



Apache Cassandra™ 1.1 Documentation

June 26, 2014

Apache, Apache Cassandra, Apache Hadoop, Hadoop and the eye logo
are trademarks of the Apache Software Foundation

Contents

What's New in Apache Cassandra 1.1	1
Introducing Cassandra Architecture	3
About Internode Communications (Gossip)	3
About Data Partitioning in Cassandra	4
About Replication in Cassandra	8
About Client Requests in Cassandra	14
Hadoop Integration	16
Planning a Cassandra Cluster Deployment	17
Anti-patterns in Cassandra	21
Installing a Cassandra Cluster	25
Installing Cassandra RHEL or CentOS Packages	25
Installing Cassandra Debian Packages	26
Installing the Cassandra Binary Tarball Distribution	27
Recommended Settings for Production Installations	28
Installing the JRE and JNA	30
Installing a Cassandra Cluster on Amazon EC2	33
Expanding a Cassandra AMI cluster	33
Upgrading Cassandra	35
Initializing a Cassandra Cluster	39
Initializing a Multiple Node Cluster in a Single Data Center	39
Initializing Multiple Data Center Clusters on Cassandra	42
Generating Tokens	45
Managing a Cassandra Cluster	49
Running Routine Node Repair	49
Adding Capacity to an Existing Cluster	49
Changing the Replication Factor	51
Replacing a Dead Node	51
Backing Up and Restoring Data	53
Taking a Snapshot	53
Deleting Snapshot Files	53
Enabling Incremental Backups	54
Restoring from a Snapshot	54
Understanding the Cassandra Data Model	55
Data Modeling	55
About Keyspaces	61
About Indexes in Cassandra	62
Planning Your Data Model	64

Managing Data	66
About Writes in Cassandra	66
About Reads in Cassandra	68
About Data Consistency in Cassandra	69
Cassandra Client APIs	72
Getting Started Using the Cassandra CLI	73
Querying Cassandra	77
Configuration	84
Node and Cluster Configuration (cassandra.yaml)	84
Keyspace and Column Family Storage Configuration	94
Configuring the heap dump directory	99
Java and System Environment Settings Configuration	100
Authentication and Authorization Configuration	101
Logging Configuration	102
Commit Log Archive Configuration	104
Performance Monitoring and Tuning	106
Monitoring a Cassandra Cluster	106
Tuning Cassandra	111
References	118
CQL 3 Language Reference	118
nodetool	153
cassandra	157
Cassandra Bulk Loader	160
cassandra-stress	161
sstable2json / json2sstable	164
CQL Commands Quick Reference	166
Install Locations	166
Configuring Firewall Port Access	167
Starting and Stopping a Cassandra Cluster	168
Troubleshooting Guide	169
Reads are getting slower while writes are still fast	169
Nodes seem to freeze after some period of time	169
Nodes are dying with OOM errors	169
Nodetool or JMX connections failing on remote nodes	170
View of ring differs between some nodes	170
Java reports an error saying there are too many open files	170
Insufficient user resource limits errors	170
Cannot initialize class org.xerial.snappy.Snappy	171
DataStax Community Release Notes	173

Issues	173
Fixes and New Features in Cassandra 1.1	173

What's New in Apache Cassandra 1.1

In Cassandra 1.1, key improvements have been made in the areas of CQL, performance, and management ease of use:

Cassandra Query Language (CQL) Enhancements

One of the main objectives of Cassandra 1.1 was to bring CQL up to parity with the legacy API and command line interface (CLI) that has shipped with Cassandra for several years. This release achieves that goal. CQL is now the primary interface into the DBMS. The CQL specification has now been promoted to CQL 3, although CQL 2 remains the default in 1.1 because CQL3 is not backwards compatible. A number of the new CQL enhancements have been rolled out in prior Cassandra 1.0.x point releases. These are covered in the [CQL Reference](#).

Composite Primary Key Columns

The most significant enhancement of CQL is support for [composite primary key columns](#) and wide rows. Composite keys distribute column family data among the nodes. New querying capabilities are a beneficial side effect of wide-row support. You use an [ORDER BY](#) clause to sort the result set. A new compact storage directive provides backward-compatibility for applications created with CQL 2. If this directive is used, then instead of each non-primary key column being stored in a way where each column corresponds to one column on disk, an entire row is stored in a single column on disk. The drawback is that updates to that column's data are not allowed. The default is non-compact storage.

CQL Shell Utility

The CQL shell utility (cqlsh) contains a number of new features. First is the [SOURCE](#) command, which reads CQL commands from an external file and runs them. Next, the [CAPTURE](#) command writes the output of a session to a specified file. Finally, the [DESCRIBE COLUMNFAMILIES](#) command shows all the column families that exist in a certain keyspace.

Global Row and Key Caches

Memory caches for column families are now managed globally instead of at the individual column family level, simplifying configuration and tuning. Cassandra automatically distributes memory for various column families based on the overall workload and specific column family usage. Two [new configuration parameters](#), `key_cache_size_in_mb` and `row_cache_size_in_mb` replace the per column family cache sizing options. Administrators can choose to include or exclude column families from being cached via the caching parameter that is used when creating or modifying column families.

Off-Heap Cache for Windows

The serializing cache provider (the off heap cache) has been rewritten to no longer require the external JNA library. This is particularly good news for Microsoft Windows users, as Cassandra never supported JNA on that platform. But with the JNA requirement now being eliminated, the off heap cache is available on the Windows platform, which provides the potential for additional performance gains.

Row-Level Isolation

Full [row-level isolation](#) is now in place so that writes to a row are isolated to the client performing the write and are not visible to any other user until they are complete. From a transactional ACID (atomic, consistent, isolated, durable) standpoint, this enhancement now gives Cassandra transactional ACID support. Consistency in the ACID sense typically involves referential integrity with foreign keys among related tables, which Cassandra does not have. Cassandra offers tunable consistency not in the ACID sense, but in the CAP theorem sense where data is made consistent across all the nodes in a distributed database cluster. A user can pick and choose on a per operation basis how many nodes must receive a DML command or respond to a SELECT query.

Concurrent Schema Change Support

Cassandra has supported online schema changes since 0.7, however the potential existed for nodes in a cluster to have a disagreement over the sequence of changes made to a particular column family. The end result was the nodes in question had to rebuild their schema.

In version 1.1, large numbers of schema changes can simultaneously take place in a cluster without the fear of having a schema disagreement occur.

A side benefit of the *support for schema changes* is new nodes are added much faster. The new node is sent the full schema instead of all the changes that have occurred over the life of the cluster. Subsequent changes correctly modify that schema.

Fine-grained Data Storage Control

Cassandra 1.1 provides fine-grained control of *column family storage* on disk. Until now, you could only use a separate disk per keyspace, not per column family. Cassandra 1.1 stores data files by using separate column family directories within each keyspace directory. In 1.1, data files are stored in this format:

```
/var/lib/cassandra/data/ks1/cf1/ks1-cf1-hc-1-Data.db
```

Now, you can mount an SSD on a particular directory (in this example cf1) to boost the performance for a particular column family. The new file name format includes the keyspace name to distinguish which keyspace and column family the file contains when streaming or bulk loading.

Write Survey Mode

Using the *write survey mode*, you can add a node to a database cluster so that it accepts all the write traffic as if it were part of the normal database cluster, without the node itself actually being part of the cluster where supporting user activity is concerned. It never officially joins the ring. In write survey mode, you can test out new compaction and compression strategies on that node and benchmark the write performance differences, without affecting the production cluster.

To see how read performance is affected by the various modifications, you apply changes to the dummy node, stop the node, bring it up as a standalone machine, and then benchmark read operations on the node.

Abortable Compactions

In Cassandra 1.1, you can stop a compaction, validation, and several other operations from continuing to run. For example, if a compaction has a negative impact on the performance of a node during a critical time of the day, for example, you can terminate the operation using the *nodetool stop [operation type]* command.

Hadoop Integration

The following low-level features have been added to Cassandra's support for Hadoop:

- *Secondary index support* for the column family input format. Hadoop jobs can now make use of Cassandra secondary indexes.
- *Wide row support*. Previously, wide rows that had, for example, millions of columns could not be accessed, but now they can be read and paged through in Hadoop.
- The *bulk output format* provides a more efficient way to load data into Cassandra from a Hadoop job.

Introducing Cassandra Architecture

A Cassandra instance is a collection of independent nodes that are configured together into a *cluster*. In a Cassandra cluster, all nodes are peers, meaning there is no master node or centralized management process. A node joins a Cassandra cluster based on its configuration. This section explains key aspects of the Cassandra cluster architecture.

About Internode Communications (Gossip)

Cassandra uses a protocol called *gossip* to discover location and state information about the other nodes participating in a Cassandra cluster. Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about.

In Cassandra, the gossip process runs every second and exchanges state messages with up to three other nodes in the cluster. The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster. A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

About Cluster Membership and Seed Nodes

When a node first starts up, it looks at its configuration file to determine the name of the Cassandra cluster it belongs to and which node(s), called *seeds*, to contact to obtain information about the other nodes in the cluster. These cluster contact points are configured in the *cassandra.yaml* configuration file for a node.

To prevent partitions in gossip communications, all nodes in a cluster should have the *same* list of seed nodes listed in their configuration file. This is most critical the *first* time a node starts up. By default, a node will remember other nodes it has gossiped with between subsequent restarts.

Note

The seed node designation has no purpose other than bootstrapping the gossip process for new nodes joining the cluster. Seed nodes are *not* a single point of failure, nor do they have any other special purpose in cluster operations beyond the bootstrapping of nodes.

To know what range of data it is responsible for, a node must also know its own *token* and those of the other nodes in the cluster. When initializing a new cluster, you should generate tokens for the entire cluster and assign an initial token to each node before starting up. Each node will then gossip its token to the others. See [About Data Partitioning in Cassandra](#) for more information about partitioners and tokens.

Configuring Gossip Settings

The gossip settings control a nodes participation in a cluster and how the node is known to the cluster.

Property	Description
<i>cluster_name</i>	Name of the cluster that this node is joining. Should be the same for every node in the cluster.
<i>listen_address</i>	The IP address or hostname that other Cassandra nodes will use to connect to this node. Should be changed from <code>localhost</code> to the public address for the host.
<i>seeds</i>	A list of comma-delimited hosts (IP addresses) that gossip uses to learn the topology of the ring. Every node should have the same list of seeds. In multiple data-center clusters, the seed list should include a node from each data center.

<i>storage_port</i>	The intra-node communication port (default is 7000). Should be the same for every node in the cluster.
<i>initial_token</i>	The initial token is used to determine the range of data this node is responsible for.

Purging Gossip State on a Node

Gossip information is also persisted locally by each node to use immediately next restart without having to wait for gossip. To clear gossip history on node restart (for example, if node IP addresses have changed), add the following line to the `cassandra-env.sh` file. This file is located in `/usr/share/cassandra` or `<install_location>/conf`.

```
-Dcassandra.load_ring_state=false
```

About Failure Detection and Recovery

Failure detection is a method for locally determining, from gossip state, if another node in the system is up or down. Failure detection information is also used by Cassandra to avoid routing client requests to unreachable nodes whenever possible. (Cassandra can also avoid routing requests to nodes that are alive, but performing poorly, through the *dynamic snitch*.)

The gossip process tracks heartbeats from other nodes both directly (nodes gossiping directly to it) and indirectly (nodes heard about secondhand, thirdhand, and so on). Rather than have a fixed threshold for marking nodes without a heartbeat as down, Cassandra uses an accrual detection mechanism to calculate a per-node threshold that takes into account network conditions, workload, or other conditions that might affect perceived heartbeat rate. During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. The value of `phi` is based on the distribution of inter-arrival time values across all nodes in the cluster. In Cassandra, configuring the *phi_convict_threshold* property adjusts the sensitivity of the failure detector. The default value is fine for most situations, but DataStax recommends increasing it to 12 for Amazon EC2 due to the network congestion frequently experienced on that platform.

Node failures can result from various causes such as hardware failures, network outages, and so on. Node outages are often transient but can last for extended intervals. A node outage rarely signifies a permanent departure from the cluster, and therefore does not automatically result in permanent removal of the node from the ring. Other nodes will still try to periodically initiate gossip contact with failed nodes to see if they are back up. To permanently change a node's membership in a cluster, administrators must explicitly add or remove nodes from a Cassandra cluster using the *nodetool* utility.

When a node comes back online after an outage, it may have missed writes for the replica data it maintains. Once the failure detector marks a node as down, missed writes are stored by other replicas for a period of time providing *hinted handoff* is enabled. If a node is down for longer than *max_hint_window_in_ms* (1 hour by default), hints are no longer saved. Because nodes that die may have stored undelivered hints, you should run a repair after recovering a node that has been down for an extended period. Moreover, you should routinely run *nodetool repair* on all nodes to ensure they have consistent data.

For more explanation about recovery, see *Modern hinted handoff*.

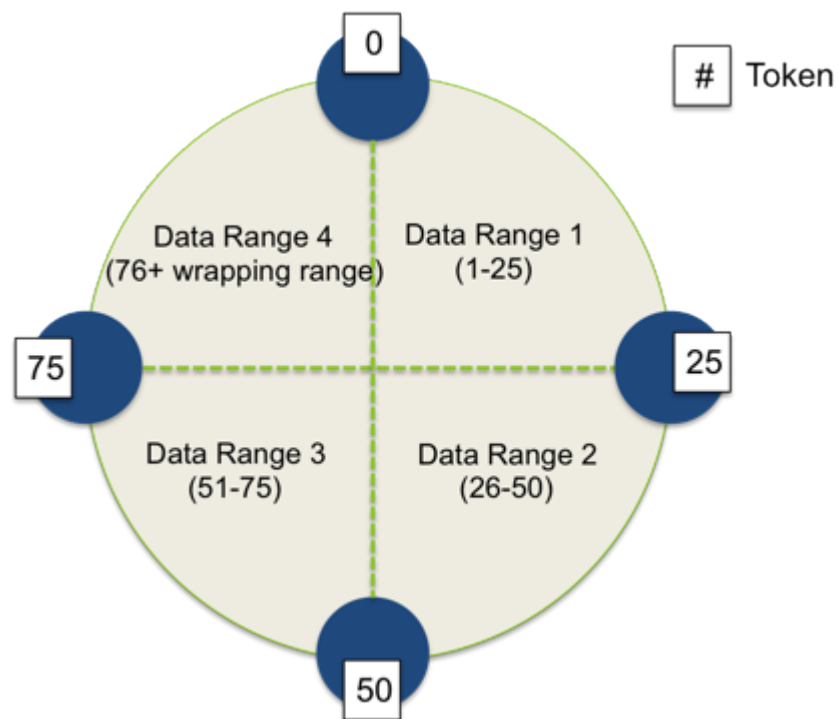
About Data Partitioning in Cassandra

A partitioner determines how data is distributed across the nodes in the cluster (including replicas). Basically, a partitioner is a hash function for computing the token (it's hash) of a row key. Each row of data is uniquely identified by a row key and distributed across the cluster by the value of the token.

Data Distribution in the Ring

In Cassandra, the total amount of data managed by the cluster is represented as a ring. The ring is divided into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the data. Before a node can join the ring, it must be assigned a token. The token value determines the node's position in the ring and its range of data. Column family data is partitioned across the nodes based on the row key. To determine the node where the first replica of a row will live, the ring is walked clockwise until it locates the node with a token value greater than that of the row key. Each node is responsible for the region of the ring between itself (inclusive) and its predecessor (exclusive). With the nodes sorted in token order, the last node is considered the predecessor of the first node; hence the ring representation.

For example, consider a simple four node cluster where all of the row keys managed by the cluster were numbers in the range of 0 to 100. Each node is assigned a token that represents a point in this range. In this simple example, the token values are 0, 25, 50, and 75. The first node, the one with token 0, is responsible for the *wrapping range* (75-0). The node with the lowest token also accepts row keys less than the lowest token and more than the highest token.



Understanding the Partitioner Types

When you deploy a Cassandra cluster, you must assign a partitioner and assign each node an *initial_token* value so each node is responsible for roughly an equal amount of data (load balancing). DataStax strongly recommends using the RandomPartitioner (default) for all cluster deployments.

To calculate the tokens for nodes in a single data center cluster, you divide the range by the total number of nodes in the cluster. In multiple data center deployments, you calculate the tokens such that each data center is individually load balanced. See [Generating Tokens](#) for the different approaches to generating tokens for nodes in single and multiple data center clusters.

Unlike almost every other configuration choice in Cassandra, the partitioner may not be changed without reloading all of your data. Therefore, it is important to choose and configure the correct partitioner before initializing your cluster. You set the partitioner in the *cassandra.yaml* file.

Cassandra offers the following partitioners:

- *RandomPartitioner*

- *ByteOrderedPartitioner*

About the RandomPartitioner

The RandomPartitioner uses tokens to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace. This is true even if the tables use different row keys, such as usernames or timestamps. Moreover, the read and write requests to the cluster are also evenly distributed and load balancing is simplified because each part of the hash range receives an equal number of rows on average. The RandomPartitioner (`org.apache.cassandra.dht.RandomPartitioner`) is the default partitioning strategy for a Cassandra cluster, and in almost all cases is the right choice. The RandomPartitioner distributes data evenly across the nodes using an MD5 hash value of the row key. The possible range of hash values is from 0 to $2^{127} - 1$.

When using the RandomPartitioner for single data center deployments, you calculate the tokens by dividing the hash range by the number of nodes in the cluster. For multiple data center deployments, you calculate the tokens per data center so that the hash range is evenly divided for the nodes in each data center. See *About Partitioning in Multiple Data Center Clusters*.

About the ByteOrderedPartitioner

Cassandra provides the ByteOrderedPartitioner (`org.apache.cassandra.dht.ByteOrderedPartitioner`) for ordered partitioning. This partitioner orders rows lexically by key bytes. You calculate tokens by looking at the actual values of your row key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you could assign an A token using its hexadecimal representation of 41.

Using the ordered partitioner allows range scans over rows. This means you can scan rows as though you were moving a cursor through a traditional index. For example, if your application has user names as the row key, you can scan rows for users whose names fall between Jake and Joe. This type of query is not possible with randomly partitioned row keys, since the keys are stored in the order of their MD5 hash (not sequentially). However, you can achieve the same functionality using column family *indexes*. Most applications can be designed with a data model that supports ordered queries as slices over a set of columns rather than range scans over a set of rows.

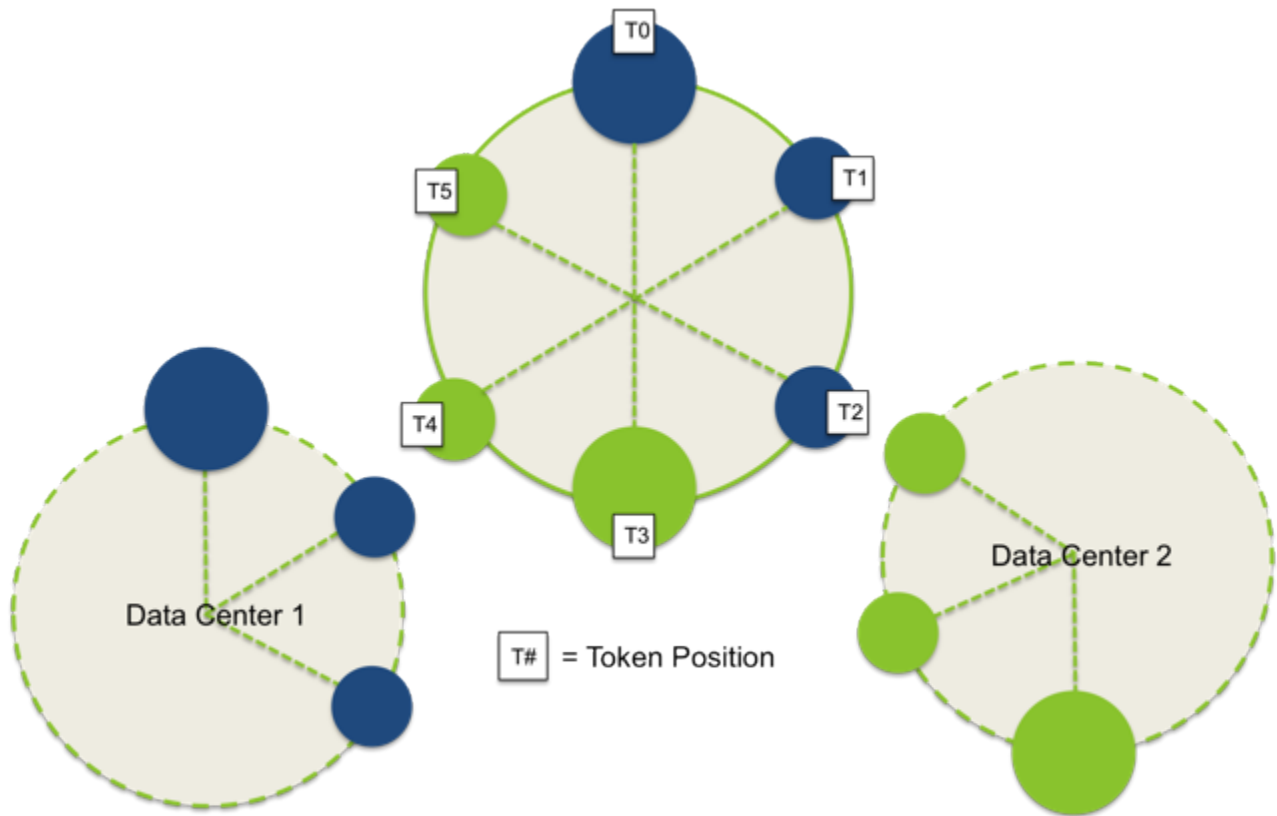
Unless absolutely required by your application, DataStax strongly recommends against using the ordered partitioner for the following reasons:

- **Sequential writes can cause hot spots:** If your application tends to write or update a sequential block of rows at a time, then these writes are not distributed across the cluster; they all go to one node. This is frequently a problem for applications dealing with timestamped data.
- **More administrative overhead to load balance the cluster:** An ordered partitioner requires administrators to manually calculate token ranges based on their estimates of the row key distribution. In practice, this requires actively moving node tokens around to accommodate the actual distribution of data once it is loaded.
- **Uneven load balancing for multiple column families:** If your application has multiple column families, chances are that those column families have different row keys and different distributions of data. An ordered partitioner that is balanced for one column family may cause hot spots and uneven distribution for another column family in the same cluster.

About Partitioning in Multiple Data Center Clusters

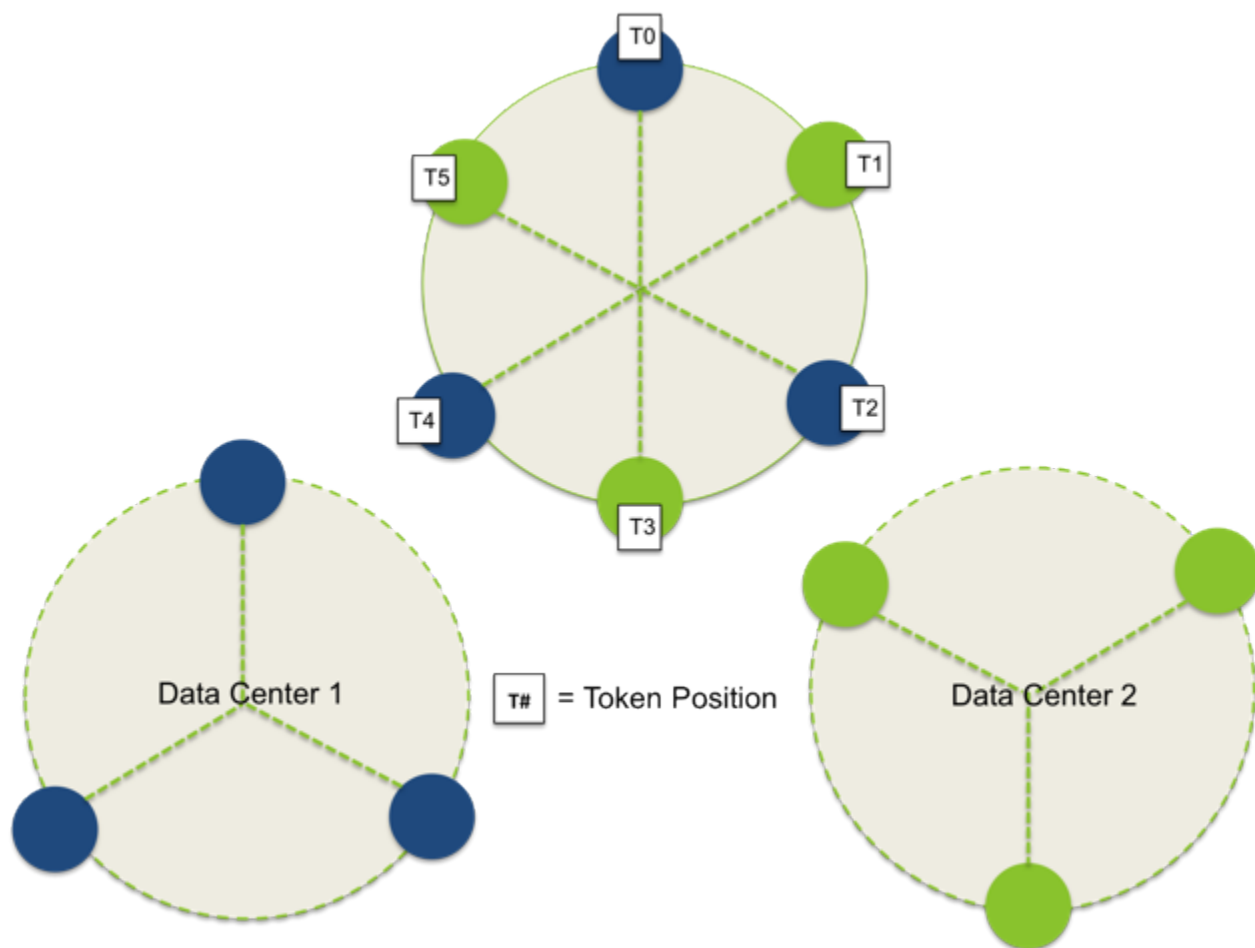
The preferred replication placement strategy for multiple data center deployments is the *NetworkTopologyStrategy*, which calculates replica placement per data center. This strategy places the first replica for each row by the token value assigned to each node. It places additional replicas in the same data center by walking the ring clockwise until it reaches the first node in another rack. This means that you must calculate partitioner tokens so that the data ranges are evenly distributed for each data center, uneven data distribution within a data center may occur:

Uneven Data Distribution



To ensure that the nodes for each data center have token assignments that evenly divide the overall range, each data center should be partitioned as if it were its own distinct ring. This averts having a disproportionate number of row keys in any one data center. However, you must avoid assigning tokens that may conflict with other token assignments elsewhere in the cluster. To make sure that each node has a unique token, see [Generating Tokens](#).

Even Data Distribution



About Replication in Cassandra

Replication is the process of storing copies of data on multiple nodes to ensure reliability and fault tolerance.

Cassandra stores copies, called replicas, of each row based on the row key. You set the number of replicas when you create a keyspace using the *replica placement strategy*. In addition to setting the number of replicas, this strategy sets the distribution of the replicas across the nodes in the cluster depending on the cluster's topology.

The total number of replicas across the cluster is referred to as the *replication factor*. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no *primary* or *master* replica. As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes afterwards. When replication factor exceeds the number of nodes, writes are rejected, but reads are served as long as the desired *consistency level* can be met.

To determine the physical location of nodes and their proximity to each other, you need to configure a *snitch* for your cluster in addition to the replication strategy.

Replication Strategy

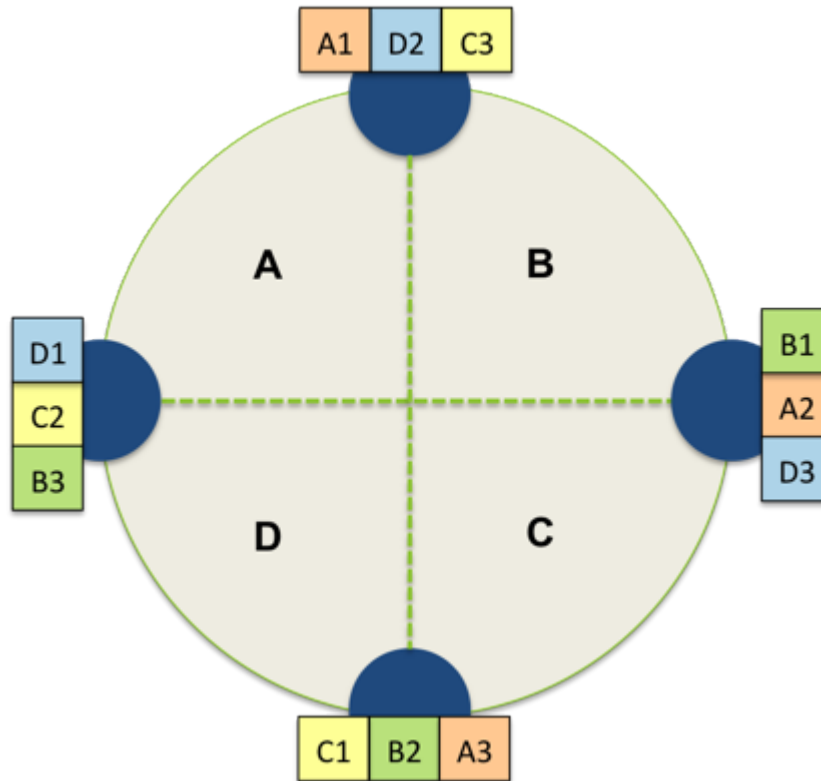
The available strategies are:

- *SimpleStrategy*
- *NetworkTopologyStrategy*

SimpleStrategy

Use SimpleStrategy for simple single data center clusters. This strategy is the default replica placement strategy when creating a keyspace using the Cassandra CLI. See [Creating a Keyspace](#). When using the Cassandra Query Language interface, you must explicitly specify a strategy. See [CREATE KEYSPACE](#).

SimpleStrategy places the first replica on a node determined by the *partitioner*. Additional replicas are placed on the next nodes clockwise in the ring without considering rack or data center location. The following graphic shows three replicas of three rows placed across four nodes:



NetworkTopologyStrategy

Use NetworkTopologyStrategy when you have (or plan to have) your cluster deployed across multiple data centers. This strategy specifies how many replicas you want in each data center.

When deciding how many replicas to configure in each data center, the two primary considerations are (1) being able to satisfy reads locally, without incurring cross-datacenter latency, and (2) failure scenarios. The two most common ways to configure multiple data center clusters are:

- **Two replicas in each data center.** This configuration tolerates the failure of a single node per replication group and still allows local reads at a *consistency level* of ONE.
- **Three replicas in each data center.** This configuration tolerates the failure of a one node per replication group at a strong *consistency level* of LOCAL_QUORUM or tolerates multiple node failures per data center using consistency level ONE.

Asymmetrical replication groupings are also possible. For example, you can have three replicas per data center to serve real-time application requests and use a single replica for running analytics.

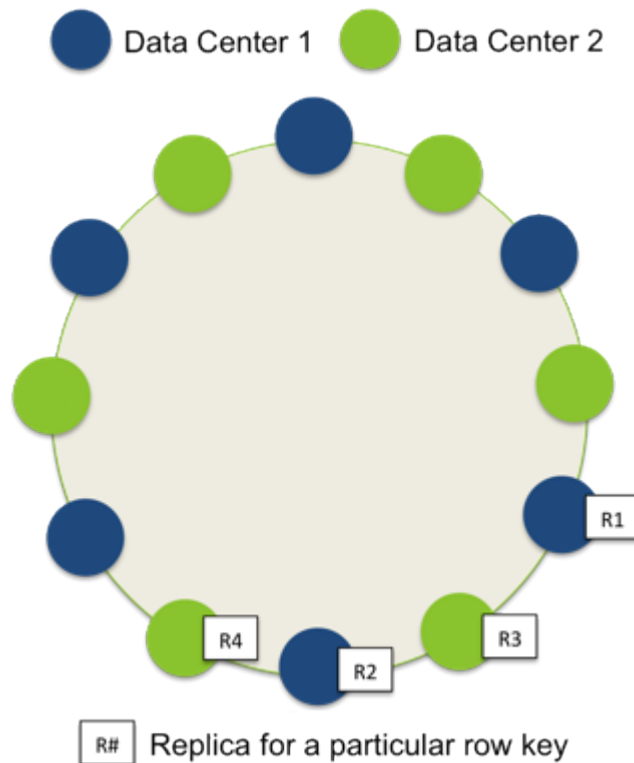
The NetworkTopologyStrategy determines replica placement independently within each data center as follows:

- The first replica is placed according to the *partitioner* (same as with SimpleStrategy).

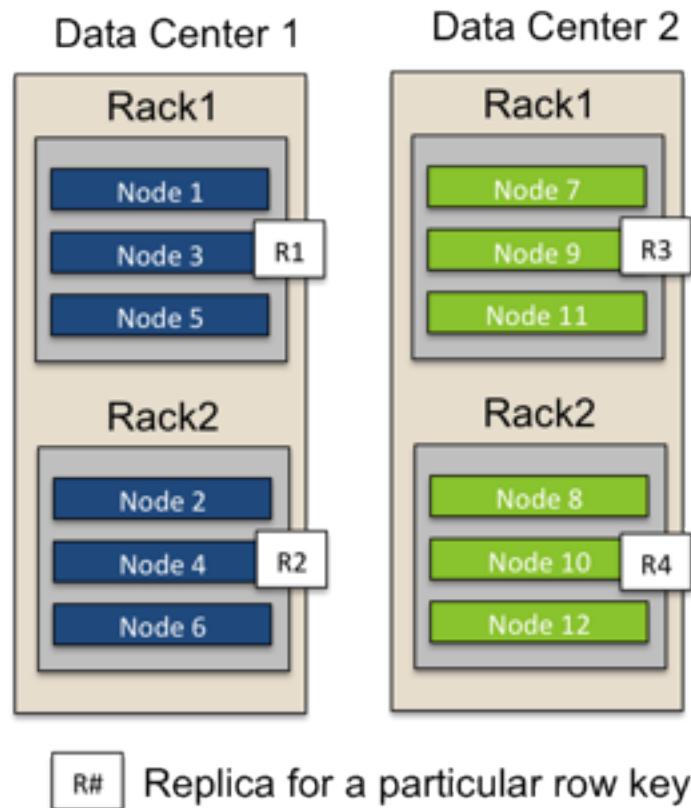
- Additional replicas are placed by walking the ring clockwise until a node in a different rack is found. If no such node exists, additional replicas are placed in different nodes in the same rack.

`NetworkTopologyStrategy` attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) can fail at the same time due to power, cooling, or network issues.

The following example shows how `NetworkTopologyStrategy` places replicas spanning two data centers with a total replication factor of 4. When using `NetworkTopologyStrategy`, you set the number of replicas per data center.



In the following graphic, notice the tokens are assigned to alternating racks. For more information, see [Generating Tokens](#).



`NetworkTopologyStrategy` relies on a properly configured *snitch* to place replicas correctly across data centers and racks. It is important to configure your cluster to use the type of snitch that correctly determines the locations of nodes in your network.

Note

Be sure to use `NetworkTopologyStrategy` instead of the `OldNetworkTopologyStrategy`, which supported only a limited configuration of 3 replicas across 2 data centers, without control over which data center got the two replicas for any given row key. This strategy meant that some rows had two replicas in the first and one replica in the second, while others had two in the second and one in the first.

About Snitches

A snitch maps IPs to racks and data centers. It defines how the nodes are grouped together within the overall network topology. Cassandra uses this information to route inter-node requests as efficiently as possible. The snitch does not affect requests between the client application and Cassandra and it does not control which node a client connects to.

You configure snitches in the `cassandra.yaml` configuration file. All nodes in a cluster must use the same snitch configuration.

The following snitches are available:

SimpleSnitch

The `SimpleSnitch` (the default) does not recognize data center or rack information. Use it for single-data center deployments (or single-zone in public clouds).

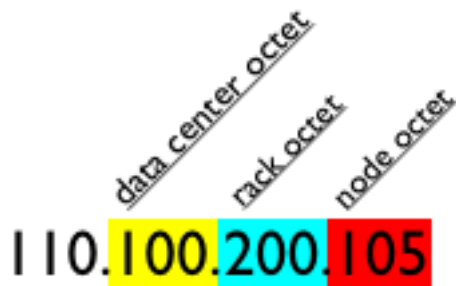
When defining your keyspace *strategy_options*, use `replication_factor=<#>`.

DseSimpleSnitch

For more information about this snitch, see [DataStax Enterprise documentation](#).

RackInferringSnitch

The RackInferringSnitch infers (assumes) the topology of the network by the octet of the node's IP address. Use this snitch as an example of writing a custom Snitch class.



When defining your keyspace *strategy_options*, use the second octet number of your node IPs for your data center name. In the above graphic, you would use 100 for the data center name.

PropertyFileSnitch

The PropertyFileSnitch determines the location of nodes by rack and data center. This snitch uses a user-defined description of the network details located in the property file `cassandra-topology.properties`. Use this snitch when your node IPs are not uniform or if you have complex replication grouping requirements.

When using this snitch, define your data center names as desired and make sure that the data center names defined in the `cassandra-topology.properties` file correlates to the name of your data centers in your keyspace *strategy_options*. Every node in the cluster should be described in this file, and the file should be exactly the same on every node in the cluster.

For example, if your cluster had non-uniform IPs and two physical data centers with two racks in each, and a third logical data center for replicating analytics data, the `cassandra-topology.properties` file might look like this:

```
# Data Center One

175.56.12.105=DC1:RAC1
175.50.13.200=DC1:RAC1
175.54.35.197=DC1:RAC1

120.53.24.101=DC1:RAC2
120.55.16.200=DC1:RAC2
120.57.102.103=DC1:RAC2

# Data Center Two

110.56.12.120=DC2:RAC1
110.50.13.201=DC2:RAC1
110.54.35.184=DC2:RAC1

50.33.23.120=DC2:RAC2
50.45.14.220=DC2:RAC2
50.17.10.203=DC2:RAC2

# Analytics Replication Group

172.106.12.120=DC3:RAC1
```

```
172.106.12.121=DC3:RAC1
172.106.12.122=DC3:RAC1

# default for unknown nodes
default=DC3:RAC1
```

GossipingPropertyFileSnitch

The `GossipingPropertyFileSnitch` allows you to define a local node's data center and rack and use gossip for propagating the information to other nodes. To define the data center and rack, create a `cassandra-rackdc.properties` file in the node's `conf` directory. For example:

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

The location of the `conf` directory depends on the type of installation; see [Cassandra Configuration Files Locations](#) or [DataStax Enterprise Configuration Files Locations](#)

To migrate from the `PropertyFileSnitch` to the `GossipingPropertyFileSnitch`, update one node at a time to allow gossip time to propagate. The `PropertyFileSnitch` is used as a fallback when `cassandra-topologies.properties` is present.

EC2Snitch

Use the `EC2Snitch` for simple cluster deployments on Amazon EC2 where all nodes in the cluster are within a single region. The region is treated as the data center and the availability zones are treated as racks within the data center. For example, if a node is in `us-east-1a`, `us-east` is the data center name and `1a` is the rack location. Because private IPs are used, this snitch does not work across multiple regions.

When defining your keyspace *strategy_options*, use the EC2 region name (for example, `us-east`) as your data center name.

EC2MultiRegionSnitch

Use the `EC2MultiRegionSnitch` for deployments on Amazon EC2 where the cluster spans multiple regions. As with the `EC2Snitch`, regions are treated as data centers and availability zones are treated as racks within a data center. For example, if a node is in `us-east-1a`, `us-east` is the data center name and `1a` is the rack location.

This snitch uses public IPs as `broadcast_address` to allow cross-region connectivity. This means that you must configure each Cassandra node so that the *listen_address* is set to the *private* IP address of the node, and the *broadcast_address* is set to the *public* IP address of the node. This allows Cassandra nodes in one EC2 region to bind to nodes in another region, thus enabling multiple data center support. (For intra-region traffic, Cassandra switches to the private IP after establishing a connection.)

Additionally, you must set the addresses of the seed nodes in the `cassandra.yaml` file to that of the *public* IPs because private IPs are not routable between networks. For example:

```
seeds: 50.34.16.33, 60.247.70.52
```

To find the public IP address, run this command from each of the seed nodes in EC2:

```
curl http://instance-data/latest/meta-data/public-ipv4
```

Finally, be sure that the *storage_port* or *ssl_storage_port* is open on the public IP firewall.

When defining your keyspace *strategy_options*, use the EC2 region name, such as `us-east`, as your data center names.

About Dynamic Snitching

By default, all snitches also use a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes. The dynamic snitch is enabled by default; it is recommended for use in most deployments.

Configure dynamic snitch thresholds for each node in the `cassandra.yaml` configuration file. For more information, see the properties listed under *Inter-node Communication and Fault Detection Properties*.

About Client Requests in Cassandra

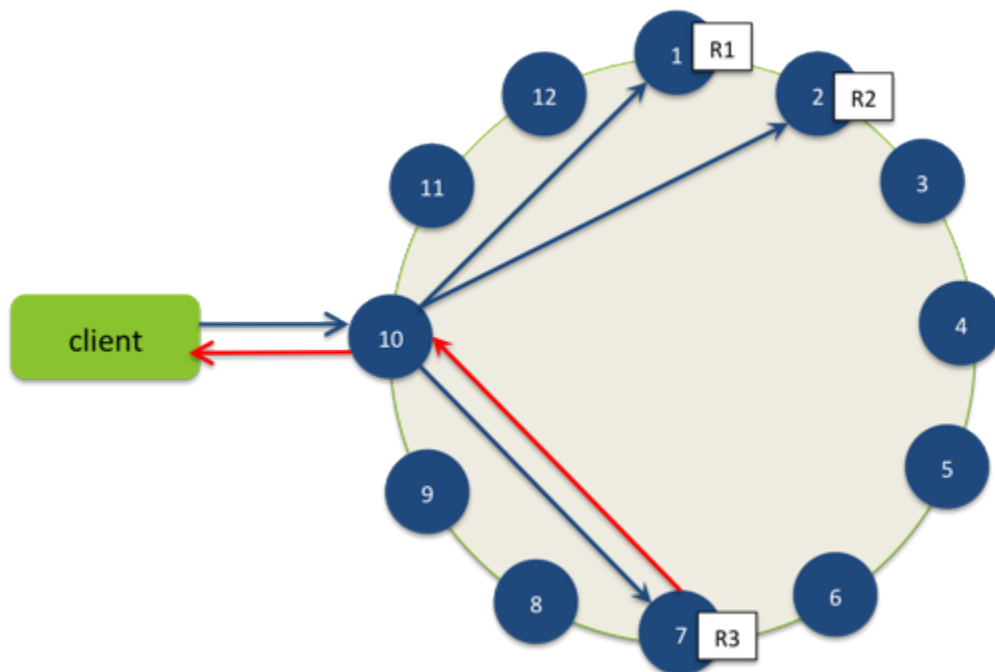
All nodes in Cassandra are peers. A client read or write request can go to any node in the cluster. When a client connects to a node and issues a read or write request, that node serves as the *coordinator* for that particular client operation.

