestors generation in metadata component
- hf (1.1.6): marker that replay position corresponds to 1.1.5+ millis-based id (see CASSANDRA-4782)
- ia (1.2.0):
  - column indexes are promoted to the index file
  - records estimated histogram of deletion times in tombstones
  - bloom filter (keys and columns) upgraded to Murmur3
- ib (1.2.1): tracks min client timestamp in metadata component
- ic (1.2.5): omits per-row bloom filter of column names

Version 2

- ja (2.0.0):
  - super columns are serialized as composites (note that there is no real format change, this is mostly a marker to know if we should expect super columns or not. We do need a major version bump however, because we should not allow streaming of super columns into this new format)
  - tracks max local deletiontime in sstable metadata
  - records bloom_filter_fp_chance in metadata component
  - remove data size and column count from data file (CASSANDRA-4180)
  - tracks max/min column values (according to comparator)
- jb (2.0.1):
  - switch from crc32 to adler32 for compression checksums
  - checksum the compressed data
- ka (2.1.0):
  - new Statistics.db file format
  - index summaries can be downsampled and the sampling level is persisted
  - switch uncompressed checksums to adler32
  - tracks presence of legacy (local and remote) counter shards

- la (2.2.0): new file name format
- lb (2.2.7): commit log lower bound included

Version 3

- ma (3.0.0):
  - swap bf hash order
  - store rows natively
- mb (3.0.7, 3.7): commit log lower bound included
- mc (3.0.8, 3.9): commit log intervals included
- md (3.0.18, 3.11.4): corrected sstable min/max clustering
- me (3.0.25, 3.11.11): added hostId of the node from which the sstable originated

Version 4

- na (4.0-rc1): uncompressed chunks, pending repair session, isTransient, checksummed sstable metadata file, new Bloomfilter format
- nb (4.0.0): originating host id

Version 5

- oa (5.0): improved min/max, partition level deletion presence marker, key range (CASSANDRA-18134)
  - Long deletionTime to prevent TTL overflow
  - token space coverage

Trie-indexed Based SSTable Versions (BTI)

Cassandra 5.0 introduced new SSTable formats BTI for Trie-indexed SSTables. To use the BTI formats configure it `cassandra.yaml` like

```
sstable:
  selected_format: bti
```

Versions come from (BtiFormat#BtiVersion.

For implementation docs see (BtiFormat.md.

Version 5

- da (5.0): initial version of the BIT format

Example Code

The following example is useful for finding all sstables that do not match the "ib" SSTable version

```
find /var/lib/cassandra/data/ -type f | grep -v -- -ib- | grep -v "/snapshots"
```

Guarantees

Guarantees

Apache Cassandra is a highly scalable and reliable database. Cassandra is used in web-based applications that serve large number of clients and the quantity of data processed is web-scale (Petabyte) large. Cassandra makes some guarantees about its scalability, availability and reliability. To fully understand the inherent limitations of a storage system in an environment in which a certain level of network partition failure is to be expected and taken into account when designing the system, it is important to first briefly introduce the CAP theorem.

What is CAP?

According to the CAP theorem, it is not possible for a distributed data store to provide more than two of the following guarantees simultaneously.

- Consistency: Consistency implies that every read receives the most recent write or errors out
- Availability: Availability implies that every request receives a response. It is not guaranteed that the response contains the most recent write or data.
- Partition tolerance: Partition tolerance refers to the tolerance of a storage system to failure of a network partition. Even if some of the messages are dropped or delayed the system continues to operate.

The CAP theorem implies that when using a network partition, with the inherent risk of partition failure, one has to choose between consistency and availability and both cannot be guaranteed at the same time. CAP theorem is illustrated in Figure 1.