The job of the coordinator is to act as a proxy between the client application and the nodes (or replicas) that own the data being requested. The coordinator determines which nodes in the ring should get the request based on the cluster configured *partitioner* and *replica placement strategy*.

About Write Requests

For writes, the coordinator sends the write to *all* replicas that own the row being written. As long as all replica nodes are up and available, they will get the write regardless of the *consistency level* specified by the client. The write consistency level determines how many replica nodes must respond with a success acknowledgement in order for the write to be considered successful.

For example, in a single data center 10 node cluster with a replication factor of 3, an incoming write will go to all 3 nodes that own the requested row. If the write consistency level specified by the client is ONE, the first node to complete the write responds back to the coordinator, which then proxies the success message back to the client. A consistency level of ONE means that it is possible that 2 of the 3 replicas could miss the write if they happened to be down at the time the request was made. If a replica misses a write, the row will be made consistent later via one of Cassandra's *built-in repair mechanisms*: hinted handoff, read repair or anti-entropy node repair.

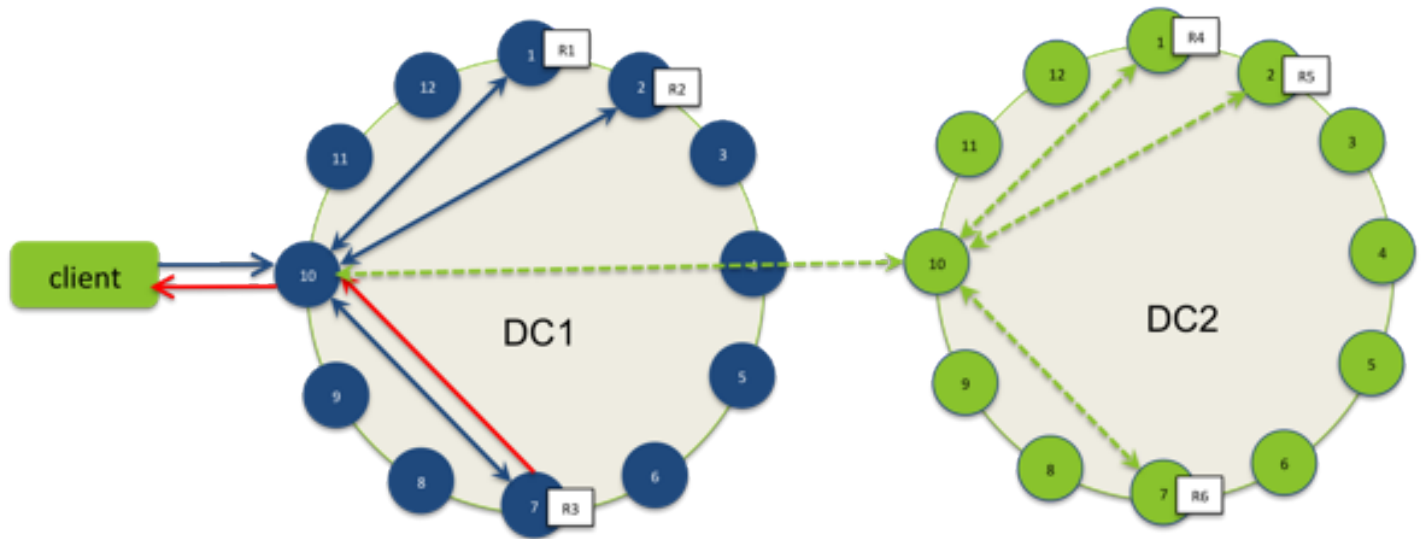


Also see *About Writes in Cassandra* for more information about how Cassandra processes writes locally at the node level.

About Multiple Data Center Write Requests

In multi data center deployments, Cassandra optimizes write performance by choosing one coordinator node in each remote data center to handle the requests to replicas within that data center. The coordinator node contacted by the client application only needs to forward the write request to one node in each remote data center.

If using a *consistency level* of ONE or LOCAL_QUORUM, only the nodes in the same data center as the coordinator node must respond to the client request in order for the request to succeed. This way, geographical latency does not impact client request response times.



About Read Requests

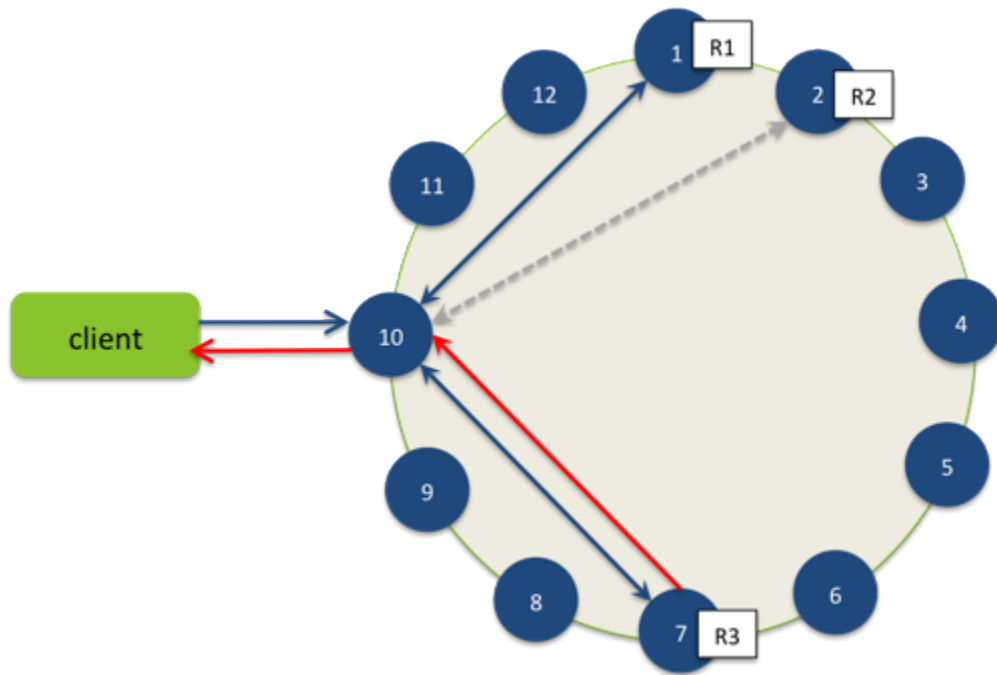
For reads, there are two types of read requests that a coordinator can send to a replica; a direct read request and a background *read repair* request. The number of replicas contacted by a direct read request is determined by the *consistency level* specified by the client. Background read repair requests are sent to any additional replicas that did not receive a direct request. Read repair requests ensure that the requested row is made consistent on all replicas.

Thus, the coordinator first contacts the replicas specified by the consistency level. The coordinator sends these requests to the replicas that respond promptly. The nodes contacted respond with the requested data; if multiple nodes are contacted, the rows from each replica are compared for consistency in memory. If replicas are inconsistent, the following events occur:

1. Regardless of the `read_repair_chance` setting, a foreground read repair occurs on the data.
2. The coordinator uses the replica that has the most recent data (based on the timestamp) to forward the result back to the client.
3. In the background, the coordinator compares the data from all the remaining replicas that own the row.
4. If the data from the replicas is inconsistent, the coordinator issues writes to the out-of-date replicas, updating the row to reflect the most recently written values.

The process described in step 4 is known as *read repair*. Read repair can be configured per column family (using *read_repair_chance*), and is enabled by default.

For example, in a cluster with a replication factor of 3, and a read consistency level of QUORUM, 2 of the 3 replicas for the given row are contacted to fulfill the read request. Supposing the contacted replicas had different versions of the row, the replica with the most recent version would return the requested data. In the background, the third replica is checked for consistency with the first two, and if needed, the most recent replica issues a write to the out-of-date replicas.



Also see [About Reads in Cassandra](#) for more information about how Cassandra processes reads locally at the node level.

Hadoop Integration

Hadoop integration with Cassandra includes support for:

- MapReduce
- Apache Pig
- Apache Hive

For more detailed information, see [Hadoop Support](#).

Starting with Cassandra 1.1, the following low-level features have been added to Cassandra-Hadoop integration:

- Secondary index support for the column family input format - Hadoop jobs can now make use of Cassandra secondary indexes.
- Wide row support - Previously, wide rows had, for example, millions of columns that could not be accessed by Hadoop. Now they can be read and paged through in Hadoop.
- BulkOutputFormat - Better efficiency when loading data into Cassandra from a Hadoop job.

Secondary Index Support

Hadoop jobs can now make use of indexes and can specify expressions that direct Cassandra to scan over specific data and send the data back to the map job. This scheme pushes work down to the server instead transferring data back and forth.

You can overload the `ConfigHelper.setInputRange` class, which specifies input data to use a list of expressions that verify the data before including it in the Hadoop result set.

```
IndexExpression expr =  
    new IndexExpression(  
        ByteBufferUtil.bytes("int4"),  
        IndexOperator.EQ,  
        ByteBufferUtil.bytes(0)
```

```
);  
  
ConfigHelper.setInputRange(  
    job.getConfiguration(),  
    Arrays.asList(expr)  
);
```

The Cassandra WordCount example has been revised to demonstrate secondary index support. It is available in the example directory of the [Apache Cassandra project repository](#).

Wide Row Support

The [Cassandra HDFS interface](#) includes a new parameter to support wide rows:

```
ConfigHelper.setInputColumnFamily(  
    job.getConfiguration(),  
    KEYSPACE,  
    COLUMN_FAMILY,  
    true  
);
```

If the Boolean parameter is true, column family rows are formatted as individual columns. You can paginate wide rows into column-slices, similar application query pagination. For example, the paging can occur in thousand-column chunks instead of one row (at least) chunks.

Bulk Loading Hadoop Data

When loading data into Cassandra from a Hadoop job, you can manage the Hadoop output using a new class: BulkOutputFormat. Use this class to set the input column family format.

```
ConfigHelper.setInputColumnFamily(  
    BulkOutputFormat.class);
```

Alternatively, you can still use the ColumnFamilyOutputFormat class.

Setting the output format class to BulkOutputFormat instead of ColumnFamilyOutputFormat improves throughput of big jobs. You bulk load large amounts of data and MapReduce over it to format it for your Cassandra application. The Hadoop job streams the data directly to the nodes in a binary format that Cassandra handles very efficiently. The downside is that you no longer see the data arriving incrementally in Cassandra when you send the data to the nodes in bulk chunks.

The new options (and defaults), which can be used for both classes are:

- OUTPUT_LOCATION (system default)
- BUFFER_SIZE_IN_MB (64)
- STREAM_THROTTLE_MBITS (unlimited)

You should not need to change the defaults for these options.

Planning a Cassandra Cluster Deployment

When planning a Cassandra cluster deployment, you should have a good idea of the initial volume of data you plan to store and a good estimate of the typical application workload. After reading this section, it is recommended that you read [Anti-patterns in Cassandra](#).

Selecting Hardware for Enterprise Implementations

As with any application, choosing appropriate hardware depends on selecting the right balance of the following resources: memory, CPU, disks, number of nodes, and network.

Memory

The more memory a Cassandra node has, the better read performance. More RAM allows for larger cache sizes and reduces disk I/O for reads. More RAM also allows memory tables (memtables) to hold more recently written data. Larger memtables lead to a fewer number of SSTables being flushed to disk and fewer files to scan during a read. The ideal amount of RAM depends on the anticipated size of your hot data.

- For dedicated hardware, the optimal price-performance sweet spot is 16GB to 64GB; the minimum is 8GB.
- For a virtual environments, the optimal range may be 8GB to 16GB; the minimum is 4GB.
- For testing light workloads, Cassandra can run on a virtual machine as small as 256MB.
- Java heap space should be set to a maximum of 8GB or half of your total RAM, whichever is lower. (A greater heap size has more intense garbage collection periods.)

CPU

Insert-heavy workloads are CPU-bound in Cassandra before becoming memory-bound. Cassandra is highly concurrent and uses as many CPU cores as available.

- For dedicated hardware, 8-core processors are the current price-performance sweet spot.
- For virtual environments, consider using a provider that allows CPU bursting, such as Rackspace Cloud Servers.

Disk

What you need for your environment depends a lot on the usage, so it's important to understand the mechanism. Cassandra writes data to disk for two purposes:

- All data is written to the commit log for durability.
- When thresholds are reached, Cassandra periodically flushes in-memory data structures (memtables) to SSTable data files for persistent storage of column family data.

Commit logs receive every write made to a Cassandra node, but are only read during node start up. Commit logs are purged after the corresponding data is flushed. Conversely, SSTable (data file) writes occur asynchronously and are read during client look-ups. Additionally, SSTables are periodically compacted. Compaction improves performance by merging and rewriting data and discarding old data. However, during compaction (or node repair), disk utilization and data directory volume can substantially increase. For this reason, DataStax recommends leaving an adequate amount of free disk space available on a node (50% [worst case] for tiered compaction, 10% for leveled compaction).

The following links provide information about compaction:

- [The Apache Cassandra storage engine](#)
- [Leveled Compaction in Apache Cassandra](#)
- [When to Use Leveled Compaction](#)

Recommendations:

- When choosing disks, consider both capacity (how much data you plan to store) and I/O (the write/read throughput rate). Most workloads are best served by using less expensive SATA disks and scaling disk capacity and I/O by adding more nodes (with more RAM).
- Solid-state drives (SSDs) are the recommended choice for Cassandra. Cassandra's sequential, streaming write patterns minimize the undesirable effects of **write amplification** associated with SSDs. This means that Cassandra deployments can take advantage of inexpensive consumer-grade SSDs.
- Ideally Cassandra needs at least two disks (if using HDDs), one for the commit log and the other for the data directories. At a minimum the commit log should be on its own partition.
- Commit log disk - this disk does not need to be large, but it should be fast enough to receive all of your writes as appends (sequential I/O).

- Data disks - use one or more disks and make sure they are large enough for the data volume and fast enough to both satisfy reads that are not cached in memory and to keep up with compaction.
- RAID - compaction can temporarily require up to 100% of the free in-use disk space on a single data directory volume. This means when approaching 50% of disk capacity, you should use RAID 0 or RAID 10 for your data directory volumes. RAID also helps smooth out I/O hotspots within a single SSTable.
 - Use RAID0 if disk capacity is a bottleneck and rely on Cassandra's replication capabilities for disk failure tolerance. If you lose a disk on a node, you can recover lost data through Cassandra's built-in repair.
 - Use RAID10 to avoid large repair operations after a single disk failure, or if you have disk capacity to spare.
 - Generally RAID is not needed for the commit log disk because replication adequately prevents data loss. If you need the extra redundancy, use RAID 1.
- Extended file systems - On ext2 or ext3, the maximum file size is 2TB even using a 64-bit kernel. On ext4 it is 16TB.

Because Cassandra can use almost half your disk space for a single file, use XFS when raiding large disks together, particularly if using a 32-bit kernel. XFS file size limits are 16TB max on a 32-bit kernel, and essentially unlimited on 64-bit.

Number of Nodes

The amount of data on each disk in the array isn't as important as the total size per node. Using a greater number of smaller nodes is better than using fewer larger nodes because of potential bottlenecks on larger nodes during compaction.

Network

Since Cassandra is a distributed data store, it puts load on the network to handle read/write requests and replication of data across nodes. Be sure to choose reliable, redundant network interfaces and make sure that your network can handle traffic between nodes without bottlenecks.

- Recommended bandwidth is 1000 Mbit/s (Gigabit) or greater.
- Bind the Thrift interface (*listen_address*) to a specific NIC (Network Interface Card).
- Bind the RPC server interface (*rpc_address*) to another NIC.

Cassandra is efficient at routing requests to replicas that are geographically closest to the coordinator node handling the request. Cassandra will pick a replica in the same rack if possible, and will choose replicas located in the same data center over replicas in a remote data center.

Firewall

If using a firewall, make sure that nodes within a cluster can reach each other on these ports. See [Configuring Firewall Port Access](#).

Note

Generally, when you have firewalls between machines, it is difficult to run JMX across a network and maintain security. This is because JMX connects on port 7199, handshakes, and then uses any port within the 1024+ range. Instead use SSH to execute commands remotely connect to JMX locally or use the DataStax OpsCenter.

Planning an Amazon EC2 Cluster

DataStax provides an Amazon Machine Image (AMI) to allow you to quickly deploy a multi-node Cassandra cluster on Amazon EC2. The DataStax AMI initializes all nodes in one availability zone using the [SimpleSnitch](#). If you want an EC2 cluster that spans multiple regions and availability zones, do not use the DataStax AMI. Instead, install Cassandra on

your EC2 instances as described in *Installing Cassandra Debian Packages*, and then configure the cluster as a *multiple data center cluster*.

Use the following guidelines when setting up your cluster:

EBS volumes are **not** recommended for Cassandra data volumes. Their network performance and disk I/O are not good fits for Cassandra for the following reasons:

- EBS volumes contend directly for network throughput with standard packets. This means that EBS throughput is likely to fail if you saturate a network link.
- EBS volumes have unreliable performance. I/O performance can be exceptionally slow, causing the system to backload reads and writes until the entire cluster becomes unresponsive.
- Adding capacity by increasing the number of EBS volumes per host does not scale. You can easily surpass the ability of the system to keep effective buffer caches and concurrently serve requests for all of the data it is responsible for managing.

For more information and graphs related to ephemeral versus EBS performance, see the blog article at <http://blog.scalyr.com/2012/10/16/a-systematic-look-at-ec2-io/>.

Calculating Usable Disk Capacity

To calculate how much data your Cassandra nodes can hold, calculate the usable disk capacity per node and then multiply that by the number of nodes in your cluster. Remember that in a production cluster, you will typically have your commit log and data directories on different disks. This calculation is for estimating the usable capacity of the data volume.

Start with the raw capacity of the physical disks:

```
raw_capacity = disk_size * number_of_disks
```

Account for file system formatting overhead (roughly 10 percent) and the RAID level you are using. For example, if using RAID-10, the calculation would be:

```
(raw_capacity * 0.9) / 2 = formatted_disk_space
```

During normal operations, Cassandra routinely requires disk capacity for compaction and repair operations. For optimal performance and cluster health, DataStax recommends that you do not fill your disks to capacity, but run at 50-80 percent capacity. With this in mind, calculate the usable disk space as follows (example below uses 50%):

```
formatted_disk_space * 0.5 = usable_disk_space
```

Calculating User Data Size

As with all data storage systems, the size of your raw data will be larger once it is loaded into Cassandra due to storage overhead. On average, raw data will be about 2 times larger on disk after it is loaded into the database, but could be much smaller or larger depending on the characteristics of your data and column families. The calculations in this section account for data persisted to disk, not for data stored in memory.

- **Column Overhead** - Every column in Cassandra incurs 15 bytes of overhead. Since each row in a column family can have different column names as well as differing numbers of columns, metadata is stored for *each* column. For counter columns and expiring columns, add an additional 8 bytes (23 bytes column overhead). So the total size of a *regular* column is:

```
total_column_size = column_name_size + column_value_size + 15
```

- **Row Overhead** - Just like columns, every row also incurs some overhead when stored on disk. Every row in Cassandra incurs 23 bytes of overhead.

- **Primary Key Index** - Every column family also maintains a primary index of its row keys. Primary index overhead becomes more significant when you have lots of *skinny* rows. Sizing of the primary row key index can be estimated as follows (in bytes):

```
primary_key_index = number_of_rows * (32 + average_key_size)
```

- **Replication Overhead** - The replication factor obviously plays a role in how much disk capacity is used. For a replication factor of 1, there is no overhead for replicas (as only one copy of your data is stored in the cluster). If replication factor is greater than 1, then your total data storage requirement will include replication overhead.

```
replication_overhead = total_data_size * (replication_factor - 1)
```

Choosing Node Configuration Options

A major part of planning your Cassandra cluster deployment is understanding and setting the various node configuration properties. These properties are set in the *cassandra.yaml* configuration file. Each node should be correctly configured before starting it for the first time:

- **Storage settings:** By default, a node is configured to store the data it manages in the `/var/lib/cassandra` directory. In a production cluster deployment, you should change the *commitlog_directory* to a different disk drive from the *data_file_directories*.
- **Gossip settings:** The *gossip-related settings* control a node's participation in a cluster and how the node is known to the cluster.
- **Purging gossip state on a node:** Gossip information is also persisted locally by each node to use immediately when a node restarts. You may want to *purge gossip history* on node restart for various reasons, such as when the node's IP addresses has changed.
- **Partitioner settings:** You must set the *partitioner type* and assign each node an *initial_token* value.
- **Snitch settings:** You need to configure a *snitch* when you create a cluster. The snitch is responsible for knowing the location of nodes within your network topology and distributing replicas by grouping machines into data centers and racks.
- **Keyspace replication settings:** When you create a keyspace, you must define the *replica placement strategy* and the number of replicas you want.

Anti-patterns in Cassandra

The anti-patterns described here are implementation or design patterns that ineffective and/or counterproductive in Cassandra production installations. Correct patterns are suggested in most cases.

Network Attached Storage

Storing SSTables on a Network Attached Storage (NAS) device is of limited use. Using a NAS device often results in network related bottlenecks resulting from high levels of I/O wait time on both reads and writes. The causes of these bottlenecks include:

- Router latency.
- The Network Interface Card (NIC) in the node.
- The NIC in the NAS device.

There are exceptions to this pattern. If you use NAS, ensure that each drive is accessed only by one machine and each drive is physically close to the node.

Shared Network File Systems

Shared Network File Systems (NFS) have the same limitations as NAS. The temptation with NFS implementations is to place all SSTables in a node into one NFS. Doing this deprecates one of Cassandra's strongest features: No Single Point of Failure (SPOF). When all SSTables from all nodes are stored onto a single NFS, the NFS becomes a SPOF. To best use Cassandra, avoid using NFS.

Excessive Heap Space Size

DataStax recommends using the default heap space size for most use cases. Exceeding this size can impair the Java virtual machine's (JVM) ability to perform fluid garbage collections (GC). The following table shows a comparison of heap space performances reported by a Cassandra user:

Heap	CPU utilization	Queries per second	Latency
40 GB	50%	750	1 second
8 GB	5%	8500 ^[1]	10 ms

Cassandra's Rack Feature

Defining one rack for the entire cluster is the simplest and most common implementation. Multiple racks should be avoided for the following reasons:

- Most users tend to ignore or forget rack requirements that racks should be organized in an alternating order. This order allows the data to get distributed safely and appropriately.
- Many users are not using the rack information effectively. For example, setting up with as many racks as nodes (or similar non-beneficial scenarios).
- Expanding a cluster when using racks can be tedious. The procedure typically involves several node moves and must ensure that racks are distributing data correctly and evenly. When clusters need immediate expansion, racks should be the last concern.

To use racks correctly:

- Use the same number of nodes in each rack.
- Use one rack and place the nodes in different racks in an alternating pattern. This allows you to still get the benefits of Cassandra's rack feature, and allows for quick and fully functional expansions. Once the cluster is stable, you can swap nodes and make the appropriate moves to ensure that nodes are placed in the ring in an alternating fashion with respect to the racks.

Multiple-gets

Multiple-gets may cause problems. One sure way to kill a node is to buffer 300MB of data, timeout, and then try again from 50 different clients.

You should architect your application using many single requests for different rows. This method ensures that if a read fails on a node, due to a backlog of pending requests, an unmet consistency, or other error, only the failed request needs to be retried.

Ideally, use the same key reading for the entire key or slices. Be sure to keep the row sizes in mind to prevent out-of-memory (OOM) errors by reading too many entire ultra-wide rows in parallel.

Super columns

Do not use super columns. They are a legacy design from a pre-open source release. This design was structured for a specific use case and does not fit most use cases. Super columns read entire super columns and all its sub-columns into memory for each read request. This results in severe performance issues. Additionally, super columns are not supported in CQL 3.

Use composite columns instead. Composite columns provide most of the same benefits as super columns without the performance issues.

Using Byte Ordered Partitioner (BOP)

The Byte Ordered Partitioner is not recommended.

The RandomPartitioner is recommended because all writes occur on the MD5 hash of the key and are therefore spread out throughout the ring amongst tokens ranging from 0 to $2^{127} - 1$. This partitioner ensures that your cluster evenly distributes data by placing the key at the correct token using the key's hash value. Even if data becomes stale and needs to be deleted, this ensures that data removal also takes place while evenly distributing data around the cluster.

Reading Before Writing

Reads take time for every request, as they typically have multiple disk hits for uncached reads. In workflows requiring reads before writes, this small amount of latency can affect overall throughput. All write I/O in Cassandra is sequential so there is very little performance difference regardless of data size or key distribution.

Load Balancers

Cassandra was designed to avoid the need for load balancers. Putting load balancers between Cassandra and Cassandra clients is harmful to performance, cost, availability, debugging, testing, and scaling. All high-level clients, such as Astyanax and pycassa, implement load balancing directly.

Insufficient testing

Be sure to test at scale and production loads. This the best way to ensure your system will function properly when your application goes live. The information you gather from testing is the best indicator of what throughput per node is needed for future expansion calculations.

To properly test, set up a small cluster with production loads. There will be a maximum throughput associated with each node count before the cluster can no longer increase performance. Take the maximum throughput at this cluster size and apply it linearly to a cluster size of a different size. Next extrapolate (graph) your results to predict the correct cluster sizes for required throughputs for your production cluster. This allows you to predict the correct cluster sizes for required throughputs in the future. The [Netflix case study](#) shows an excellent example for testing.

Lack of familiarity with Linux

Linux has a great collection of tools. Become familiar with the Linux built-in tools. It will help you greatly and ease operation and management costs in normal, routine functions. The essential list of tools and techniques to learn are:

- **Parallel SSH and Cluster SSH:** The pssh and cssh tools allow SSH access to multiple nodes. This is useful for inspections and cluster wide changes.
- **Passwordless SSH:** SSH authentication is carried out by using public and private keys. This allows SSH connections to easily hop from node to node without password access. In cases where more security is required, you can implement a password Jump Box and/or VPN.

- **Useful common command-line tools include:**

- **top:** Provides an ongoing look at processor activity in real time.
- **System performance tools:** Tools such as iostat, mpstat, iftop, sar, lsof, netstat, htop, vmstat, and similar can collect and report a variety of metrics about the operation of the system.
- **vmstat:** Reports information about processes, memory, paging, block I/O, traps, and CPU activity.
- **iftop:** Shows a list of network connections. Connections are ordered by bandwidth usage, with the pair of hosts responsible for the most traffic at the top of list. This tool makes it easier to identify the hosts causing network congestion.

Running Without the Recommended Settings

Be sure to use the *recommended settings* in the Cassandra documentation.

Also be sure to consult the *Planning a Cassandra Cluster Deployment* documentation, which discusses hardware and other recommendations before making your final hardware purchases.

More Anti-Patterns

For more about anti-patterns, visit the [Matt Dennis slideshare](#).

Installing a Cassandra Cluster

Installing a Cassandra cluster involves installing the Cassandra software on each node. After each node is installed, configure each node as described in *Initializing a Cassandra Cluster*.

Note

For information on installing for evaluation and on Windows, see the [Quick Start Documentation](#).

Installing Cassandra RHEL or CentOS Packages

DataStax provides `yum` repositories for CentOS and RedHat Enterprise. For a complete list of supported platforms, see [DataStax Community – Supported Platforms](#).

Note

By downloading community software from DataStax you agree to the terms of the [DataStax Community EULA](#) (End User License Agreement) posted on the DataStax web site.

Prerequisites

Before installing Cassandra make sure the following prerequisites are met:

- Yum Package Management application installed.
- Root or sudo access to the install machine.
- The latest version of Oracle Java SE Runtime Environment (JRE) **6** is installed. Java 7 is not recommended.
- Java Native Access (JNA) is required for production installations. See *Installing JNA*.
- Also see *Recommended Settings for Production Installations*.

Steps to Install Cassandra

The packaged releases create a `cassandra` user. When starting Cassandra as a service, the service runs as this user.

1. Check which version of Java is installed by running the following command in a terminal window:

```
java -version
```

Use the latest version of Java 6 on all nodes. Java 7 is not recommended. If you need help installing Java, see *Installing the JRE on RHEL or CentOS Systems*.

2. (RHEL 5.x/CentOS 5.x only) Make sure you have EPEL (Extra Packages for Enterprise Linux) installed. EPEL contains dependent packages required by DSE, such as `jna` and `jpackage-utils`. For both 32- and 64-bit systems:

```
$ sudo rpm -Uvh http://dl.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rpm
```

3. Add a `yum` repository specification for the DataStax repository in `/etc/yum.repos.d`. For example:

```
$ sudo vi /etc/yum.repos.d/datastax.repo
```

4. In this file add the following lines for the DataStax repository:

```
[datastax]
name= DataStax Repo for Apache Cassandra
baseurl=http://rpm.datastax.com/community
enabled=1
gpgcheck=0
```

5. Install the package using `yum`.

```
$ sudo yum install dsc1.1
```

This installs the DataStax Community distribution of Cassandra and the OpsCenter Community Edition.

Next Steps

- [Initializing a Multiple Node Cluster in a Single Data Center](#)
- [Initializing Multiple Data Center Clusters on Cassandra](#)
- [Install Locations](#)

Installing Cassandra Debian Packages

DataStax provides Debian package repositories for Debian and Ubuntu. For a complete list of supported platforms, see [DataStax Community – Supported Platforms](#).

Note

By downloading community software from DataStax you agree to the terms of the [DataStax Community EULA](#) (End User License Agreement) posted on the DataStax web site.

Prerequisites

Before installing Cassandra make sure the following prerequisites are met:

- Aptitude Package Manager installed.
- Root or sudo access to the install machine.
- The latest version of Oracle Java SE Runtime Environment (JRE) **6** is installed. Java 7 is not recommended.
- Java Native Access (JNA) is required for production installations. See [Installing JNA](#). If you are using Ubuntu 10.04 LTS, you need to update to JNA 3.4, as described in [Install JNA on Ubuntu 10.04](#).
- Also see [Recommended Settings for Production Installations](#).

Steps to Install Cassandra

The packaged releases create a `cassandra` user. When starting Cassandra as a service, the service runs as this user.

1. Check which version of Java is installed by running the following command in a terminal window:

```
java -version
```

Use the latest version of Java 6 on all nodes. Java 7 is not recommended. If you need help installing Java, see [Installing the JRE on Debian or Ubuntu Systems](#).

Installing the Cassandra Binary Tarball Distribution

2. Add the DataStax Community repository to the `/etc/apt/sources.list.d/cassandra.sources.list`.

```
deb http://debian.datastax.com/community stable main
```

3. (Debian Systems Only) In `/etc/apt/sources.list`, find the line that describes your source repository for Debian and add `contrib non-free` to the end of the line. This allows installation of the Oracle JVM instead of the OpenJDK JVM. For example:

```
deb http://some.debian.mirror/debian/ $distro main contrib non-free
```

Save and close the file when you are done adding/editing your sources.

4. Add the DataStax repository key to your aptitude trusted keys.

```
$ curl -L http://debian.datastax.com/debian/repo_key | sudo apt-key add -
```

5. Install the Python CQL driver and the DataStax Community package.

```
$ sudo apt-get update
$ sudo apt-get install python-cql
$ sudo apt-get install dsc1.1 cassandra=1.1.9
```

This installs the Python CQL driver, the DataStax Community distribution of Cassandra, and the OpsCenter Community Edition.

By default, the Debian packages start the Cassandra service automatically.

6. To stop the service and clear the initial gossip history that gets populated by this initial start:

```
$ sudo service cassandra stop
$ sudo rm -rf /var/lib/cassandra/data/system/*
```

Next Steps

- [Initializing a Multiple Node Cluster in a Single Data Center](#)
- [Initializing Multiple Data Center Clusters on Cassandra](#)
- [Install Locations](#)

Installing the Cassandra Binary Tarball Distribution

DataStax provides binary tarball distributions of Cassandra for installing on platforms that do not have package support, such as Mac, or if you do not have or want to do a root installation. For a complete list of supported platforms, see [DataStax Community – Supported Platforms](#).

Note

By downloading community software from DataStax you agree to the terms of the [DataStax Community EULA](#) (End User License Agreement) posted on the DataStax web site.

Prerequisites

Before installing Cassandra make sure the following prerequisites are met:

- The latest version of Oracle Java SE Runtime Environment (JRE) **6** is installed. Java 7 is not recommended.
- Java Native Access (JNA) is required for production installations. See [Installing JNA](#). If you are using Ubuntu 10.04 LTS, you need to update to JNA 3.4, as described in [Install JNA on Ubuntu 10.04](#).

- Also see *Recommended Settings for Production Installations*.

Steps to Install Cassandra

1. Download the Cassandra DataStax Community tarball:

```
$ curl -OL http://downloads.datastax.com/community/dsc.tar.gz
```

2. Check which version of Java is installed by running the following command in a terminal window:

```
java -version
```

Use the latest version of Java 6 on all nodes. Java 7 is not recommended. If you need help installing Java, see *Installing the JRE on Debian or Ubuntu Systems*.

3. Unpack the distribution:

```
$ tar -xvzf dsc.tar.gz
$ rm *.tar.gz
```

4. By default, Cassandra installs files into the `/var/lib/cassandra` and `/var/log/cassandra` directories.

If you do not have root access to the default directories, ensure you have write access as follows:

```
$ sudo mkdir /var/lib/cassandra
$ sudo mkdir /var/log/cassandra
$ sudo chown -R $USER:$GROUP /var/lib/cassandra
$ sudo chown -R $USER:$GROUP /var/log/cassandra
```

Next Steps

- *Initializing a Multiple Node Cluster in a Single Data Center*
- *Initializing Multiple Data Center Clusters on Cassandra*
- *Install Locations*

Recommended Settings for Production Installations

The following recommendations are for production environments. You may need to adjust them accordingly for your implementation.

File Descriptors

Cassandra generally needs more than the default amount (1024) of file descriptors. To increase the number of file descriptors, change the security limits on your Cassandra nodes. For example:

```
echo "* soft nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "* hard nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "root soft nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "root hard nofile 32768" | sudo tee -a /etc/security/limits.conf
```

For more information, see *Java reports an error saying there are too many open files*.

User Resource Limits

Recommended Settings for Production Installations

Cassandra requires greater user resource limits than the default settings. Add the following entries to your `/etc/security/limits.conf` file:

```
* soft nfile 32768
* hard nfile 32768
root soft nfile 32768
root hard nfile 32768
* soft memlock unlimited
* hard memlock unlimited
root soft memlock unlimited
root hard memlock unlimited
* soft as unlimited
* hard as unlimited
root soft as unlimited
root hard as unlimited
```

In addition, you may need to be run the following command:

```
sysctl -w vm.max_map_count=131072
```

The command enables more mapping. It is not in the `limits.conf` file.

On CentOS, RHEL, OEL Sysems, change the system limits from 1024 to 10240 in `/etc/security/limits.d/90-nproc.conf` and then start a new shell for these changes to take effect.

```
* soft nproc 10240
```

For more information, see [Insufficient user resource limits errors](#).

Disable Swap

Disable swap entirely. This prevents the Java Virtual Machine (JVM) from responding poorly because it is buried in swap and ensures that the OS OutOfMemory (OOM) killer does not kill Cassandra.

```
sudo swapoff --all
```

For more information, see [Nodes seem to freeze after some period of time](#).

Synchronize Clocks

The clocks on all nodes should be synchronized using NTP (Network Time Protocol).

This is required because columns are only overwritten if the timestamp in the new version of the column is more recent than the existing column.

Optimum blockdev --setra Settings for RAID

Typically, a setra of 512 is recommended, especially on Amazon EC2 RAID0 devices.

Check to ensure setra is not set to 65536:

```
sudo blockdev --report /dev/<device>
```

To set setra:

```
sudo blockdev --setra 512 /dev/<device>
```

Java Virtual Machine

Installing the JRE and JNA

The latest 64-bit version of Java 6 is recommended, not the OpenJDK. At a minimum, use JRE 1.6.0_32. Java 7 is not recommended. See *Installing the JRE and JNA*.

Java Native Access

Java Native Access (JNA) is required for production installations.

Installing the JRE and JNA

Cassandra is Java program and requires the Java Runtime Environment (JRE). The latest 64-bit version of Java 6 is recommended. At a minimum, use JRE 1.6.0_32. Java 7 is not recommended. Java Native Access (JNA) is needed for production installations.

Installing Oracle JRE

Select one of the following:

- *Installing the JRE on RHEL or CentOS Systems*
- *Installing the JRE on Debian or Ubuntu Systems*
- *Installing the JRE on SUSE Systems*

Note

After installing the JRE, you may need to set JAVA_HOME:

```
export JAVA_HOME=<path_to_java>
```

Installing the JRE on RHEL or CentOS Systems

You must configure your operating system to use the Oracle JRE, not OpenJDK.

1. Check which version of the JRE your system is using:

```
java -version
```

2. If necessary, go to [Oracle Java SE Downloads](#), accept the license agreement, and download the Linux x64-RPM Installer or Linux x86-RPM Installer (depending on your platform).

Note

If installing the JRE in a cloud environment, accept the license agreement, download the installer to your local client, and then use scp (secure copy) to transfer the file to your cloud machines.

3. Go to the directory where you downloaded the JRE package, and change the permissions so the file is executable:

```
$ chmod a+x jre-6u43-linux-x64-rpm.bin
```

4. Extract and run the RPM file. For example:

```
$ sudo ./jre-6u43-linux-x64-rpm.bin
```

The RPM installs the JRE into `/usr/java/`.

5. Configure your system so that it is using the Oracle JRE instead of the OpenJDK JRE. Use the `alternatives` command to add a symbolic link to the Oracle JRE installation. For example:

```
$ sudo alternatives --install /usr/bin/java java /usr/java/jre1.6.0_43/bin/java 20000
```

6. Make sure your system is now using the correct JRE. For example:

```
$ java -version
java version "1.6.0_43"
Java(TM) SE Runtime Environment (build 1.6.0_43-b05)
Java HotSpot(TM) 64-Bit Server VM (build 20.7-b02, mixed mode)
```

7. If the OpenJDK JRE is still being used, use the `alternatives` command to switch it. For example:

```
$ sudo alternatives --config java
There are 2 programs which provide 'java'.
```

Selection	Command
1	/usr/lib/jvm/jre-1.6.0-openjdk.x86_64/bin/java
*+ 2	/usr/java/jre1.6.0_43/bin/java

```
Enter to keep the current selection[+], or type selection number: 2
```

Installing the JRE on Debian or Ubuntu Systems

The Oracle Java Runtime Environment (JRE) has been removed from the official software repositories of Ubuntu and only provides a binary (.bin) version. You can get the JRE from the [Java SE Downloads](#).

1. Check which version of the JRE your system is using:

```
java -version
```

2. If necessary, download the appropriate version of the JRE, such as `jre-6u43-linux-x64.bin`, for your system and place it in `/usr/java/latest`.

Note

If installing the JRE in a cloud environment, accept the license agreement, download the installer to your local client, and then use `scp` (secure copy) to transfer the file to your cloud machines.

3. Make the file executable:

```
sudo chmod a+x /usr/java/latest/jre-6u43-linux-x64.bin
```

4. Go to the new folder:

```
cd /usr/java/latest
```

5. Execute the file:

```
sudo ./jre-6u43-linux-x64.bin
```

6. Tell the system that there's a new Java version available:

```
sudo update-alternatives --install "/usr/bin/java" "java" "/usr/java/latest/jre1.6.0_43/bin/java" 1
```

Note

If updating from a previous version that was removed manually, execute the above command twice, because you'll get an error message the first time.

7. Set the new JRE as the default:

```
sudo update-alternatives --set java /usr/java/latest/jre1.6.0_43/bin/java
```

8. Make sure your system is now using the correct JRE:

```
$ java -version

java version "1.6.0_43"
Java(TM) SE Runtime Environment (build 1.6.0_43-b05)
Java HotSpot(TM) 64-Bit Server VM (build 20.23-b02, mixed mode)
```

Installing the JRE on SUSE Systems

You must configure your operating system to use the Oracle JRE.

1. Check which version of the JRE your system is using:

```
java -version
```

2. If necessary, go to [Oracle Java SE Downloads](#), accept the license agreement, and download the Linux x64-RPM installer.

Note

If installing the JRE in a cloud environment, accept the license agreement, download the installer to your local client, and then use scp (secure copy) to transfer the file to your cloud machines.

3. Go to the directory where you downloaded the JRE package, and change the permissions so the file is executable:

```
$ chmod a+x jre-6u43-linux-x64-rpm.bin
```

4. Extract and run the RPM file. For example:

```
$ sudo ./jre-6u43-linux-x64-rpm.bin
```

The RPM installs the JRE into `/usr/java/`.

5. Make sure your system is now using the correct JRE. For example:

```
$ java -version
java version "1.6.0_43"
Java(TM) SE Runtime Environment (build 1.6.0_43-b05)
Java HotSpot(TM) 64-Bit Server VM (build 20.23-b02, mixed mode)
```

Installing JNA

Java Native Access (JNA) is required for production installations. Installing JNA can improve Cassandra memory usage. When installed and configured, Linux does not swap out the JVM, and thus avoids related performance issues.

Debian or Ubuntu Systems

```
$ sudo apt-get install libjna-java
```

Ubuntu 10.04 LTS

For Ubuntu 10.04 LTS, you need to update to JNA 3.4.

Installing a Cassandra Cluster on Amazon EC2

1. Download the `jna.jar` from <https://github.com/twall/jna>.
2. Remove older versions of the JNA from the `/usr/share/java/` directory.
3. Place the new `jna.jar` file in `/usr/share/java/` directory.
4. Create a symbolic link to the file:

```
ln -s /usr/share/java/jna.jar <install_location>/lib
```

RHEL or CentOS Systems

```
# yum install jna
```

SUSE Systems

```
# curl -o jna.jar -L https://github.com/twall/jna/blob/3.4.1/dist/jna.jar?raw=true
# curl -o platform.jar -L https://github.com/twall/jna/blob/3.4.1/dist/platform.jar?raw=true
# mv jna.jar /usr/share/java
# mv platform.jar /usr/share/java
```

Tarball Installations

1. Download `jna.jar` from <https://github.com/twall/jna>.
2. Add `jna.jar` to `<install_location>/lib/` (or place it in the `CLASSPATH`).
3. Add the following lines in the `/etc/security/limits.conf` file for the user/group that runs Cassandra:

```
$USER soft memlock unlimited
$USER hard memlock unlimited
```

Installing a Cassandra Cluster on Amazon EC2

For instructions on installing the DataStax AMI (Amazon Machine Image), see the latest [AMI documentation](#).

Expanding a Cassandra AMI cluster

This section contains instructions for expanding a cluster that uses the DataStax Community Edition AMI (Amazon Machine Image) prior to Cassandra 1.2. If your AMI uses Cassandra 1.2, see the [1.2 instructions](#).

As a best practice when expanding a cluster, DataStax recommends doubling the size of the cluster size with each expansion. You should calculate the node number and token for each node based on the final ring size. For example, suppose that you ultimately want a 12 node cluster, starting with three node cluster. The initial three nodes are 0, 4, and 8. For the first expansion, you double the number of nodes to six by adding nodes, 2, 6, and 10. For the final expansion, you add six more nodes: 1, 3, 5, 7, 9, and 11. Using the [tokengentool](#), calculate tokens for 12 nodes and enter the corresponding values in the `initial_token` property in the `cassandra.yaml` file. For more information, see [Adding Capacity to an Existing Cluster](#).

Steps to expand a Cassandra AMI cluster

Installing a Cassandra Cluster on Amazon EC2

1. In the AWS Management Console, create the number of nodes you need in another cluster with a temporary name. See *Installing a Cassandra Cluster on Amazon EC2*.

The temporary name prevents the new nodes from joining the cluster with the wrong tokens.

2. After the nodes are initialized, login to each node and stop the service:

```
sudo service cassandra stop
```

3. Clear the data in each node:

- a. Check the `cassandra.yaml` for the location of the data directories:

```
data_file_directories:
  - /raid0/cassandra/data
```

- b. Remove the data directories:

```
sudo rm -rf /raid0/cassandra/*
```

You must clear the data because new nodes have existing data from the initial start with the temporary cluster name and token.

4. For each node, change the `/etc/dse/cassandra/cassandra.yaml` settings to match the `cluster_name` and `- seeds` list of the other cluster. For example:

```
cluster_name: 'NameOfExistingCluster'
...
initial_token: 28356863910078205288614550619314017621
...
seed_provider:
  - class_name: org.apache.cassandra.locator.SimpleSeedProvider
    parameters:
      - seeds: "110.82.155.0,110.82.155.3"
```

5. Assign the correct `initial_token` to each node; it determines the node's placement within the ring.

6. If adding nodes to an **existing** data center:

- a. Set `auto_bootstrap: true`. (If `auto_bootstrap` is not in the `cassandra.yaml` file, it automatically defaults to `true`.)
- b. Go to step 8.

7. If adding nodes to a **new** data center:

- a. Set `auto_bootstrap: false`.
- b. After all new nodes have joined the ring, do either of the following:
 - Run a rolling *nodetool repair* on all new nodes in the cluster.
 - Run a rolling `nodetool repair -pr` on all nodes in the cluster.

For example:

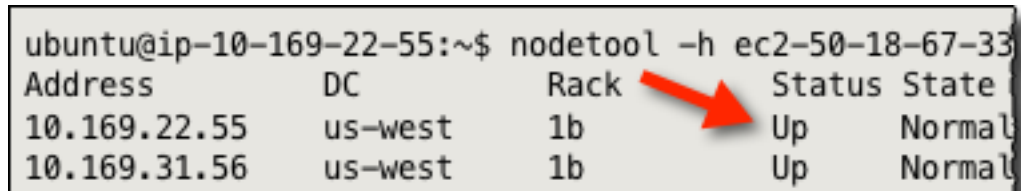
```
$ nodetool repair -h 10.46.123.12 keyspace_name -pr
```

8. Start each node in **two** minute intervals:

```
$ sudo service cassandra start
```


9. Verify that each node has finished joining the ring:

```
nodetool -h <hostname> ring
```



Address	DC	Rack	Status	State
10.169.22.55	us-west	1b	Up	Normal
10.169.31.56	us-west	1b	Up	Normal

Each node should show **Up** not **Joining**.

Upgrading Cassandra

This section describes how to upgrade an earlier version of Cassandra or DataStax Community Edition to Cassandra 1.1. This section contains the following topics:

- *Best Practices for Upgrading Cassandra*
- *Upgrading to Cassandra 1.1.9*
- *Upgrade Steps for Binary Tarball and Packaged Releases Installations*
- *New and Changed Features*
- *Upgrading Between Minor Releases of Cassandra 1.1.x*
- *New Parameters between 1.0 and 1.1*

Best Practices for Upgrading Cassandra

The following steps are recommended before upgrading Cassandra:

- Take a *snapshot* before the upgrade. This allows you to rollback to the previous version if necessary. Cassandra is able to read data files created by the previous version, but the inverse is not always true.
Taking a snapshot is fast, especially if you have JNA installed, and takes effectively zero disk space until you start compacting the live data files again.
- Check <https://github.com/apache/cassandra/blob/trunk/NEWS.txt> for any new information about upgrading.
- For a list of fixes and new features, see <https://github.com/apache/cassandra/blob/trunk/CHANGES.txt>.

Upgrading to Cassandra 1.1.9

If you are upgrading to Cassandra 1.1.9 from a version earlier than 1.1.7, all nodes must be upgraded before any streaming can take place. Until you upgrade all nodes, you cannot add version 1.1.7 nodes or later to a 1.1.7 or earlier cluster.

Upgrade Steps for Binary Tarball and Packaged Releases Installations

Upgrading from version 0.8 or later can be done with a rolling restart, one node at a time. You do not need to bring down the whole cluster at once.

To upgrade a Binary Tarball Installation

1. Save the `cassandra.yaml` file from the old installation to a safe place.
2. On each node, download and unpack the binary tarball package from the [downloads section](#) of the Cassandra website.
3. In the new installation, open the `cassandra.yaml` for writing.
4. In the old installation, open the `cassandra.yaml`.
5. Diff the new and old `cassandra.yaml` files.
6. Merge the diffs by hand from the old file into the new one.
7. Follow steps for [completing the upgrade](#).

To upgrade a RHEL or CentOS Installation

1. On each of your Cassandra nodes, run `sudo yum install apache-cassandra1`. The installer creates the file `cassandra.yaml.rpmnew` in `/etc/cassandra/default.conf/`.
2. Open the old and new `cassandra.yaml` files and diff them.
3. Merge the diffs by hand from the old file into the new one. Save the file as `cassandra.yaml`.
4. Follow steps for [completing the upgrade](#).

To Upgrade a Debian or Ubuntu Installation

1. Save the `cassandra.yaml` file from the old installation to a safe place.
2. On each of your Cassandra nodes, run `sudo apt-get install cassandra1`.
3. Open the old and new `cassandra.yaml` files and diff them.
4. Merge the diffs by hand from the old file into the new one.
5. Follow steps for [completing the upgrade](#).

Completing the Upgrade

To complete the upgrade, perform the following steps:

1. Account for [New Parameters between 1.0 and 1.1](#) in `cassandra.yaml`.
2. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
3. Run `nodetool drain` before shutting down the existing Cassandra service. This will prevent overcounts of counter data, and will also speed up restart post-upgrade.
4. Stop the old Cassandra process, then start the new binary process.
5. Monitor the log files for any issues.
6. If you are upgrading from Cassandra 1.1.3 or earlier to Cassandra 1.1.5 or later, skip steps 7 and 8 of this procedure and go to [Completing the upgrade from Cassandra 1.1.3 or earlier to Cassandra 1.1.5 or later](#).
7. After upgrading and restarting all Cassandra processes, restart client applications.
8. After upgrading, run `nodetool upgradesstables` against each node before running repair, moving nodes, or adding new ones. If you are using Cassandra 1.0.3 and earlier, use `nodetool scrub` instead of `nodetool upgradesstables`.

Completing the upgrade from Cassandra 1.1.3 or earlier to Cassandra 1.1.5 or later

If you created column families having the *CQL compaction_strategy_class* storage option set to `LeveledCompactionStrategy`, you need to scrub the SSTables that store those column families.

First, upgrade all nodes to the latest Cassandra version, according to the platform-specific instructions presented earlier in this document. Next, complete steps 1-5 of *Completing the Upgrade*. At this point, all nodes are upgraded and started. Finally, follow these steps to scrub the affected SSTables:

To scrub SSTables:

1. Shut down the nodes, one-at-a-time.
2. On each offline node, run the `sstablescrib` utility, which is located in `<install directory>/bin` (tarball distributions) or in `/usr/bin` (packaged distributions). Help for `sstablescrib` is:

```
usage: sstablescrib [options] <keyspace> <column_family>
--
Scrub the sstable for the provided column family.
--
Options are:
--debug display stack traces
-h,--help display this help message
-m,--manifest-check only check and repair the leveled manifest, without
actually scrubbing the sstables
-v,--verbose verbose output
```

For example, on a tarball installation:

```
cd <install directory>/bin
./sstablescrib mykeyspace mycolumnfamily
```

3. Restart each node and client applications, one node at-a-time.

If you do not scrub the affected SSTables, you might encounter the following error during compactions on column families using `LeveledCompactionStrategy`:

```
ERROR [CompactionExecutor:150] 2012-07-05 04:26:15,570 AbstractCassandraDaemon.java (line 134)
Exception in thread Thread[CompactionExecutor:150,1,main]
java.lang.AssertionError
at org.apache.cassandra.db.compaction.LeveledManifest.promote
(LeveledManifest.java:214)
```

Upgrading Between Minor Releases of Cassandra 1.1.x

The upgrade procedure between minor releases of Cassandra 1.1.x is identical to the upgrade procedure between major releases with one exception: Do *not* perform the last step of *Completing the Upgrade* to run `nodetool upgradesstables` or `nodetool scrub` after upgrading.

New and Changed Features

The following list provides information about new and changed features in Cassandra 1.1. Also see *New Parameters between 1.0 and 1.1*.

- Compression is enabled by default on newly created column families and unchanged for column families created prior to upgrading.
- If running a multi data-center, you should **upgrade** to the latest 1.0.x (or 0.8.x) release before upgrading to this version. Versions 0.8.8 and 1.0.3-1.0.5 generate cross-data center forwarding that is incompatible with 1.1.

Cross-data center forwarding means optimizing cross-data center replication. If DC1 needs to replicate a write to three replicas in DC2, only one message is sent across data centers; one node in DC2 forwards the message to the two other replicas, instead of sending three message across data centers.

- `EACH_QUORUM` `ConsistencyLevel` is only supported for writes and now throws an `InvalidRequestException` when used for reads. (Previous versions would silently perform a `LOCAL_QUORUM` read.)
- `ANY` `ConsistencyLevel` is supported only for writes and now throw an `InvalidRequestException` when used for reads. (Previous versions would silently perform a `ONE` read for range queries; `single-row` and `multiget` reads already rejected `ANY`.)
- The largest mutation batch accepted by the `commitlog` is now 128MB. (In practice, batches larger than ~10MB always caused poor performance due to load volatility and GC promotion failures.) Larger batches will continue to be accepted but are not durable. Consider setting `durable_writes=false` if you really want to use such large batches.
- Make sure that global settings: `key_cache_{size_in_mb, save_period}` and `row_cache_{size_in_mb, save_period}` in `conf/cassandra.yaml` configuration file are used instead of per-ColumnFamily options.
- JMX methods no longer return custom Cassandra objects. JMX methods now return standard Maps, Lists, and so on.
- Hadoop input and output details are now separated. If you were previously using methods such as `getRpcPort` you now need to use `getInputRpcPort` or `getOutputRpcPort`, depending on the circumstance.
- CQL changes: Prior to Cassandra 1.1, you could use the CQL 2 keyword, `KEY`, instead of using the actual name of the primary key column in some select statements. In Cassandra 1.1 and later, you must use the name of the primary key.
- The `sliced_buffer_size_in_kb` option has been removed from the `cassandra.yaml` configuration file (this option was a no-op since 1.0).

New Parameters between 1.0 and 1.1

This table lists parameters in the `cassandra.yaml` configuration files that have changed between 1.0 and 1.1. See the [cassandra.yaml reference](#) for details on these parameters.

Option	Default Value
1.1 Release	
<code>key_cache_size_in_mb</code>	empty
<code>key_cache_save_period</code>	14400 (4 hours)
<code>row_cache_size_in_mb</code>	0 (disabled)
<code>row_cache_save_period</code>	0 (disabled)
1.0 Release (Column Family Attributes)	
<code>key_cache_size</code>	2MB (ignored in 1.1)
<code>key_cache_save_period_in_seconds</code>	N/A
<code>row_cache_size</code>	0 (ignored in 1.1)
<code>row_cache_save_period_in_seconds</code>	N/A

Initializing a Cassandra Cluster

Initializing a Cassandra cluster involves configuring each node so that it is prepared to join the cluster. After each node is configured, start each node sequentially beginning with the seed node(s). For considerations on choosing the right configuration options for your environment, see *Planning a Cassandra Cluster Deployment*.

Initializing a Multiple Node Cluster in a Single Data Center

In this scenario, data replication is distributed across a single data center.

Data replicates across the data centers automatically and transparently; no ETL work is necessary to move data between different systems or servers. You can configure the number of *copies of the data* in each data center and Cassandra handles the rest, replicating the data for you. To configure a multiple data center cluster, see *Initializing Multiple Data Center Clusters on Cassandra*.

Note

In Cassandra, the term data center is a grouping of nodes. Data center is synonymous with replication group, that is, a grouping of nodes configured together for replication purposes. The data replication protects against hardware failure and other problems that cause data loss in a single cluster.

Prerequisites

To correctly configure a multi-node cluster, requires the following:

- Cassandra is installed on each node.
- The total number of nodes in the cluster.
- A name for the cluster.
- The IP addresses of each node in the cluster.
- Which nodes will serve as the seed nodes. (Cassandra nodes use this host list to find each other and learn the topology of the ring.)
- The *snitch* you plan to use.
- If the nodes are behind a firewall, make sure you know what ports you need to open. See *Configuring Firewall Port Access*.
- Other configuration settings you may need are described in *Choosing Node Configuration Options* and *Node and Cluster Configuration (cassandra.yaml)*.

This information is used to configure the *Node and Cluster Initialization Properties* in the *cassandra.yaml* configuration file on each node in the cluster. Each node should be correctly configured before starting up the cluster.

Configuration Example

This example describes installing a six node cluster spanning two racks in a single data center.

You set properties for each node in the *cassandra.yaml* file. The location of this file depends on the type of installation; see *Cassandra Configuration Files Locations* or *DataStax Enterprise Configuration Files Locations*.

Note

After changing properties in the *cassandra.yaml* file, you must restart the node for the changes to take effect.

To configure a mixed-workload cluster:

Initializing a Cassandra Cluster

1. The nodes have the following IPs, and one node per rack will serve as a seed:

- node0 110.82.155.0 (seed1)
- node1 110.82.155.1
- node2 110.82.155.2
- node3 110.82.156.3 (seed2)
- node4 110.82.156.4
- node5 110.82.156.5

2. Calculate the token assignments using the *Token Generating Tool*.

Node	Token
node0	0
node1	28356863910078205288614550619314017621
node2	56713727820156410577229101238628035242
node3	85070591730234615865843651857942052864
node4	113427455640312821154458202477256070485
node5	141784319550391026443072753096570088106

3. If you have a firewall running on the nodes in your cluster, you must open certain ports to allow communication between the nodes. See *Configuring Firewall Port Access*.

4. Stop the nodes and clear the data.

- For packaged installs, run the following commands:

```
$ sudo service cassandra stop (stops the service)
```

```
$ sudo rm -rf /var/lib/cassandra/* (clears the data from the default directories)
```
- For binary installs, run the following commands from the install directory:

```
$ ps auwx | grep cassandra (finds the Cassandra Java process ID [PID])
```

```
$ sudo kill <pid> (stops the process)
```

```
$ sudo rm -rf /var/lib/cassandra/* (clears the data from the default directories)
```

5. Modify the following property settings in the `cassandra.yaml` file for each node:

Note

In the `- seeds` list property, include the internal IP addresses of each seed node.

node0

```
cluster_name: 'MyDemoCluster'
initial_token: 0
seed_provider:
  - class_name: org.apache.cassandra.locator.SimpleSeedProvider
    parameters:
      - seeds: "110.82.155.0,110.82.155.3"
listen_address: 110.82.155.0
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

node1 to node5

The properties for the rest of the nodes are the same as **Node0** except for the `initial_token` and `listen_address`:

node1

```
initial_token: 28356863910078205288614550619314017621
listen_address: 110.82.155.1
```

node2

```
initial_token: 56713727820156410577229101238628035242
listen_address: 110.82.155.2
```

node3

```
initial_token: 85070591730234615865843651857942052864
listen_address: 110.82.155.3
```

node4

```
initial_token: 113427455640312821154458202477256070485
listen_address: 110.82.155.4
```

node5

```
initial_token: 141784319550391026443072753096570088106
listen_address: 110.82.155.5
```

6. After you have installed and configured Cassandra on all nodes, start the seed nodes one at a time, and then start the rest of the nodes.

Note

If the node has restarted because of automatic restart, you must stop the node and clear the data directories, as described in above.

- Packaged installs: `sudo service cassandra start`
- Binary installs, run one of the following commands from the install directory:
`bin/cassandra` (starts in the background)
`bin/cassandra -f` (starts in the foreground)

7. Check that your ring is up and running:

- Packaged installs: `nodetool ring -h localhost`
- Binary installs:
`cd /<install_directory>`
`$ bin/nodetool ring -h localhost`

The ring status is displayed. This can give you an idea of the load balanced within the ring and if any nodes are down. If your cluster is not properly configured, different nodes may show a different ring; this is a good way to check that every node views the ring the same way.

Initializing Multiple Data Center Clusters on Cassandra

In this scenario, data replication can be distributed across multiple, geographically dispersed data centers, between different physical racks in a data center, or between public cloud providers and on-premise managed data centers.

Data replicates across the data centers automatically and transparently; no ETL work is necessary to move data between different systems or servers. You can configure the number of *copies of the data* in each data center and Cassandra handles the rest, replicating the data for you. To configure a multiple data center cluster, see *Initializing a Multiple Node Cluster in a Single Data Center*.

Prerequisites

To correctly configure a multi-node cluster with multiple data centers, requires:

- Cassandra is installed on each node.
- The total number of nodes in the cluster.
- A name for the cluster.
- The IP addresses of each node in the cluster.
- Which nodes will serve as the seed nodes. (Cassandra nodes use this host list to find each other and learn the topology of the ring.)
- The *snitch* you plan to use.
- If the nodes are behind a firewall, make sure you know what ports you need to open. See *Configuring Firewall Port Access*.
- Other configuration settings you may need are described in *Choosing Node Configuration Options* and *Node and Cluster Configuration*.

This information is used to configure the following properties on each node in the cluster:

- The *Node and Cluster Initialization Properties* in the *cassandra.yaml* file.
- Assigning the data center and rack names to the IP addresses of each node in the *cassandra-topology.properties* file.

Configuration Example

This example describes installing a six node cluster spanning two data centers.

You set properties for each node in the *cassandra.yaml* and *cassandra-topology.properties* files. The location of these files depends on the type of installation; see *Cassandra Configuration Files Locations* or *DataStax Enterprise Configuration Files Locations*.

Note

After changing properties in these files, you must restart the node for the changes to take effect.

To configure a cluster with multiple data centers:

1. Suppose you install Cassandra on these nodes:

```
10.168.66.41 (seed1)
10.176.43.66
10.168.247.41
10.176.170.59 (seed2)
10.169.61.170
10.169.30.138
```

2. Assign tokens so that data is evenly distributed within each data center or replication group by calculating the token assignments with the *Token Generating Tool* and then offset the tokens for the second data center:

Node	IP Address	Token	Offset	Data Center
node0	10.168.66.41	0	NA	DC1
node1	10.176.43.66	56713727820156410577229101238628035242	NA	DC1
node2	10.168.247.41	113427455640312821154458202477256070485	NA	DC1
node3	10.176.170.59	10	10	DC2
node4	10.169.61.170	56713727820156410577229101238628035252	10	DC2
node5	10.169.30.138	113427455640312821154458202477256070495	10	DC2

For more information, see *Generating Tokens*.

3. Stop the nodes and clear the data.

- For packaged installs, run the following commands:

```
$ sudo service cassandra stop (stops the service)
$ sudo rm -rf /var/lib/cassandra/* (clears the data from the default directories)
```

- For binary installs, run the following commands from the install directory:

```
$ ps aux | grep cassandra (finds the Cassandra Java process ID [PID])
$ sudo kill <pid> (stops the process)
$ sudo rm -rf /var/lib/cassandra/* (clears the data from the default directories)
```

4. Modify the following property settings in the `cassandra.yaml` file for each node:

- `endpoint_snitch` <name of snitch> - See [endpoint_snitch](#).
- `initial_token`: <token from previous step>
- `-seeds`: <internal IP_address of each seed node>
- `listen_address`: <localhost IP address>

node0:

```
end_point_snitch: org.apache.cassandra.locator.PropertyFileSnitch
initial_token: 0
seed_provider:
  - class_name: org.apache.cassandra.locator.SimpleSeedProvider
    parameters:
      - seeds: "10.168.66.41,10.176.170.59"
listen_address: 10.176.43.66
```

Note

You must include at least one seed node from each data center. It is a best practice to have at more than one seed node per data center and the seed list should be the same for each node.

node1 to node5

The properties for the rest of the nodes are the same as **Node0** except for the `initial_token` and `listen_address`:

5. Determine a naming convention for each data center and rack, for example: DC1, DC2 or 100, 200 and RAC1, RAC2 or R101, R102.
6. In the `cassandra-topology.properties` file, assign data center and rack names to the IP addresses of each node. For example:

```
# Cassandra Node IP=Data Center:Rack
10.168.66.41=DC1:RAC1
10.176.43.66=DC2:RAC1
10.168.247.41=DC1:RAC1
10.176.170.59=DC2:RAC1
10.169.61.170=DC1:RAC1
10.169.30.138=DC2:RAC1
```

7. Also, in the `cassandra-topologies.properties` file, assign a default data center name and rack name for unknown nodes.

```
# default for unknown nodes
default=DC1:RAC1
```

8. After you have installed and configured Cassandra on all nodes, start the seed nodes one at a time, and then start the rest of the nodes.

Note

If the node has restarted because of automatic restart, you must stop the node and clear the data directories, as described above.

- Packaged installs: `sudo service cassandra start`
- Binary installs, run one of the following commands from the install directory:
`bin/cassandra` (starts in the background)
`bin/cassandra -f` (starts in the foreground)

9. Check that your ring is up and running:

- Packaged installs: `nodetool ring -h localhost`
- Binary installs:
`cd /<install_directory>`
`$ bin/nodetool ring -h localhost`

The ring status is displayed. This can give you an idea of the load balanced within the ring and if any nodes are down. If your cluster is not properly configured, different nodes may show a different ring; this is a good way to check that every node views the ring the same way.

Balancing the Data Center Nodes

When you deploy a Cassandra cluster, you need to use the *partitioner* to distribute roughly an equal amount of data to nodes. You also use identifiers for each data center in a formula to calculate tokens that balance nodes within a data center (DC). For example, assign each DC a numerical name that is a multiple of 100. Then for each DC, determine the tokens as follows: $\text{token} = (2^{127} / \text{num_nodes_in_dc} * n + \text{DC_ID})$ where n is the node for which the token is being calculated and DC_ID is the numerical name.

More Information About Configuring Data Centers

Links to more information about configuring a data center:

- [Configuring nodes](#)
- [Choosing keyspace replication options](#)
- [Replication in a physical or virtual data center](#)

Generating Tokens

Tokens assign a range of data to a particular node within a data center.

When you start a Cassandra cluster, data is distributed across the nodes in the cluster based on the row key using a *partitioner*. You must assign each node in a cluster a token and that token determines the node's position in the ring and its range of data. The tokens assigned to your nodes need to be distributed throughout the entire possible range of tokens (0 to $2^{127} - 1$). Each node is responsible for the region of the ring between itself (inclusive) and its predecessor (exclusive). To illustrate using a simple example, if the range of possible tokens was 0 to 100 and you had four nodes, the tokens for your nodes should be 0, 25, 50, and 75. This approach ensures that each node is responsible for an equal range of data. When using more than one data center, each data center should be partitioned as if it were its own distinct ring.

Note

Each node in the cluster must be assigned a token before it is started for the first time. The token is set with the *initial_token* property in the *cassandra.yaml* configuration file.

Token Generating Tool

Cassandra includes a tool for generating tokens using the maximum possible range (0 to $2^{127} - 1$) for use with the *RandomPartitioner*.

Usage

- **Packaged installs:** `token-generator <nodes_in_DC1> <nodes_in_DC2> ...`
- **Binary** **installs:**
`<install_location>/tools/bin/token-generator <nodes_in_DC1> <nodes_in_DC2> ...`
- **Interactive Mode:** Use `token-generator` without options and messages will guide you through the process.

The available options are:

Long Option	
Short Option	Description
<code>--help</code> <code>-h</code>	Show help.
<code>--ringrange <RINGRANGE></code>	Specify a numeric maximum token value for your ring, if different from the default value of $2^{127} - 1$.
<code>--graph</code>	Displays a rendering of the generated tokens as line segments in a circle, colored according to data center.
<code>--nts</code> <code>-n</code>	Optimizes multiple cluster distribution for <i>NetworkTopologyStrategy</i> (default).
<code>--onts</code> <code>-o</code>	Optimizes multiple cluster distribution for the <i>OldNetworkTopologyStrategy</i> .
<code>--test</code> <code>-o</code>	Run in test mode. Opens Firefox and displays an HTML file that shows various ring arrangements.

Examples

- Generate tokens for nodes in a single data center:

```
./tools/bin/token-generator 4
```

```
Node #1: 0
Node #2: 42535295865117307932921825928971026432
Node #3: 85070591730234615865843651857942052864
Node #4: 127605887595351923798765477786913079296
```

Generating Tokens

- Generate tokens for multiple data centers using NetworkTopologyStrategy (default):

```
./tools/bin/token-generator 4 4
```

DC #1:

```
Node #1: 0
Node #2: 42535295865117307932921825928971026432
Node #3: 85070591730234615865843651857942052864
Node #4: 127605887595351923798765477786913079296
```

DC #2:

```
Node #1: 169417178424467235000914166253263322299
Node #2: 41811290829115311202148688466350243003
Node #3: 84346586694232619135070514395321269435
Node #4: 126881882559349927067992340324292295867
```

Replica placement is independent within each data center.

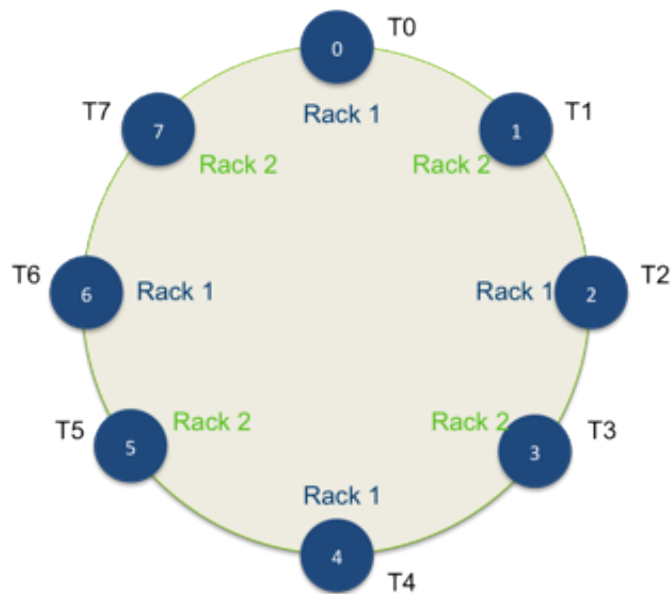
- Generate tokens for multiple racks in a single data center:

```
./tools/bin/token-generator 8
```

DC #1:

```
Node #1: 0
Node #2: 21267647932558653966460912964485513216
Node #3: 42535295865117307932921825928971026432
Node #4: 63802943797675961899382738893456539648
Node #5: 85070591730234615865843651857942052864
Node #6: 106338239662793269832304564822427566080
Node #7: 127605887595351923798765477786913079296
Node #8: 148873535527910577765226390751398592512
```

As a best practice, each rack should have the same number of nodes. This allows you to alternate the rack assignments: rack1, rack2, rack3, rack1, rack2, rack3, and so on:

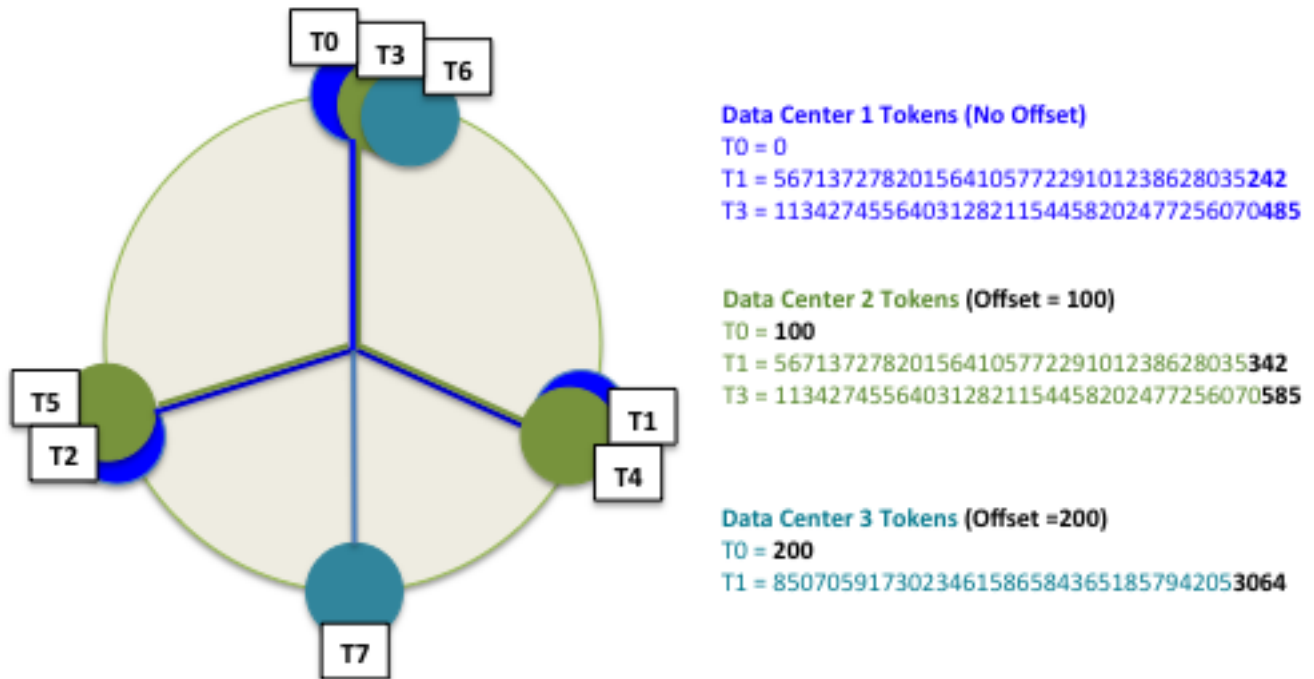


Assign tokens to nodes in alternating racks

Token Assignments when Adding Nodes

When adding nodes to a cluster, you must avoid token collisions. You can do this by offsetting the token values, which allows room for the new nodes.

The following graphic shows an example using an offset of +100:



Note

It is more important that the nodes within each data center manage an equal amount of data than the distribution of the nodes within the cluster. See *balancing the load*.

Managing a Cassandra Cluster

This section discusses the following routine management and maintenance tasks:

- *Running Routine Node Repair*
- *Adding Capacity to an Existing Cluster*
- *Changing the Replication Factor*
- *Replacing a Dead Node*

Running Routine Node Repair

The *nodetool repair* command repairs inconsistencies across all of the replicas for a given range of data. Repair should be run at regular intervals during normal operations, as well as during node recovery scenarios, such as bringing a node back into the cluster after a failure.

Unless Cassandra applications perform no deletes at all, production clusters require periodic, scheduled repairs on all nodes. The hard requirement for repair frequency is the value of *gc_grace_seconds*. *Make sure you run a repair operation at least once on each node within this time period.* Following this important guideline ensures that deletes are properly handled in the cluster.

Note

Repair requires heavy disk and CPU consumption. Use caution when running node repair on more than one node at a time. Be sure to schedule regular repair operations during low-usage hours.

In systems that seldom delete or overwrite data, it is possible to raise the value of *gc_grace_seconds* at a minimal cost in extra disk space used. This allows wider intervals for scheduling repair operations with the *nodetool* utility.

Adding Capacity to an Existing Cluster

Cassandra allows you to add capacity to a cluster by introducing new nodes to the cluster in stages and by adding an entire data center. When a new node joins an existing cluster, it needs to know:

- Its position in the ring and the range of data it is responsible for, which is assigned by the *initial_token* and the *partitioner*.
- The *seed* nodes it should contact to learn about the cluster and establish the *gossip* process.
- The name of the cluster it is joining and how the node should be addressed within the cluster.
- Any other non-default *settings* made to *cassandra.yaml* on your existing cluster should also be made on the new node before it is started.

Calculating Tokens For the New Nodes

When you add a node to a cluster, it needs to know its position in the ring. There are a few different approaches for calculating tokens for new nodes:

- **Add capacity by doubling the cluster size.** Adding capacity by doubling (or tripling or quadrupling) the number of nodes is less complicated when assigning tokens. Existing nodes can keep their existing token assignments, and new nodes are assigned tokens that bisect (or trisect) the existing token ranges. For example, when you generate tokens for 6 nodes, three of the generated token values will be the same as if you generated for 3 nodes. You just need to determine the token values that are already in use, and assign the newly calculated token values to the newly added nodes.

- **Recalculate new tokens for all nodes and move nodes around the ring.** If you need to increase capacity by a non-uniform number of nodes, you must recalculate tokens for the entire cluster, and then use *nodetool move* to assign the new tokens to the existing nodes. After all nodes are restarted with their new token assignments, run a *nodetool cleanup* to remove unused keys on all nodes. These operations are resource intensive and should be planned for low-usage times.
- **Add one node at a time and leave the `initial_token` property empty.** When the *initial_token* is empty, Cassandra splits the token range of the heaviest loaded node and places the new node into the ring at that position. This approach will probably not result in a perfectly balanced ring, but it will alleviate hot spots.

Adding Nodes to a Cluster

1. Install Cassandra on the new nodes, but do not start them.
2. Calculate the tokens for the nodes based on the expansion strategy you are using using the *Token Generating Tool*. You can skip this step if you want the new nodes to automatically pick a token range when joining the cluster.
3. Set the *Node and Cluster Initialization Properties* for the new nodes.
4. Set the *initial_token* according to your token calculations (or leave it unset if you want the new nodes to automatically pick a token range when joining the cluster).
5. Start Cassandra on each new node. Allow **two** minutes between node initializations. You can monitor the startup and data streaming process using *nodetool netstats*.
6. After the new nodes are fully bootstrapped, assign the new *initial_token* property value to the nodes that required new tokens, and then run `nodetool move <new_token>`, one node at a time.
7. After all nodes have their new tokens assigned, run *nodetool cleanup* on each of the existing nodes to remove the keys no longer belonging to those nodes. Wait for cleanup to complete on one node before doing the next. Cleanup may be safely postponed for low-usage hours.

Adding a Data Center to a Cluster

The following steps describe adding a data center to an existing cluster. Before starting this procedure, please read the guidelines in *Adding Capacity to an Existing Cluster* above.

1. Ensure that you are using *NetworkTopologyStrategy* for all of your custom keyspaces.
2. For each new node, edit the *configuration properties* in the *cassandra.yaml* file:
 - Set `auto_bootstrap` to `False`.
 - Set the `initial_token`. Be sure to offset the tokens in the new data center, see *Generating Tokens*.
 - Set the `cluster_name`.
 - Set any other non-default settings.
 - Set the *seed lists*. Every node in the cluster must have the same list of seeds and include at least one node from each data center. Typically one to three seeds are used per data center.
3. If using the *PropertyFileSnitch*, update the `cassandra-topology.properties` file on all servers to include the new nodes. You do not need to restart.

The location of this file depends on the type of installation; see *Cassandra Configuration Files Locations* or *DataStax Enterprise Configuration Files Locations*.
4. Ensure that your client does not autodetect the new nodes so that they aren't contacted by the client until explicitly directed. For example in Hector, set `hostConfig.setAutoDiscoverHosts(false)` ;
5. If using a QUORUM *consistency level* for reads or writes, check the `LOCAL_QUORUM` or `EACH_QUORUM` consistency level to see if the level meets your requirements for multiple data centers.

6. *Start the new nodes.*
7. After all nodes are running in the cluster:
 - a. Change the *strategy_options* for your keyspace to the desired replication factor for the new data center using CLI. For example: `strategy_options={DC1:2,DC2:2}`
 - b. Run *nodetool rebuild* on each node in the new data center.

Changing the Replication Factor

Increasing the replication factor increases the total number of copies of each row of data. You update the number of replicas by updating the keyspace using the CLI.

1. Update each keyspace in the cluster and change its replication strategy options. For example, to update the number of replicas in Cassandra CLI when using SimpleStrategy replica placement strategy:

```
[default@unknown] UPDATE KEYSPACE demo
WITH strategy_options = {replication_factor:3};
```

Or if using NetworkTopologyStrategy:

```
[default@unknown] UPDATE KEYSPACE demo
WITH strategy_options = {datacenter1:6,datacenter2:6};
```

2. On each node in the cluster, run *nodetool repair* for each keyspace that was updated. Wait until repair completes on a node before moving to the next node.

Replacing a Dead Node

To replace a node that has died (due to hardware failure, for example), bring up a new node using the token of the dead node as described in the next procedure. This token used must already be part of the ring.

To replace a dead node:

1. Confirm that the node is dead using the *nodetool ring* command on any live node in the cluster.

Trying to replace a node using a token from a live node results in an exception. The *nodetool ring* command shows a *Down* status for the token value of the dead node:

```
$ nodetool ring -h localhost
```

Address	DC	Rack	Status	State	Load	Owns	Token
10.46.123.11	datacenter1	rack1	Up	Normal	179.58 KB	16.67%	0
10.46.123.12	datacenter1	rack1	Down	Normal	315.21 KB	16.67%	28356863910078205288614550619314017621
10.46.123.13	datacenter1	rack1	Up	Normal	267.71 KB	16.67%	56713727820156410577229101238628035242
10.46.123.14	datacenter1	rack1	Up	Normal	315.21 KB	16.67%	85070591730234615865843651857942052863
10.46.123.15	datacenter1	rack1	Up	Normal	292.36 KB	16.67%	113427455640312821154458202477256070485
10.46.123.16	datacenter1	rack1	Up	Normal	300.02 KB	16.67%	141784319550391026443072753096570088106

2. *Install Cassandra* on the replacement node.
3. Remove any preexisting Cassandra data on the replacement node:

```
sudo rm -rf /var/lib/cassandra/*
```

4. Set `auto_bootstrap: true`. (If `auto_bootstrap` is not in the *cassandra.yaml* file, it automatically defaults to `true`.)

Changing the Replication Factor

5. Set the `initial_token` in the `cassandra.yaml` file to the value of the dead node's token -1. Using the value from the above graphic, this is 28356863910078205288614550619314017621-1:

```
initial_token: 28356863910078205288614550619314017620
```

6. Configure any non-default settings in the node's `cassandra.yaml` to match your existing cluster.
7. *Start the new node.*
8. After the new node has finished bootstrapping, check that it is marked up using the `nodetool ring` command.
9. Run *nodetool repair* on each keyspace to ensure the node is fully consistent. For example:

```
$ nodetool repair -h 10.46.123.12 keyspace_name
```

10. Remove the dead node.

Backing Up and Restoring Data

Cassandra backs up data by taking a snapshot of all on-disk data files (SSTable files) stored in the data directory. You can take a snapshot on all keyspaces, a single keyspace, or a single column family while the system is online. However, to restore a snapshot, you must take the nodes offline.

Using a parallel ssh tool (such as `pssh`), you can snapshot an entire cluster. This provides an *eventually consistent* backup. Although no one node is guaranteed to be consistent with its replica nodes at the time a snapshot is taken, a restored snapshot resumes consistency using Cassandra's built-in consistency mechanisms.

After a system-wide snapshot is performed, you can enable incremental backups on each node to backup data that has changed since the last snapshot: each time an SSTable is flushed, a hard link is copied into a `/backups` subdirectory of the data directory (provided JNA is enabled).

Note

If JNA is enabled, snapshots are performed by hard links. If not enabled, I/O activity increases as the files are copied from one location to another, which significantly reduces efficiency.

Taking a Snapshot

Snapshots are taken per node using the `nodetool snapshot` command. To take a global snapshot, run the `nodetool snapshot` command using a parallel ssh utility, such as `pssh`.

A snapshot first flushes all in-memory writes to disk, then makes a hard link of the SSTable files for each keyspace. By default the snapshot files are stored in the `/var/lib/cassandra/data/<keyspace_name>/<column_family_name>/snapshots` directory.

You must have enough free disk space on the node to accommodate making snapshots of your data files. A single snapshot requires little disk space. However, snapshots can cause your disk usage to grow more quickly over time because a snapshot prevents old obsolete data files from being deleted. After the snapshot is complete, you can move the backup files to another location if needed, or you can leave them in place.

To create a snapshot of a node

Run the `nodetool snapshot` command, specifying the hostname, JMX port, and keyspace. For example:

```
$ nodetool -h localhost -p 7199 snapshot demodb
```

The snapshot is created in `<data_directory_location>/<keyspace_name>/<column_family_name>/snapshots/<snapshot_name>`. Each snapshot folder contains numerous `.db` files that contain the data at the time of the snapshot.

Deleting Snapshot Files

When taking a snapshot, previous snapshot files are not automatically deleted. You should remove old snapshots that are no longer needed.

The `nodetool clearsnapshot` command removes all existing snapshot files from the snapshot directory of each keyspace. You should make it part of your back-up process to clear old snapshots before taking a new one.

- To delete all snapshots for a node, run the `nodetool clearsnapshot` command. For example:

```
$ nodetool -h localhost -p 7199 clearsnapshot
```

- To delete snapshots on all nodes at once, run the `nodetool clearsnapshot` command using a parallel ssh utility.

Enabling Incremental Backups

When incremental backups are enabled (disabled by default), Cassandra hard-links each flushed SSTable to a backups directory under the keyspace data directory. This allows storing backups offsite without transferring entire snapshots. Also, incremental backups combine with snapshots to provide a dependable, up-to-date backup mechanism.

To enable incremental backups, edit the *cassandra.yaml* configuration file on each node in the cluster and change the value of *incremental_backups* to `true`.

As with snapshots, Cassandra does not automatically clear incremental backup files. DataStax recommends setting up a process to clear incremental backup hard-links each time a new snapshot is created.

Restoring from a Snapshot

Restoring a keyspace from a snapshot requires all snapshot files for the column family, and if using incremental backups, any incremental backup files created after the snapshot was taken. You can restore a snapshot in several ways:

- Use the *sstableloader* tool.
- Copy the snapshot SSTable directory (see *Taking a Snapshot*) to the data directory (`/var/lib/cassandra/data/<keyspace>/<column_family>/`), and then call the JMX method `loadNewSSTables()` in the column family MBean for each column family through JConsole. Instead of using the `loadNewSSTables()` call, you can also use *nodetool refresh*.
- Use the Node Restart Method described below.

Node Restart Method

If restoring a single node, you must first shutdown the node. If restoring an entire cluster, you must shutdown all nodes, restore the snapshot data, and then start all nodes again.

Note

Restoring from snapshots and incremental backups temporarily causes intensive CPU and I/O activity on the node being restored.

To restore a node from a snapshot and incremental backups:

1. Shut down the node.
2. Clear all files in `/var/lib/cassandra/commitlog`.
3. Delete all `*.db` files in `<data_directory_location>/<keyspace_name>/<column_family_name>` directory, but **DO NOT** delete the `/snapshots` and `/backups` subdirectories.
4. Locate the most recent snapshot folder in `<data_directory_location>/<keyspace_name>/<column_family_name>/snapshots/<snapshot_name>`, and copy its contents into the `<data_directory_location>/<keyspace_name>/<column_family_name>` directory.
5. If using incremental backups, copy all contents of `<data_directory_location>/<keyspace_name>/<column_family_name>/backups` into `<data_directory_location>/<keyspace_name>/<column_family_name>`.
6. Restart the node.

Restarting causes a temporary burst of I/O activity and consumes a large amount of CPU resources.

Understanding the Cassandra Data Model

The Cassandra data model is a dynamic schema, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by your application up front, as each row is not required to have the same set of columns. Columns and their metadata can be added by your application as they are needed without incurring downtime to your application.

Data Modeling

When comparing Cassandra to a relational database, the column family is similar to a table in that it is a container for columns and rows. However, a column family requires a major shift in thinking for those coming from the relational world.

In a relational database, you define tables, which have defined columns. The table defines the column names and their data types, and the client application then supplies rows conforming to that schema: each row contains the same fixed set of columns.

In Cassandra, you define column families. Column families can (and should) define metadata about the columns, but the actual columns that make up a row are determined by the client application. Each row can have a different set of columns. There are two types of column families:

- *Static column family*: Typical Cassandra column family design.
- *Dynamic column family*: Use with a custom data type.

Column families consist of these kinds of columns:

- *Standard*: Has one primary key.
- *Composite*: Has more than one primary key, recommended for managing wide rows.
- *Expiring*: Gets deleted during compaction.
- *Counter*: Counts occurrences of an event.
- *Super*: Used to manage wide rows, inferior to using composite columns.

Although column families are very flexible, in practice a column family is not entirely schema-less.

Designing Column Families

Each row of a column family is uniquely identified by its row *key*, similar to the primary key in a relational table. A column family is partitioned on its row key, and the row key is implicitly indexed.

Static Column Families

A static column family uses a relatively static set of column names and is similar to a relational database table. For example, a column family storing user data might have columns for the user name, address, email, phone number and so on. Although the rows generally have the same set of columns, they are not required to have all of the columns defined. Static column families typically have column metadata pre-defined for each column.

row key	columns ...			
jbellis	name	email	address	state
	jonathan	jb@ds.com	123 main	TX
dhutch	name	email	address	state
	daria	dh@ds.com	45 2 nd St.	CA
egilmore	name	email		
	eric	eg@ds.com		

Dynamic Column Families

A dynamic column family takes advantage of Cassandra's ability to use arbitrary application-supplied column names to store data. A dynamic column family allows you to pre-compute result sets and store them in a single row for efficient data retrieval. Each row is a snapshot of data meant to satisfy a given query, sort of like a materialized view. For example, a column family that tracks the users that subscribe to a particular user's blog is dynamic.

row key	columns ...			
jbellis	dhutch	egilmore	datastax	mzcassie
dhutch	egilmore			
egilmore	datastax	mzcassie		

Instead of defining metadata for individual columns, a dynamic column family defines the type information for column names and values (comparators and validators), but the actual column names and values are set by the application when a column is inserted.

Standard Columns

The column is the smallest increment of data in Cassandra. It is a tuple containing a name, a value and a timestamp.

column_name
value
timestamp

A column must have a name, and the name can be a static label (such as "name" or "email") or it can be dynamically set when the column is created by your application.

Columns can be indexed on their name (see [secondary indexes](#)). However, one limitation of column indexes is that they do not support queries that require access to ordered data, such as time series data. In this case a secondary index on a timestamp column would not be sufficient because you cannot control column sort order with a secondary index. For cases where sort order is important, manually maintaining a column family as an 'index' is another way to lookup column data in sorted order.

It is not required for a column to have a value. Sometimes all the information your application needs to satisfy a given query can be stored in the column name itself. For example, if you are using a column family as a materialized view to lookup rows from other column families, all you need to store is the row key that you are looking up; the value can be

empty.

Cassandra uses the column timestamp to determine the most recent update to a column. The timestamp is provided by the client application. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one that will eventually persist. See [About Transactions and Concurrency Control](#) for more information about how Cassandra handles conflict resolution.

Composite Columns

Cassandra's storage engine uses composite columns under the hood to store clustered rows. All the logical rows with the same partition key get stored as a single, physical wide row. Using this design, Cassandra supports up to 2 billion columns per (physical) row.

Composite columns comprise fully denormalized wide rows by using composite primary keys. You create and query composite columns using CQL 3.

Tweets Example

For example, in the database you store the tweet, user, and follower data, and you want to use one query to return all the tweets of a user's followers.

First, set up a tweets table and a timeline table.

- The tweets table is the data table where the tweets live. The table has an author column, a body column, and a **surrogate** UUID key.

Note

UUIDs are handy for sequencing the data or automatically incrementing synchronization across multiple machines.

- The timeline table denormalizes the tweets, setting up composite columns by virtue of the composite primary key.