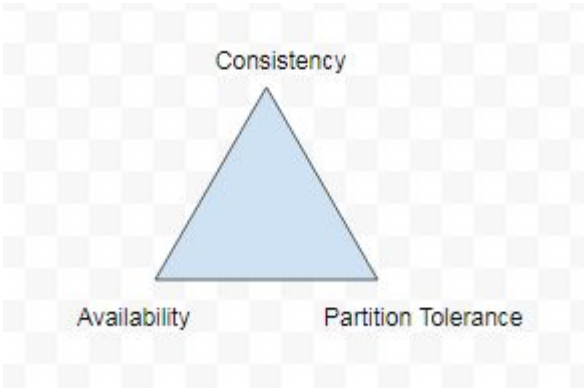


Figure 1. CAP Theorem

High availability is a priority in web-based applications and to this objective Cassandra chooses Availability and Partition Tolerance from the CAP guarantees, compromising on data Consistency to some extent.

Cassandra makes the following guarantees.

- High Scalability
- High Availability
- Durability
- Eventual Consistency of writes to a single table
- Lightweight transactions with linearizable consistency
- Batched writes across multiple tables are guaranteed to succeed completely or not at all
- Secondary indexes are guaranteed to be consistent with their local replicas' data

High Scalability

Cassandra is a highly scalable storage system in which nodes may be added/removed as needed. Using gossip-based protocol, a unified and consistent membership list is kept at each node.

High Availability

Cassandra guarantees high availability of data by implementing a fault-tolerant storage system. Failure of a node is detected using a gossip-based protocol.

Durability

Cassandra guarantees data durability by using replicas. Replicas are multiple copies of a data stored on different nodes in a cluster. In a multi-datacenter environment the replicas may be stored on different datacenters. If one replica is lost due to unrecoverable node/datacenter failure, the data is not completely lost, as replicas are still available.

Eventual Consistency

Meeting the requirements of performance, reliability, scalability and high availability in production, Cassandra is an eventually consistent storage system. Eventually consistency implies that all updates reach all replicas eventually. Divergent versions of the same data may exist temporarily, but they are eventually reconciled to a consistent state. Eventual consistency is a tradeoff to achieve high availability, and it involves some read and write latencies.

Lightweight transactions with linearizable consistency

Data must be read and written in a sequential order. The Paxos consensus protocol is used to implement lightweight transactions. The Paxos protocol implements lightweight transactions that are able to handle concurrent operations using linearizable consistency. Linearizable consistency is sequential consistency with real-time constraints, and it ensures transaction isolation with compare-and-set (CAS) transactions. With CAS replica data is compared and data that is found to be out of date is set to the most consistent value. Reads with linearizable consistency allow reading the current state of the data, which may possibly be uncommitted, without making a new addition or update.

Batched Writes

The guarantee for batched writes across multiple tables is that they will eventually succeed, or none will. Batch data is first written to batchlog system data, and when the batch data has been successfully stored in the cluster, the batchlog data is removed. The batch is replicated to another node to ensure that the full batch completes in the event if coordinator node fails.

Secondary Indexes

A secondary index is an index on a column, and it’s used to query a table that is normally not queryable. Secondary indexes, when built, are guaranteed to be consistent with their local replicas.

Improved Internode Messaging

Improved Internode Messaging

Apache Cassandra 4.0 has added several new improvements to internode messaging.

Optimized Internode Messaging Protocol

The internode messaging protocol has been optimized (CASSANDRA-14485). Previously the `IPAddressAndPort` of the sender was included with each message that was sent even though the `IPAddressAndPort` had already been sent once when the initial connection/session was established. In Cassandra 4.0 `IPAddressAndPort` has been removed from every separate message sent and only sent when connection/session is initiated.