```
CREATE TABLE tweets (  
    tweet_id uuid PRIMARY KEY,  
    author varchar,  
    body varchar  
);  
  
CREATE TABLE timeline (  
    user_id varchar,  
    tweet_id uuid,  
    author varchar,  
    body varchar,  
    PRIMARY KEY (user_id, tweet_id)  
);
```


The combination of the user_id and tweet_id in the timeline table uniquely identifies a row in the timeline table. You can have more than one row with the same user ID as long as the rows contain different tweetIDs. The following figure shows three sample tweets of Patrick Henry, George Washington, and George Mason from different years. The tweet_ids are unique.

Tweets Table

tweet_id	author	body
1742	gWashington	I chopped down the cherry tree
1765	phenry	Give me liberty or give me death
1778	jadams	A government of laws, not men

The next figure shows how the tweets are denormalized for two of the users who are following these men. George Mason follows Patrick Henry and George Washington and Alexander Hamilton follow John Adams and George Washington.

Timeline Table




user_id	tweet_id	author	body
gmason	1765	phenry	Give me liberty or give me death
gmason	1742	gWashington	I chopped down the cherry tree
ahamilton	1797	jadams	A government of laws, not men
ahamilton	1742	gWashington	I chopped down the cherry tree

Cassandra uses the first column name in the primary key definition as the partition key, which is the same as the row key to the underlying storage engine. For example, in the timeline table, user_id is the partition key. The data for each partition key will be clustered by the remaining columns of the primary key definition. Clustering means that the storage engine creates an index and always keeps the data clustered by that index. Because the user_id is the partition key, all the tweets for gmason's friends, are clustered in the order of the remaining tweet_id column.

The storage engine guarantees that the columns are clustered according to the tweet_id. The next figure shows explicitly how the data maps to the storage engine: the gmason partition key designates a single storage engine row in which the rows of the logical view of the data share the same tweet_id part of a composite column name.

Timeline Physical Layout



gmason	[1765, author]: phenry	[1765, body]: Give me liberty or give me death	[1742, author]: gWashington	[1742, body]: I chopped down the cherry tree
ahamilton	[1797, author]: jadams	[1797, body]: A government of laws not men	[1742, author]: gWashington	[1742, body]: I chopped down the cherry tree

The gmason columns are next to each other as are the ahamilton columns. All the gmason or ahamilton columns are stored sequentially, ordered by the tweet_id columns within the respective gmason or ahamilton partition key. In the gmason row, the first field is the tweet_id, 1765, which is the composite column name, shared by the row data. Likewise, the 1742 row data share the 1742 component. The second field, named author in one column and body in another, contains the literal data that Cassandra stores. The physical representation of the row achieves the same sparseness using a compound primary key column as a standard Cassandra column.

Using the CQL 3 model, you can query a single sequential set of data on disk to get the tweets of a user's followers.

```
SELECT * FROM timeline WHERE user_id = gmason
ORDER BY tweet_id DESC LIMIT 50;
```

Compatibility with Older Applications

The query, expressed in SQL-like CQL 3 replaces the CQL 2 query that uses a range and the REVERSE keyword to slice 50 tweets out of the timeline material as viewed in the gmason row. The custom comparator or default_validation class that you had to set when dealing with wide rows in CQL 2 is no longer necessary in CQL 3.

The WITH COMPACT STORAGE directive is provided for backward compatibility with older Cassandra applications; new applications should avoid it. Using compact storage prevents you from adding new columns that are not part of the PRIMARY KEY. With compact storage, each logical row corresponds to exactly one physical column:

gmason	[1765, author]: phenry Give me liberty or give me death	[1742, author]: gwashington I chopped down the cherry tree
ahamilton	[1797, author]: jadams A government of laws, not men	[1742, author]: gwashington I chopped down the cherry tree

Expiring Columns

A column can also have an optional expiration date called TTL (time to live). Whenever a column is inserted, the client request can specify an optional TTL value, defined in seconds, for the column. TTL columns are marked as deleted (with a tombstone) after the requested amount of time has expired. Once they are marked with a tombstone, they are automatically removed during the normal compaction (defined by the *gc_grace_seconds*) and repair processes.

Use CQL *to set the TTL* for a column.

If you want to change the TTL of an expiring column, you have to re-insert the column with a new TTL. In Cassandra the insertion of a column is actually an *insertion* or *update* operation, depending on whether or not a previous version of the column exists. This means that to update the TTL for a column with an unknown value, you have to read the column and then re-insert it with the new TTL value.

TTL columns have a precision of one second, as calculated on the server. Therefore, a very small TTL probably does not make much sense. Moreover, the clocks on the servers should be synchronized; otherwise reduced precision could be observed because the expiration time is computed on the primary host that receives the initial insertion but is then interpreted by other hosts on the cluster.

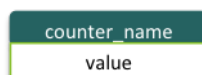
An expiring column has an additional overhead of 8 bytes in memory and on disk (to record the TTL and expiration time) compared to standard columns.

Counter Columns

A counter is a special kind of column used to store a number that incrementally counts the occurrences of a particular event or process. For example, you might use a counter column to count the number of times a page is viewed.

Counter column families must use CounterColumnType as the validator (the column value type). This means that currently, counters may only be stored in dedicated column families.

Counter columns are different from regular columns in that once a counter is defined, the client application then updates the column value by incrementing (or decrementing) it. A client update to a counter column passes the name of the counter and the increment (or decrement) value; no timestamp is required.



Internally, the structure of a counter column is a bit more complex. Cassandra tracks the distributed state of the counter as well as a server-generated timestamp upon deletion of a counter column. For this reason, it is important that all nodes in your cluster have their clocks synchronized using network time protocol (NTP).

A counter can be read or written at any of the available *consistency levels*. However, it's important to understand that unlike normal columns, a write to a counter requires a read in the background to ensure that distributed counter values remain consistent across replicas. If you write at a consistency level of ONE, the implicit read will not impact write latency, hence, ONE is the most common consistency level to use with counters.

Super Columns

Do not use super columns. They are a legacy design from a pre-open source release. This design was structured for a specific use case and does not fit most use cases. Super columns read entire super columns and all its sub-columns into memory for each read request. This results in severe performance issues. Additionally, super columns are not supported in CQL 3.

Use composite columns instead. Composite columns provide most of the same benefits as super columns without the performance issues.

About Data Types (Comparators and Validators)

In a relational database, you must specify a data type for each column when you define a table. The data type constrains the values that can be inserted into that column. For example, if you have a column defined as an integer datatype, you would not be allowed to insert character data into that column. Column names in a relational database are typically fixed labels (strings) that are assigned when you define the table schema.

In Cassandra, the data type for a column (or row key) *value* is called a *validator*. The data type for a column *name* is called a *comparator*. You can define data types when you create your column family schemas (which is recommended), but Cassandra does not require it. Internally, Cassandra stores column names and values as hex byte arrays (`ByteType`). This is the default client encoding used if data types are not defined in the column family schema (or if not specified by the client request).

Cassandra comes with the following built-in data types, which can be used as both validators (row key and column value data types) or comparators (column name data types). One exception is `CounterColumnType`, which is only allowed as a column value (not allowed for row keys or column names).

Internal Type	CQL Name	Description
ByteType	blob	Arbitrary hexadecimal bytes (no validation)
AsciiType	ascii	US-ASCII character string
UTF8Type	text, varchar	UTF-8 encoded string
IntegerType	varint	Arbitrary-precision integer
Int32Type	int	4-byte integer
LongType	bigint	8-byte long
UUIDType	uuid	Type 1 or type 4 UUID
TimeUUIDType	timeuuid	Type 1 UUID only (CQL3)
DateType	timestamp	Date plus time, encoded as 8 bytes since epoch
BooleanType	boolean	true or false
FloatType	float	4-byte floating point
DoubleType	double	8-byte floating point
DecimalType	decimal	Variable-precision decimal
CounterColumnType	counter	Distributed counter value (8-byte long)

Composite Types

Additional new composite types exist for indirect use through CQL. Using these types through an API client is not recommended. Composite types used through CQL 3 support Cassandra wide rows using composite column names to *create tables*.

About Validators

Using the CLI you can define a default row key validator for a column family using the *key_validation_class* property. Using CQL, you use built-in *key validators* to validate row key values. For static column families, define each column and its associated type when you define the column family using the *column_metadata* property.

Key and column validators may be added or changed in a column family definition at any time. If you specify an invalid validator on your column family, client requests that respect that metadata will be confused, and data inserts or updates that do not conform to the specified validator will be rejected.

For dynamic column families (where column names are not known ahead of time), you should specify a *default_validation_class* instead of defining the per-column data types.

Key and column validators may be added or changed in a column family definition at any time. If you specify an invalid validator on your column family, client requests that respect that metadata will be confused, and data inserts or updates that do not conform to the specified validator will be rejected.

About the Comparator

Within a row, columns are always stored in sorted order by their *column name*. The *comparator* specifies the data type for the column name, as well as the sort order in which columns are stored within a row. Unlike validators, the comparator may *not* be changed after the column family is defined, so this is an important consideration when defining a column family in Cassandra.

Typically, static column family names will be strings, and the sort order of columns is not important in that case. For dynamic column families, however, sort order is important. For example, in a column family that stores time series data (the column names are timestamps), having the data in sorted order is required for slicing result sets out of a row of columns.

Compressing Column Family Data

Cassandra application-transparent compression maximizes the storage capacity of your Cassandra nodes by reducing the volume of data on disk. In addition to the space-saving benefits, compression also reduces disk I/O, particularly for read-dominated workloads. To compress column family data, *use CLI or CQL*.

About Keyspaces

In Cassandra, the *keyspace* is the container for your application data, similar to a schema in a relational database. Keyspaces are used to group column families together. Typically, a cluster has one keyspace per application.

Replication is controlled on a per-keyspace basis, so data that has different replication requirements should reside in different keyspace. Keyspaces are not designed to be used as a significant map layer within the data model, only as a way to control data replication for a set of column families. DataStax recommends *NetworkTopologyStrategy* for single and multiple data center clusters because it allows for expansion to multiple data centers in the future. By configuring a flexible replication strategy in the beginning, you can avoid having to reconfigure replication later after you have loaded data into your cluster.

Defining Keyspaces

Data Definition Language (DDL) commands for defining and altering keyspace are provided in the various client interfaces, such as Cassandra CLI and CQL. For example, to define a keyspace in CQL:

Note

These examples work with pre-release CQL 3 in Cassandra 1.1.x. The syntax differs in the release version of CQL 3 in Cassandra 1.2 and later.

```
CREATE KEYSPACE test
  WITH strategy_class = NetworkTopologyStrategy
  AND strategy_options:"us-east" = 6;
```

Or for a multiple data center cluster:

```
CREATE KEYSPACE test2
  WITH strategy_class = NetworkTopologyStrategy
  AND strategy_options:DC1 = 3
  AND strategy_options:DC2 = 6;
```

When declaring the keyspace *strategy_options*, use the data center names defined for the *snitch* that you chose for the cluster. To place replicas in the correct location, Cassandra requires a keyspace definition that uses the snitch-aware data center names. For example, if the cluster uses the *PropertyFileSnitch* create the keyspace using the user-defined data center and rack names in the `cassandra-topologies.properties` file. If the cluster uses the *EC2Snitch*, create the keyspace using EC2 data center and rack names. All *snitches* are compatible with both *replica placement strategies*.

For more information on DDL commands, see *Querying Cassandra*.

About Indexes in Cassandra

An index is a data structure that allows for fast, efficient lookup of data matching a given condition.

About Primary Indexes

In relational database design, a primary key is the unique key used to identify each row in a table. A primary key index, like any index, speeds up random access to data in the table. The primary key also ensures record uniqueness, and may also control the order in which records are physically clustered, or stored by the database.

In Cassandra, the primary index for a column family is the index of its row keys. Each node maintains this index for the data it manages.

Rows are assigned to nodes by the cluster-configured *partitioner* and the keyspace-configured *replica placement strategy*. The primary index in Cassandra allows looking up of rows by their row key. Since each node knows what ranges of keys each node manages, requested rows can be efficiently located by scanning the row indexes only on the relevant replicas.

With randomly partitioned row keys (the default in Cassandra), row keys are partitioned by their MD5 hash and cannot be scanned in order like traditional b-tree indexes. The CQL3 *token function* introduced in Cassandra 1.1.1 now makes it possible to page through non-ordered partitioner results.

About Secondary Indexes

Secondary indexes in Cassandra refer to indexes on column values (to distinguish them from the primary row key index for a column family). Cassandra implements secondary indexes as a hidden table, separate from the table that contains the values being indexed. Using CQL, you can create a secondary index on a column after defining a column family. Using CQL 3, for example, define a static column family, and then create a secondary index on one of its named columns:

```
CREATE TABLE users (
  userID uuid,
  firstname text,
```

```
    lastname text,  
    state text,  
    zip int,  
    PRIMARY KEY (userID)  
);
```

```
CREATE INDEX users_state  
ON users (state);
```

Secondary index names, in this case `users_state`, must be unique within the keyspace. After creating a secondary index for the state column and inserting values into the column family, you can query Cassandra directly for users who live in a given state:

```
SELECT * FROM users WHERE state = 'TX';
```

userid	firstname	lastname	state	zip
b70de1d0-9908-4ae3-be34-5573e5b09f14	john	smith	TX	77362

Secondary indexes allow for efficient querying by specific values using equality predicates (where column x = value y).

Creating Multiple Secondary Indexes

You can create multiple secondary indexes, for example on the `firstname` and `lastname` columns of the `users` table, and use multiple conditions in the `WHERE` clause to filter the results:

```
CREATE INDEX users_fname  
ON users (firstname);
```

```
CREATE INDEX users_lname  
ON users (lastname);
```

```
SELECT * FROM users WHERE users_fname = 'bob' AND users_lname = 'smith';
```

When there are multiple conditions in a `WHERE` clause, Cassandra's selects the least-frequent occurrence of a condition for processing first for efficiency. In this example, Cassandra queries on the last name first if there are fewer Smiths than Bobs in the database or on the first name first if there are fewer Bobs than Smiths.

When to Use Secondary Indexes

Cassandra's built-in secondary indexes are best on a column family having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a user table with a billion users and wanted to look up users by the state they lived in. Many users will share the same column value for state (such as CA, NY, TX, etc.). This would be a good candidate for a secondary index.

When Not to Use Secondary Indexes

Do not use secondary indexes to query a huge volume of records for a small number of results. For example, if you create indexes on columns that have many distinct values, a query between the fields will incur many seeks for very few results. In the column family with a billion users, looking up users by their email address (a value that is typically unique for each user) instead of by their state, is likely to be very inefficient. It would probably be more efficient to manually maintain a dynamic column family as a form of an index instead of using a secondary index. For columns containing unique data, it is sometimes fine performance-wise to use secondary indexes for convenience, as long as the query volume to the indexed column family is moderate and not under constant load.

Building and Using Secondary Indexes

An advantage of secondary indexes is the operational ease of populating and maintaining the index. Secondary indexes are built in the background automatically, without blocking reads or writes. Client-maintained *column families as indexes* must be created manually; for example, if the state column had been indexed by creating a column family such as `users_by_state`, your client application would have to populate the column family with data from the `users` column family.

Maintaining Secondary Indexes

To perform a hot rebuild of a secondary index, use the `nodetool` utility `rebuild_index` command.

Planning Your Data Model

Planning a data model in Cassandra involves different design considerations than you may be used to if you work with relational databases. Ultimately, the data model you design depends on the data you want to capture and how you plan to access it. However, there are some common design considerations for Cassandra data model planning.

Start with Queries

The best way to approach data modeling for Cassandra is to start with your queries and work back from there. Think about the actions your application needs to perform, how you want to access the data, and then design column families to support those access patterns. A good rule of a thumb is one column family per query since you optimize column families for read performance.

For example, start with listing the use cases your application needs to support. Think about the data you want to capture and the lookups your application needs to do. Also note any ordering, filtering or grouping requirements. For example, needing events in chronological order or considering only the last 6 months worth of data would be factors in your data model design.

Denormalize to Optimize

In the relational world, the data model is usually designed up front with the goal of normalizing the data to minimize redundancy. Normalization typically involves creating smaller, well-structured tables and then defining relationships between them. During queries, related tables are joined to satisfy the request.

Cassandra does not have foreign key relationships like a relational database does, which means you cannot join multiple column families to satisfy a given query request. Cassandra performs best when the data needed to satisfy a given query is located in the *same* column family. Try to plan your data model so that one or more rows in a single column family are used to answer each query. This sacrifices disk space (one of the cheapest resources for a server) in order to reduce the number of disk seeks and the amount of network traffic.

Planning for Concurrent Writes

Within a column family, every row is known by its row key, a string of virtually unbounded length. The key has no required form, but it must be unique within a column family. Unlike the primary key in a relational database, Cassandra does not enforce unique-ness. Inserting a duplicate row key will *upsert* the columns contained in the insert statement rather than return a unique constraint violation.

Using Natural or Surrogate Row Keys

One consideration is whether to use surrogate or natural keys for a column family. A surrogate key is a generated key (such as a UUID) that uniquely identifies a row, but has no relation to the actual data in the row.

For some column families, the data may contain values that are guaranteed to be unique and are not typically updated after a row is created. For example, the username in a `users` column family. This is called a natural key. Natural keys make the data more readable, and remove the need for additional indexes or denormalization. However, unless your client application ensures unique-ness, there is potential of over-writing column data.

Also, the natural key approach does not easily allow updates to the row key. For example, if your row key was an email address and a user wanted to change their email address, you would have to create a new row with the new email address and copy all of the existing columns from the old row to the new row.

UUID Type for Column Names

The UUID comparator type (universally unique id) is used to avoid collisions in column names in CQL. Alternatively, as of Cassandra 1.1.1 in CQL3 you can use the *timeuuid*. For example, if you wanted to identify a column (such as a blog entry or a tweet) by its timestamp, multiple clients writing to the same row key simultaneously could cause a timestamp collision, potentially overwriting data that was not intended to be overwritten. Using the UUIDType to represent a type-1 (time-based) UUID can avoid such collisions.

Managing Data

To manage data, you need to understand how Cassandra *writes* data, eventually *tuning data consistency*, and then compacts and *stores* data on disk. Atomicity, consistency, isolation, and durability of data is *described in this section*. You can access and manage data in Cassandra using a utility, such as the Cassandra Command Line Interface (CLI) and the Cassandra Query Language (CQL). Application programming interfaces (APIs) can be used for developing applications that use Cassandra for data storage and retrieval.

About Writes in Cassandra

Relational databases typically structure tables in order to keep data duplication at a minimum. The various pieces of information needed to satisfy a query are stored in various related tables that adhere to a pre-defined structure. Because of the data structure in a relational database, writing data is expensive, as the database server has to do additional work to ensure data integrity across the various related tables. As a result, relational databases unlike Cassandra usually are not performant on writes.

Cassandra writes are first written to a commit log (for durability), and then to an in-memory table structure called a *memtable*. A write is successful once it is written to the commit log and memory, so there is very minimal disk I/O at the time of write. Writes are batched in memory and periodically written to disk to a persistent table structure called an *SSTable* (sorted string table). Memtables and SSTables are maintained per column family. Memtables are organized in sorted order by row key and flushed to SSTables sequentially (no random seeking as in relational databases).

SSTables are immutable (they are not written to again after they have been flushed). This means that a row is typically stored across multiple SSTable files. At read time, a row must be combined from all SSTables on disk (as well as unflushed memtables) to produce the requested data. To optimize this piecing-together process, Cassandra uses an in-memory structure called a *Bloom filter*. Each SSTable has a Bloom filter associated with it that checks if a requested row key exists in the SSTable before doing any disk seeks.

For a detailed explanation of how client read and write requests are handled in Cassandra, see *About Client Requests in Cassandra*.

Managing Stored Data

Cassandra now writes column families to disk using this directory and file naming format:

```
/var/lib/cassandra/data/ks1/cf1/ks1-cf1-hc-1-Data.db
```

Cassandra creates a subdirectory for each column family, which allows a developer or admin to symlink a column family to a chosen physical drive or data volume. This provides the ability to move very active column families to faster media, such as SSD's for better performance and also divvy up column families across all attached storage devices for better I/O balance at the storage layer.

The keyspace name is part of the SST name, which makes bulk loading easier. You no longer need to put sstables into a subdirectory named for the destination keyspace or specify the keyspace a second time on the command line when you create sstables to stream off to the cluster.

About Compaction

In the background, Cassandra periodically merges SSTables together into larger SSTables using a process called *compaction*. Compaction merges row fragments together, removes expired tombstones (deleted columns), and rebuilds primary and secondary indexes. Since the SSTables are sorted by row key, this merge is efficient (no random disk I/O). Once a newly merged SSTable is complete, the input SSTables are marked as obsolete and eventually deleted by the JVM garbage collection (GC) process. However, during compaction, there is a temporary spike in disk space usage and disk I/O.

Compaction impacts read performance in two ways. While a compaction is in progress, it temporarily increases disk I/O and disk utilization which can impact read performance for reads that are not fulfilled by the cache. However, after a compaction has been completed, off-cache read performance improves since there are fewer SSTable files on disk that

need to be checked in order to complete a read request.

As of Cassandra 1.0, there are two different compaction strategies that you can configure on a column family - size-tiered compaction or leveled compaction. See *Tuning Compaction* for a description of these compaction strategies.

Starting with Cassandra 1.1, a startup option allows you to test various compaction strategies without affecting the production workload. See *Testing Compaction and Compression*. Additionally, you can now stop compactions

About Transactions and Concurrency Control

Cassandra does not offer fully ACID-compliant transactions, the standard for transactional behavior in a relational database systems:

- **Atomic.** Everything in a transaction succeeds or the entire transaction is rolled back.
- **Consistent.** A transaction cannot leave the database in an inconsistent state.
- **Isolated.** Transactions cannot interfere with each other.
- **Durable.** Completed transactions persist in the event of crashes or server failure.

As a non-relational database, Cassandra does not support joins or foreign keys, and consequently does not offer consistency in the ACID sense. For example, when moving money from account A to B the total in the accounts does not change. Cassandra supports atomicity and isolation at the row-level, but trades transactional isolation and atomicity for high availability and fast write performance. Cassandra writes are durable.

Atomicity in Cassandra

In Cassandra, a write is atomic at the row-level, meaning inserting or updating columns for a given row key will be treated as one write operation. Cassandra does not support transactions in the sense of bundling multiple row updates into one all-or-nothing operation. Nor does it roll back when a write succeeds on one replica, but fails on other replicas. It is possible in Cassandra to have a write operation report a failure to the client, but still actually persist the write to a replica.

For example, if using a write consistency level of QUORUM with a replication factor of 3, Cassandra will send the write to 2 replicas. If the write fails on one of the replicas but succeeds on the other, Cassandra will report a write failure to the client. However, the write is not automatically rolled back on the other replica.

Cassandra uses timestamps to determine the most recent update to a column. The timestamp is provided by the client application. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one that will eventually persist.

Tunable Consistency in Cassandra

There are no locking or transactional dependencies when concurrently updating multiple rows or column families. Cassandra supports *tuning between availability and consistency*, and always gives you partition tolerance. Cassandra can be tuned to give you strong consistency in the CAP sense where data is made consistent across all the nodes in a distributed database cluster. A user can pick and choose on a per operation basis how many nodes must receive a DML command or respond to a SELECT query.

Isolation in Cassandra

Prior to Cassandra 1.1, it was possible to see partial updates in a row when one user was updating the row while another user was reading that same row. For example, if one user was writing a row with two thousand columns, another user could potentially read that same row and see some of the columns, but not all of them if the write was still in progress.

Full row-level isolation is now in place so that writes to a row are isolated to the client performing the write and are not visible to any other user until they are complete.

From a transactional ACID (atomic, consistent, isolated, durable) standpoint, this enhancement now gives Cassandra transactional AID support. A write is isolated at the row-level in the storage engine.

Durability in Cassandra

Writes in Cassandra are durable. All writes to a replica node are recorded both in memory and in a commit log before they are acknowledged as a success. If a crash or server failure occurs before the memory tables are flushed to disk, the commit log is replayed on restart to recover any lost writes.

About Inserts and Updates

Any number of columns may be inserted at the same time. When inserting or updating columns in a column family, the client application specifies the row key to identify which column records to update. The row key is similar to a primary key in that it must be unique for each row within a column family. However, unlike a primary key, inserting a duplicate row key will not result in a primary key constraint violation - it will be treated as an `UPSERT` (update the specified columns in that row if they exist or insert them if they do not).

Columns are only overwritten if the timestamp in the new version of the column is more recent than the existing column, so precise timestamps are necessary if updates (overwrites) are frequent. The timestamp is provided by the client, so the clocks of all client machines should be synchronized using NTP (network time protocol).

About Deletes

When deleting a column in Cassandra, there are a few things to be aware of that may differ from what one would expect in a relational database.

1. **Deleted data is not immediately removed from disk.** Data that is inserted into Cassandra is persisted to SSTables on disk. Once an SSTable is written, it is immutable (the file is not updated by further DML operations). This means that a deleted column is not removed immediately. Instead a marker called a *tombstone* is written to indicate the new column status. Columns marked with a tombstone exist for a configured time period (defined by the *gc_grace_seconds* value set on the column family), and then are permanently deleted by the compaction process after that time has expired.
2. **A deleted column can reappear if routine node repair is not run.** Marking a deleted column with a tombstone ensures that a replica that was down at the time of delete will eventually receive the delete when it comes back up again. However, if a node is down longer than the configured time period for keeping tombstones (defined by the *gc_grace_seconds* value set on the column family), then the node can possibly miss the delete altogether, and replicate deleted data once it comes back up again. To prevent deleted data from reappearing, administrators must run regular node repair on every node in the cluster (by default, every 10 days).

About Reads in Cassandra

When a read request for a row comes in to a node, the row must be combined from all SSTables on that node that contain columns from the row in question, as well as from any unflushed memtables, to produce the requested data. To optimize this piecing-together process, Cassandra uses an in-memory structure called a *Bloom filter*. Each SSTable has a Bloom filter associated with it that checks if any data for the requested row exists in the SSTable before doing any disk I/O. As a result, Cassandra is very performant on reads when compared to other storage systems, even for read-heavy workloads.

As with any database, reads are fastest when the most in-demand data (or *hot* working set) fits into memory. Although all modern storage systems rely on some form of caching to allow for fast access to hot data, not all of them degrade gracefully when the cache capacity is exceeded and disk I/O is required. Cassandra's read performance benefits from built-in caching, but it also does not dip dramatically when random disk seeks are required. When I/O activity starts to increase in Cassandra due to increased read load, it is easy to remedy by adding more nodes to the cluster.

For rows that are accessed frequently, Cassandra has a built-in key cache (and an optional row cache). For more information about optimizing read performance using the built-in caching feature, see *Tuning Data Caches*.

For a detailed explanation of how client read and write requests are handled in Cassandra, also see [About Client Requests in Cassandra](#).

About Data Consistency in Cassandra

In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replicas. Cassandra extends the concept of eventual consistency by offering *tunable consistency*. For any given read or write operation, the client application decides how consistent the requested data should be.

In addition to tunable consistency, Cassandra has a number of *built-in repair mechanisms* to ensure that data remains consistent across replicas.

In this Cassandra version, many schema changes can take place simultaneously in a cluster without any schema disagreement among nodes. For example, if one client sets a column to an integer and another client sets the column to text, one or the other will be instantly agreed upon, which one is unpredictable.

The new schema resolution design eliminates delays caused by schema changes when a new node joins the cluster. As soon as the node joins the cluster, it receives the current schema with instantaneous reconciliation of changes.

Tunable Consistency for Client Requests

Consistency levels in Cassandra can be set on any read or write query. This allows application developers to tune consistency on a per-query basis depending on their requirements for response time versus data accuracy. Cassandra offers a number of consistency levels for both reads and writes.

About Write Consistency

When you do a write in Cassandra, the consistency level specifies on how many replicas the write must succeed before returning an acknowledgement to the client application.

The following consistency levels are available, with ANY being the lowest consistency (but highest availability), and ALL being the highest consistency (but lowest availability). QUORUM is a good middle-ground ensuring strong consistency, yet still tolerating some level of failure.

A quorum is calculated as (rounded down to a whole number):

$$(\text{replication_factor} / 2) + 1$$

For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 replica down). With a replication factor of 6, a quorum is 4 (can tolerate 2 replicas down).

Level	Description
ANY	A write must be written to at least one node. If all replica nodes for the given row key are down, the write can still succeed once a <i>hinted handoff</i> has been written. Note that if all replica nodes are down at write time, an ANY write will not be readable until the replica nodes for that row key have recovered.
ONE	A write must be written to the commit log and memory table of at least one replica node.
TWO	A write must be written to the commit log and memory table of at least two replica nodes.
THREE	A write must be written to the commit log and memory table of at least three replica nodes.
QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes.
LOCAL_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in the same data center as the coordinator node. Avoids latency of inter-data center communication.
EACH_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in <i>all</i> data centers.

ALL	A write must be written to the commit log and memory table on all replica nodes in the cluster for that row key.
-----	--

About Read Consistency

When you do a read in Cassandra, the consistency level specifies how many replicas must respond before a result is returned to the client application.

Cassandra checks the specified number of replicas for the most recent data to satisfy the read request (based on the timestamp).

The following consistency levels are available, with ONE being the lowest consistency (but highest availability), and ALL being the highest consistency (but lowest availability). QUORUM is a good middle-ground ensuring strong consistency, yet still tolerating some level of failure.

A quorum is calculated as (rounded down to a whole number):

```
(replication_factor / 2) + 1
```

For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 replica down). With a replication factor of 6, a quorum is 4 (can tolerate 2 replicas down).

Level	Description
ONE	Returns a response from the closest replica (as determined by the snitch). By default, a read repair runs in the background to make the other replicas consistent.
TWO	Returns the most recent data from two of the closest replicas.
THREE	Returns the most recent data from three of the closest replicas.
QUORUM	Returns the record with the most recent timestamp after a quorum of replicas has responded.
LOCAL_QUORUM	Returns the record with the most recent timestamp after a quorum of replicas in the current data center as the coordinator node has reported. Avoids latency of inter-data center communication.
EACH_QUORUM	Returns the record with the most recent timestamp after a quorum of replicas in each data center of the cluster has responded.
ALL	Returns the record with the most recent timestamp after all replicas have responded. The read operation fails if a replica does not respond.

Choosing Client Consistency Levels

Choosing a consistency level for reads and writes involves determining your requirements for consistent results (always reading the most recently written data) versus read or write latency (the time it takes for the requested data to be returned or for the write to succeed).

If latency is a top priority, consider a consistency level of ONE (only one replica node must successfully respond to the read or write request). There is a higher probability of stale data being read with this consistency level (as the replicas contacted for reads may not always have the most recent write). For some applications, this may be an acceptable trade-off. If it is an absolute requirement that a write never fail, you may also consider a write consistency level of ANY. This consistency level has the highest probability of a read not returning the latest written values (see *hinted handoff*).

If consistency is top priority, you can ensure that a read will always reflect the most recent write by using the following formula:

```
(nodes_written + nodes_read) > replication_factor
```

For example, if your application is using the QUORUM consistency level for both write and read operations and you are using a replication factor of 3, then this ensures that 2 nodes are always written and 2 nodes are always read. The combination of nodes written and read (4) being greater than the replication factor (3) ensures strong read consistency.

Consistency Levels for Multiple Data Center Clusters

A client read or write request to a Cassandra cluster always specifies the *consistency level* it requires. Ideally, you want a client request to be served by replicas in the same data center in order to avoid latency. Contacting multiple data centers for a read or write request can slow down the response. `LOCAL_QUORUM` and `EACH_QUORUM` are used in multiple data center clusters of real-time Cassandra nodes using a rack-aware replica placement strategy (such as `NetworkTopologyStrategy`) and a properly configured snitch. These consistency levels will fail when using `SimpleStrategy`.

A consistency level of `ONE` is also fine for applications with less stringent consistency requirements. A majority of Cassandra users do writes at consistency level `ONE`. With this consistency, the request will always be served by the replica node closest to the coordinator node that received the request (unless the *dynamic snitch* determines that the node is performing poorly and routes it elsewhere).

Keep in mind that even at consistency level `ONE` or `LOCAL_QUORUM`, the write is still sent to all replicas for the written key, even replicas in other data centers. The consistency level just determines how many replicas are required to respond that they received the write.

Specifying Client Consistency Levels

Consistency level is specified by the client application when a read or write request is made. The default consistency level may differ depending on the client you are using.

For example, in CQL the default consistency level for reads and writes is `ONE`. If you wanted to use `QUORUM` instead, you could specify that consistency level in the client request as follows:

```
SELECT * FROM users USING CONSISTENCY QUORUM WHERE state='TX' ;
```

About Cassandra's Built-in Consistency Repair Features

Cassandra has a number of built-in repair features to ensure that data remains consistent across replicas. These features are:

- **Read Repair** - For reads, there are two types of read requests that a coordinator can send to a replica; a direct read request and a background *read repair* request. The number of replicas contacted by a direct read request is determined by the *consistency level* specified by the client. Background read repair requests are sent to any additional replicas that did not receive a direct request. To ensure that frequently-read data remains consistent, the coordinator compares the data from all the remaining replicas that own the row in the background, and if they are inconsistent, issues writes to the out-of-date replicas to update the row to reflect the most recently written values. Read repair can be configured per column family (using *read_repair_chance*), and is enabled by default.
- **Anti-Entropy Node Repair** - For data that is not read frequently, or to update data on a node that has been down for a while, the *nodetool repair* process (also referred to as anti-entropy repair) ensures that all data on a replica is made consistent. Node repair should be run routinely as part of regular cluster maintenance operations.
- **Hinted Handoff** - Writes are always sent to all replicas for the specified row regardless of the consistency level specified by the client. If a node happens to be down at the time of write, its corresponding replicas will save hints about the missed writes, and then handoff the affected rows once the node comes back online. Hinted handoff ensures data consistency due to short, transient node outages. The hinted handoff feature is configurable at the node-level in the *cassandra.yaml* file

About Hinted Handoff Writes

Hinted handoff is an optional feature of Cassandra that reduces the time to restore a failed node to consistency once the failed node returns to the cluster. It can also be used for absolute write availability for applications that cannot tolerate a failed write, but can tolerate inconsistent reads.

When a write is made, Cassandra attempts to write to all replicas for the affected row key. If a replica is known to be down before the write is forwarded to it, or if it fails to acknowledge the write for any reason, the coordinator will store a hint for it. The hint consists of the target replica, as well as the mutation to be replayed.

If all replicas for the affected row key are down, it is still possible for a write to succeed when using a *write consistency* level of ANY. Under this scenario, the hint and write data are stored on the coordinator node but not available to reads until the hint is replayed to the actual replicas that own the row. The ANY consistency level provides absolute write availability at the cost of consistency; there is no guarantee as to when write data is available to reads because availability depends on how long the replicas are down. The coordinator node stores hints for dead replicas regardless of consistency level unless hinted handoff is *disabled*. A `TimedOutException` is reported if the coordinator node cannot replay to the replica. In Cassandra, a timeout is not a failure for writes.

Note

By default, hints are only saved for one hour after a replica fails because if the replica is down longer than that, it is likely permanently dead. In this case, you should run a *repair* to re-replicate the data before the failure occurred. You can configure this time using the *max_hint_window_in_ms* property in the `cassandra.yaml` file.

Hint creation does not count towards any consistency level besides ANY. For example, if no replicas respond to a write at a consistency level of ONE, hints are created for each replica but the request is reported to the client as timed out. However, since hints are replayed at the earliest opportunity, a timeout here represents a write-in-progress, rather than failure. The only time Cassandra will fail a write entirely is when too few replicas are alive when the coordinator receives the request. For a complete explanation of how Cassandra deals with replica failure, see *When a timeout is not a failure: how Cassandra delivers high availability*.

When a replica that is storing hints detects via gossip that the failed node is alive again, it will begin streaming the missed writes to catch up the out-of-date replica.

Note

Hinted handoff does not completely replace the need for regular node repair operations. In addition to the time set by *max_hint_window_in_ms*, the coordinator node storing hints could fail before replay. You should *always* run a full repair after losing a node or disk.

Cassandra Client APIs

When Cassandra was first released, it originally provided a *Thrift* RPC-based API as the foundation for client developers to build upon. This proved to be suboptimal: Thrift is too low-level to use without a more idiomatic client wrapping it, and supporting new features (such as secondary indexes in 0.7 and counters in 0.8) became hard to maintain across these clients for many languages. Also, by not having client development hosted within the Apache Cassandra project itself, incompatible clients proliferated in the open source community, all with different levels of stability and features. It became hard for application developers to choose the best API to fit their needs.

About Cassandra CLI

Cassandra 0.7 introduced a stable version of its command-line client interface, `cassandra-cli`, that can be used for common data definition (DDL), data manipulation (DML), and data exploration. Although not intended for application development, it is a good way to get started defining your data model and becoming familiar with Cassandra.

About CQL

CQL syntax is based on SQL (Structured Query Language). Although CQL has many similarities to SQL, it does not change the underlying Cassandra data model, which is not relational. There is no support for JOINS, for example.

Cassandra 0.8 was the first release to include the Cassandra Query Language (CQL). As with SQL, clients built on CQL only need to know how to interpret query `resultset` objects. CQL is the future of Cassandra client API development. CQL drivers are hosted within the Apache Cassandra project.

CQL version 2.0, which has improved support for several commands, is compatible with Cassandra version 1.0 but not version 0.8.x.

Getting Started Using the Cassandra CLI

In Cassandra 1.1, CQL became the primary interface into the DBMS. The CQL specification was promoted to CQL 3, although CQL 2 remains the default because CQL3 is pre-release and not backwards compatible. The most significant enhancement of CQL 3 is support for *compound primary key columns*.

The Python driver includes a command-line interface, `cql.sh`. See *Querying Cassandra*.

Other High-Level Clients

The Thrift API will continue to be supported for backwards compatibility. Using a high-level client is highly recommended over using raw Thrift calls.

A list of other available clients may be found on the [Client Options](#) page.

The Java, Python, and PHP clients are well supported.

Java: Hector Client API

Hector provides Java developers with features lacking in Thrift, including connection pooling, JMX integration, failover and extensive logging. Hector is the first client to implement CQL.

For more information, see the [Hector web site](#).

Python: Pycassa Client API

Pycassa is a Python client API with features such as connection pooling and a method to map existing classes to Cassandra column families.

For more information, see the [Pycassa documentation](#).

PHP: Phpcassa Client API

Phpcassa is a PHP client API with features such as connection pooling, a method for counting rows, and support for secondary indexes.

For more information, see the [Phpcassa documentation](#).

Getting Started Using the Cassandra CLI

The Cassandra CLI client utility can be used to do basic data definition (DDL) and data manipulation (DML) within a Cassandra cluster. It is located in `/usr/bin/cassandra-cli` in packaged installations or `<install_location>/bin/cassandra-cli` in binary installations.

To start the CLI and connect to a particular Cassandra instance, launch the script together with `-host` and `-port` options. It will connect to the cluster name specified in the *cassandra.yaml* file (which is *Test Cluster* by default). For example, if you have a single-node cluster on localhost:

```
$ cassandra-cli -host localhost -port 9160
```

Or to connect to a node in a multi-node cluster, give the IP address of the node:

```
$ cassandra-cli -host 110.123.4.5 -port 9160
```

To see help on the various commands available:

```
[default@unknown] help;
```

For detailed help on a specific command, use `help <command>;`. For example:

```
[default@unknown] help SET;
```

Note

A command is not sent to the server unless it is terminated by a semicolon (;). Hitting the return key without a semicolon at the end of the line echos an ellipsis (. . .), which indicates that the CLI expects more input.

Creating a Keyspace

You can use the Cassandra CLI commands described in this section to create a keyspace. In this example, we create a keyspace called `demo`, with a replication factor of 1 and using the `SimpleStrategy` replica placement strategy.

Note the single quotes around the string value of `placement_strategy`:

```
[default@unknown] CREATE KEYSPACE demo
with placement_strategy = 'SimpleStrategy'
and strategy_options = {replication_factor:1};
```

You can verify the creation of a keyspace with the `SHOW KEYSPACES` command. The new keyspace is listed along with the system keyspace and any other existing keyspaces.

Creating a Column Family

First, connect to the keyspace where you want to define the column family with the `USE` command.

```
[default@unknown] USE demo;
```

In this example, we create a `users` column family in the `demo` keyspace. In this column family we are defining a few columns; `full_name`, `email`, `state`, `gender`, and `birth_year`. This is considered a *static* column family - we are defining the column names up front and most rows are expected to have more-or-less the same columns.

Notice the settings of `comparator`, `key_validation_class` and `validation_class`. These are setting the default encoding used for column names, row key values and column values. In the case of column names, the comparator also determines the sort order.

```
[default@unknown] USE demo;

[default@demo] CREATE COLUMN FAMILY users
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
AND column_metadata = [
{column_name: full_name, validation_class: UTF8Type}
{column_name: email, validation_class: UTF8Type}
{column_name: state, validation_class: UTF8Type}
{column_name: gender, validation_class: UTF8Type}
{column_name: birth_year, validation_class: LongType}
];
```

Next, create a *dynamic* column family called `blog_entry`. Notice that here we do not specify column definitions as the column names are expected to be supplied later by the client application.

```
[default@demo] CREATE COLUMN FAMILY blog_entry
WITH comparator = TimeUUIDType
AND key_validation_class=UTF8Type
AND default_validation_class = UTF8Type;
```

Creating a Counter Column Family

A counter column family contains counter columns. A counter column is a specific kind of column whose user-visible value is a 64-bit signed integer that can be incremented (or decremented) by a client application. The counter column tracks the most recent value (or count) of all updates made to it. A counter column cannot be mixed in with regular columns of a column family, you must create a column family specifically to hold counters.

To create a column family that holds counter columns, set the `default_validation_class` of the column family to `CounterColumnType`. For example:

```
[default@demo] CREATE COLUMN FAMILY page_view_counts
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND comparator=UTF8Type;
```

To insert a row and counter column into the column family (with the initial counter value set to 0):

```
[default@demo] INCR page_view_counts['www.datastax.com']['home'] BY 0;
```

To increment the counter:

```
[default@demo] INCR page_view_counts['www.datastax.com']['home'] BY 1;
```

Inserting Rows and Columns

The following examples illustrate using the `SET` command to insert columns for a particular row key into the `users` column family. In this example, the row key is `bobbyjo` and we are setting each of the columns for this user. Notice that you can only set one column at a time in a `SET` command.

```
[default@demo] SET users['bobbyjo']['full_name']='Robert Jones';
[default@demo] SET users['bobbyjo']['email']='bobjones@gmail.com';
[default@demo] SET users['bobbyjo']['state']='TX';
[default@demo] SET users['bobbyjo']['gender']='M';
[default@demo] SET users['bobbyjo']['birth_year']='1975';
```

In this example, the row key is `yomama` and we are just setting some of the columns for this user.

```
[default@demo] SET users['yomama']['full_name']='Cathy Smith';
[default@demo] SET users['yomama']['state']='CA';
[default@demo] SET users['yomama']['gender']='F';
[default@demo] SET users['yomama']['birth_year']='1969';
```

In this example, we are creating an entry in the `blog_entry` column family for row key `yomama`:

```
[default@demo] SET blog_entry['yomama'][timeuuid()] = 'I love my new shoes!';
```

Note

The Cassandra CLI sets the consistency level for the client. The level defaults to `ONE` for all write and read operations. For more information, see [About Data Consistency in Cassandra](#).

Reading Rows and Columns

Getting Started Using the Cassandra CLI

Use the GET command within Cassandra CLI to retrieve a particular row from a column family. Use the LIST command to return a batch of rows and their associated columns (default limit of rows returned is 100).

For example, to return the first 100 rows (and all associated columns) from the users column family:

```
[default@demo] LIST users;
```

Cassandra stores all data internally as hex byte arrays by default. If you do not specify a default row key validation class, column comparator and column validation class when you define the column family, Cassandra CLI will expect input data for row keys, column names, and column values to be in hex format (and data will be returned in hex format).

To pass and return data in human-readable format, you can pass a value through an encoding function. Available encodings are:

- `ascii`
- `bytes`
- `integer` (a generic variable-length integer type)
- `lexicalUUID`
- `long`
- `utf8`

For example to return a particular row key and column in UTF8 format:

```
[default@demo] GET users[utf8('bobbyjo')][utf8('full_name')];
```

You can also use the ASSUME command to specify the encoding in which column family data should be returned for the entire client session. For example, to return row keys, column names, and column values in ASCII-encoded format:

```
[default@demo] ASSUME users KEYS AS ascii;  
[default@demo] ASSUME users COMPARATOR AS ascii;  
[default@demo] ASSUME users VALIDATOR AS ascii;
```

Setting an Expiring Column

When you set a column in Cassandra, you can optionally set an expiration time, or *time-to-live* (TTL) attribute for it.

For example, suppose we are tracking coupon codes for our users that expire after 10 days. We can define a `coupon_code` column and set an expiration date on that column. For example:

```
[default@demo] SET users['bobbyjo']  
[utf8('coupon_code')] = utf8('SAVE20') WITH ttl=864000;
```

After ten days, or 864,000 seconds have elapsed since the setting of this column, its value will be marked as deleted and no longer be returned by read operations. Note, however, that the value is not actually deleted from disk until normal Cassandra compaction processes are completed.

Indexing a Column

The CLI can be used to create secondary indexes (indexes on column values). You can add a secondary index when you create a column family or add it later using the UPDATE COLUMN FAMILY command.

For example, to add a secondary index to the `birth_year` column of the users column family:

```
[default@demo] UPDATE COLUMN FAMILY users  
WITH comparator = UTF8Type  
AND column_metadata = [{column_name: birth_year, validation_class: LongType, index_type: KEYS}];
```

Because of the secondary index created for the column `birth_year`, its values can be queried directly for users born in a given year as follows:

```
[default@demo] GET users WHERE birth_year = 1969;
```

Deleting Rows and Columns

The Cassandra CLI provides the DEL command to delete a row or column (or subcolumn).

For example, to delete the coupon_code column for the yomama row key in the users column family:

```
[default@demo] DEL users ['yomama']['coupon_code'];
```

```
[default@demo] GET users ['yomama'];
```

Or to delete an entire row:

```
[default@demo] DEL users ['yomama'];
```

Dropping Column Families and Keyspaces

With Cassandra CLI commands you can drop column families and keyspaces in much the same way that tables and databases are dropped in a relational database. This example shows the commands to drop our example users column family and then drop the demo keyspace altogether:

```
[default@demo] DROP COLUMN FAMILY users;
```

```
[default@demo] DROP KEYSPACE demo;
```

Querying Cassandra

Cassandra CQL now has parity with the legacy API and command line interface (CLI) that has shipped with Cassandra for several years, making CQL the primary interface into the DBMS. CQL 3 is beta in Cassandra 1.1. The released version will be available in Cassandra 1.2.

CLI commands have complements in CQL based on the CQL specification 3. CQL specification 2 remains the default because CQL 2 applications are incompatible with the CQL specification 3. The *compact storage directive* used with the CREATE TABLE command provides backward compatibility with older Cassandra applications; new applications should avoid it.

You activate the CQL specification in one of these ways:

- Start the CQL shell utility using the cql<specification number> *startup option*.
- Use the set_sql_version Thrift method.
- Specify the desired CQL version in the connect() call to the Python driver:

```
connection = cql.connect('localhost:9160', cql_version='3.0')
```

CQL Specification 3 supports *composite columns*. By virtue of the composite primary keys feature in CQL, wide rows are supported with full denormalization.

CQL Specification 2 supports dynamic columns, but not composite columns.

Both versions of CQL support only a subset of all the available column family storage properties. Also, Super Columns are not supported by either CQL version; column_type and subcomparator arguments are not valid.

Using CQL

Developers can access CQL commands in a variety of ways. Drivers are available for Python, PHP, Ruby, Node.js, and JDBC-based client programs. For the purposes of administrators, using the Python-based CQLsh command-line client is

the most direct way to run simple CQL commands. Using the CQLsh client, you can run CQL commands from the command line. The CQLsh client is installed with Cassandra in `<install_location>/bin/cqlsh` for tarball installations, or `/usr/bin/cqlsh` for packaged installations.

When you start CQLsh, you can provide the IP address of a Cassandra node to connect to. The default is `localhost`. You can also provide the RPC connection port (default is 9160), and the cql specification number.

Starting CQLsh Using the CQL 2 Specification

Linux

To start CQLsh 2.0.0 using the default CQL 2 Specification from the Cassandra bin directory on Linux, for example:

```
./cqlsh
```

Windows

To start CQLsh 2.0.0 using the default CQL 2 Specification from the Cassandra bin directory on Windows in the Command Prompt, for example:

```
python cqlsh
```

Starting CQLsh Using the CQL 3 Specification

Linux

To start CQLsh using the CQL 3 Specification on Linux, for example, navigate to the bin directory and enter this command:

```
./cqlsh -3
```

To start CQLsh on a different node, specify the IP address and port:

```
./cqlsh 1.2.3.4 9160 -3
```

Windows

To start CQLsh using the CQL 3 Specification on Windows, for example, in Command Prompt navigate to the bin directory and enter this command:

```
python cqlsh -3
```

To start CQLsh on a different node, specify the IP address and port:

```
python cqlsh 1.2.3.4 9160 -3
```

Windows and Linux

To exit CQLsh:

```
cqlsh> exit
```

Using CQL Commands

Commands in CQL combine SQL-like syntax that maps to Cassandra concepts and operations. CQL 3, like CQL 2, supports [tab completion](#). Some platforms, such as Mac OSX, do not ship with tab completion installed. You can use [easy_install](#) to install tab completion capabilities on OSX:

```
easy_install readline
```

This section presents an overview of the CQL 3 commands. These commands are described in detail in the [CQL Reference](#).

Note

The examples in this documentation use CQL 3 and, in some cases, do not work if you use the default CQL 2.

For a description of CQL 2, see [the CQL reference for Cassandra 1.0](#).

Creating a Keyspace

If you haven't already started CQLsh 3, do so.

```
./cqlsh -3
```

To create a keyspace, the CQL counterpart to the SQL database, use the CREATE KEYSPACE command.

Note

This example works with pre-release CQL 3 in Cassandra 1.1.x. The syntax differs in the release version of CQL 3 in Cassandra 1.2 and later.

```
cqlsh> CREATE KEYSPACE demodb
      WITH strategy_class = 'SimpleStrategy'
      AND strategy_options:replication_factor='1';
```

The strategy_class keyspace option must be enclosed in single quotation marks. For more information about keyspace options, see [About Replication in Cassandra](#).

Using the Keyspace

After creating a keyspace, select the keyspace for use, just as you connect to a database in SQL:

```
cqlsh> USE demodb;
```

Next, create a column family, the counterpart to a table in SQL, and then populate it with data.

Creating a Column Family

To create a *users* column family in the newly created keyspace:

```
cqlsh:demodb> CREATE TABLE users (
      user_name varchar,
      password varchar,
      gender varchar,
      session_token varchar,
      state varchar,
      birth_year bigint,
      PRIMARY KEY (user_name)
    );
```

The users column family has a single primary key.

After executing the CREATE TABLE command, if you get no output, all is well. If you get an error message, you probably started CQLsh 2 instead of CQLsh 3.

Inserting and Retrieving Columns

In production scenarios, inserting columns and column values programmatically is more practical than using CQLsh, but often, being able to test queries using this SQL-like shell is very convenient.

The following example shows how to use CQLsh to create and then get a user record for "jsmith." The record includes a value for the password column. The user name "jsmith" is the row key, or in CQL terms, the primary key.

```
cqlsh:demodb> INSERT INTO users
    (user_name, password)
    VALUES ('jsmith', 'ch@ngem3a');

cqlsh:demodb> SELECT * FROM users WHERE user_name='jsmith';
```

Using the Keyspace Qualifier

Sometimes it is inconvenient to have to issue a USE statement to select a keyspace. If you use connection pooling, for example, you have multiple keyspaces to juggle. Simplify tracking multiple keyspaces using the keyspace qualifier. Use the name of the keyspace followed by a period, then the column family name. For example, History.timeline.

```
cqlsh:demodb> INSERT INTO History.timeline (user_id, tweet_id, author, body)
    VALUES (gmason, 1765, phenry, 'Give me liberty or give me death');
```

You can specify the keyspace you want to use in these statements:

- SELECT
- CREATE
- ALTER
- DELETE
- TRUNCATE

For more information, see the [CQL Reference](#).

Using Composite Primary Keys

Use a composite primary key when you need wide row support or when you want to create columns that you can query to return sorted results. To create a column family having a composite primary key:

```
cqlsh:demodb> CREATE TABLE emp (
    empID int,
    deptID int,
    first_name varchar,
    last_name varchar,
    PRIMARY KEY (empID, deptID)
);
```

The composite primary key is made up of the empID and deptID columns. The empID acts as a partition key for distributing data in the column family among the various nodes that comprise the cluster. The remaining column, the deptID, in the primary key acts as a clustering mechanism and ensures the data is stored in ascending order on disk (much like a clustered index in Microsoft SQL Server works). For more information about composite keys, see the [Composite Columns](#) section.

To insert data into the column family:

```
cqlsh:demodb> INSERT INTO emp (empID, deptID, first_name, last_name)
    VALUES (104, 15, 'jane', 'smith');
```

If you want to specify a keyspace other than the one you're using, prefix the keyspace name followed by a period (.) to the column family name:

```
cqlsh> INSERT INTO demodb.emp
    (empID, deptID, first_name, last_name)
    VALUES (130, 5, 'sughit', 'singh');
```

Retrieving and Sorting Results

Similar to a SQL query, follow the WHERE clause by an ORDER BY clause to retrieve and sort results:

```
cqlsh:demodb> SELECT * FROM emp WHERE empID IN (130,104) ORDER BY deptID DESC;
```

empid	deptid	first_name	last_name
104	15	jane	smith
130	5	sughit	singh

```
cqlsh:demodb> SELECT * FROM emp where empID IN (130,104) ORDER BY deptID ASC;
```

empid	deptid	first_name	last_name
130	5	sughit	singh
104	15	jane	smith

See the *tweets example* for more information about using composite primary keys.

Adding Columns with ALTER TABLE

The ALTER TABLE command adds new columns to a column family. For example, to add a coupon_code column with the varchar validation type to the users column family:

```
cqlsh:demodb> ALTER TABLE users ADD coupon_code varchar;
```

This creates the column metadata and adds the column to the column family schema, but does not update any existing rows.

Altering Column Metadata

Using ALTER TABLE, you can change the type of a column after it is defined or added to a column family. For example, to change the coupon_code column to store coupon codes as integers instead of text, change the validation type as follows:

```
cqlsh:demodb> ALTER TABLE users ALTER coupon_code TYPE int;
```

Only newly inserted values, not existing coupon codes are validated against the new type.

Specifying Column Expiration with TTL

Both the INSERT and UPDATE commands support setting a column expiration time (TTL). In the INSERT example above for the key jsmith we set the password column to expire at 86400 seconds, or one day. If we wanted to extend the expiration period to five days, we could use the UPDATE command as shown:

```
cqlsh:demodb> INSERT INTO users
    (user_name, password)
VALUES ('cbrown', 'ch@ngem4a') USING TTL 86400;
```

```
cqlsh:demodb> UPDATE users USING TTL 432000 SET 'password' = 'ch@ngem4a'
WHERE user_name = 'cbrown';
```

Dropping Column Metadata

To remove a column's metadata entirely, including the column name and validation type, use ALTER TABLE <columnFamily> DROP <column>. The following command removes the name and validator without

affecting or deleting any existing data:

```
cqlsh:demodb> ALTER TABLE users DROP coupon_code;
```

After you run this command, clients can still add new columns named `coupon_code` to the `users` column family, but the columns are not validated until you explicitly add a type.

Indexing a Column

CQLsh can be used to create secondary indexes, or indexes on column values. This example creates an index on the `state` and `birth_year` columns in the `users` column family.

```
cqlsh:demodb> CREATE INDEX state_key ON users (state);
cqlsh:demodb> CREATE INDEX birth_year_key ON users (birth_year);
```

Because of the secondary index created for the two columns, their values can be queried directly as follows:

```
cqlsh:demodb> SELECT * FROM users
                WHERE gender='f' AND
                state='TX' AND
                birth_year='1968';
```

Deleting Columns and Rows

CQLsh provides the `DELETE` command to delete a column or row. This example deletes user `jsmith`'s session token column, and then delete `jsmith`'s entire row.

```
cqlsh:demodb> DELETE session_token FROM users where pk = 'jsmith';
cqlsh:demodb> DELETE FROM users where pk = 'jsmith';
```

Dropping Column Families and Keyspaces

Using CQLsh commands, you can drop column families and keyspaces in much the same way that tables and databases are dropped in relational models. This example shows the commands to drop our example `users` column family and then drop the `demodb` keyspace:

```
cqlsh:demodb> DROP TABLE users;
cqlsh:demodb> DROP KEYSPACE demodb;
```

Paging through Non-ordered Partitioner Results

When using the `RandomPartitioner`, Cassandra rows are ordered by the MD5 hash of their value and hence the order of rows is not meaningful. Despite that fact, given the following definition:

```
CREATE TABLE test (
  k int PRIMARY KEY,
  v1 int,
  v2 int
);
```

and assuming `RandomPartitioner`, CQL2 allowed queries like:

```
SELECT * FROM test WHERE k > 42;
```

The semantics of such a query was to query all rows for which the MD5 of the key was bigger than the MD5 of 42. The query would return results where $k \leq 42$, which is unintuitive. CQL3 forbids such a query unless the partitioner in use is ordered.

Even when using the random partitioner, it can sometimes be useful to page through all rows. For this purpose, CQL3 includes the token function:

```
SELECT * FROM test WHERE token(k) > token(42);
```

The ByteOrdered partitioner arranges tokens the same way as key values, but the RandomPartitioner distributes tokens in a completely unordered manner. Using the token function actually queries results directly using tokens. Underneath, the token function makes token-based comparisons and does not convert keys to tokens (not $k > 42$).

Querying System Tables

An alternative to the Thrift API `describe_keyspaces` function is querying the system tables directly in CQL 3. For example, you can query the defined keyspaces as follows:

```
SELECT * from system.schema_keyspaces;
```

The output includes information about defined keyspaces. For example:

keyspace	durable_writes	name	strategy_class	strategy_options
history	True	history	SimpleStrategy	{"replication_factor":"1"}
ks_info	True	ks_info	SimpleStrategy	{"replication_factor":"1"}

You can also retrieve information about tables by querying `system.schema_columnfamilies` and about column metadata by querying `system.schema_columns`.

Configuration

Like any modern server-based software, Cassandra has a number of configuration options to tune the system towards specific workloads and environments. Substantial efforts have been made to provide meaningful default configuration values, but given the inherently complex nature of distributed systems coupled with the wide variety of possible workloads, most production deployments will require some modifications of the default configuration.

Node and Cluster Configuration (cassandra.yaml)

The `cassandra.yaml` file is the main configuration file for Cassandra. It is located in the following directories:

- Cassandra packaged installs: `/etc/cassandra/conf`
- Cassandra binary installs: `<install_location>/conf`
- DataStax Enterprise packaged installs: `/etc/dse/cassandra`
- DataStax Enterprise binary installs: `<install_location>/resources/cassandra/conf`

After changing properties in this file, you must restart the node for the changes to take effect.

Note

** Some default values are set at the class level and may be missing or commented out in the `cassandra.yaml` file. Additionally, values in commented out options may not match the default value: they are the recommended value when changing from the default.

Option	Option
authenticator	max_hint_window_in_ms
authority	memtable_flush_queue_size
authorizer	memtable_flush_writers
auth_replication_options	memtable_total_space_in_mb
auth_replication_strategy	partitioner
auto_bootstrap	permissions_validity_in_ms
auto_snapshot	phi_convict_threshold
broadcast_address	populate_io_cache_on_flush
cluster_name	reduce_cache_capacity_to
column_index_size_in_kb	reduce_cache_sizes_at
commitlog_directory	request_scheduler
commitlog_segment_size_in_mb	request_scheduler_id
commitlog_sync	request_scheduler_options
commitlog_total_space_in_mb	row_cache_keys_to_save
compaction_preheat_key_cache	row_cache_provider
compaction_throughput_mb_per_sec	row_cache_size_in_mb
concurrent_reads	rpc_address
concurrent_writes	rpc_keepalive
data_file_directories	rpc_max_threads

dynamic_snitch_badness_threshold	rpc_min_threads
dynamic_snitch_reset_interval_in_ms	rpc_port
dynamic_snitch_update_interval_in_ms	rpc_recv_buff_size_in_bytes
encryption_options	rpc_send_buff_size_in_bytes
endpoint_snitch	rpc_server_type
flush_largest_memtables_at	rpc_timeout_in_ms
hinted_handoff_enabled	saved_caches_directory
hinted_handoff_throttle_delay_in_ms	seed_provider
in_memory_compaction_limit_in_mb	snapshot_before_compaction
incremental_backups	ssl_storage_port
index_interval	storage_port
initial_token	stream_throughput_outbound_megabits_per_sec
key_cache_keys_to_save	streaming_socket_timeout_in_ms
key_cache_save_period	thrift_framed_transport_size_in_mb
key_cache_size_in_mb	thrift_max_message_length_in_mb
listen_address	trickle_fsync

Node and Cluster Initialization Properties

The following properties are used to initialize a new cluster or when introducing a new node to an established cluster. They control how a node is configured within a cluster, including inter-node communication, data partitioning, and replica placement. DataStax recommends that you carefully evaluate your requirements and make any changes before starting a node for the first time.

auto_bootstrap

(Default: `true`) This setting has been removed from default configuration. It makes new (non-seed) nodes automatically migrate the right data to themselves. It is referenced here because `auto_bootstrap: true` is explicitly added to the `cassandra.yaml` file in an [AMI installation](#). Setting this property to false is not recommended and is necessary only in rare instances.

broadcast_address

(Default: `listen_address`**) If your Cassandra cluster is deployed across multiple Amazon EC2 regions and you use the [EC2MultiRegionSnitch](#), set the `broadcast_address` to public IP address of the node and the `listen_address` to the private IP.

cluster_name

(Default: `Test Cluster`) The name of the cluster; used to prevent machines in one logical cluster from joining another. All nodes participating in a cluster must have the same value.

commitlog_directory

(Default: `/var/lib/cassandra/commitlog`) The directory where the commit log is stored. For optimal write performance, DataStax recommends the commit log be on a separate disk partition (ideally, a separate physical device) from the data file directories.

data_file_directories

(Default: `/var/lib/cassandra/data`) The directory location where column family data (SSTables) are stored.

initial_token

(Default: N/A) The initial token assigns the node token position in the ring, and assigns a range of data to the node when it first starts up. If the initial token is left unset when introducing a new node to an established cluster, Cassandra requests a token that bisects the range of the heaviest-loaded existing node. If no load information is available (for example in a new cluster), Cassandra picks a random token, which may lead to hot spots. For information about calculating tokens that position nodes in the ring, see *Generating Tokens*.

listen_address

(Default: `localhost`) The IP address or hostname that other Cassandra nodes use to connect to this node. If left unset, the hostname must resolve to the IP address of this node using `/etc/hostname`, `/etc/hosts`, or DNS. Do not specify `0.0.0.0`.

partitioner

(Default: `org.apache.cassandra.dht.RandomPartitioner`) Distributes rows (by key) across nodes in the cluster. Any `IPartitioner` may be used, including your own as long as it is on the classpath. Cassandra provides the following partitioners:

- *org.apache.cassandra.dht.RandomPartitioner*
- *org.apache.cassandra.dht.ByteOrderedPartitioner*
- `org.apache.cassandra.dht.OrderPreservingPartitioner` (deprecated)
- `org.apache.cassandra.dht.CollatingOrderPreservingPartitioner` (deprecated)

rpc_address

(Default: `localhost`) The listen address for client connections (Thrift remote procedure calls). Valid values are:

- **0.0.0.0**: Listens on all configured interfaces.
- **IP address**
- **hostname**
- **unset**: Resolves the address using the hostname configuration of the node.

If left unset, the hostname must resolve to the IP address of this node using `/etc/hostname`, `/etc/hosts`, or DNS.

Note

In DataStax Enterprise 3.0.x, the default is `0.0.0.0`.

rpc_port

(Default: `9160`) The port for the Thrift RPC service, which is used for client connections.

saved_caches_directory

(Default: `/var/lib/cassandra/saved_caches`) The directory location where column family key and row caches are stored.

seed_provider

(Default: `org.apache.cassandra.locator.SimpleSeedProvider`) A list of comma-delimited hosts (IP addresses) to use as contact points when a node joins a cluster. Cassandra also uses this list to learn the topology of the ring. When running multiple nodes, you must change the `- seeds` list from the default value (127.0.0.1). In multiple data-center clusters, the `- seeds` list should include at least one node from each data center (replication group).

storage_port

(Default: 7000) The port for inter-node communication.

endpoint_snitch

(Default: `org.apache.cassandra.locator.SimpleSnitch`) Sets which snitch Cassandra uses for locating nodes and routing requests. For descriptions of the snitches, see [About Snitches](#). In DataStax Enterprise, the default snitch is `com.datastax.bdp.snitch.DseDelegateSnitch`.

Global Row and Key Caches Properties

When creating or modifying column families, you enable or disable the key or row caches at the column family level by setting the [caching parameter](#). Other row and key cache tuning and configuration options are set at the global (node) level. Cassandra uses these settings to automatically distribute memory for each column family based on the overall workload and specific column family usage. You can also configure the save periods for these caches globally. For more information, see [Tuning Data Caches](#).

key_cache_keys_to_save

(Default: disabled - all keys are saved**) Number of keys from the key cache to save.

key_cache_save_period

(Default: 14400- 4 hours) Duration in seconds that keys are saved in cache. Caches are saved to [saved_caches_directory](#). Saved caches greatly improve cold-start speeds and has relatively little effect on I/O.

key_cache_size_in_mb

(Default: empty, which automatically sets it to the smaller of 5% of the available heap, or 100MB) A global cache setting for column families. It is the maximum size of the key cache in memory. To disable set to 0.

row_cache_keys_to_save

(Default: disabled - all keys are saved**) Number of keys from the row cache to save.

row_cache_size_in_mb

(Default: 0 - disabled) A global cache setting for column families. Holds the entire row in memory so reads can be satisfied without using disk

row_cache_save_period

(Default: 0 - disabled) Duration in seconds that rows are saved in cache. Caches are saved to [saved_caches_directory](#).

row_cache_provider

(Default: `SerializingCacheProvider`) Specifies what kind of implementation to use for the row cache.

- **SerializingCacheProvider:** Serializes the contents of the row and stores it in native memory, that is, off the JVM Heap. Serialized rows take significantly less memory than live rows in the JVM, so you can cache more rows in a given memory footprint. Storing the cache off-heap means you can use smaller heap sizes, which reduces the impact of garbage collection pauses. It is valid to specify the fully-qualified class name to a class that implements `org.apache.cassandra.cache.ILogCacheProvider`.
- **ConcurrentLinkedHashCacheProvider:** Rows are cached using the JVM heap, providing the same row cache behavior as Cassandra versions prior to 0.8.

The `SerializingCacheProvider` is 5 to 10 times more memory-efficient than `ConcurrentLinkedHashCacheProvider` for applications that are not blob-intensive. However, `SerializingCacheProvider` may perform worse in update-heavy workload situations because it invalidates cached rows on update instead of updating them in place as `ConcurrentLinkedHashCacheProvider` does.

Performance Tuning Properties

The following properties are used to tune performance and system resource utilization, such as memory, disk I/O, and CPU, for reads and writes.

column_index_size_in_kb

(Default: 64) Add column indexes to a row when the data reaches this size. This value defines how much row data must be deserialized to read the column. Increase this setting if your column values are large or if you have a very large number of columns. If consistently reading only a few columns from each row or doing many partial-row reads, keep it small. All index data is read for each access, so take that into consideration when setting the index size.

commitlog_segment_size_in_mb

(Default: 32 for 32-bit JVMs, 1024 for 64-bit JVMs) Sets the size of the individual commitlog file segments. A commitlog segment may be archived, deleted, or recycled after all its data has been flushed to SSTables. This amount of data can potentially include commitlog segments from every column family in the system. The default size is usually suitable for most commitlog archiving, but if you want a finer granularity, 8 or 16 MB is reasonable. See [Commit Log Archive Configuration](#).

commitlog_sync

(Default: periodic) The method that Cassandra uses to acknowledge writes in milliseconds:

- **periodic:** Used with `commitlog_sync_period_in_ms` (default: 10000 - 10 seconds) to control how often the commit log is synchronized to disk. Periodic syncs are acknowledged immediately.
- **batch:** Used with `commitlog_sync_batch_window_in_ms` (default: disabled**) to control how long Cassandra waits for other writes before performing a sync. When using this method, writes are not acknowledged until fsynced to disk.

commitlog_total_space_in_mb

(Default: 32 for 32-bit JVMs, 1024 for 64-bit JVMs**) Total space used for commitlogs. If the used space goes above this value, Cassandra rounds up to the next nearest segment multiple and flushes memtables to disk for the oldest commitlog segments, removing those log segments. This reduces the amount of data to replay on startup, and prevents infrequently-updated tables from indefinitely keeping commitlog segments. A small total commitlog space tends to cause more flush activity on less-active column families.

compaction_preheat_key_cache

(Default: true) When set to `true`, cached row keys are tracked during compaction, and re-cached to their new positions in the compacted SSTable. If you have extremely large key caches for your column families, set to `false`; see [Global Row and Key Caches Properties](#).

compaction_throughput_mb_per_sec

(Default: 16) Throttles compaction to the given total throughput across the entire system. The faster you insert data, the faster you need to compact in order to keep the SSTable count down. The recommended Value is 16 to 32 times the rate of write throughput (in MBs/second). Setting to 0 disables compaction throttling.

concurrent_compactors

(Default: 1 per CPU core**) Sets the number of concurrent compaction processes allowed to run simultaneously on a node, not including validation compactions for anti-entropy repair. Simultaneous compactions help preserve read performance in a mixed read-write workload by mitigating the tendency of small SSTables to accumulate during a single long-running compaction. If compactions run too slowly or too fast, change ***compaction_throughput_mb_per_sec*** first.

concurrent_reads

(Default: 32) For workloads with more data than can fit in memory, the bottleneck is reads fetching data from disk. Setting to (16 * number_of_drives) allows operations to queue low enough in the stack so that the OS and drives can reorder them.

concurrent_writes

(Default: 32) Writes in Cassandra are rarely I/O bound, so the ideal number of concurrent writes depends on the number of CPU cores in your system. The recommended value is (8 * number_of_cpu_cores).

flush_largest_memtables_at

(Default: 0.75) When Java heap usage (after a full concurrent mark sweep (CMS) garbage collection) exceeds the set value, Cassandra flushes the largest memtables to disk to free memory. This parameter is an emergency measure to prevent sudden out-of-memory (OOM) errors. Do **not** use it as a tuning mechanism. It is most effective under light to moderate loads or read-heavy workloads; it will fail under massive write loads. A value of 0.75 flushes memtables when Java heap usage is above 75% total heap size. Set to 1.0 to disable. Other emergency measures are ***reduce_cache_capacity_to*** and ***reduce_cache_sizes_at***.

in_memory_compaction_limit_in_mb

(Default: 64) Size limit for rows being compacted in memory. Larger rows spill to disk and use a slower two-pass compaction process. When this occurs, a message is logged specifying the row key. The recommended value is 5 to 10 percent of the available Java heap size.

index_interval

(Default: 128) Controls the sampling of entries from the primary row index. The interval corresponds to the number of index entries that are skipped between taking each sample. By default Cassandra samples one row key out of every 128. The larger the interval, the smaller and less effective the sampling. The larger the sampling, the more effective the index, but with increased memory usage. Generally, the best trade off between memory usage and performance is a value between 128 and 512 in combination with a large table key cache. However, if you have small rows (many to an OS page), you may want to increase the sample size, which often lowers memory usage without an impact on performance. For large rows, decreasing the sample size may improve read performance.

memtable_flush_queue_size

(Default: 4) The number of full memtables to allow pending flush, that is, waiting for a writer thread. At a minimum, set to the maximum number of secondary indexes created on a single column family.

memtable_flush_writers

(Default: 1 per data directory**) Sets the number of memtable flush writer threads. These threads are blocked by disk I/O, and each one holds a memtable in memory while blocked. If you have a large Java heap size and many *data directories*, you can increase the value for better flush performance.

memtable_total_space_in_mb

(Default: 1/3 of the heap**) Specifies the total memory used for all column family memtables on a node. This replaces the per-column family storage settings `memtable_operations_in_millions` and `memtable_throughput_in_mb`.

populate_io_cache_on_flush

(Default: `false`**) Populates the page cache on memtable flush and compaction. Enable this setting **only** when the whole node's data fits in memory.

reduce_cache_capacity_to

(Default: 0.6) Sets the size percentage to which maximum cache capacity is reduced when Java heap usage reaches the threshold defined by `reduce_cache_sizes_at`. Together with `flush_largest_memtables_at`, these properties are an emergency measure for preventing sudden out-of-memory (OOM) errors.

reduce_cache_sizes_at

(Default: 0.85) When Java heap usage (after a full concurrent mark sweep (CMS) garbage collection) exceeds this percentage, Cassandra reduces the cache capacity to the fraction of the current size as specified by `reduce_cache_capacity_to`. To disable set to 1.0.

stream_throughput_outbound_megabits_per_sec

(Default: 400**) Throttles all outbound streaming file transfers on a node to the specified throughput. Cassandra does mostly sequential I/O when streaming data during bootstrap or repair, which can lead to saturating the network connection and degrading client (RPC) performance.

trickle_fsync

(Default: `false`) When doing sequential writing, enabling this option tells `fsync` to force the operating system to flush the dirty buffers at a set interval (`trickle_fsync_interval_in_kb` [default: 10240]). Enable this parameter to avoid sudden dirty buffer flushing from impacting read latencies. Recommended to use on SSDs, but not on HDDs.

Remote Procedure Call (RPC) Tuning Properties

The following properties are used to configure and tune remote procedure calls (client connections).

request_scheduler

(Default: `org.apache.cassandra.scheduler.NoScheduler`) Defines a scheduler to handle incoming client requests according to a defined policy. This scheduler is useful for throttling client requests in single clusters containing multiple keyspaces. Valid values are:

- **`org.apache.cassandra.scheduler.NoScheduler`**: No scheduling takes place and does not have any options.
- **`org.apache.cassandra.scheduler.RoundRobinScheduler`**: See `request_scheduler_options` properties.
- A Java class that implements the `RequestScheduler` interface.

request_scheduler_id

(Default: `keyspace`) An identifier on which to perform request scheduling. Currently the only valid value is `keyspace`.

request_scheduler_options

(Default: disabled**) Contains a list of properties that define configuration options for `request_scheduler`:

- **throttle_limit**: (Default: 80) The number of active requests per client. Requests beyond this limit are queued up until running requests complete. Recommended value is $((\text{concurrent_reads} + \text{concurrent_writes}) * 2)$.
- **default_weight**: (Default: 1**) How many requests are handled during each turn of the RoundRobin.
- **weights**: (Default: 1 or default_weight) How many requests are handled during each turn of the RoundRobin, based on the `request_scheduler_id`. Takes a list of keyspaces: weights.

rpc_keepalive

(Default: `true`) Enable or disable keepalive on client connections.

rpc_max_threads

(Default: unlimited**) Cassandra uses one thread-per-client for remote procedure calls. For a large number of client connections, this can cause excessive memory usage for the thread stack. Connection pooling on the client side is highly recommended. Setting a maximum thread pool size acts as a safeguard against misbehaved clients. If the maximum is reached, Cassandra will block additional connections until a client disconnects.

rpc_min_threads

(Default: 16**) Sets the minimum thread pool size for remote procedure calls.

rpc_recv_buff_size_in_bytes

(Default: N/A**) Sets the receiving socket buffer size for remote procedure calls.

rpc_send_buff_size_in_bytes

(Default: N/A**) Sets the sending socket buffer size for remote procedure calls.

rpc_timeout_in_ms

(Default: 10000) The time in milliseconds that a node will wait on a reply from other nodes before the command is failed.

streaming_socket_timeout_in_ms

(Default: 0 - never timeout streams**) Enable or disable socket timeout for streaming operations. When a timeout occurs during streaming, streaming is retried from the start of the current file. Avoid setting this value too low, as it can result in a significant amount of data re-streaming.

rpc_server_type

(Default: `sync`) Cassandra provides three options for the RPC server. On Windows, `sync` is about 30% slower than `hsha`. On Linux, `sync` and `hsha` performance is about the same, but `hsha` uses less memory.

- **sync**: (Default) One connection per thread in the RPC pool. For a very large number of clients, memory is the limiting factor. On a 64 bit JVM, 128KB is the minimum stack size per thread. Connection pooling is strongly recommended.
- **hsha**: Half synchronous, half asynchronous. The RPC thread pool is used to manage requests, but the threads are multiplexed across the different clients.

- **async:** (Deprecated) Nonblocking server implementation with one thread to serve RPC connections. It is not recommended for high throughput use cases. It is about 50% slower than `sync` or `hsha`. It will be removed in the 1.2 release.

thrift_framed_transport_size_in_mb

(Default: 15) Frame size (maximum field length) for Thrift.

thrift_max_message_length_in_mb

(Default: 16) The maximum length of a Thrift message in megabytes, including all fields and internal Thrift overhead.

Inter-node Communication and Fault Detection Properties

dynamic_snitch_badness_threshold

(Default: 0.0) Sets the performance threshold for dynamically routing requests away from a poorly performing node. A value of 0.2 means Cassandra continues to prefer the static snitch values until the node response time is 20% worse than the best performing node. Until the threshold is reached, incoming client requests are statically routed to the closest replica (as determined by the snitch). Having requests consistently routed to a given replica can help keep a working set of data hot when *read repair* is less than 1.

dynamic_snitch_reset_interval_in_ms

(Default: 600000) Time interval in milliseconds to reset all node scores, which allows a bad node to recover.

dynamic_snitch_update_interval_in_ms

(Default: 100) The time interval in milliseconds for calculating read latency.

hinted_handoff_enabled

(Default: `true`) Enables or disables hinted handoff. A hint indicates that the write needs to be replayed to an unavailable node. Where Cassandra writes the hint depends on the version:

- Prior to 1.0: Writes to a live replica node.
- 1.0 and later: Writes to the coordinator node.

max_hint_window_in_ms

(Default: 3600000 - 1 hour) Defines how long in milliseconds to generate and save hints for an unresponsive node. After this interval, new hints are no longer generated until the node is back up and responsive. If the node goes down again, a new interval begins. This setting can prevent a sudden demand for resources when a node is brought back online and the rest of the cluster attempts to replay a large volume of hinted writes.

hinted_handoff_throttle_delay_in_ms

(Default: 1) When a node detects that a node for which it is holding hints has recovered, it begins sending the hints to that node. This setting specifies the sleep interval in milliseconds after delivering each hint.

phi_convict_threshold

(Default: 8 **) Adjusts the sensitivity of the failure detector on an exponential scale. Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures will cause a node failure. In unstable network environments (such as EC2 at times), raising the value to 10 or 12 helps

prevent false failures. Values higher than 12 and lower than 5 are not recommended.

Automatic Backup Properties

auto_snapshot

(Default: `true`) Enable or disable whether a snapshot is taken of the data before keyspace truncation or dropping of column families. To prevent data loss, using the default setting is **strongly** advised. If you set to `false`, you will lose data on truncation or drop.

incremental_backups

(Default: `false`) Backs up data updated since the last snapshot was taken. When enabled, Cassandra creates a hard link to each SSTable flushed or streamed locally in a `backups/` subdirectory of the keyspace data. Removing these links is the operator's responsibility.

snapshot_before_compaction

(Default: `false`) Enable or disable taking a snapshot before each compaction. This option is useful to back up data when there is a data format change. Be careful using this option because Cassandra does not clean up older snapshots automatically.

Security Properties

authenticator

(Default: `org.apache.cassandra.auth.AllowAllAuthenticator`) The authentication backend. It implements `IAuthenticator`, which is used to identify users.

The following authenticator options are only available in DataStax Enterprise 3.0:

- `com.datastax.bdp.cassandra.auth.PasswordAuthenticator`
- `com.datastax.bdp.cassandra.auth.KerberosAuthenticator`

authority

For backwards compatibility only.

authorizer

(Default: `org.apache.cassandra.auth.AllowAllAuthorizer`) The authorization backend. It implements `IAuthorizer`, which limits access and provides permissions. (Available only in DataStax Enterprise 3.0.x.)

permissions_validity_in_ms

(Default: `2000`) How long permissions in cache remain valid. Available only in DataStax Enterprise 3.0. Depending on the authorizer, fetching permissions can be resource intensive. This setting is automatically disabled when `AllowAllAuthorizer` is set. (Available only in DataStax Enterprise 3.0.x.)

auth_replication_strategy

(Default: `org.apache.cassandra.locator.SimpleStrategy`) The replication strategy for the auth keyspace. (Available only in DataStax Enterprise 3.0.x; see [Configuring dse_auth keyspace replication](#).)

auth_replication_options

Replication options for the authorization and authentication (`dse_auth`) keyspace. (Available only in DataStax Enterprise 3.0.x; see [Configuring dse_auth keyspace replication](#).)

replication_factor: (Default: 1) For SimpleStrategy:

```
auth_replication_options:  
  replication_factor: 3
```

For NetworkTopologyStrategy, use the same options for creating a keyspace. For example:

```
auth_replication_options:  
  DC1: 3  
  DC2: 3
```

encryption_options

Enable or disable inter-node encryption. The available options are:

- **internode_encryption:** (Default: none) Enable or disable encryption of inter-node communication using the TLS_RSA_WITH_AES_128_CBC_SHA cipher suite for authentication, key exchange, and encryption of data transfers.
- **keystore:** (Default: conf/.keystore) The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), which is the Java version of the Secure Sockets Layer (SSL), and Transport Layer Security (TLS) protocols. The keystore contains the private key used to encrypt outgoing messages.
- **keystore_password:** (Default: cassandra) Password for the keystore.
- **truststore:** (Default: conf/.truststore) The location of the truststore containing the trusted certificate for authenticating remote servers.
- **truststore_password:** (Default: cassandra) Password for the truststore.

The passwords used in these options must match the passwords used when generating the keystore and truststore. For instructions on generating these files, see: [Creating a Keystore to Use with JSSE](#). The advanced settings are:

- **protocol:** (Default: TLS)
- **algorithm:** (Default: SunX509)
- **store_type:** (Default: JKS)
- **cipher_suites:** (Default: TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA)
- **require_client_auth:** (Default: false) Enables peer certificate authentication.

ssl_storage_port

(Default: 7001) The SSL port for encrypted communication. Unused if encryption is disabled.

Keyspace and Column Family Storage Configuration

Cassandra stores the storage configuration attributes listed in this section in the `system` keyspace. You set storage configuration attributes on a per-keyspace or per-column family basis programmatically or using a client application, such as CQL or CLI. The attribute names documented in this section are the names recognized by the Thrift API and the CLI. When using CQL, you need to use slightly *different names in CQL* for a few of the attributes.

Keyspace Attributes

A keyspace must have a user-defined name, a replica placement strategy, and a replication factor that specifies the number of replicas per data center or node.

Option	Default Value
<i>name</i>	N/A (A user-defined value is required)
<i>placement_strategy</i>	org.apache.cassandra.locator.SimpleStrategy
<i>strategy_options</i>	N/A (container attribute)
<i>durable_writes</i>	True

name

Required. The name for the keyspace.

placement_strategy

Required. Determines how Cassandra distributes replicas for a keyspace among nodes in the ring.

Allowed values are:

- org.apache.cassandra.locator.SimpleStrategy
- SimpleStrategy
- org.apache.cassandra.locator.NetworkTopologyStrategy
- NetworkTopologyStrategy

NetworkTopologyStrategy requires a *properly configured snitch* to be able to determine rack and data center locations of a node. For more information about the replica placement strategy, see *About Replication in Cassandra*.

strategy_options

Specifies configuration options for the chosen replication strategy.

For examples and more information about configuring the replication placement strategy for a cluster and data centers, see *About Replication in Cassandra*.

durable_writes

(Default: true) When set to false, data written to the keyspace bypasses the commit log. Be careful using this option because you risk losing data. Change the durable_writes attribute using the CQL *CREATE KEYSPACE* or *ALTER KEYSPACE* statement.

Column Family Attributes

The following attributes can be declared per column family.

Option	Default Value
<i>bloom_filter_fp_chance</i>	0.000744
<i>caching</i>	keys_only
<i>column_metadata</i>	n/a (container attribute)
<i>column_type</i>	Standard
<i>comment</i>	n/a
<i>compaction_strategy</i>	SizeTieredCompactionStrategy

<code>compaction_strategy_options</code>	n/a (container attribute)
<code>comparator</code>	BytesType
<code>compare_subcolumns_with</code>	BytesType
<code>compression_options</code>	sstable_compression='SnappyCompressor'
<code>default_validation_class</code>	n/a
<code>dclocal_read_repair_chance</code>	0.0
<code>gc_grace_seconds</code>	864000 (10 days)
<code>key_validation_class</code>	n/a
<code>max_compaction_threshold</code>	32
<code>min_compaction_threshold</code>	4
<code>memtable_flush_after_mins</code> ^[1]	n/a
<code>memtable_operations_in_millions</code> ^[1]	n/a
<code>memtable_throughput_in_mb</code> ^[1]	n/a
<code>name</code>	n/a
<code>read_repair_chance</code>	0.1 or 1 (See description below.)
<code>replicate_on_write</code>	true

bloom_filter_fp_chance

(Default: 0.000744) Desired false-positive probability for SSTable Bloom filters. When data is requested, the Bloom filter checks if the requested row exists in the SSTable before doing any disk I/O. Valid values are 0 to 1.0. A setting of 0 means that the unmodified (effectively the largest possible) Bloom filter is enabled. Setting the Bloom Filter at 1.0 disables it. The higher the setting, the less memory Cassandra uses. The maximum recommended setting is 0.1, as anything above this value yields diminishing returns. For information about how Cassandra uses Bloom filters and tuning Bloom filters, see *Tuning Bloom Filters*.

caching

(Default: `keys_only`) Optimizes the use of cache memory without manual tuning. Set caching to one of the following values: `all`, `keys_only`, `rows_only`, or `none`. Cassandra weights the cached data by size and access frequency. Using Cassandra 1.1 and later, you use this parameter to specify a key or row cache instead of a table cache, as in earlier versions.

column_metadata

(Default: N/A - container attribute) Column metadata defines attributes of a column. Values for `name` and `validation_class` are required, though the `default_validation_class` for the column family is used if no `validation_class` is specified. Note that `index_type` must be set to create a secondary index for a column. `index_name` is not valid unless `index_type` is also set.

Option	Description
<code>name</code>	Binds a <code>validation_class</code> and (optionally) an index to a column.
<code>validation_class</code>	Type used to check the column value.
<code>index_name</code>	Name for the secondary index.
<code>index_type</code>	Type of index. Currently the only supported value is <code>KEYS</code> .

Setting and updating column metadata with the Cassandra CLI requires a slightly different command syntax than other attributes; note the brackets and curly braces in this example:

```
[default@demo] UPDATE COLUMN FAMILY users WITH comparator=UTF8Type
AND column_metadata=[{column_name: full_name, validation_class: UTF8Type, index_type: KEYS}];
```

column_type

(Default: Standard) Use `Standard` for regular column families and `Super` for super column families.

Note

Super columns are not recommended for any reason and may result in poor performance and `OutOfMemory` exceptions.

comment

(Default: N/A) A human readable comment describing the column family.

compaction_strategy

(Default: `SizeTieredCompactionStrategy`) Sets the compaction strategy for the column family. The available strategies are:

- **SizeTieredCompactionStrategy:** This is the default compaction strategy and the only compaction strategy available in pre-1.0 releases. This strategy triggers a minor compaction whenever there are a number of similar sized SSTables on disk (as configured by `min_compaction_threshold`). This strategy causes bursts in I/O activity while a compaction is in process, followed by longer and longer lulls in compaction activity as SSTable files grow larger in size. These I/O bursts can negatively effect read-heavy workloads, but typically do not impact write performance. Watching disk capacity is also important when using this strategy, as compactions can temporarily double the size of SSTables for a column family while a compaction is in progress.
- **LeveledCompactionStrategy:** The leveled compaction strategy creates SSTables of a fixed, relatively small size (5 MB by default) that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables. This can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after [Google's leveldb](#) implementation.

compaction_strategy_options

(Default: N/A - container attribute) Sets options related to the chosen `compaction_strategy`. Currently only `LeveledCompactionStrategy` has options:

sstable_size_in_mb: (Default 5) A compaction runs whenever more data exists in Level $N > 0$ than desired ($sstable_size_in_mb * 10^{level}$), or whenever there is anything in L0. In practice, this often means multiple tables are flushed to L0 while Cassandra is busy compacting higher levels, but in theory a single L0 SSTable can definitely be compacted with L1.

Setting and updating compaction options using CQL requires a slightly different command syntax than other attributes. For example:

```
ALTER TABLE users
WITH compaction_strategy_class='SizeTieredCompactionStrategy'
AND min_compaction_threshold = 6;
```

For more information, see [Configuring Compaction](#).

comparator

(Default: `ByteType`) Defines the data types used to validate and sort column names. There are several built-in *column comparators* available. Note that the comparator cannot be changed after a column family is created.

compare_subcolumns_with

(Default: `ByteType`) Required when `column_type` is "Super" (not recommended). Same as *comparator* but for the sub-columns of a SuperColumn.

For attributes of columns, see *column_metadata*.

compression_options

(Default: `sstable_compression='SnappyCompressor'`) This is a container attribute for setting compression options on a column family. It contains the following options:

Option	Description
<code>sstable_compression</code>	Specifies the compression algorithm to use when compressing SSTable files. Cassandra supports two built-in compression classes: <code>SnappyCompressor</code> (Snappy compression library) and <code>DeflateCompressor</code> (Java zip implementation). Snappy compression offers faster compression/decompression while the Java zip compression offers better compression ratios. Choosing the right one depends on your requirements for space savings over read performance. For read-heavy workloads, Snappy compression is recommended. Developers can also implement custom compression classes using the <code>org.apache.cassandra.io.compress.ICompressor</code> interface.
<code>chunk_length_kb</code>	Sets the compression chunk size in kilobytes. The default value (64) is a good middle-ground for compressing column families with either wide rows or with skinny rows. With wide rows, it allows reading a 64kb slice of column data without decompressing the entire row. For skinny rows, although you may still end up decompressing more data than requested, it is a good trade-off between maximizing the compression ratio and minimizing the overhead of decompressing more data than is needed to access a requested row. The compression chunk size can be adjusted to account for read/write access patterns (how much data is typically requested at once) and the average size of rows in the column family.
<code>crc_check_chance</code>	Specifies how often to perform the block checksum calculation.

Use *CQL to configure compression*. Using CQL, you disable, set, and update compression options. In CQL `compression_options` are called `compression_parameters`.

default_validation_class

(Default: N/A) Defines the data type used to validate column values. There are several built-in *column validators* available.

dclocal_read_repair_chance

(Default: 0.0) Specifies the probability with which read repairs are invoked over all replicas in the current data center. Contrast *read_repair_chance*.

gc_grace_seconds

(Default: 864000 [10 days]) Specifies the time to wait before garbage collecting tombstones (deletion markers). The default value allows a great deal of time for consistency to be achieved prior to deletion. In many deployments this interval can be reduced, and in a single-node cluster it can be safely set to zero.

Note

This property is called `gc_grace` in the `cassandra-cli` client.

key_validation_class

(Default: N/A) Defines the data type used to validate row key values. There are several built-in *key validators* available, however `CounterColumnType` (distributed counters) cannot be used as a row key validator.

max_compaction_threshold

(Default: 32) Ignored in `LeveledCompactionStrategy`. In `SizeTieredCompactionStrategy`, sets the maximum number of `SSTables` to allow in a minor compaction.

min_compaction_threshold

(Default: 4) Sets the minimum number of `SSTables` to trigger a minor compaction when `compaction_strategy=sizeTieredCompactionStrategy`. Raising this value causes minor compactions to start less frequently and be more I/O-intensive.

memtable_flush_after_mins

Deprecated as of Cassandra 1.0. Can still be declared (for backwards compatibility) but settings will be ignored. Use the `cassandra.yaml` parameter *commitlog_total_space_in_mb* instead.

memtable_operations_in_millions

Deprecated as of Cassandra 1.0. Can still be declared (for backwards compatibility) but settings will be ignored. Use the `cassandra.yaml` parameter *commitlog_total_space_in_mb* instead.

memtable_throughput_in_mb

Deprecated as of Cassandra 1.0. Can still be declared (for backwards compatibility) but settings will be ignored. Use the `cassandra.yaml` parameter *commitlog_total_space_in_mb* instead.

name

(Default: N/A) Required. The user-defined name of the column family.

read_repair_chance

(Default: 0.1 or 1) Specifies the probability with which read repairs should be invoked on non-quorum reads. The value must be between 0 and 1. For column families created in versions of Cassandra before 1.0, it defaults to 1. For column families created in versions of Cassandra 1.0 and higher, it defaults to 0.1. However, for Cassandra 1.0, the default is 1.0 if you use CLI or any Thrift client, such as Hector or pycassa, and is 0.1 if you use CQL.

replicate_on_write

(Default: `true`) Applies only to counter column families. When set to `true`, replicates writes to all affected replicas regardless of the consistency level specified by the client for a write request. It should always be set to `true` for counter column families.

Configuring the heap dump directory

Cassandra starts Java with the option `-XX:-HeapDumpOnOutOfMemoryError`. Using this option triggers a heap dump in the event of an out-of-memory condition. The heap dump file consists of references to objects that cause the heap to overflow. Analyzing the heap dump file can help troubleshoot memory problems. By default, Cassandra puts the file a subdirectory of the working, root directory when running as a service. If Cassandra does not have write permission to the root directory, the heap dump fails. If the root directory is too small to accommodate the heap dump, the server crashes.

For a heap dump to succeed and to prevent crashes, configure a heap dump directory that meets these requirements:

- Accessible to Cassandra for writing
- Large enough to accommodate a heap dump

Base the size of the directory on the value of the Java `-mx` option.

To configure the heap dump directory:

1. Open the `cassandra-env.sh` file for editing. This file is located in:

- Packaged installs
`/etc/dse/cassandra`
- Binary installs
`<install_location>/resources/cassandra/conf`

2. Scroll down to the comment about the heap dump path:

```
# set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
```

3. On the line after the comment, set the `CASSANDRA_HEAPDUMP_DIR` to the path you want to use:

```
# set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
CASSANDRA_HEAPDUMP_DIR=<path>
```

4. Save the `cassandra-env.sh` file and restart DataStax Enterprise.

Java and System Environment Settings Configuration

There are two files that control environment settings for Cassandra:

- `conf/cassandra-env.sh` - Java Virtual Machine (JVM) configuration settings
- `bin/cassandra-in.sh` - Sets up Cassandra environment variables such as `CLASSPATH` and `JAVA_HOME`.

Heap Sizing Options

If you decide to change the Java heap sizing, both `MAX_HEAP_SIZE` and `HEAP_NEWSIZE` should be set together in `conf/cassandra-env.sh` (if you set one, set the other as well). See the section on [Tuning the Java Heap](#) for more information on choosing the right Java heap size.

- `MAX_HEAP_SIZE` - Sets the maximum heap size for the JVM. The same value is also used for the minimum heap size. This allows the heap to be locked in memory at process start to keep it from being swapped out by the OS. Defaults to half of available physical memory.
- `HEAP_NEWSIZE` - The size of the young generation. The larger this is, the longer GC pause times will be. The shorter it is, the more expensive GC will be (usually). A good guideline is 100 MB per CPU core.

JMX Options

Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications

and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX. JConsole, *nodetool* and DataStax OpsCenter are examples of JMX-compliant management tools.

By default, the `conf/cassandra-env.sh` file configures JMX to listen on port 7199 without authentication. See the table below for more information on commonly changed JMX configuration properties.

- `com.sun.management.jmxremote.port` - The port on which Cassandra listens from JMX connections.
- `com.sun.management.jmxremote.ssl` - Enable/disable SSL for JMX.
- `com.sun.management.jmxremote.authenticate` - Enable/disable remote authentication for JMX.
- `-Djava.rmi.server.hostname` - Sets the interface hostname or IP that JMX should use to connect. Uncomment and set if you are having trouble connecting.

Further Reading on JVM Tuning

The remaining options are optimal across a wide variety of workloads and environments and are not frequently changed. See [the Sun JVM options list](#) for more information on JVM tuning parameters.

Authentication and Authorization Configuration

Note

As of release 1.0, the `SimpleAuthenticator` and `SimpleAuthority` classes have been moved to the example directory of the [Apache Cassandra project repository](#). They are no longer available in the packaged and binary distributions. They are only examples and do not provide actual security in their current state. DataStax does not officially support them and does not recommend their use.

Using authentication and authorization requires configuration changes in `cassandra.yaml` and two additional files: one for assigning users and their permissions to keyspaces and column families, and the other for assigning passwords to those users. These files are named `access.properties` and `passwd.properties`, respectively, and are located in the `examples` directory of the [Apache Cassandra project repository](#). To test simple authentication, you can move these files to the `conf` directory.

The location of the `cassandra.yaml` file depends on the type of installation; see [Cassandra Configuration Files Locations](#) or [DataStax Enterprise Configuration Files Locations](#).

To set up simple authentication and authorization

1. Edit `cassandra.yaml`, setting `org.apache.cassandra.auth.SimpleAuthenticator` as the `authenticator` value. The default value of `AllowAllAuthenticator` is equivalent to no authentication.
2. Edit `access.properties`, adding entries for users and their permissions to read and write to specified keyspaces and column families. See [access.properties](#) below for details on the correct format.
3. Make sure that users specified in `access.properties` have corresponding entries in `passwd.properties`. See [passwd.properties](#) below for details and examples.
4. After making the required configuration changes, you must specify the properties files when starting Cassandra with the flags `-Dpasswd.properties` and `-Daccess.properties`. For example:

```
cd <install_location>
sh bin/cassandra -f -Dpasswd.properties=conf/passwd.properties -Daccess.properties=conf/access.properties
```

access.properties

This file contains entries in the format `KEYSPACE[.COLUMNFAMILY].PERMISSION=USERS` where

- `KEYSPACE` is the keyspace name.
- `COLUMNFAMILY` is the column family name.
- `PERMISSION` is one of `<ro>` or `<rw>` for read-only or read-write respectively.
- `USERS` is a comma delimited list of users from `passwd.properties`.

For example, to control access to `Keyspace1` and give `jsmith` and `Elvis` read-only permissions while allowing `dilbert` full read-write access to add and remove column families, you would create the following entries:

```
Keyspace1.<ro>=jsmith,Elvis Presley
Keyspace1.<rw>=dilbert
```

To provide a finer level of access control to the `Standard1` column family in `Keyspace1`, you would create the following entry to allow the specified users read-write access:

```
Keyspace1.Standard1.<rw>=jsmith,Elvis Presley,dilbert
```

The `access.properties` file also contains a simple list of users who have permissions to modify the list of keyspaces:

```
<modify-keyspaces>=jsmith
```

passwd.properties

This file contains name/value pairs in which the names match users defined in `access.properties` and the values are user passwords. Passwords are in clear text unless the `passwd.mode=MD5` system property is provided.

```
jsmith=havebadpass
Elvis Presley=graceland4ever
dilbert=nomoovertime
```

Logging Configuration

Cassandra provides logging functionality using Simple Logging Facade for Java (SLF4J) with a log4j backend. Additionally, the `output.log` captures the stdout of the Cassandra process, which is configurable using the standard Linux logrotate facility. You can also change logging levels via JMX using the *JConsole* tool.

Changing the Rotation and Size of Cassandra Logs

You can control the rotation and size of both the `system.log` and `output.log`. Cassandra's `system.log` logging configuration is controlled by the `log4j-server.properties` file in the following directories:

- **Packaged installs:** `/etc/dse/cassandra`
- **Binary installs:** `<install_location>/resources/cassandra/conf`

system.log

The maximum log file size and number of backup copies are controlled by the following lines:

```
log4j.appender.R.maxFileSize=20MB
log4j.appender.R.maxBackupIndex=50
```

The default configuration rolls the log file once the size exceeds 20MB and maintains up to 50 backups. When the `maxFileSize` is reached, the current log file is renamed to `system.log.1` and a new `system.log` is started. Any previous backups are renumbered from `system.log.n` to `system.log.n+1`, which means the higher the number, the older the file. When the maximum number of backups is reached, the oldest file is deleted.

If an issue occurred but has already been rotated out of the current `system.log`, check to see if it is captured in an older backup. If you want to keep more history, increase the `maxFileSize`, `maxBackupIndex`, or both. However, make sure you have enough space to store the additional logs.

By default, logging output is placed the `/var/log/cassandra/system.log`. You can change the location of the output by editing the `log4j.appender.R.File` path. Be sure that the directory exists and is writable by the process running Cassandra.

output.log

The `output.log` stores the stdout of the Cassandra process; it is not controllable from log4j. However, you can rotate it using the standard Linux logrotate facility. To configure logrotate to work with cassandra, create a file called `/etc/logrotate.d/cassandra` with the following contents:

```
/var/log/cassandra/output.log {
    size 10M
    rotate 9
    missingok
    copytruncate
    compress
}
```

The `copytruncate` directive is critical because it allows the log to be rotated without any support from Cassandra for closing and reopening the file. For more information, refer to the [logrotate](#) man page.

Changing Logging Levels

If you need more diagnostic information about the runtime behavior of a specific Cassandra node than what is provided by Cassandra's JMX MBeans and the `nodetool` utility, you can increase the logging levels on specific portions of the system using log4j. The logging levels from most to least verbose are:

```
TRACE
DEBUG
INFO
WARN
ERROR
FATAL
```

Note

Be aware that increasing logging levels can generate a lot of logging output on even a moderately trafficked cluster.

Logging Levels

The default logging level is determined by the following line in the `log4j-server.properties` file:

```
log4j.rootLogger=INFO,stdout,R
```

To exert more fine-grained control over your logging, you can specify the logging level for specific categories. The categories usually (but not always) correspond to the package and class name of the code doing the logging.

For example, the following setting logs DEBUG messages from all classes in the `org.apache.cassandra.db` package:

```
log4j.logger.org.apache.cassandra.db=DEBUG
```

In this example, DEBUG messages are logged specifically from the `StorageProxy` class in the `org.apache.cassandra.service` package:

```
log4j.logger.org.apache.cassandra.service.StorageProxy=DEBUG
```

Finding the Category of a Log Message

To determine which category a particular message in the log belongs to, change the following line:

```
log4j.appender.R.layout.ConversionPattern=%5p [%t] %d{ISO8601} %F (line %L) %m%n
```

1. Add `%c` at the beginning of the conversion pattern:

```
log4j.appender.R.layout.ConversionPattern=%c %5p [%t] %d{ISO8601} %F (line %L) %m%n
```

Each log message is now prefixed with the category.

2. After Cassandra runs for a while, use the following command to determine which categories are logging the most messages:

```
cat system.log.* | egrep 'TRACE/DEBUG/INFO/WARN/ERROR/FATAL' | awk '{ print $1 }' | sort | uniq -c | sort -n
```

3. If you find that a particular class logs too many messages, use the following format to set a less verbose logging level for that class by adding a line for that class:

```
loggerorg4j.logger.package.class=WARN
```

For example a busy Solr node can log numerous INFO messages from the SolrCore, LogUpdateProcessorFactory, and SolrIndexSearcher classes. To suppress these messages, add the following lines:

```
log4j.logger.org.apache.solr.core.SolrCore=WARN
log4j.logger.org.apache.solr.update.processor.LogUpdateProcessorFactory=WARN
log4j.logger.org.apache.solr.search.SolrIndexSearcher=WARN
```

4. After determining which category a particular message belongs to you may want to revert the messages back to the default format. Do this by removing `%c` from the ConversionPattern.

Commit Log Archive Configuration

Cassandra provides commitlog archiving and point-in-time recovery starting with version 1.1.1. You configure this feature in the `commitlog_archiving.properties` configuration file, which is located in the following directories:

- Cassandra packaged installs: `/etc/cassandra/conf`
- Cassandra binary installs: `<install_location>/conf`
- DataStax Enterprise packaged installs: `/etc/dse/cassandra`
- DataStax Enterprise binary installs: `<install_location>/resources/cassandra/conf`

Commands

The commands `archive_command` and `restore_command` expect only a single command with arguments. STDOUT and STDIN or multiple commands cannot be executed. To workaround, you can script multiple commands and add a pointer to this file. To disable a command, leave it blank.

Archive a Segment

Archive a particular commitlog segment.

Command	archive_command=	
Parameters	<path>	Fully qualified path of the segment to archive.
	<name>	Name of the commit log.

Example	archive_command=/bin/ln <path> /backup/<name>
---------	---

Restore an Archived Commitlog

Make an archived commitlog live again.

Command	restore_command=	
Parameters	<from>	Fully qualified path of the an archived commitlog segment from the <restore_directories>.
	<to>	Name of live commit log directory.
Example	restore_command=cp -f <from> <to>	

Restore Directory Location

Set the directory for the recovery files are placed.

Command	restore_directories=
Format	restore_directories=<restore_directory location>

Restore Mutations

Restore mutations created up to and including the specified timestamp.

Command	restore_point_in_time=
Format	<timestamp>
Example	restore_point_in_time=2012-08-16 20:43:12

Note

Restore stops when the first client-supplied timestamp is greater than the restore point timestamp. Because the order in which Cassandra receives mutations does not strictly follow the timestamp order, this can leave some mutations unrecovered.

Performance Monitoring and Tuning

Monitoring a Cassandra Cluster

Understanding the performance characteristics of your Cassandra cluster is critical to diagnosing issues and planning capacity.

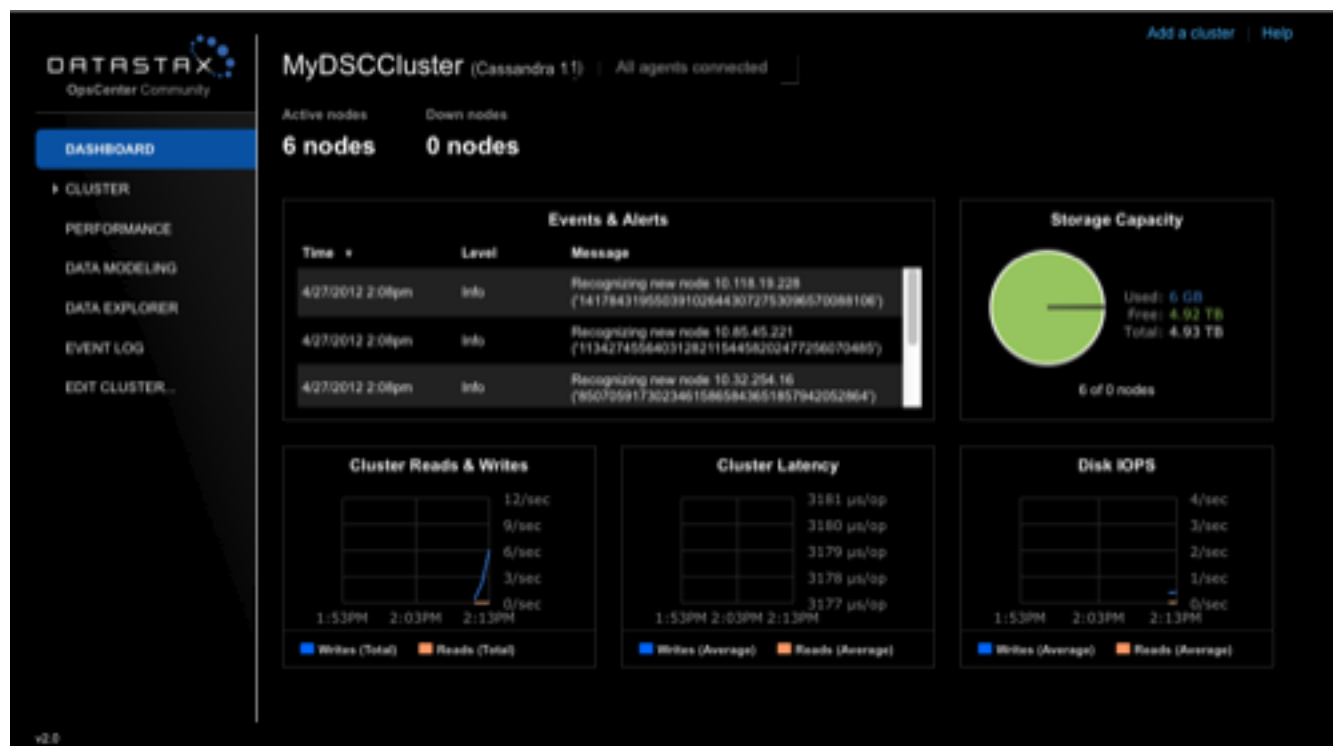
Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX.

During normal operation, Cassandra outputs information and statistics that you can monitor using JMX-compliant tools such as JConsole, the Cassandra *nodetool* utility, or the **DataStax OpsCenter** management console. With the same tools, you can perform certain administrative commands and operations such as flushing caches or doing a repair.

Monitoring Using DataStax OpsCenter

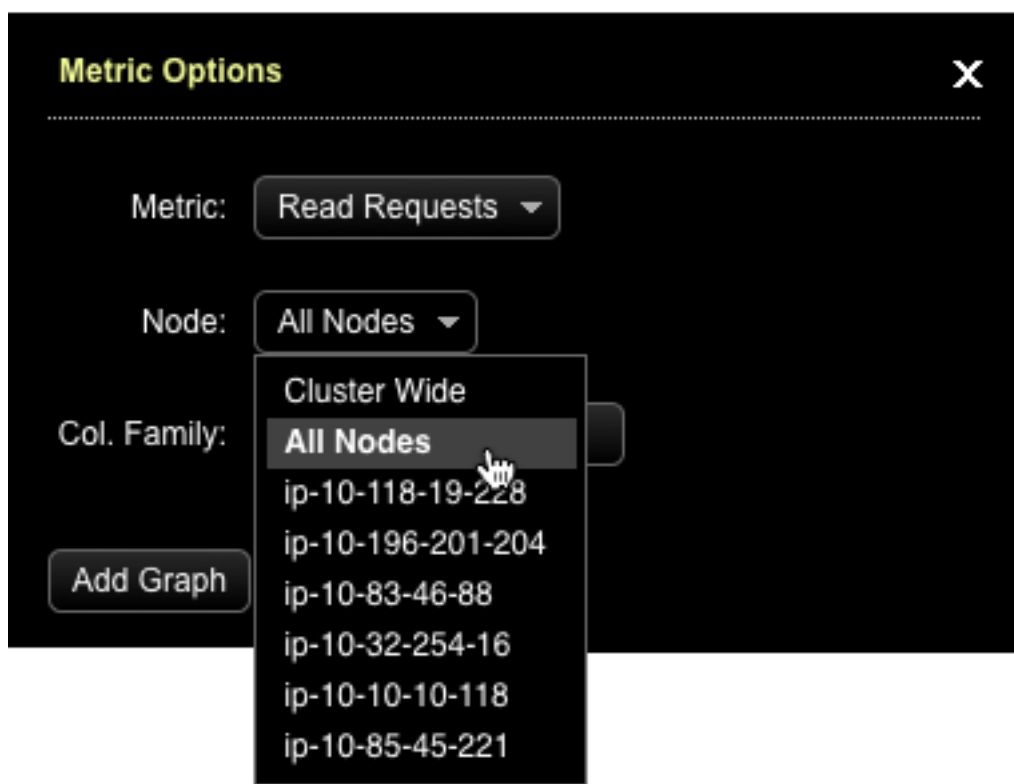
DataStax OpsCenter is a graphical user interface for monitoring and administering all nodes in a Cassandra cluster from one centralized console. DataStax OpsCenter is bundled with DataStax support offerings, or you can register for a free version licensed for development or non-production use.

OpsCenter provides a graphical representation of performance trends in a summary view that is hard to obtain with other monitoring tools. The GUI provides views for different time periods as well as the capability to drill down on single data points. Both real-time and historical performance data for a cluster are available in OpsCenter. OpsCenter metrics are captured and stored within Cassandra.



The performance metrics viewed within OpsCenter can be customized according to your monitoring needs. Administrators can also perform routine node administration tasks from OpsCenter. Metrics within OpsCenter are divided into three general categories: column family metrics, cluster metrics, and OS metrics. For many of the available

metrics, you can choose to view aggregated cluster-wide information, or view information on a per-node basis.



Monitoring Using *nodetool*

The *nodetool* utility is a command-line interface for monitoring Cassandra and performing routine database operations. It is included in the Cassandra distribution and is typically run directly from an operational Cassandra node.

The *nodetool* utility supports the most important JMX metrics and operations, and includes other useful commands for Cassandra administration. This utility is commonly used to output a quick summary of the ring and its current state of general health with the *ring* command. For example:

```
# nodetool -h localhost -p 7199 ring
```

Address	Status	State	Load	Owns	Range	Ring
10.194.171.160	Down	Normal	?	39.98	95315431979199388464207182617231204396	
10.196.14.48	Up	Normal	3.16 KB	30.01	61078635599166706937511052402724559481	<--
10.196.14.239	Up	Normal	3.16 KB	30.01	78197033789183047700859117509977881938	
					95315431979199388464207182617231204396	-->

The *nodetool* utility provides commands for viewing detailed metrics for column family metrics, server metrics, and compaction statistics. Commands are also available for important operations such as decommissioning a node, running repair, and moving partitioning tokens.

Monitoring Using *JConsole*

JConsole is a JMX-compliant tool for monitoring Java applications such as Cassandra. It is included with Sun JDK 5.0 and higher. JConsole consumes the JMX metrics and operations exposed by Cassandra and displays them in a well-organized GUI. For each node monitored, JConsole provides these six separate tab views:

- **Overview** - Displays overview information about the Java VM and monitored values.
- **Memory** - Displays information about memory use.
- **Threads** - Displays information about thread use.

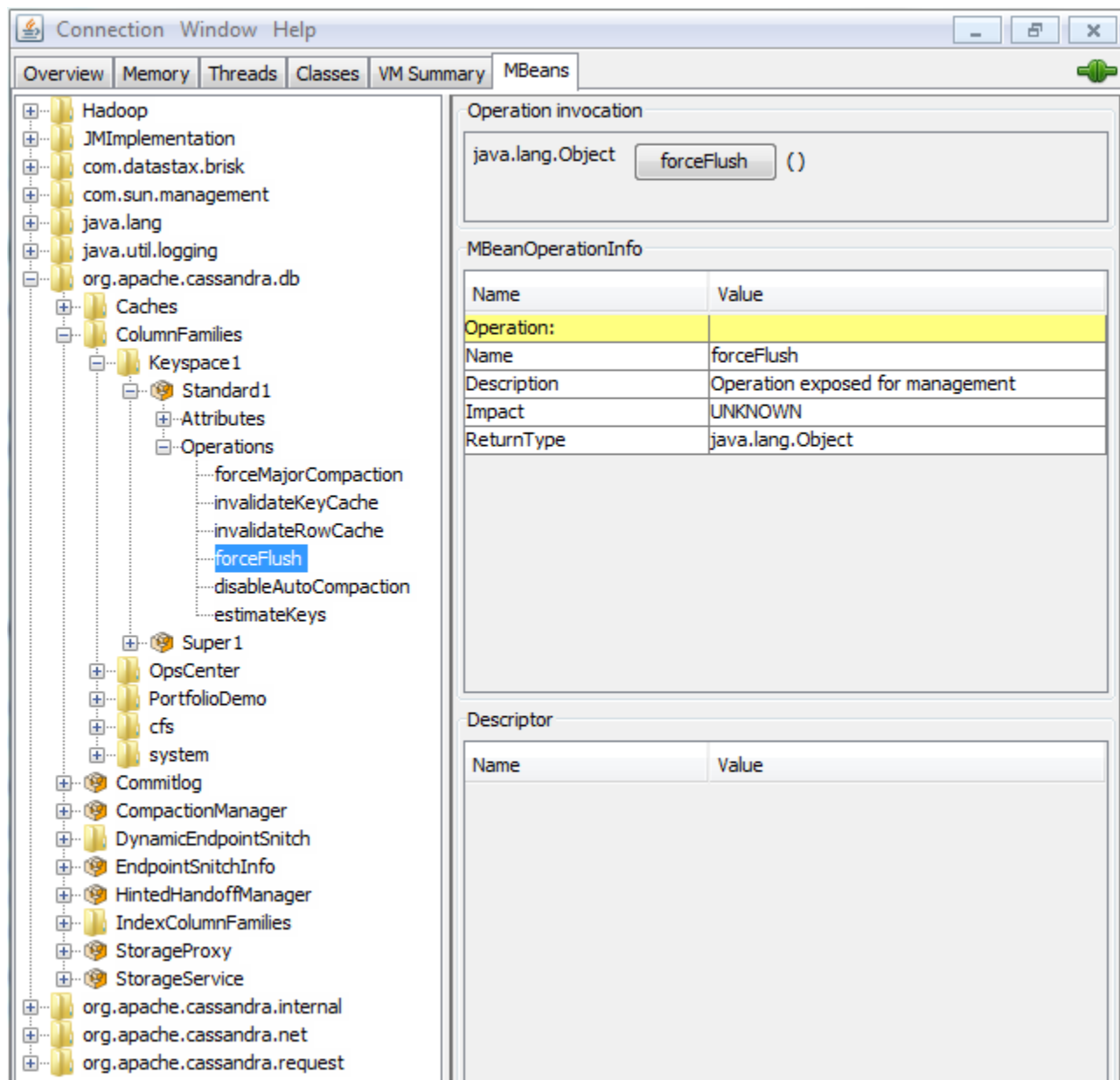
- **Classes** - Displays information about class loading.
- **VM Summary** - Displays information about the Java Virtual Machine (VM).
- **Mbeans** - Displays information about MBeans.

The **Overview** and **Memory** tabs contain information that is very useful for Cassandra developers. The Memory tab allows you to compare heap and non-heap memory usage, and provides a control to immediately perform Java garbage collection.

For specific Cassandra metrics and operations, the most important area of JConsole is the **MBeans** tab. This tab lists the following Cassandra MBeans:

- `org.apache.cassandra.db` - Includes caching, column family metrics, and compaction.
- `org.apache.cassandra.internal` - Internal server operations such as gossip and hinted handoff.
- `org.apache.cassandra.net` - Inter-node communication including `FailureDetector`, `MessagingService` and `StreamingService`.
- `org.apache.cassandra.request` - Tasks related to read, write, and replication operations.

When you select an MBean in the tree, its `MBeanInfo` and `MBean Descriptor` are both displayed on the right, and any attributes, operations or notifications appear in the tree below it. For example, selecting and expanding the `org.apache.cassandra.db` MBean to view available actions for a column family results in a display like the following:



If you choose to monitor Cassandra using JConsole, keep in mind that JConsole consumes a significant amount of system resources. For this reason, DataStax recommends running JConsole on a remote machine rather than on the same host as a Cassandra node.

Compaction Metrics

Monitoring compaction performance is an important aspect of knowing when to add capacity to your cluster. The following attributes are exposed through `CompactionManagerMBean`:

Attribute	Description
CompletedTasks	Number of completed compactions since the last start of this Cassandra instance
PendingTasks	Number of estimated tasks remaining to perform
ColumnFamilyInProgress	ColumnFamily currently being compacted. <code>null</code> if no compactions are in progress.
BytesTotalInProgress	Total number of data bytes (index and filter are not included) being compacted. <code>null</code> if no compactions are in progress.

BytesCompacted	The progress of the current compaction. <code>null</code> if no compactions are in progress.
----------------	--

Thread Pool Statistics

Cassandra maintains distinct thread pools for different stages of execution. Each of these thread pools provide statistics on the number of tasks that are active, pending and completed. Watching trends on these pools for increases in the pending tasks column is an excellent indicator of the need to add additional capacity. Once a baseline is established, alarms should be configured for any increases past normal in the pending tasks column. See below for details on each thread pool (this list can also be obtained via command line using `nodetool tpstats`).

Thread Pool	Description
AE_SERVICE_STAGE	Shows anti-entropy tasks
CONSISTENCY-MANAGER	Handles the background consistency checks if they were triggered from the client's <i>consistency level</i> <code><consistency></code> .
FLUSH-SORTER-POOL	Sorts flushes that have been submitted.
FLUSH-WRITER-POOL	Writes the sorted flushes.
GOSSIP_STAGE	Activity of the Gossip protocol on the ring.
LB-OPERATIONS	The number of load balancing operations.
LB-TARGET	Used by nodes leaving the ring.
MEMTABLE-POST-FLUSHER	Memtable flushes that are waiting to be written to the commit log.
MESSAGE-STREAMING-POOL	Streaming operations. Usually triggered by bootstrapping or decommissioning nodes.
MIGRATION_STAGE	Tasks resulting from the call of <code>system_*</code> methods in the API that have modified the schema.
MISC_STAGE	
MUTATION_STAGE	API calls that are modifying data.
READ_STAGE	API calls that have read data.
RESPONSE_STAGE	Response tasks from other nodes to message streaming from this node.
STREAM_STAGE	Stream tasks from this node.

Read/Write Latency Metrics

Cassandra keeps tracks latency (averages and totals) of read, write and slicing operations at the server level through `StorageProxyMBean`.

ColumnFamily Statistics

For individual column families, `ColumnFamilyStoreMBean` provides the same general latency attributes as `StorageProxyMBean`. Unlike `StorageProxyMBean`, `ColumnFamilyStoreMBean` has a number of other statistics that are important to monitor for performance trends. The most important of these are listed below:

Attribute	Description
MemtableDataSize	The total size consumed by this column family's data (not including meta data).
MemtableColumnsCount	Returns the total number of columns present in the memtable (across all keys).
MemtableSwitchCount	How many times the memtable has been flushed out.
RecentReadLatencyMicros	The average read latency since the last call to this bean.

RecentWriterLatencyMicros	The average write latency since the last call to this bean.
LiveSSTableCount	The number of live SSTables for this ColumnFamily.

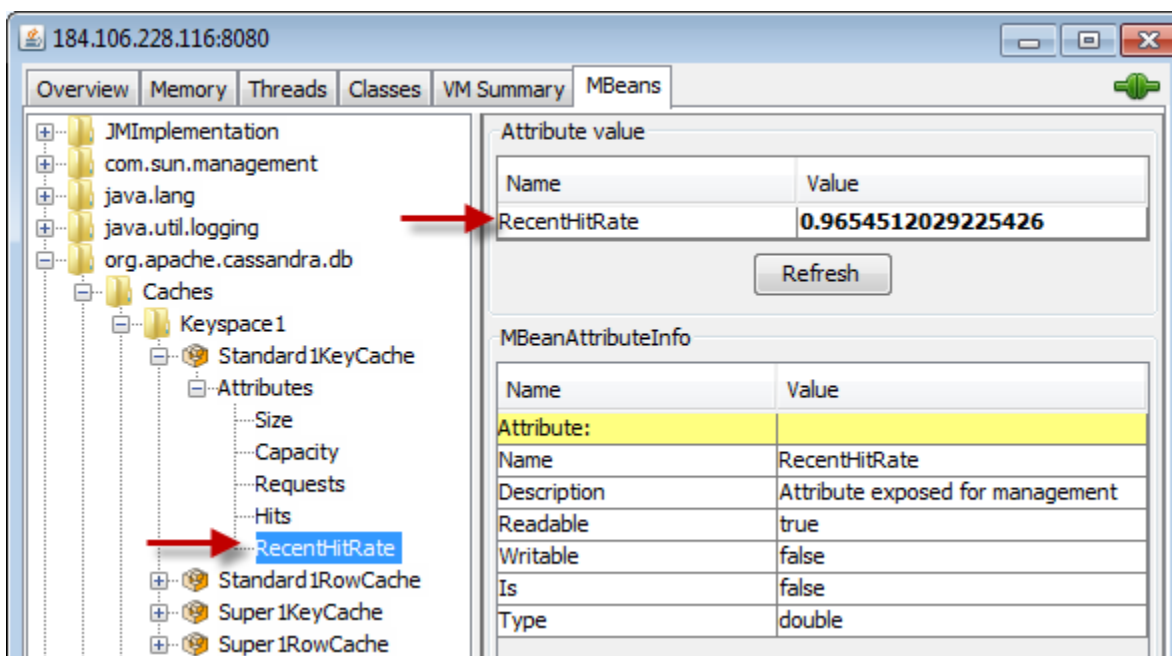
The recent read latency and write latency counters are important in making sure that operations are happening in a consistent manner. If these counters start to increase after a period of staying flat, it is probably an indication of a need to add cluster capacity.

`LiveSSTableCount` can be monitored with a threshold to ensure that the number of SSTables for a given ColumnFamily does not become too great.

Monitoring and Adjusting Cache Performance

Making small incremental cache changes followed by careful monitoring is the best way to maximize benefit from Cassandra's built-in caching features. It is best to monitor Cassandra as a whole for unintended impact on the system. Adjustments that increase cache hit rate are likely to use more system resources, such as memory.

For each node and each column family, you can view cache hit rate, cache size, and number of hits by expanding `org.apache.cassandra.db` in the MBeans tab. For example:



Monitor new cache settings not only for hit rate, but also to make sure that memtables and heap size still have sufficient memory for other operations. If you cannot maintain the desired key cache hit rate of 85% or better, add nodes to the system and re-test until you can meet your caching requirements.

Tuning Cassandra

Tuning Cassandra and Java resources is recommended in the event of a performance degradation, high memory consumption, and other atypical situations described in this section. After completion of tuning operations, follow recommendations in this section to monitor and test changes. Tuning Cassandra includes the following tasks:

- *Tuning Bloom Filters*
- *Tuning Data Caches*
- *Tuning the Java Heap*
- *Tuning Java Garbage Collection*

- *Tuning Compaction*
- *Tuning Column Family Compression*
- Modifying *performance-related properties* in the `cassandra.yaml` file.

Tuning Bloom Filters

Each SSTable has a Bloom filter. A Bloom filter tests whether an element is a member of a set. False positive retrieval results are possible, but false negatives are not. In Cassandra when data is requested, the Bloom filter checks if the requested row exists in the SSTable before doing any disk I/O. High memory consumption can result from the Bloom filter false positive ratio being set too low. The higher the Bloom filter setting, the lower the memory consumption. By tuning a Bloom filter, you are setting the chance of false positives; the lower the chances of false positives, the larger the Bloom filter. The maximum recommended setting is 0.1, as anything above this value yields diminishing returns.

Bloom filter settings range from 0.000744 (default) to 1.0 (disabled). For example, to run an analytics application that heavily scans a particular column family, you would want to inhibit the Bloom filter on the column family by setting it high. Setting it high ensures that an analytics application will never ask for keys that don't exist.

To change the *Bloom filter attribute* on a column family, use CQL. For example:

```
ALTER TABLE addamsFamily WITH bloom_filter_fp_chance = 0.01;
```

After updating the value of `bloom_filter_fp_chance` on a column family, Bloom filters need to be regenerated in one of these ways:

- *Initiate compaction*
- *Upgrade SSTables*

You do not have to restart Cassandra after regenerating SSTables.

Tuning Data Caches

These caches are built into Cassandra and provide very efficient data caching:

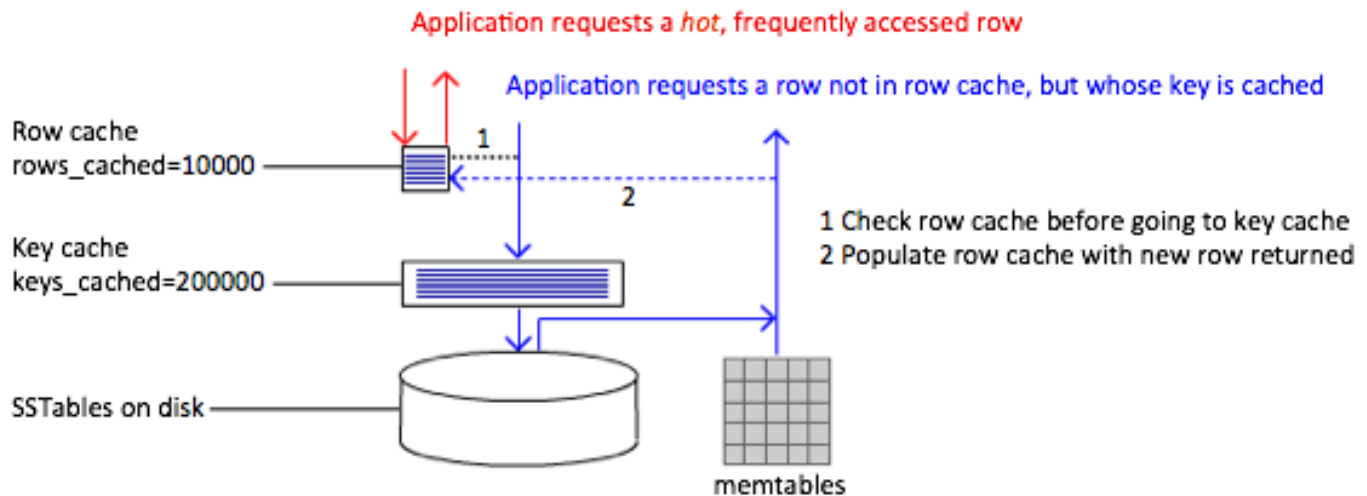
- Key cache: a cache of the *primary key index* for a Cassandra table. Enabled by default.
- Row cache: similar to a traditional cache like memcached. Holds the entire row in memory so reads can be satisfied without using disk. Disabled by default.

If read performance is critical, you can leverage the built-in caching to effectively pry dedicated caching tools, such as memcached, completely out of the stack. Such deployments remove a redundant layer and strengthen cache functionality in the lower tier where the data is already being stored. Caching never needs to be restarted in a completely *cold* state.

With proper tuning, key cache hit rates of 85% or better are possible with Cassandra, and each hit on a key cache can save one disk seek per SSTable. Row caching, when feasible, can save the system from performing any disk seeks at all when fetching a cached row. When growth in the read load begins to impact your hit rates, you can add capacity to restore optimal levels of caching. Typically, expect a 90% hit rate for row caches. If row cache hit rates are 30% or lower, it may make more sense to leave row caching disabled (the default). Using only the key cache makes the row cache available for other column families that need it.

How Caching Works

When both row and key caches are configured, the row cache returns results whenever possible. In the event of a row cache miss, the key cache might still provide a hit that makes the disk seek much more efficient. This diagram depicts two read operations on a column family with both caches already populated.



One read operation hits the row cache, returning the requested row without a disk seek. The other read operation requests a row that is not present in the row cache but is present in the key cache. After accessing the row in the SSTable, the system returns the data and populates the row cache with this read operation.

When to Use Key Caching

Because the key cache holds the location of keys in memory on a per-column family basis, turning this value up can have an immediate, positive impact on column family reads as soon as the cache warms.

High levels of key caching are recommended for most scenarios. Cases for row caching are more specialized, but whenever it can coexist peacefully with other demands on memory resources, row caching provides the most dramatic gains in efficiency.

Using the default key cache setting, or a higher one, works well in most cases. Tune key cache sizes in conjunction with the *Java heap size*.

When to Use Row Caching

Row caching saves more time than key caching, but it is extremely space consuming. Row caching is recommended in these cases:

- Data access patterns follow a normal (Gaussian) distribution.
- Rows contain heavily-read data and queries frequently return data from most or all of the columns.

General Cache Usage Tips

Some tips for efficient cache use are:

- Store lower-demand data or data with extremely long rows in a column family with minimal or no caching.
- Deploy a large number of Cassandra nodes under a relatively light load per node.
- Logically separate heavily-read data into discrete column families.

Cassandra's memtables have overhead for index structures on top of the actual data they store. If the size of the values stored in the heavily-read columns is small compared to the number of columns and rows themselves (long, narrow rows), this overhead can be substantial. Short, narrow rows, on the other hand, lend themselves to highly efficient row caching.

Enabling the Key and Row Caches

Enable the key and row caches at the column family level using the CQL *caching* parameter. Unlike earlier Cassandra versions, cache sizes do not need to be specified per table. Just set caching to all, keys_only, rows_only, or none, and Cassandra weights the cached data by size and access frequency, and thus make optimal use of the cache memory without manual tuning. For archived tables, *disable caching* entirely because these tables are read infrequently.

Setting Cache Options

In the *cassandra.yaml* file, tune caching by changing these options:

- *key_cache_size_in_mb*: The capacity in megabytes of all key caches on the node.
- *row_cache_size_in_mb*: The capacity in megabytes of all row caches on the node.
- *key_cache_save_period*: How often to save the key caches to disk.
- *row_cache_save_period*: How often to save the row caches to disk.
- *row_cache_provider*: The implementation used for row caches on the node.

Monitoring Cache Tune Ups

Make changes to cache options in small, incremental adjustments, then monitor the effects of each change using one of the following tools:

- OpsCenter
- *nodetool cfstats*
- JConsole

About the Off-Heap Row Cache

Cassandra can store cached rows in native memory, outside the Java heap. This results in both a smaller per-row memory footprint and reduced JVM heap requirements, which helps keep the heap size in the sweet spot for JVM garbage collection performance.

Using the off-heap row cache requires the JNA library to be installed; otherwise, Cassandra falls back on the on-heap cache provider.

Tuning the Java Heap

Because Cassandra is a database, it spends significant time interacting with the operating system's I/O infrastructure through the JVM, so a well-tuned Java heap size is important. Cassandra's default configuration opens the JVM with a heap size that is based on the total amount of system memory:

System Memory	Heap Size
Less than 2GB	1/2 of system memory
2GB to 4GB	1GB
Greater than 4GB	1/4 system memory, but not more than 8GB

General Guidelines

Many users new to Cassandra are tempted to turn up Java heap size too high, which consumes the majority of the underlying system's RAM. In most cases, increasing the Java heap size is actually detrimental for these reasons:

- In most cases, the capability of Java 6 to gracefully handle garbage collection above 8GB quickly diminishes.
- Modern operating systems maintain the OS page cache for frequently accessed data and are very good at keeping this data in memory, but can be prevented from doing its job by an elevated Java heap size.

To change a JVM setting, modify the *cassandra-env.sh* file.

Because MapReduce runs outside the JVM, changes to the JVM do not affect Hadoop operations directly.

Tuning Java Garbage Collection

Cassandra's `GCInspector` class logs information about garbage collection whenever a garbage collection takes longer than 200ms. Garbage collections that occur frequently and take a moderate length of time to complete (such as ConcurrentMarkSweep taking a few seconds), indicate that there is a lot of garbage collection pressure on the JVM. Remedies include adding nodes, lowering cache sizes, or adjusting the JVM options regarding garbage collection.

Tuning Compaction

In addition to consolidating SSTables, the compaction process merges keys, combines columns, discards tombstones, and creates a new index in the merged SSTable.

To tune compaction, set a *compaction_strategy* for each column family based on its access patterns. The compaction strategies are:

- Size-Tiered Compaction

Appropriate for append-mostly workloads, which add new rows, and to a lesser degree, new columns to old rows.

- Leveled Compaction

Appropriate for workloads with many updates that change the values of existing columns, such as time-bound data in columns marked for *expiration using TTL*.

For example, to update a column family to use the leveled compaction strategy using Cassandra CQL:

```
ALTER TABLE users WITH
  compaction_strategy_class='LeveledCompactionStrategy'
AND  compaction_strategy_options:sstable_size_in_mb:10;
```

Tuning Compaction for Size-Tiered Compaction

Control the frequency and scope of a minor compaction of a column family that uses the default size-tiered compaction strategy by setting the *min_compaction_threshold*. The size-tiered compaction strategy triggers a minor compaction when a number SSTables on disk are of the size configured by `min_compaction_threshold`.

By default, a minor compaction can begin any time Cassandra creates four SSTables on disk for a column family. A minor compaction *must* begin before the total number of SSTables reaches 32.

Configure this value per column family using CQL. For example:

```
ALTER TABLE users WITH min_compaction_threshold = 6;
```

This CQL example shows how to change the `compaction_strategy_class` and set a minimum compaction threshold:

```
ALTER TABLE users
  WITH compaction_strategy_class='SizeTieredCompactionStrategy'
AND  min_compaction_threshold = 6;
```

A full compaction applies only to `SizeTieredCompactionStrategy`. It merges all SSTables into one large SSTable. Generally, a full compaction is not recommended because the large SSTable that is created will not be compacted until the amount of actual data increases four-fold (or `min_compaction_threshold`). Additionally, during runtime, full compaction is I/O and CPU intensive and can temporarily double disk space usage when no old versions or tombstones are evicted.

To initiate a full compaction for all tables in a keyspace use the *nodetool compact* command.

Cassandra provides a startup option for *testing compaction strategies* without affecting the production workload.

Tuning Column Family Compression

Compression maximizes the storage capacity of Cassandra nodes by reducing the volume of data on disk and disk I/O, particularly for read-dominated workloads. Cassandra quickly finds the location of rows in the SSTable index and decompresses the relevant row chunks.

Write performance is not negatively impacted by compression in Cassandra as it is in traditional databases. In traditional relational databases, writes require overwrites to existing data files on disk. The database has to locate the relevant pages on disk, decompress them, overwrite the relevant data, and finally recompress. In a relational database, compression is an expensive operation in terms of CPU cycles and disk I/O. Because Cassandra SSTable data files are immutable (they are not written to again after they have been flushed to disk), there is no recompression cycle necessary in order to process writes. SSTables are compressed only once when they are written to disk. Writes on compressed tables can show up to a 10 percent performance improvement.

How to Enable and Disable Compression

Compression is enabled by default in Cassandra 1.1. To disable compression, use CQL to set the compression parameters to an empty string:

```
CREATE TABLE DogTypes (
    block_id uuid,
    species text,
    alias text,
    population varint,
    PRIMARY KEY (block_id)
)
WITH compression_parameters:sstable_compression = '';
```

To enable or change compression on an existing column family, use ALTER TABLE and set the *compression_parameters* sstable_compression to SnappyCompressor or DeflateCompressor.

How to Change and Tune Compression

Change or tune data compression on a per-column family basis using CQL to alter a column family and set the *compression_parameters* attributes:

```
ALTER TABLE users
WITH compression_parameters:sstable_compression = 'DeflateCompressor'
AND compression_parameters:chunk_length_kb = 64;
```

When to Use Compression

Compression is best suited for column families that have many rows and each row has the same columns, or at least as many columns, as other rows. For example, a column family containing user data such as username, email, and state, is a good candidate for compression. The greater the similarity of the data across rows, the greater the compression ratio and gain in read performance.

A column family that has rows of different sets of columns, or a few wide rows, is not well-suited for compression. Dynamic column families do not yield good compression ratios.

Don't confuse column family compression with *compact storage* of columns, which is used for backward compatibility of old applications with CQL 3.

Depending on the data characteristics of the column family, compressing its data can result in:

- 2x-4x reduction in data size
- 25-35% performance improvement on reads
- 5-10% performance improvement on writes

After configuring compression on an existing column family, subsequently created SSTables are compressed. Existing SSTables on disk are not compressed immediately. Cassandra compresses existing SSTables when the normal Cassandra compaction process occurs. Force existing SSTables to be rewritten and compressed by using *nodetool upgradesstables* (Cassandra 1.0.4 or later) or *nodetool scrub*.

Testing Compaction and Compression

Write survey mode is a Cassandra startup option for testing new compaction and compression strategies. Using write survey mode, experiment with different strategies and benchmark write performance differences without affecting the production workload.

Write survey mode adds a node to a database cluster. The node accepts all write traffic as if it were part of the normal Cassandra cluster, but the node does not officially join the ring.

To enable write survey mode, start a Cassandra node using the option shown in this example:

```
bin/cassandra - Dcassandra.write_survey=true
```

Also use write survey mode to try out a new Cassandra version. The nodes you add in write survey mode to a cluster must be of the same major release version as other nodes in the cluster. The write survey mode relies on the streaming subsystem that transfers data between nodes in bulk and differs from one major release to another.

If you want to see how read performance is affected by modifications, stop the node, bring it up as a standalone machine, and then benchmark read operations on the node.

References

CQL 3 Language Reference

Cassandra Query Language (CQL) is a SQL (Structured Query Language)-like language for querying Cassandra. Although CQL has many similarities to SQL, there are some fundamental differences. For example, the CQL adaptation to the Cassandra data model and architecture, doesn't support operations, such as JOINS, which make no sense in a non-relational database.

This reference describes the Beta version of CQL 3.0.0. For a description of CQL 2.0.0, see [the CQL reference for Cassandra 1.0](#).

About the CQL 3 Reference

In addition to describing CQL commands, this reference includes introductory topics that briefly describe how commands are structured, the keywords and identifiers used in CQL, Cassandra data types, how to format dates and times, and CQL counterparts to Cassandra storage types. These topics are covered in the following sections.

CQL Lexical Structure

CQL input consists of statements. Like SQL, statements change data, look up data, store data, or change the way data is stored. Statements end in a semicolon (;).

For example, the following is valid CQL syntax:

```
SELECT * FROM MyColumnFamily;

UPDATE MyColumnFamily
  SET SomeColumn = 'SomeValue'
 WHERE columnName = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

This is a sequence of two CQL statements. This example shows one statement per line, although a statement can usefully be split across lines as well.

CQL Case-Sensitivity

In CQL 3, identifiers, such as keyspace and table names, are case-insensitive unless enclosed in double quotation marks. You can force the case by using double quotation marks. For example:

```
CREATE TABLE test (
  Foo int PRIMARY KEY,
  "Bar" int
)
```

The following table shows partial queries that work and do not work to return results from the test table:

Queries that Work	Queries that Don't Work
SELECT foo FROM ...	SELECT "Foo" FROM ...
SELECT Foo FROM ...	SELECT "BAR" FROM ...
SELECT FOO FROM ...	SELECT bar FROM ...
SELECT "foo" FROM ...	
SELECT "Bar" FROM ...	

SELECT "foo" FROM ... works because internally, Cassandra stores foo in lowercase.

CQL keywords are case-insensitive. For example, the keywords `SELECT` and `select` are equivalent, although this document shows keywords in uppercase.

Keywords and Identifiers

Column names that contain characters that CQL cannot parse need to be enclosed in double quotation marks in CQL3. In CQL2, single quotation marks were used.

Valid expressions consist of these kinds of values:

- **identifier:** A letter followed by any sequence of letters, digits, or the underscore.
- **string literal:** Characters enclosed in single quotation marks. To use a single quotation mark itself in a string literal, escape it using a single quotation mark. For example, `' '`.
- **integer:** An optional minus sign, `-`, followed by one or more digits.
- **uuid:** 32 hex digits, `0-9` or `a-f`, which are case-insensitive, separated by dashes, `-`, after the 8th, 12th, 16th, and 20th digits. For example: `01234567-0123-0123-0123-0123456789ab`
- **float:** A series of one or more decimal digits, followed by a period, `.`, and one or more decimal digits. There is no provision for exponential, `e`, notation, no optional `+` sign, and the forms `.42` and `42.` are unacceptable. Use leading or trailing zeros before and after decimal points. For example, `0.42` and `42.0`.
- **whitespace:** Separates terms and used inside string literals, but otherwise CQL ignores whitespace.

CQL Data Types

Cassandra has a schema-optional data model. You can define data types when you create your column family schemas. Creating the schema is recommended, but not required. Column names, column values, and row key values can be typed in Cassandra.

CQL comes with the following built-in data types, which can be used for column names and column/row key values. One exception is `counter`, which is allowed only as a column value (not allowed for row key values or column names).

CQL Type	Description
<code>ascii</code>	US-ASCII character string
<code>bigint</code>	64-bit signed long
<code>blob</code>	Arbitrary bytes (no validation), expressed as hexadecimal
<code>boolean</code>	true or false
<code>counter</code>	Distributed counter value (64-bit long)
<code>decimal</code>	Variable-precision decimal
<code>double</code>	64-bit IEEE-754 floating point
<code>float</code>	32-bit IEEE-754 floating point
<code>int</code>	32-bit signed integer
<code>text</code>	UTF-8 encoded string
<code>timestamp</code>	Date plus time, encoded as 8 bytes since epoch
<code>uuid</code>	Type 1 or type 4 UUID
<code>timeuuid</code>	Type 1 UUID only (CQL3)
<code>varchar</code>	UTF-8 encoded string
<code>varint</code>	Arbitrary-precision integer

In addition to the CQL types listed in this table, you can use a string containing the name of a class (a sub-class of `AbstractType` loadable by Cassandra) as a CQL type. The class name should either be fully qualified or relative to the

org.apache.cassandra.db.marshall package.

CQL3 timeuuid Type

The timeuuid type in CQL3 uses the Thrift comparator TimeUUIDType underneath. This type accepts type 1 UUID only; consequently, using this type prevents mistakenly inserting UUID that do not represent a time and allows the CQL date syntax, such as 2012-06-24 11:04:42, to input the UUID.

Working with Dates and Times

Values serialized with the `timestamp` type are encoded as 64-bit signed integers representing a number of milliseconds since the standard base time known as the *epoch*: January 1 1970 at 00:00:00 GMT.

Timestamp types can be input in CQL as simple long integers, giving the number of milliseconds since the epoch.

Timestamp types can also be input as string literals in any of the following ISO 8601 formats:

```
yyyy-mm-dd HH:mm
yyyy-mm-dd HH:mm:ss
yyyy-mm-dd HH:mmZ
yyyy-mm-dd HH:mm:ssZ
yyyy-mm-dd 'T' HH:mm
yyyy-mm-dd 'T' HH:mmZ
yyyy-mm-dd 'T' HH:mm:ss
yyyy-mm-dd 'T' HH:mm:ssZ
yyyy-mm-dd
yyyy-mm-ddZ
```

For example, for the date and time of Jan 2, 2003, at 04:05:00 AM, GMT:

```
2011-02-03 04:05+0000
2011-02-03 04:05:00+0000
2011-02-03T04:05+0000
2011-02-03T04:05:00+0000
```

The `+0000` is the RFC 822 4-digit time zone specification for GMT. US Pacific Standard Time is `-0800`. The time zone may be omitted. For example:

```
2011-02-03 04:05
2011-02-03 04:05:00
2011-02-03T04:05
2011-02-03T04:05:00
```

If no time zone is specified, the time zone of the Cassandra coordinator node handling the write request is used. For accuracy, DataStax recommends specifying the time zone rather than relying on the time zone configured on the Cassandra nodes.

If you only want to capture date values, the time of day can also be omitted. For example:

```
2011-02-03
2011-02-03+0000
```

In this case, the time of day defaults to 00:00:00 in the specified or default time zone.

CQL Comments

Comments can be used to document CQL statements in your application code. Single line comments can begin with a double dash (`--`) or a double slash (`//`) and extend to the end of the line. Multi-line comments can be enclosed in `/*` and `*/` characters.

Specifying Consistency Level

In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replica nodes. For any given read or write operation, the client request specifies a consistency level, which determines how many replica nodes must successfully respond to the request.

In CQL, the default consistency level is `ONE`. You can set the consistency level for any read (`SELECT`) or write (`INSERT`, `UPDATE`, `DELETE`, `BATCH`) operation. For example:

```
SELECT * FROM users USING CONSISTENCY QUORUM WHERE state='TX' ;
```

Consistency level specifications are made up the keywords `USING CONSISTENCY`, followed by a consistency level identifier. Valid consistency level identifiers are:

- `ANY` (applicable to writes only)
- `ONE` (default)
- `TWO`
- `THREE`
- `QUORUM`
- `LOCAL_QUORUM` (applicable to multi-data center clusters only)
- `EACH_QUORUM` (applicable to multi-data center clusters only)
- `ALL`

See [tunable consistency](#) for more information about the different consistency levels.

CQL Storage Parameters

Certain CQL commands allow a `WITH` clause for setting certain properties on a keyspace or column family. Note that CQL does not currently offer support for using *all of the Cassandra column family attributes* as CQL storage parameters, just a subset.

CQL Keyspace Storage Parameters

The `CREATE KEYSPACE` and `ALTER KEYSPACE` commands support setting the following keyspace properties.

- *strategy_class* The name of the replication strategy: `SimpleStrategy` or `NetworkTopologyStrategy`
- *strategy_options* Replication strategy option names are appended to the `strategy_options` keyword using a colon (:). For example: `strategy_options:DC1=1` or `strategy_options:replication_factor=3`
- *durable_writes* True or false. Exercise caution as described in [durable_writes](#).

CQL 3 Column Family Storage Parameters

CQL supports Cassandra column family attributes through the CQL parameters in the following table. In a few cases, the CQL parameters have slightly different names than their corresponding *column family attributes*:

- The CQL parameter `compaction_strategy_class` corresponds to the column family attribute `compaction_strategy`.
- The CQL parameter `compression_parameters` corresponds to the column family attribute `compression_options`.

CQL Parameter Name	Default Value
<i>bloom_filter_fp_chance</i>	0
<i>caching</i>	keys_only
<i>compaction_strategy_class</i>	SizeTieredCompactionStrategy
<i>compaction_strategy_options</i>	none

<i>compression_parameters</i>	SnappyCompressor
<i>comment</i>	"(an empty string)"
<i>dclocal_read_repair_chance</i>	0.0
<i>gc_grace_seconds</i>	864000
<i>max_compaction_threshold</i>	32
<i>min_compaction_threshold</i>	4
<i>read_repair_chance</i>	0.1
<i>replicate_on_write</i>	false

A brief tutorial on *using CQL3 commands* is also provided.

CQL Command Reference

The command reference covers CQL and CQLsh, which is the CQL client. Using cqlsh, you can query the Cassandra database from the command line. All of the commands included in CQL are available on the CQLsh command line. The CQL command reference includes a synopsis, description, and examples of each CQL 3 command. These topics are covered in the following sections.

ALTER TABLE

Manipulates the column metadata of a column family.

Synopsis

```
ALTER TABLE [<keyspace_name>].<column_family>
  (ALTER <column_name> TYPE <data_type>
  | ADD <column_name> <data_type>
  | DROP <column_name>
  | WITH <optionname> = <val> [AND <optionname> = <val> [...]]);
```

Description

ALTER TABLE manipulates the column family metadata. You can change the data storage type of columns, add new columns, drop existing columns, and change column family properties. No results are returned.

You can also use the alias ALTER COLUMNFAMILY.

See *CQL Data Types* for the available data types and *CQL 3 Column Family Storage Parameters* for column properties and their default values.

First, specify the name of the column family to be changed after the ALTER TABLE keywords, followed by the type of change: ALTER, ADD, DROP, or WITH. Next, provide the rest of the needed information, as explained in the following sections.

You can qualify column family names by keyspace. For example, to alter the addamsFamily table in the monsters keyspace:

```
ALTER TABLE monsters.addamsFamily ALTER lastKnownLocation TYPE uuid;
```

Changing the Type of a Typed Column

To change the storage type for a column of type ascii to type text, use ALTER TABLE and the ALTER and TYPE keywords in the following way:

```
ALTER TABLE addamsFamily ALTER lastKnownLocation TYPE text;
```


References

The column must already have a type in the column family metadata and the old type must be compatible with the new type. The column may or may not already exist in current rows (no validation of existing data occurs). The bytes stored in values for that column remain unchanged, and if existing data is not deserializable according to the new type, your CQL driver or interface might report errors.

Adding a Typed Column

To add a typed column to a column family, use `ALTER TABLE` and the `ADD` keyword in the following way:

```
ALTER TABLE addamsFamily ADD gravesite varchar;
```

The column must not already have a type in the column family metadata. The column may or may not already exist in current rows (no validation of existing data occurs).

Dropping a Typed Column

To drop a typed column from the column family metadata, use `ALTER TABLE` and the `DROP` keyword in the following way:

```
ALTER TABLE addamsFamily DROP gender;
```

Dropping a typed column does not remove the column from current rows; it just removes the metadata saying that the bytes stored under that column are expected to be deserializable according to a certain type.

Modifying Column Family Options

To change the column family storage options established during creation of the column family, use `ALTER TABLE` and the `WITH` keyword. To change multiple properties, use `AND` as shown in this example:

```
ALTER TABLE addamsFamily WITH comment = 'A most excellent and useful column family'
AND read_repair_chance = 0.2;
```

See [CQL 3 Column Family Storage Parameters](#) for the column family options you can define.

Changing any compaction or compression setting erases all previous `compaction_strategy_options` or `compression_parameters` settings, respectively.

Examples

```
ALTER TABLE users ALTER email TYPE varchar;
```

```
ALTER TABLE users ADD gender varchar;
```

```
ALTER TABLE users DROP gender;
```

```
ALTER TABLE users WITH comment = 'active users' AND read_repair_chance = 0.2;
```

```
ALTER TABLE addamsFamily WITH
compression_parameters:ssstable_compression = 'DeflateCompressor'
AND compression_parameters:chunk_length_kb = 64;
```

```
ALTER TABLE users
WITH compaction_strategy_class='SizeTieredCompactionStrategy'
AND min_compaction_threshold = 6;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>

<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

ALTER KEYSPACE

Change attributes of a keyspace.

Synopsis

```
ALTER KEYSPACE <ks_name>
  WITH strategy_class = <value>
  [ AND strategy_options:<option> = <value> [...]
  [ AND durable_writes = true | false ] ];
```

Description

You can use ALTER KEYSPACE in Cassandra 1.1.6 and later to change the keyspace attributes:

- *replica placement strategy*
- *strategy options*
- *durable_writes*

You cannot change the name of the keyspace. The ALTER KEYSPACE statement must include setting the strategy class even when there is no change to the strategy class.

Example

Continuing with the last example in *CREATE KEYSPACE*, change the definition of the MyKeyspace keyspace to set durable_writes to false for purposes of this example. Changing the durable_writes attribute is not recommended because it could result in data loss.

Note

These examples work with pre-release CQL 3 in Cassandra 1.1.x. The syntax differs in the release version of CQL 3 in Cassandra 1.2 and later.

```
ALTER KEYSPACE "MyKeyspace"
  WITH strategy_class = NetworkTopologyStrategy
  AND durable_writes = false;
```

References

Check the keyspace settings:

```
SELECT * from system.schema_keyspaces;
```

The output describes the defined keyspace and looks something like this:

keyspace	durable_writes	name	strategy_class	strategy_options
MyKeyspace	False	MyKeyspace	NetworkTopologyStrategy	

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

BATCH

Sets a global consistency level and client-supplied timestamp for all columns written by the statements in the batch.

Synopsis

```
BEGIN BATCH
```

```
[ USING <write_option> [ AND <write_option> [...] ] ];
```

```
<dml_statement>
```

```
<dml_statement>
```

```
[...]
```

```
APPLY BATCH;
```

<write_option> is:

```
USING CONSISTENCY <consistency_level>
```

```
TIMESTAMP <integer>
```

Description

A **BATCH** statement combines multiple data modification (DML) statements into a single logical operation. **BATCH** supports setting a client-supplied, global consistency level and timestamp that is used for each of the operations included in the batch.

You can specify these global options in the **USING** clause:

- *Consistency level*
- Timestamp (current time)

Batched statements default to a consistency level of **ONE** when unspecified.

After the **USING** clause, you can add only these DML statements:

- *INSERT*
- *UPDATE*
- *DELETE*

Individual DML statements inside a **BATCH** cannot specify a consistency level or timestamp. These individual statements can specify a TTL (time to live). TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

Close the batch statement with **APPLY BATCH**.

BATCH is not an analogue for SQL ACID transactions. Column updates are considered atomic and isolated within a given record (row) only.

Example

```
BEGIN BATCH USING CONSISTENCY QUORUM
  INSERT INTO users (userID, password, name) VALUES ('user2', 'ch@ngem3b', 'second user')
  UPDATE users SET password = 'ps22dhds' WHERE userID = 'user2'
  INSERT INTO users (userID, password) VALUES ('user3', 'ch@ngem3c')
  DELETE name FROM users WHERE userID = 'user2'
  INSERT INTO users (userID, password, name) VALUES ('user4', 'ch@ngem3c', 'Andrew')
APPLY BATCH;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	

<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

CREATE TABLE

Define a new column family.

Synopsis

```
CREATE TABLE <column family name>
  (<column_name> <type>,
  [<column_name2> <type>, ...]
  PRIMARY KEY (<column_name> <type>
  [, <column_name2> <type>,...])
  [WITH <option name> = <value>
  [AND <option name> = <value> [...]]];
```

Description

CREATE TABLE creates new column family namespaces under the current keyspace. You can also use the alias CREATE COLUMNFAMILY. Valid column family names are strings of alphanumeric characters and underscores, which begin with a letter.

The only schema information that must be defined for a column family is the primary key (or row key) and its associated data type. Other column metadata, such as the size of the associated row and key caches, can be defined.

```
CREATE TABLE users (
  user_name varchar PRIMARY KEY,
  password varchar,
  gender varchar,
  session_token varchar,
  state varchar,
  birth_year bigint
);
```

```
CREATE TABLE emp (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID)
);
```

Specifying the Key Type

When creating a new column family, specify PRIMARY KEY. It probably does not make sense to use counter for a key. The key type must be compatible with the partitioner in use. For example, OrderPreservingPartitioner and CollatingOrderPreservingPartitioner (deprecated partitioners) require UTF-8 keys.

Using Composite Primary Keys

When you use composite keys in CQL, Cassandra supports wide Cassandra rows using composite column names. In CQL 3, a primary key can have any number (1 or more) of component columns, but there must be at least one column in the column family that is not part of the primary key. The new wide row technique consumes more storage because for every piece of data stored, the column name is stored along with it.

References

```
cqlsh> CREATE TABLE History.tweets (  
    tweet_id uuid PRIMARY KEY,  
    author varchar,  
    body varchar);  
  
cqlsh> CREATE TABLE timeline (  
    user_id varchar,  
    tweet_id uuid,  
    author varchar,  
    body varchar,  
    PRIMARY KEY (user_id, tweet_id));
```

Using Compact Storage

When you create a table using composite primary keys, rows can become very wide because for every piece of data stored, the column name needs to be stored along with it. Instead of each non-primary key column being stored such that each column corresponds to one column on disk, an entire row is stored in a single column on disk. If you need to conserve disk space, use the `WITH COMPACT STORAGE` directive that stores data essentially the same as it was stored under CQL 2.

```
CREATE TABLE sblocks (  
    block_id uuid,  
    subblock_id uuid,  
    data blob,  
    PRIMARY KEY (block_id, subblock_id)  
)  
WITH COMPACT STORAGE;
```

Using the compact storage directive prevents you from defining more than one column that is not part of a compound primary key. A compact table using a primary key that is not compound can have multiple columns that are not part of the primary key. Each logical row corresponds to exactly one physical column, as shown in the *tweets timeline example*.

Updates to data in a table created with compact storage are not allowed.

Unless you specify `WITH COMPACT STORAGE`, CQL creates a column family with non-compact storage.

Specifying Column Types

You assign columns a type during column family creation. Column types are specified as a parenthesized, comma-separated list of column term and type pairs. See *CQL Data Types* for the available types.

Column Family Options (not required)

Using the `WITH` clause and optional keyword arguments, you can control the configuration of a new column family. See *CQL 3 Column Family Storage Parameters* for the column family options you can define.

Examples

```
cqlsh> use zoo;  
  
cqlsh:zoo> CREATE TABLE MonkeyTypes (  
    block_id uuid,  
    species text,  
    alias text,  
    population varint,  
    PRIMARY KEY (block_id)  
)  
WITH comment='Important biological records'  
AND read_repair_chance = 1.0;
```

References

```
cqlsh:zoo> CREATE TABLE DogTypes (  
    block_id uuid,  
    species text,  
    alias text,  
    population varint,  
    PRIMARY KEY (block_id)  
)  
WITH compression_parameters:sstable_compression = 'SnappyCompressor'  
AND compression_parameters:chunk_length_kb = 128;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

CREATE INDEX

Define a new, secondary index on a single, typed column of a column family.

Synopsis

```
CREATE INDEX [<index_name>]  
ON <cf_name> (<column_name>);
```

Description

CREATE INDEX creates a new, automatic secondary index on the given column family for the named column. Optionally, specify a name for the index itself before the ON keyword. Enclose a single column name in parentheses. It is not necessary for the column to exist on any current rows because Cassandra is schema-optional. The column must already have a type specified when the family was created, or added afterward by altering the column family.

```
CREATE INDEX userIndex ON NerdMovies (user);  
CREATE INDEX ON Mutants (abilityId);
```

Examples

References

Define a static column family and then create a secondary index on two of its named columns:

```
CREATE TABLE users (  
    userID uuid,  
    firstname text,  
    lastname text,  
    email text,  
    address text,  
    zip int,  
    state text,  
    PRIMARY KEY (userID)  
);
```

```
CREATE INDEX user_state  
ON users (state);
```

```
CREATE INDEX ON users (zip);
```

CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	ASSUME
ALTER KEYSPACE	CAPTURE
BATCH	COPY
CREATE TABLE	DESCRIBE
CREATE INDEX	EXIT
CREATE KEYSPACE	SHOW
DELETE	SOURCE
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

CREATE KEYSPACE

Define a new keyspace and its replica placement strategy.

Synopsis

```
CREATE KEYSPACE <ks_name>  
    WITH strategy_class = <value>  
    [ AND strategy_options:<option> = <value> [...] ]  
    [ AND durable_writes = true | false ] ;
```

Description

`CREATE KEYSPACE` creates a top-level namespace and sets the replica placement strategy, associated replication factor options, and the *durable writes* option for the keyspace. Valid keyspace names are strings of alpha-numeric characters and underscores, and must begin with a letter. Enclose keyspace names in double quotation marks to preserve case-sensitive names; otherwise, Cassandra stores keyspace names in lowercase. As of Cassandra 1.1.6, you do not need to enclose the noncase-sensitive keyspace name, strategy, and option strings in single quotation marks. Properties such as replication strategy and count are specified during creation using the following accepted keyword arguments:

Keyword	Description
<code>strategy_class</code>	Required. The name of the <i> replica placement strategy </i> for the new keyspace, such as <code>SimpleStrategy</code> and <code>NetworkTopologyStrategy</code> .
<code>strategy_options</code>	Optional. Additional arguments appended to the option name.
<code>durable_writes</code>	Optional. True or false. Use caution. See <i> durable writes </i> .

Use the `strategy_options` keyword, separated by a colon to specify a strategy option. For example, a strategy option of `DC1` with a value of `1` would be specified as `strategy_options:DC1 = 1`; `replication_factor` for `SimpleStrategy` could be `strategy_options:replication_factor=3`.

To use the `NetworkTopologyStrategy`, you specify the number of replicas per data center in this format:

```
strategy_options:<datacenter_name>=<number>
```

See *About Keyspaces* for more information.

Examples

Define a new keyspace using the simple replication strategy:

Note

These examples work with pre-release CQL 3 in Cassandra 1.1.x. The syntax differs in the release version of CQL 3 in Cassandra 1.2 and later.

```
CREATE KEYSPACE History WITH strategy_class = SimpleStrategy
AND strategy_options:replication_factor = 1;
```

Define the number of replicas for one data center using the network-aware replication strategy, `NetworkTopologyStrategy`:

```
CREATE KEYSPACE test
WITH strategy_class = 'NetworkTopologyStrategy'
AND strategy_options:"us-east" = 6;
```

Double quotation marks must be used to enclose the `us-east` data center name because it contains a hyphen.

For a multiple data center cluster, assuming you are using the *PropertyFileSnitch* and your data centers are named `DC1` and `DC2` in the `cassandra-topology.properties` file.:

```
CREATE KEYSPACE test2
WITH strategy_class = 'NetworkTopologyStrategy'
AND strategy_options:DC1 = 3
AND strategy_options:DC2 = 6;
```

Change the `durable_writes` attribute of a keyspace.

```
CREATE KEYSPACE test2
WITH strategy_class = NetworkTopologyStrategy
AND strategy_options:DC1 = 3 AND durable_writes=false;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

DELETE

Removes one or more columns from the named row(s).

Synopsis

```
DELETE [<column_name> [, ...]]
  FROM [keyspace.]<column_family>
[USING CONSISTENCY <consistency_level> [AND TIMESTAMP <integer>]]
WHERE <row_specification>;
```

<row_specification> is:

```
<primary key name> = <key_value>
<primary key name> IN (<key_value> [,...])
```

Description

A **DELETE** statement removes one or more columns from one or more rows in the named column family.

Specifying Columns

After the **DELETE** keyword, optionally list column names, separated by commas.

```
DELETE col1, col2, col3 FROM Planetears USING CONSISTENCY ONE WHERE userID = 'Captain';
```

When no column names are specified, the entire row(s) specified in the **WHERE** clause are deleted.

```
DELETE FROM MastersOfTheUniverse WHERE mastersID IN ('Man-At-Arms', 'Teela');
```

Specifying the Column Family

The column family name follows the list of column names and the keyword **FROM**.

Specifying Options

You can specify these options:

- *Consistency level*
- Timestamp (current time)

When a column is deleted, it is not removed from disk immediately. The deleted column is marked with a tombstone and then removed after the configured grace period has expired. The optional timestamp defines the new tombstone record. See [About Deletes](#) for more information about how Cassandra handles deleted columns and rows.