Another improvement is that at several instances (listed) a fixed 4-byte integer value has been replaced with `vint` as a `vint` is almost always less than 1 byte:

- The `paramSize` (the number of parameters in the header)
- Each individual parameter value
- The `payloadSize`

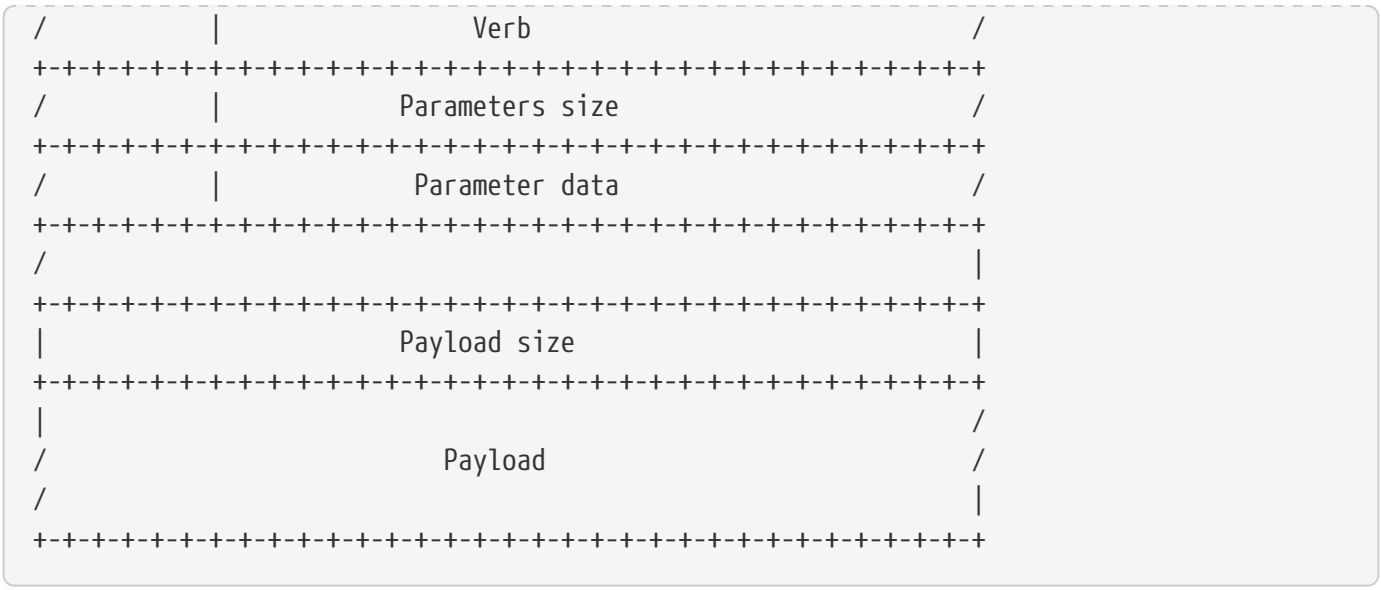
NIO Messaging

In Cassandra 4.0 peer-to-peer (internode) messaging has been switched to non-blocking I/O (NIO) via Netty (CASSANDRA-8457).

As serialization format, each message contains a header with several fixed fields, an optional key-value parameters section, and then the message payload itself. Note: the IP address in the header may be either IPv4 (4 bytes) or IPv6 (16 bytes).

> The diagram below shows the IPv4 address for brevity.

```
1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 6 6
0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       PROTOCOL MAGIC                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Message ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Timestamp                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Addr len |           IP Address (IPv4)                       /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



An individual parameter has a String key and a byte array value. The key is serialized with its length, encoded as two bytes, followed by the UTF-8 byte encoding of the string. The body is serialized with its length, encoded as four bytes, followed by the bytes of the value.

Resource limits on Queued Messages

System stability is improved by enforcing strict resource limits (CASSANDRA-15066) on the number of outbound messages that are queued, measured by the `serializedSize` of the message. There are three separate limits imposed simultaneously to ensure that progress is always made without any reasonable combination of failures impacting a node's stability.

1. Global, per-endpoint and per-connection limits are imposed on messages queued for delivery to other nodes and waiting to be processed on arrival from other nodes in the cluster. These limits are applied to the on-wire size of the message being sent or received.
2. The basic per-link limit is consumed in isolation before any endpoint or global limit is imposed. Each node-pair has three links: urgent, small and large. Any given node may have a maximum of `N*3 * (internode_application_send_queue_capacity in bytes + internode_application_receive_queue_capacity in bytes)` of messages data queued without any coordination between them although in practice, with token-aware routing, only RF*tokens nodes should need to communicate with significant bandwidth.
3. The per-endpoint limit is imposed on all messages exceeding the per-link limit, simultaneously with the global limit, on all links to or from a single node in the cluster. The global limit is imposed on all messages exceeding the per-link limit, simultaneously with the per-endpoint limit, on all links to or from any node in the cluster. The following configuration settings have been added to `cassandra.yaml` for resource limits on queued messages.

```
internode_application_send_queue_capacity: 4MiB
internode_application_send_queue_reserve_endpoint_capacity: 128MiB
```

```
internode_application_send_queue_reserve_global_capacity: 512MiB
internode_application_receive_queue_capacity: 4MiB
internode_application_receive_queue_reserve_endpoint_capacity: 128MiB
internode_application_receive_queue_reserve_global_capacity: 512MiB
```

Virtual Tables for Messaging Metrics

Metrics is improved by keeping metrics using virtual tables for inter-node inbound and outbound messaging (CASSANDRA-15066). For inbound messaging a virtual table (`internode_inbound`) has been added to keep metrics for:

- Bytes and count of messages that could not be serialized or flushed due to an error
- Bytes and count of messages scheduled
- Bytes and count of messages successfully processed
- Bytes and count of messages successfully received
- Nanos and count of messages throttled
- Bytes and count of messages expired
- Corrupt frames recovered and unrecovered

A separate virtual table (`internode_outbound`) has been added for outbound inter-node messaging. The outbound virtual table keeps metrics for:

- Bytes and count of messages pending
- Bytes and count of messages sent
- Bytes and count of messages expired
- Bytes and count of messages that could not be sent due to an error
- Bytes and count of messages overloaded
- Active Connection Count
- Connection Attempts
- Successful Connection Attempts

Hint Messaging

A specialized version of hint message that takes an already encoded in a `ByteBuffer` hint and sends it verbatim has been added. It is an optimization for when dispatching a hint file of the current messaging version to a node of the same messaging version, which is the most common case. It saves on extra `ByteBuffer` allocations one redundant hint deserialization-serialization cycle.

Internode Application Timeout

A configuration setting has been added to `cassandra.yaml` for the maximum continuous period a connection may be unwritable in application space.

```
# internode_application_timeout_in_ms = 30000
```

Some other new features include logging of message size to trace message for tracing a query.

Paxos prepare and propose stage for local requests optimized

In pre-4.0 Paxos prepare and propose messages always go through entire `MessagingService` stack in Cassandra even if request is to be served locally, we can enhance and make local requests severed w/o involving `MessagingService`. Similar things are done elsewhere in Cassandra which skips `MessagingService` stage for local requests.

This is what it looks like in pre 4.0 if we have tracing on and run a light-weight transaction:

```
Sending PAXOS_PREPARE message to /A.B.C.D [MessagingService-Outgoing-/A.B.C.D] | 2017-09-
11
21:55:18.971000 | A.B.C.D | 15045
··· REQUEST_RESPONSE message received from /A.B.C.D [MessagingService-Incoming-/A.B.C.D] |
2017-09-11 21:55:18.976000 | A.B.C.D | 20270
··· Processing response from /A.B.C.D [SharedPool-Worker-4] | 2017-09-11 21:55:18.976000 |
A.B.C.D | 20372
```

Same thing applies for Propose stage as well.

In version 4.0 Paxos prepare and propose stage for local requests are optimized (CASSANDRA-13862).

Quality Assurance

Several other quality assurance improvements have been made in version 4.0 (CASSANDRA-15066).

Framing

Version 4.0 introduces framing to all internode messages, i.e. the grouping of messages into a single logical payload with headers and trailers; these frames are guaranteed to either contain at most one message, that is split into its own unique sequence of frames (for large messages), or that a frame contains only complete messages.

Corruption prevention

Previously, intra-datacenter internode messages would be unprotected from corruption by default, as only LZ4 provided any integrity checks. All messages to post 4.0 nodes are written to explicit frames, which may be:

- LZ4 encoded
- CRC protected

The Unprotected option is still available.

Resilience

For resilience, all frames are written with a separate CRC protected header, of 8 and 6 bytes respectively. If corruption occurs in this header, the connection must be reset, as before. If corruption occurs anywhere outside of the header, the corrupt frame will be skipped, leaving the connection intact and avoiding the loss of any messages unnecessarily.

Previously, any issue at any point in the stream would result in the connection being reset, with the loss of any in-flight messages.

Efficiency

The overall memory usage, and number of byte shuffles, on both inbound and outbound messages is reduced.

Outbound the Netty LZ4 encoder maintains a chunk size buffer (64KiB), that is filled before any compressed frame can be produced. Our frame encoders avoid this redundant copy, as well as freeing 192KiB per endpoint.

Inbound, frame decoders guarantee only to copy the number of bytes necessary to parse a frame, and to never store more bytes than necessary. This improvement applies twice to LZ4 connections, improving both the message decode and the LZ4 frame decode.

Inbound Path

Version 4.0 introduces several improvements to the inbound path.

An appropriate message handler is used based on whether large or small messages are expected on a particular connection as set in a flag. `NonblockingBufferHandler`, running on event loop, is used for small messages, and `BlockingBufferHandler`, running off event loop, for large messages. The single implementation of `InboundMessageHandler` handles messages of any size effectively by deriving size of the incoming message from the byte stream. In addition to deriving size of the message from the stream, incoming message expiration time is proactively read, before attempting to deserialize the

entire message. If it's expired at the time when a message is encountered the message is just skipped in the byte stream altogether. And if a message fails to be deserialized while still on the receiving side - say, because of table id or column being unknown - bytes are skipped, without dropping the entire connection and losing all the buffered messages. An immediately reply back is sent to the coordinator node with the failure reason, rather than waiting for the coordinator callback to expire. This logic is extended to a corrupted frame; a corrupted frame is safely skipped over without dropping the connection.

Inbound path imposes strict limits on memory utilization. Specifically, the memory occupied by all parsed, but unprocessed messages is bound - on per-connection, per-endpoint, and global basis. Once a connection exceeds its local unprocessed capacity and cannot borrow any permits from per-endpoint and global reserve, it simply stops processing further messages, providing natural backpressure - until sufficient capacity is regained.

Outbound Connections

Opening a connection

A consistent approach is adopted for all kinds of failure to connect, including: refused by endpoint, incompatible versions, or unexpected exceptions;

- Retry forever, until either success or no messages waiting to deliver.
- Wait incrementally longer periods before reconnecting, up to a maximum of 1s.
- While failing to connect, no reserve queue limits are acquired.

Closing a connection

- Correctly drains outbound messages that are waiting to be delivered (unless disconnected and fail to reconnect).
- Messages written to a closing connection are either delivered or rejected, with a new connection being opened if the old is irrevocably closed.
- Unused connections are pruned eventually.

Reconnecting

We sometimes need to reconnect a perfectly valid connection, e.g. if the preferred IP address changes. We ensure that the underlying connection has no in-progress operations before closing it and reconnecting.

Message Failure

Propagates to callbacks instantly, better preventing overload by reclaiming committed memory.

Expiry

- No longer experiences head-of-line blocking (e.g. undroppable message preventing all droppable messages from being expired).
- While overloaded, expiry is attempted eagerly on enqueuing threads.
- While disconnected we schedule regular pruning, to handle the case where messages are no longer being sent, but we have a large backlog to expire.

Overload

- Tracked by bytes queued, as opposed to number of messages.

Serialization Errors

- Do not result in the connection being invalidated; the message is simply completed with failure, and then erased from the frame.
- Includes detected mismatch between calculated serialization size to actual.

Failures to flush to network, perhaps because the connection has been reset are not currently notified to callback handlers, as the necessary information has been discarded, though it would be possible to do so in future if we decide it is worth our while.

QoS

"Gossip" connection has been replaced with a general purpose "Urgent" connection, for any small messages impacting system stability.

Metrics

We track, and expose via Virtual Table and JMX, the number of messages and bytes that: we could not serialize or flush due to an error, we dropped due to overload or timeout, are pending, and have successfully sent.

Added a Message size limit

Cassandra pre-4.0 doesn’t protect the server from allocating huge buffers for the inter-node Message objects. Adding a message size limit would be good to deal with issues such as a malfunctioning cluster participant. Version 4.0 introduced max message size config param, akin to max mutation size - set to endpoint reserve capacity by default.

Recover from unknown table when deserializing internode messages

As discussed in (CASSANDRA-9289) it would be nice to gracefully recover from seeing an unknown table in a message from another node. Pre-4.0, we close the connection and reconnect, which can cause other concurrent queries to fail. Version 4.0 fixes the issue by wrapping message in-stream with `TrackedDataInputPlus`, catching `UnknownCFException`, and skipping the remaining bytes in this message. TCP won't be closed and it will remain connected for other messages.

Improved Streaming

Improved Streaming

Apache Cassandra 4.0 has made several improvements to streaming. Streaming is the process used by nodes of a cluster to exchange data in the form of SSTables. Streaming of SSTables is performed for several operations, such as:

- SSTable Repair
- Host Replacement
- Range movements
- Bootstrapping
- Rebuild
- Cluster expansion

Streaming based on Netty

Streaming in Cassandra 4.0 is based on Non-blocking Input/Output (NIO) with Netty (CASSANDRA-12229). It replaces the single-threaded (or sequential), synchronous, blocking model of streaming messages and transfer of files. Netty supports non-blocking, asynchronous, multi-threaded streaming with which multiple connections are opened simultaneously. Non-blocking implies that threads are not blocked as they don't wait for a response for a sent request. A response could be returned in a different thread. With asynchronous, connections and threads are decoupled and do not have a 1:1 relation. Several more connections than threads may be opened.

Zero Copy Streaming

Pre-4.0, during streaming Cassandra reifies the SSTables into objects. This creates unnecessary garbage and slows down the whole streaming process as some SSTables can be transferred as a whole file rather than individual partitions. Cassandra 4.0 has added support for streaming entire SSTables when possible (CASSANDRA-14556) for faster Streaming using ZeroCopy APIs. If enabled, Cassandra will use ZeroCopy for eligible SSTables significantly speeding up transfers and increasing throughput. A zero-copy path avoids bringing data into user-space on both sending and receiving side. Any streaming related operations will notice corresponding improvement. Zero copy streaming is hardware bound; only limited by the hardware limitations (Network and Disk IO).

High Availability

In benchmark tests Zero Copy Streaming is 5x faster than partitions based streaming. Faster streaming provides the benefit of improved availability. A cluster's recovery mainly depends on the streaming speed, Cassandra clusters with failed nodes will be able to recover much more quickly (5x faster). If a

node fails, SSTables need to be streamed to a replacement node. During the replacement operation, the new Cassandra node streams SSTables from the neighboring nodes that hold copies of the data belonging to this new node's token range. Depending on the amount of data stored, this process can require substantial network bandwidth, taking some time to complete. The longer these range movement operations take, the more the cluster availability is lost. Failure of multiple nodes would reduce high availability greatly. The faster the new node completes streaming its data, the faster it can serve traffic, increasing the availability of the cluster.

Enabling Zero Copy Streaming

Zero copy streaming is enabled by setting the following setting in `cassandra.yaml`.

```
stream_entire_sstables: true
```

By default zero copy streaming is enabled.

SSTables Eligible for Zero Copy Streaming

Zero copy streaming is used if all partitions within the SSTable need to be transmitted. This is common when using `LeveledCompactionStrategy` or when partitioning SSTables by token range has been enabled. All partition keys in the SSTables are iterated over to determine the eligibility for Zero Copy streaming.

Benefits of Zero Copy Streaming

When enabled, it permits Cassandra to zero-copy stream entire eligible SSTables between nodes, including every component. This speeds up the network transfer significantly subject to throttling specified by `stream_throughput_outbound`.

Enabling this will reduce the GC pressure on sending and receiving node. While this feature tries to keep the disks balanced, it cannot guarantee it. This feature will be automatically disabled if internode encryption is enabled. Currently this can be used with Leveled Compaction.

Configuring for Zero Copy Streaming

Throttling would reduce the streaming speed. The `stream_throughput_outbound` throttles all outbound streaming file transfers on a node to the given total throughput in Mbps. When unset, the default is 200 Mbps or 24 MiB/s.

```
stream_throughput_outbound: 24MiB/s
```

To run any Zero Copy streaming benchmark the `stream_throughput_outbound` must be set to a really high value otherwise, throttling will be significant and the benchmark results will not be meaningful.

The `inter_dc_stream_throughput_outbound` throttles all streaming file transfer between the datacenters, this setting allows users to throttle inter dc stream throughput in addition to throttling all network stream traffic as configured with `stream_throughput_outbound`. When unset, the default is 200 Mbps or 25 MB/s.

```
inter_dc_stream_throughput_outbound: 24MiB/s
```

SSTable Components Streamed with Zero Copy Streaming

Zero Copy Streaming streams entire SSTables. SSTables are made up of multiple components in separate files. SSTable components streamed are listed in Table 1.

Table 1. SSTable Components

SSTable Component	Description
Data.db	The base data for an SSTable: the remaining components can be regenerated based on the data component.
Index.db	Index of the row keys with pointers to their positions in the data file.
Filter.db	Serialized bloom filter for the row keys in the SSTable.
CompressionInfo.db	File to hold information about uncompressed data length, chunk offsets etc.
Statistics.db	Statistical metadata about the content of the SSTable.
Digest.crc32	Holds CRC32 checksum of the data file size_bytes.
CRC.db	Holds the CRC32 for chunks in an uncompressed file.
Summary.db	Holds SSTable Index Summary (sampling of Index component)
TOC.txt	Table of contents, stores the list of all components for the SSTable.

Custom component, used by e.g. custom compaction strategy may also be included.

Repair Streaming Preview

Repair with `nodetool repair` involves streaming of repaired SSTables and a repair preview has been added to provide an estimate of the amount of repair streaming that would need to be performed. Repair preview (CASSANDRA-13257) is invoke with `nodetool repair --preview` using option:

```
-prv, --preview
```

It determines ranges and amount of data to be streamed, but doesn't actually perform repair.

Parallelizing of Streaming of Keyspaces

The streaming of the different keyspaces for bootstrap and rebuild has been parallelized in Cassandra 4.0 (CASSANDRA-4663).

Unique nodes for Streaming in Multi-DC deployment

Range Streamer picks unique nodes to stream data from when number of replicas in each DC is three or more (CASSANDRA-4650). What the optimization does is to even out the streaming load across the cluster. Without the optimization, some node can be picked up to stream more data than others. This patch allows to select dedicated node to stream only one range.

This will increase the performance of bootstrapping a node and will also put less pressure on nodes serving the data. This does not affect if N < 3 in each DC as then it streams data from only 2 nodes.

Stream Operation Types

It is important to know the type or purpose of a certain stream. Version 4.0 (CASSANDRA-13064) adds an `enum` to distinguish between the different types of streams. Stream types are available both in a stream request and a stream task. The different stream types are:

- Restore replica count
- Unbootstrap
- Relocation
- Bootstrap
- Rebuild
- Bulk Load
- Repair

Disallow Decommission when number of Replicas will drop below configured RF

CASSANDRA-12510 guards against decommission that will drop # of replicas below configured replication factor (RF), and adds the `--force` option that allows decommission to continue if intentional; force decommission of this node even when it reduces the number of replicas to below configured RF.