Specifying Rows

The WHERE clause specifies which row or rows to delete from the column family.

```
DELETE coll FROM SomeColumnFamily WHERE userID = 'some_key_value';
```

This form provides a list of key names using the IN notation and a parenthetical list of comma-delimited keyname terms.

```
DELETE coll FROM SomeColumnFamily WHERE userID IN (key1, key2);
```

Example

```
DELETE email, phone
FROM users
USING CONSISTENCY QUORUM AND TIMESTAMP 1318452291034
WHERE user_name = 'jsmith';
```

```
DELETE phone FROM users WHERE user_name IN ('jdoe', 'jsmith');
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

DROP TABLE

References

Removes the named column family.

Synopsis

```
DROP TABLE <name>;
```

Description

A **DROP TABLE** statement results in the immediate, irreversible removal of a column family, including all data contained in the column family. You can also use the alias **DROP TABLE**.

Example

```
DROP TABLE worldSeriesAttendees;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

DROP INDEX

Drops the named secondary index.

Synopsis

```
DROP INDEX <name>;
```

Description

A **DROP INDEX** statement removes an existing secondary index. If the index was not given a name during creation, the index name is `<columnfamily_name>_<column_name>_idx`.

Example

References

```
DROP INDEX user_state;
```

```
DROP INDEX users_zip_idx;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

DROP KEYSPACE

Removes the keyspace.

Synopsis

```
DROP KEYSPACE <name>;
```

Description

A `DROP KEYSPACE` statement results in the immediate, irreversible removal of a keyspace, including all column families and data contained in the keyspace.

Example

```
DROP KEYSPACE MyTwitterClone;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>

<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

INSERT

Adds or updates one or more columns in the identified row of a column family.

Synopsis

```
INSERT INTO [keyspace.]<column_family>
  (<keyname>, <colname> [, ...]) VALUES
  (<keyvalue>, <colvalue> [, ...])
[USING <consistency>
[AND TIMESTAMP <timestamp>]
[AND TTL <timetolive>]];
```

Description

An `INSERT` writes one or more columns to a record in a Cassandra column family. No results are returned. The first column name in the `INSERT` list must be the name of the column family key. Also, there must be more than one column name specified (Cassandra rows are not considered to exist with only a key and no associated columns).

The first column value in the `VALUES` list is the row key value to insert. List column values in the same order as the column names are listed in the `INSERT` list. If a row or column does not exist, it will be inserted. If it does exist, it will be updated.

Unlike SQL, the semantics of `INSERT` and `UPDATE` are identical. In either case a record is created if none existed before, and updated when it does.

You can qualify column family names by keyspace. For example, to insert a column into the `NerdMovies` table in the `oscar_winners` keyspace:

```
INSERT INTO Hollywood.NerdMovies (user_uuid, fan)
VALUES ( 'cfd66ccc-d857-4e90-b1e5-df98a3d40cd6', 'johndoe' )
```

Specifying Options

You can specify these options:

- *Consistency level*
- Time-to-live (TTL) in seconds
- Timestamp (current time)

TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

```
INSERT INTO Hollywood.NerdMovies (user_uuid, fan)
VALUES ('cfd66ccc-d857-4e90-b1e5-df98a3d40cd6', 'johndoe')
USING CONSISTENCY LOCAL_QUORUM AND TTL 86400;
```

Example

```
INSERT INTO History.tweets (tweet_id, author, body)
VALUES (1742, 'gwashtington', 'I chopped down the cherry tree');

INSERT INTO History.timeline (user_id, tweet_id, author, body)
VALUES ('gmason', 1765, 'phenry', 'Give me liberty or give me death');
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

SELECT

Retrieves data, including Solr data, from a Cassandra column family.

Synopsis

```
SELECT <select expression>
FROM <column family>
[USING CONSISTENCY <level>]
[WHERE (<clause>) [AND (<clause>) ...]] [LIMIT <n>]
[ORDER BY <composite key 2>] [ASC, DESC]
```

<clause> syntax is:

```
<relation> [<AND relation> ...]
<primary key name> { = | < | > | <= | >= } <key_value>
<primary key name> IN (<key_value> [,...])
```

Description

A `SELECT` expression reads one or more records from a Cassandra column family and returns a result-set of rows. Each row consists of a row key and a collection of columns corresponding to the query.

Unlike the projection in a SQL `SELECT`, there is no guarantee that the results will contain all of the columns specified because Cassandra is schema-optional. An error does not occur if you request non-existent columns.

Examples

Specifying Columns

The `SELECT` expression determines which columns, if any, appear in the result:

```
SELECT * from People;
```

Select two columns, Name and Occupation, from three rows having employee ids (primary key) 199, 200, or 207:

```
SELECT Name, Occupation FROM People WHERE empID IN (199, 200, 207);
```

A simple form is a comma-separated list of column names. The list can consist of a range of column names.

Counting Returned Rows

A `SELECT` expression using `COUNT(*)` returns the number of rows that matched the query. Alternatively, you can use `COUNT(1)` to get the same result.

Count the number of rows in the users column family:

```
SELECT COUNT(*) FROM users;
```

Using the `LIMIT` option, you can specify that the query limit number of rows returned. For example, the output of these statements if you had 105,291 rows in the database would be: 50000 and 105291

```
SELECT COUNT(*) FROM big_columnfamily LIMIT 50000;
```

```
count
-----
50000
```

```
SELECT COUNT(*) FROM big_columnfamily LIMIT 200000;
```

```
count
-----
105291
```

Specifying the Column Family, FROM, Clause

The `FROM` clause specifies the column family to query. Optionally, specify a keyspace for the column family followed by a period, (`.`), then the column family name. If a keyspace is not specified, the current keyspace will be used.

Count the number of rows in the Migrations column family in the system keyspace:

```
SELECT COUNT(*) FROM system.Migrations;
```

Specifying a Consistency Level

You can optionally specify a *consistency level*, such as `QUORUM`:

```
SELECT * from People USING CONSISTENCY QUORUM;
```

See *tunable consistency* for more information about the consistency levels.

Filtering Data Using the WHERE Clause

References

The `WHERE` clause filters the rows that appear in the results. You can filter on a key name, a range of keys, or on column values if columns have a secondary index. Row keys are specified using the `KEY` keyword or key alias defined on the column family, followed by a relational operator, and then a value.

Relational operators are: `=`, `>`, `>=`, `<`, or `<=`.

To filter a indexed column, the term on the left of the operator must be the name of the column, and the term on the right must be the value to filter on.

In CQL2, the greater-than and less-than operators (`>` and `<`) result in key ranges that are inclusive of the terms. There is no supported notion of strictly greater-than or less-than; these operators are merely supported as aliases to `>=` and `<=`. In CQL3, this is no longer the case. Strict bounds are respected in CQL3. For example:

```
CREATE TABLE scores (  
    name text,  
    score int,  
    date timestamp,  
    PRIMARY KEY (name, score)  
);
```

```
INSERT INTO scores (name, score, date) VALUES ('bob', 42, '2012-06-24');  
INSERT INTO scores (name, score, date) VALUES ('bob', 47, '2012-06-25');  
INSERT INTO scores (name, score, date) VALUES ('bob', 33, '2012-06-26');  
INSERT INTO scores (name, score, date) VALUES ('bob', 40, '2012-06-27');
```

```
SELECT date, score FROM scores WHERE name='bob' AND score >= 40;
```

date	score
2012-06-27 00:00:00+0000	40
2012-06-24 00:00:00+0000	42
2012-06-25 00:00:00+0000	47

```
SELECT date, score FROM scores WHERE name='bob' AND score > 40;
```

date	score
2011-06-24 00:00:00+0000	42
2011-06-25 00:00:00+0000	47

WHERE clauses can include comparisons on columns other than the first. As long as all previous key-component columns have already been identified with strict `=` comparisons, the last given key component column can be any sort of comparison.

Sorting Filtered Data

ORDER BY clauses can only select a single column, and that column has to be the second column in a composite PRIMARY KEY. This holds even for tables with more than 2 column components in the primary key. Ordering can be done in ascending or descending order, default ascending, and specified with the `ASC` or `DESC` keywords.

The column you ORDER BY has to be part of a composite primary key.

```
SELECT * FROM emp where empid IN (103, 100) ORDER BY deptid ASC;
```

This query returns:

empid	deptid	first_name	last_name
103	11	charlie	brown
100	10	john	doe

References

```
SELECT * FROM emp where empid IN (103, 100) ORDER BY deptid DESC;
```

This query returns:

empid	deptid	first_name	last_name
100	10	john	doe
103	11	charlie	brown

Specifying Rows Returned Using LIMIT

By default, a query returns 10,000 rows maximum. Using the LIMIT clause, you can change the default limit of 10,000 rows to a lesser or greater number of rows. The default is 10,000 rows.

```
SELECT * from emp where deptid=10 LIMIT 900;
```

Querying System Tables

In CQL3, you can query system tables directly to obtain cluster schema information. For example:

```
USE my_ks;
```

```
SELECT * FROM system.schema_keyspaces;
```

Output

keyspace	durable_writes	name	strategy_class	strategy_options
my_ks	True	test	org.apache.cassandra.locator.SimpleStrategy	{"replication_factor": "1"}

Examples

Captain Reynolds keeps track of every ship registered by his sensors as he flies through space. Each day-code is a Cassandra row, and events are added in with timestamps.

```
CREATE TABLE seen_ships (  
    day text,  
    time_seen timestamp,  
    shipname text,  
    PRIMARY KEY (day, time_seen));
```

```
SELECT * FROM seen_ships  
WHERE day='199-A/4'  
AND time_seen > '7943-02-03'  
AND time_seen < '7943-02-28'  
LIMIT 12;
```

Set up the tweets and timeline tables described in the [Composite Columns section](#), *insert the example data*, and use this query to get the tweets of a George Mason's followers.

```
SELECT * FROM timeline WHERE user_id = gmason  
ORDER BY tweet_id DESC;
```

Output

user_id	tweet_id	author	body
gmason	148e9150-1dd2-11b2-0000-242d50cf1fbf	phenry	Give me liberty or give me death
gmason	148b0ee0-1dd2-11b2-0000-242d50cf1fbf	gWashington	I chopped down the cherry tree

References

As of Cassandra 1.1.3, you can also use the IN operator, as shown in this example, to get the tweets of George Mason and Alexander Hamilton's followers:

```
SELECT * FROM timeline WHERE user_id IN ('gmason', 'ahamilton') ORDER BY tweet_id;
```

Output

user_id	tweet_id	author	body
gmason	148b0ee0-1dd2-11b2-0000-e547f5a51eff	gWashington	I chopped down the cherry tree
ahamilton	148b0ee0-1dd2-11b2-0000-e547f5a51eff	gWashington	I chopped down the cherry tree
gmason	148e9150-1dd2-11b2-0000-e547f5a51eff	phenry	Give me liberty, or give me death
ahamilton	14937350-1dd2-11b2-0000-e547f5a51eff	jadams	A government of laws, not men

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

TRUNCATE

Removes all data from a column family.

Synopsis

```
TRUNCATE [keyspace.]<column_family>;
```

Description

A `TRUNCATE` statement results in the immediate, irreversible removal of all data in the named column family.

Example

```
TRUNCATE user_activity;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

UPDATE

Updates one or more columns in the identified row of a column family.

Synopsis

```
UPDATE <column_family>
[ USING <write_option> [ AND <write_option> [...] ] ];
SET <column_name> = <column_value> [, ...]
| <counter_column_name> = <counter_column_name> {+ | -} <integer>
WHERE <row_specification>;
```

<write_option> is:

```
CONSISTENCY <consistency_level>
TTL <seconds>
TIMESTAMP <integer>
```

``<row_specification>`` is:

```
<primary/composite key name> = <key_value>
<primary/composite key name> IN (<key_value> [,...])
```

Description

An `UPDATE` writes one or more columns to a record in a Cassandra column family. No results are returned. Row/column records are created if they do not exist, or overwritten if they do exist.

A statement begins with the `UPDATE` keyword followed by a Cassandra column family name. To update multiple columns, separate the name/value pairs using commas.

The `SET` clause specifies the new column name/value pairs to update or insert. Separate multiple name/value pairs using commas. If the named column exists, its value is updated, otherwise, its value is inserted. To update a counter

References

column value in a counter column family, specify a value to increment or decrement value the current value of the counter column.

Each update statement requires a precise set of row keys to be specified using a `WHERE` clause.

```
UPDATE Movies SET col1 = val1, col2 = val2 WHERE movieID = key1;
UPDATE Movies SET col3 = val3 WHERE movieID IN (key1, key2, key3);
UPDATE Movies SET col4 = 22 WHERE movieID = key4;
```

```
UPDATE NerdMovies USING CONSISTENCY ALL AND TTL 400
  SET 'A 1194' = 'The Empire Strikes Back',
    'B 1194' = 'Han Solo'
  WHERE movieID = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

```
UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
```

You can specify these options:

- *Consistency level*
- Time-to-live (TTL)
- Timestamp (current time)

TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

Examples

Update a column in several rows at once:

```
UPDATE users USING CONSISTENCY QUORUM
  SET 'state' = 'TX'
  WHERE user_uuid IN (88b8fd18-bled-4e96-bf79-4280797cba80,
                     06a8913c-c0d6-477c-937d-6c1b69a95d43,
                     bc108776-7cb5-477f-917d-869c12dffffa8);
```

Update several columns in a single row:

```
UPDATE users USING CONSISTENCY QUORUM
  SET 'name' = 'John Smith', 'email' = 'jsmith@cassie.com'
  WHERE user_uuid = 88b8fd18-bled-4e96-bf79-4280797cba80;
```

Update the value of a counter column:

```
UPDATE page_views USING CONSISTENCY QUORUM AND TIMESTAMP=1318452291034
  SET 'index.html' = 'index.html' + 1
  WHERE url_key = 'www.datastax.com';
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>

References

<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

USE

Connects the current client session to a keyspace.

Synopsis

```
USE <keyspace_name>;
```

Description

A `USE` statement tells the current client session and the connected Cassandra instance which keyspace you will be working in. All subsequent operations on column families and indexes are in the context of the named keyspace, unless otherwise specified or until the client connection is terminated or another `USE` statement is issued.

Example

```
USE PortfolioDemo;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

CQLsh Command Reference

The CQLsh Command Reference describes each command not included in CQL, but available in the CQL 3 shell program. These topics are covered in the following sections.

ASSUME

Sets the client-side encoding for a `cqlsh` session.

Synopsis

```
ASSUME [<keyspace_name>].<columnfamily_name>
      <storage_type_definition>
      [, ...] ;
```

<storage_type_definition> is:

```
(<primary key column_name>) VALUES ARE <datatype>
| NAMES ARE <datatype>
| VALUES ARE <datatype>
```

Description

Cassandra is a schema-optional data model, meaning that column families are not required to have data type information explicitly defined for column names, column values or row key values. When type information is not explicitly defined, and implicit typing cannot be determined, data is displayed as raw hex bytes (`blob` type), which is not human-readable. The `ASSUME` command allows you to specify type information for particular column family values passed between the `cqlsh` client and the Cassandra server.

The name of the column family (optionally prefix the keyspace) for which to specify assumed types follows the `ASSUME` keyword. If keyspace is not supplied, the keyspace default is the currently connected one. Next, list the assumed type for column name, preceded by `NAMES ARE`, then list the assumed type for column values preceded by `VALUES ARE`.

To declare an assumed type for a particular column, such as the row key, use the column family row key name.

Examples

```
ASSUME users NAMES ARE text, VALUES are text;
```

```
ASSUME users(user_id) VALUES are uuid;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	

<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

CAPTURE

Captures command output and appends it to a file.

Synopsis

```
CAPTURE ['<file>' | OFF];
```

Description

To start capturing the output of a CQL query, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to \$HOME:

Example

```
CAPTURE '~/mydir/myfile.txt';
```

Output is not shown on the console while it is captured. Only query result output is captured. Errors and output from cqlsh-only commands still appear.

To stop capturing output and return to normal display of output, use CAPTURE OFF.

To determine the current capture state, use CAPTURE with no arguments.

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	

USE

COPY

Imports and exports CSV (comma-separated values) data to and from Cassandra 1.1.3 and higher.

Synopsis

```
COPY <column family name> [ ( column [, ...] ) ]
FROM ( '<filename>' | STDIN )
[ WITH <option>='value' [AND ...] ];
```

```
COPY <column family name> [ ( column [, ...] ) ]
TO ( '<filename>' | STDOUT )
[ WITH <option>='value' [AND ...] ];
```

Description

Using the COPY options in a WITH clause, you can change the format of the CSV format. The following table describes these options:

COPY Options	Default Value	Use To
DELIMITER	comma (,)	Set the character that separates fields in the file.
QUOTE	quotation mark(")	Set the character that encloses field values. ^[1]
ESCAPE	backslash (\)	Set the character that escapes literal uses of the QUOTE character.
HEADER	false	Set true to indicate that first row of the file is a header.
ENCODING	UTF8	Set the COPY TO command to output unicode strings.
NULL	an empty string	Set the COPY TO command to represent the absence of a value.

The ENCODING and NULL options cannot be used in the COPY FROM command.

This table shows that, by default, Cassandra expects the CSV data to consist of fields separated by commas (,), records separated by line separators (a newline, \r\n), and field values enclosed in double-quotation marks ("""). Also, to avoid ambiguity, escape a literal double-quotation mark using a backslash inside a string enclosed in double-quotation marks (""). By default, Cassandra does not expect the CSV file to have a header record on the first line that consists of the column names. COPY TO includes the header in the output if HEADER=true. COPY FROM ignores the first line if HEADER=true.

COPY FROM a CSV File

By default, when you use the COPY FROM command, Cassandra expects every row in the CSV input to contain the same number of columns. The number of columns in the CSV input is the same as the number of columns in the Cassandra table metadata. Cassandra assigns fields in the respective order. To apply your input data to a particular set of columns, specify the column names in parentheses after the table name.

Note

COPY FROM is intended for importing small datasets (a few million rows or less) into Cassandra. For importing larger datasets, use *Cassandra Bulk Loader* or the *sstable2json / json2sstable* utility.

COPY TO a CSV File

For example, assume you have the following table in CQL:

```
cqlsh> SELECT * FROM test.airplanes;
```

name	mach	manufacturer	year
P38-Lightning	0.7	Lockheed	1937

After inserting data into the table, you can copy the data to a CSV file in another order by specifying the column names in parentheses after the table name:

```
COPY airplanes
(name, mach, year, manufacturer)
TO 'temp.csv'
```

Specifying the Source or Destination Files

Specify the source file of the CSV input or the destination file of the CSV output by a file path. Alternatively, you can use the STDIN or STDOUT keywords to import from standard input and export to standard output. When using stdin, signal the end of the CSV data with a backslash and period ("\.") on a separate line. If the data is being imported into a column family that already contains data, COPY FROM does not truncate the column family beforehand.

Examples**Copy a Column Family to a CSV File**

1. Create a keyspace, and using the keyspace, create a column family named airplanes and copy it to a CSV file.

```
CREATE TABLE airplanes (
  name text PRIMARY KEY,
  manufacturer ascii,
  year int,
  mach float
);
```

```
INSERT INTO airplanes
(name, manufacturer, year, mach)
VALUES ('P38-Lightning', 'Lockheed', 1937, '.7');
```

```
COPY airplanes (name, manufacturer, year, mach) TO 'temp.csv';
1 rows exported in 0.004 seconds.
```

2. Clear the data from the airplanes column family and import the data from the temp.csv file

```
TRUNCATE airplanes;
```

```
COPY airplanes (name, manufacturer, year, mach) FROM 'temp.csv';
1 rows imported in 0.087 seconds.
```

Copy Data from Standard Input to a Column Family

References

1. Enter data directly during an interactive cqlsh session, using the COPY command defaults.

```
COPY airplanes (name, manufacturer, year, mach) FROM STDIN;
```

2. At the [copy] prompt, enter the following data:

```
"F-14D Super Tomcat", Grumman, "1987", "2.34"  
"MiG-23 Flogger", Russian-made, "1964", "2.35"  
"Su-27 Flanker", U.S.S.R., "1981", "2.35"  
\.  
3 rows imported in 55.204 seconds.
```

3. Query the airplanes column family to see data imported from STDIN:

```
SELECT * FROM airplanes;
```

Output

name	manufacturer	year	mach
F-14D Super Tomcat	Grumman	1987	2.35
P38-Lightning	Lockheed	1937	0.7
Su-27 Flanker	U.S.S.R.	1981	2.35
MiG-23 Flogger	Russian-made	1967	2.35

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

DESCRIBE

Provides information about the connected Cassandra cluster, or about the data objects stored in the cluster.

Synopsis

```

DESCRIBE CLUSTER | SCHEMA
                  | KEYSPACE [<keyspace_name>]
                  | COLUMNFAMILIES
                  | COLUMNFAMILY <columnfamily_name>

```

Description

The DESCRIBE or DESC command outputs information about the connected Cassandra cluster, or about the data stored on it. In CQL3, you can *query the system tables* directly using SELECT. Use DESCRIBE in one of the following ways:

- **CLUSTER:** Describes the Cassandra cluster, such as the cluster name, partitioner, and snitch configured for the cluster. When connected to a non-system keyspace, this form of the command also shows the data endpoint ranges owned by each node in the Cassandra ring.
- **SCHEMA:** Lists CQL commands that you can use to recreate the column family schema. Works as though DESCRIBE KEYSPACE k was invoked for each keyspace k. May also show metadata about the column family.
- **KEYSPACE:** Yields all the information that CQL is capable of representing about the keyspace. From this information, you can recreate the given keyspace and the column families in it. Omit the <keyspace_name> argument when using a non-system keyspace to get a description of the current keyspace.
- **COLUMNFAMILIES:** Lists the names of all column families in the current keyspace or in all keyspaces if there is no current keyspace.
- **COLUMNFAMILY:** Yields all the information that CQL is capable of representing about the column family.

To obtain additional information about the cluster schema, you can *query system tables* in CQL3.

Examples

```
DESCRIBE CLUSTER;
```

```
DESCRIBE KEYSPACE PortfolioDemo;
```

```
DESCRIBE COLUMNFAMILIES;
```

```
DESCRIBE COLUMNFAMILY Stocks;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	

References

<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

EXIT

Terminates the CQL command-line client.

Synopsis

```
EXIT | QUIT;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

SHOW

Shows the Cassandra version, host, or data type assumptions for the current `cqlsh` client session.

Synopsis

```
SHOW VERSION  
| HOST  
| ASSUMPTIONS;
```

Description

A `SHOW` command displays this information about the current `cqlsh` client session:

References

- The version and build number of the connected Cassandra instance, as well as the versions of the CQL specification and the Thrift protocol that the connected Cassandra instance understands.
- The host information of the Cassandra node that the `cqlsh` session is currently connected to.
- The data type assumptions for the current `cqlsh` session as specified by the `ASSUME` command.

Examples

```
SHOW VERSION;
```

```
SHOW HOST;
```

```
SHOW ASSUMPTIONS;
```

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

SOURCE

Executes a file containing CQL statements.

Synopsis

```
SOURCE '<file>';
```

Description

To execute the contents of a file, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to `$HOME`:

Example

```
SOURCE '<USER_HOME>/mydir/myfile.txt';
```

The output for each statement, if there is any, appears in turn, including any error messages. Errors do not abort execution of the file.

Alternatively, use the `--file` option to execute a file while starting CQL.

CQL Commands

CQL Commands	CQL Shell Commands
<i>ALTER TABLE</i>	<i>ASSUME</i>
<i>ALTER KEYSPACE</i>	<i>CAPTURE</i>
<i>BATCH</i>	<i>COPY</i>
<i>CREATE TABLE</i>	<i>DESCRIBE</i>
<i>CREATE INDEX</i>	<i>EXIT</i>
<i>CREATE KEYSPACE</i>	<i>SHOW</i>
<i>DELETE</i>	<i>SOURCE</i>
<i>DROP TABLE</i>	
<i>DROP INDEX</i>	
<i>DROP KEYSPACE</i>	
<i>INSERT</i>	
<i>SELECT</i>	
<i>TRUNCATE</i>	
<i>UPDATE</i>	
<i>USE</i>	

nodetool

The `nodetool` utility is a command line interface for Cassandra. You can use it to help manage a cluster.

In binary installations, `nodetool` is located in the `<install_location>/bin` directory. Square brackets indicate optional parameters.

Standard usage:

```
nodetool -h HOSTNAME [-p JMX_PORT] COMMAND
```

RMI usage:

If a username and password for RMI authentication are set explicitly in the `cassandra-env.sh` file for the host, then you must specify credentials:

```
nodetool -h HOSTNAME [-p JMX_PORT -u JMX_USERNAME -pw JMX_PASSWORD] COMMAND
```

Options

The available options are:

Flag	Option	Description
-h	--host <i>arg</i>	Hostname of node or IP address.
-p	--port <i>arg</i>	Remote JMX agent port number.
-pr	--partitioner-range	Repair only the first range returned by the partitioner for the node.

-pw	--password <i>arg</i>	Remote JMX agent password.
-u	--username <i>arg</i>	Remote JMX agent username.
Snapshot Options Only		
-cf	--column-family <i>arg</i>	Only take a snapshot of the specified column family.
-snapshot	--with-snapshot	Repair one node at a time using snapshots.
-t	--tag <i>arg</i>	Optional name to give a snapshot.

Command List

The available commands are:

Command List		
<i>cfhistograms</i>	<i>getcompactionthreshold</i>	<i>removetoken</i>
<i>cfstats</i>	<i>getendpoints</i>	<i>repair</i>
<i>cleanup</i>	<i>getsstables</i>	<i>ring</i>
<i>clearsnapshot</i>	<i>gossipinfo</i>	<i>scrub</i>
<i>compact</i>	<i>info</i>	<i>setcachecapacity</i>
<i>compactionstats</i>	<i>invalidatekeycache</i>	<i>setcompactionthreshold</i>
<i>decommission</i>	<i>invalidaterowcache</i>	<i>setcompactionthroughput</i>
<i>describering</i>	<i>join</i>	<i>setstreamthroughput</i>
<i>disablegossip</i>	<i>move</i>	<i>snapshot</i>
<i>disablethrift</i>	<i>netstats</i>	<i>statusthrift</i>
<i>drain</i>	<i>rangekeysample</i>	<i>stop</i>
<i>enablegossip</i>	<i>rebuild</i>	<i>tpstats</i>
<i>enablethrift</i>	<i>rebuild_index</i>	<i>upgradesstables</i>
<i>flush</i>	<i>refresh</i>	<i>version</i>

Command Details

Details for each command are listed below:

cfhistograms *keyspace cf_name*

Displays statistics on the read/write latency for a column family. These statistics, which include row size, column count, and bucket offsets, can be useful for monitoring activity in a column family.

cfstats

Displays statistics for every keyspace and column family.

cleanup [*keyspace*][*cf_name*]

Triggers the immediate cleanup of keys no longer belonging to this node. This has roughly the same effect on a node that a major compaction does in terms of a temporary increase in disk space usage and an increase in disk I/O. Optionally takes a list of column family names.

clearsnapshot [*keyspaces*] -t [*snapshotName*]

Deletes snapshots for the specified keyspaces. You can remove all snapshots or remove the snapshots with the given name.

compact [*keyspace*][*cf_name*]

For column families that use the *SizeTieredCompactionStrategy*, initiates an immediate major compaction of all column families in *keyspace*. For each column family in *keyspace*, this compacts all existing SSTables into a single SSTable. This can cause considerable disk I/O and can temporarily cause up to twice as much disk space to be used. Optionally takes a list of column family names.

compactionstats

Displays compaction statistics.

decommission

Tells a live node to decommission itself (streaming its data to the next node on the ring). Use *netstats* to monitor the progress.

See also:

<http://wiki.apache.org/cassandra/NodeProbe#Decommission>

http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely

describering [*keyspace*]

Shows the token ranges for a given keyspace.

disablegossip

Disable Gossip. Effectively marks the node dead.

disablethrift

Disables the Thrift server.

drain

Flushes all memtables for a node and causes the node to stop accepting write operations. Read operations will continue to work. You typically use this command before upgrading a node to a new version of Cassandra or routinely before stopping a node to speed up the restart process. Because this operation writes the current memtables to disk, Cassandra does not need to read through the commit log when you restart the node. If you have durable writes set to false, which is unlikely, there is no commit log and you must drain the node before stopping it to prevent losing data.

enablegossip

Re-enables Gossip.

enablethrift

Re-enables the Thrift server.

flush [*keyspace*] [*cf_name*]

Flushes all memtables for a keyspace to disk, allowing the commit log to be cleared. Optionally takes a list of column family names.

getcompactionthreshold *keyspace cf_name*

Gets the current compaction threshold settings for a column family. See:

<http://wiki.apache.org/cassandra/MemtableSSTable>

getendpoints *keyspace cf key*

Displays the end points that owns the key. The *key* is only accepted in HEX format.

getsstables *keyspace cf key*

Displays the sstable filenames that own the key.

gossipinfo

Shows the gossip information for the cluster.

info

Outputs node information including the token, load info (on disk storage), generation number (times started), uptime in seconds, and heap memory usage.

invalidatekeycache [*keyspace*] [*cfnames*]

Invalidates, or deletes, the key cache. Optionally takes a keyspace or list of column family names. Leave a blank space between each column family name.

invalidaterowcache [*keyspace*] [*cfnames*]

Invalidates, or deletes, the row cache. Optionally takes a keyspace or list of column family names. Leave a blank space between each column family name.

join

Causes the node to join the ring. This assumes that the node was initially *not* started in the ring, that is, started with `-Djoin_ring=false`. Note that the joining node should be properly configured with the desired options for seed list, initial token, and auto-bootstrapping.

move *new_token*

Moves a node to a new token. This essentially combines decommission and bootstrap. See:

http://wiki.apache.org/cassandra/Operations#Moving_nodes

netstats [*host*]

Displays network information such as the status of data streaming operations (bootstrap, repair, move, and decommission) as well as the number of active, pending, and completed commands and responses.

rangekeysample

Displays the sampled keys held across all keyspaces.

rebuild [*source_dc_name*]

Rebuilds data by streaming from other nodes (similar to bootstrap). Use this command to bring up a new data center in an existing cluster. See [Adding a Data Center to a Cluster](#).

rebuild_index *keyspace cf-name index_name,index_name1*

Fully rebuilds of native secondary index for a given column family. Example of *index_names*: Standard3.IdxName,Standard3.IdxName1

refresh *keyspace cf_name*

Loads newly placed SSTables on to the system without restart.

removetoken status | force | *token*

Shows status of a current token removal, forces the the completion of a pending removal, or removes a specified token. This token's range is assumed by another node and the data is streamed there from the remaining live replicas. See:

http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely

repair *keyspace [cf_name]* [-pr]

Begins an anti-entropy node repair operation. If the `-pr` option is specified, only the first range returned by the partitioner for a node is repaired. This allows you to repair each node in the cluster in succession without duplicating work.

Without `-pr`, all replica ranges that the node is responsible for are repaired.

Optionally takes a list of column family names.

ring

Displays node status and information about the ring as determined by the node being queried. This can give you an idea of the load balance and if any nodes are down. If your cluster is not properly configured, different nodes may show a

different ring; this is a good way to check that every node views the ring the same way.

scrub [*keyspace*][*cf_name*]

Rebuilds SSTables on a node for the named column families and snapshots data files before rebuilding as a safety measure. If possible use `upgradesstables`. While `scrub` rebuilds SSTables, it also discards data that it deems broken and creates a snapshot, which you have to remove manually.

setcachecapacity *key_cache_capacity* *row_cache_capacity*

Set global key and row cache capacities in megabytes.

setcompactionthreshold *keyspace* *cf_name* *min_threshold* *max_threshold*

The *min_threshold* parameter controls how many SSTables of a similar size must be present before a minor compaction is scheduled. The *max_threshold* sets an upper bound on the number of SSTables that may be compacted in a single minor compaction. See also:

<http://wiki.apache.org/cassandra/MemtableSSTable>

setcompactionthroughput *value_in_mb*

Set the maximum throughput for compaction in the system in megabytes per second. Set to 0 to disable throttling.

setstreamthroughput *value_in_mb*

Set the maximum streaming throughput in the system in megabytes per second. Set to 0 to disable throttling.

snapshot -cf [*columnfamilyName*] [*keyspace*] -t [*snapshot-name*]

Takes an online snapshot of Cassandra's data. Before taking the snapshot, the node is flushed. The results are stored in Cassandra's data directory under the `snapshots` directory of each keyspace. See [Install Locations](#). See:

http://wiki.apache.org/cassandra/Operations#Backing_up_data

statusthrift

Status of the thrift server.

stop [*operation type*]

Stops an operation from continuing to run. Options are COMPACT, VALIDATION, CLEANUP, SCRUB, INDEX_BUILD. For example, this allows you to stop a compaction that has a negative impact on the performance of a node. After the compaction stops, Cassandra continues with the rest in the queue. Eventually, Cassandra restarts the compaction.

tpstats

Displays the number of active, pending, and completed tasks for each of the thread pools that Cassandra uses for stages of operations. A high number of pending tasks for any pool can indicate performance problems. See:

<http://wiki.apache.org/cassandra/Operations#Monitoring>

upgradesstables [*keyspace*][*cf_name*]

Rebuilds SSTables on a node for the named column families. Use when upgrading your server or changing compression options (available from Cassandra 1.0.4 onwards).

version

Displays the Cassandra release version for the node being queried.

cassandra

The `cassandra` utility starts the Cassandra Java server process.

Usage

cassandra

cassandra [OPTIONS]

Environment

Cassandra requires the following environment variables to be set:

- JAVA_HOME - The path location of your Java Virtual Machine (JVM) installation
- CLASSPATH - A path containing all of the required Java class files (.jar)
- CASSANDRA_CONF - Directory containing the Cassandra configuration files

For convenience, Cassandra uses an include file, `cassandra.in.sh`, to source these environment variables. It will check the following locations for this file:

- Environment setting for CASSANDRA_INCLUDE if set
- `<install_location>/bin`
- `/usr/share/cassandra/cassandra.in.sh`
- `/usr/local/share/cassandra/cassandra.in.sh`
- `/opt/cassandra/cassandra.in.sh`
- `<USER_HOME>/cassandra.in.sh`

Cassandra also uses the Java options set in `$CASSANDRA_CONF/cassandra-env.sh`. If you want to pass additional options to the Java virtual machine, such as maximum and minimum heap size, edit the options in that file rather than setting JVM_OPTS in the environment.

Options

-f

Start the `cassandra` process in foreground (default is to start as a background process).

-p <filename>

Log the process ID in the named file. Useful for stopping Cassandra by killing its PID.

-v

Print the version and exit.

-D <parameter>

Passes in one of the following startup parameters:

Parameter	Description
<code>access.properties=<filename></code>	The file location of the access.properties file.
<code>cassandra-pidfile=<filename></code>	Log the Cassandra server process ID in the named file. Useful for stopping Cassandra by killing its PID.
<code>cassandra.config=<directory></code>	The directory location of the Cassandra configuration files.
<code>cassandra.initial_token=<token></code>	Sets the initial partitioner token for a node the first time the node is started.
<code>cassandra.join_ring=<true false></code>	Set to false to start Cassandra on a node but not have the node join the cluster.
<code>cassandra.load_ring_state=<true false></code>	Set to false to clear all gossip state for the node on restart. Use if you have changed node information in <code>cassandra.yaml</code> (such as <code>listen_address</code>).
<code>cassandra.renew_counter_id=<true false></code>	Set to true to reset local counter info on a node. Used to recover from data loss to a counter column family. First remove all SSTables for counter column families on the node, then restart the node with <code>-Dcassandra.renew_counter_id=true</code> , then run <code>nodetool repair</code> once the node is up again.
<code>cassandra.replace_token=<token></code>	To replace a node that has died, restart a new node in its place and use this parameter to pass in the token that the new node is assuming. The new node must not have any data in its data directory and the token passed must already be a token that is part of the ring.
<code>cassandra.write_survey=true</code>	For testing new compaction and compression strategies. It allows you to experiment with different strategies and benchmark write performance differences without affecting the production workload. See Testing Compaction and Compression .

```

cassandra.framed
cassandra.host
cassandra.port=<port>
cassandra.rpc_port=<port>
cassandra.start_rpc=<true|false>
cassandra.storage_port=<port>
corrupt-sstable-root
legacy-sstable-root
mx4jaddress
mx4jport
passwd.mode
passwd.properties=<file>

```

Examples

Start Cassandra on a node and log its PID to a file:

```
cassandra -p ./cassandra.pid
```

Clear gossip state when starting a node. This is useful if the node has changed its configuration, such as its listen IP address:

```
cassandra -Dcassandra.load_ring_state=false
```

Start Cassandra on a node in stand-alone mode (do not join the cluster configured in the `cassandra.yaml` file):

```
cassandra -Dcassandra.join_ring=false
```

Cassandra Bulk Loader

The `sstableloader` tool provides the ability to bulk load external data into a cluster, load existing SSTables into another cluster with a different number nodes or replication strategy, and restore snapshots.

About *sstableloader*

The `sstableloader` tool streams a set of SSTable data files to a live cluster. It does not simply copy the set of SSTables to every node, but transfers the relevant part of the data to each node, conforming to the replication strategy of the cluster. The column family into which the data is loaded does not need to be empty.

Because `sstableloader` uses Cassandra gossip, make sure that the `cassandra.yaml` configuration file is in the classpath and set to communicate with the cluster. At least one node of the cluster must be configured as seed. If necessary, properly configure the following properties: `listen_address`, `storage_port`, `rpc_address`, and `rpc_port`.

If you use `sstableloader` to load external data, you must first generate SSTables. If you use DataStax Enterprise, you can use `Sqoop` to migrate your data or if you use Cassandra, follow the procedure described in [Using the Cassandra Bulk Loader](#) blog. Before loading the data, you must define the schema of the column families with `CLI`, Thrift, or `CQL`.

To get the best throughput from SSTable loading, you can use multiple instances of `sstableloader` to stream across multiple machines. No hard limit exists on the number of SSTables that `sstableloader` can run at the same time, so you can add additional loaders until you see no further improvement.

If you use `sstableloader` on the same machine as the Cassandra node, you can't use the same network interface as the Cassandra node. However, you can use the `JMX > StorageService > bulkload()` call from that node. This method takes the absolute path to the directory where the SSTables are located, and loads them just as `sstableloader` does. However, because the node is both source and destination for the streaming, it increases the load on that node. This means that you should load data from machines that are not Cassandra nodes when loading into a live cluster.

Using *sstableloader*

In binary installations, `sstableloader` is located in the `<install_location>/bin` directory.

The `sstableloader` bulk loads the SSTables found in the directory `<dir_path>` to the configured cluster. The parent directory of `<dir_path>` is used as the keyspace name. For example to load an SSTable named `Standard1-he-1-Data.db` into keyspace `Keyspace1`, the files `Keyspace1-Standard1-he-1-Data.db` and `Keyspace1-Standard1-he-1-Index.db` must be in a directory called `Keyspace1/Standard1/`.

```
bash sstableloader [options] <dir_path>
```

Example:

```
$ ls -l Keyspace1/Standard1/
Keyspace1-Standard1-he-1-Data.db
Keyspace1-Standard1-he-1-Index
$ <path_to_install>/bin/sstableloader -d localhost <keyspace>/<dir_name>/
```

where `<dir_name>` is the directory containing the SSTables. Only the `-Data` and `-Index` components are required; `-Statistics` and `-Filter` are ignored.

The `sstableloader` has the following options:

Option	Description
--------	-------------

-d,--nodes <initial hosts>	Connect to comma separated list of hosts for initial ring information.
--debug	Display stack traces.
-h,--help	Display help.
-i,--ignore <NODES>	Do not stream to this comma separated list of nodes.
--no-progress	Do not display progress.
-p,--port <rpc port>	RPC port (default 9160).
-t,--throttle <throttle>	Throttle speed in Mbits (default unlimited).
-v,--verbose	Verbose output.
Kerberos authentication options (Available only in DataStax Enterprise 3.0.1)	
-pr,--principal	Kerberos principal. (Optional, not required if you have run kinit.)
-k,--keytab	Keytab location. (Optional, not required if you have run kinit.)
SSL encryption options (Available only in DataStax Enterprise 3.0.1)	
--ssl-keystore	SSL keystore location.
--ssl-keystore-password	SSL keystore password.
--ssl-keystore-type	SSL keystore type.
--ssl-truststore	SSL truststore location.
--ssl-truststore-password	SSL truststore password.
--ssl-truststore-type	SSL truststore type.

cassandra-stress

The `cassandra-stress` tool is a Java-based stress testing utility for benchmarking and load testing a Cassandra cluster. The binary installation of the tool also includes a daemon, which in larger-scale testing can prevent potential skews in the test results by keeping the JVM warm.

There are different modes of operation:

- **Inserting:** Loads test data.
- **Reading:** Reads test data.
- **Indexed range slicing:** Works with RandomPartitioner on indexed column families.

You can use these modes with or without the `cassandra-stressd` daemon running (binary installs only).

Usage

- **Packaged installs:** `cassandra-stress [options]`
- **Binary installs:** `<install_location>/tools/bin/cassandra-stress [options]`

The available options are:

Long Option	
Short Option	Description
--average-size-values -V	Generate column values of average rather than specific size.

--cardinality <CARDINALITY> -C <CARDINALITY>	Number of unique values stored in columns. Default is 50.
--columns <COLUMNS> -c <COLUMNS>	Number of columns per key. Default is 5.
--column-size <COLUMN-SIZE> -S <COLUMN-SIZE>	Size of column values in bytes. Default is 34.
--compaction-strategy <COMPACTION-STRATEGY> -Z <COMPACTION-STRATEGY>	Specifies which <i>compaction strategy</i> to use.
--comparator <COMPARATOR> -U <COMPARATOR>	Specifies which column comparator to use. Supported types are: TimeUUIDType, AsciiType, and UTF8Type.
--compression <COMPRESSION> -I <COMPRESSION>	Specifies the compression to use for SSTables. Default is no compression.
--consistency-level <CONSISTENCY-LEVEL> -e <CONSISTENCY-LEVEL>	<i>Consistency level</i> to use (ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL, ANY). Default is ONE.
--create-index <CREATE-INDEX> -x <CREATE-INDEX>	Type of index to create on column families (KEYS).
--enable-cql -L	Perform queries using <i>CQL</i> (Cassandra Query Language).
--family-type <TYPE> -y <TYPE>	Sets the <i>column family</i> type.
--file <FILE> -f <FILE>	Write output to a given file.
--help -h	Show help.
--keep-going -k	Ignore errors when inserting or reading. When set, --keep-trying has no effect. Default is false.
--keep-trying <KEEP-TRYING> -K <KEEP-TRYING>	Retry on-going operation N times (in case of failure). Use a positive integer. The default is 10.
--keys-per-call <KEYS-PER-CALL> -g <KEYS-PER-CALL>	Number of keys to per call. Default is 1000.
--nodes <NODES> -d <NODES>	Nodes to perform the test against. Must be comma separated with no spaces. Default is localhost.
--nodesfile <NODESFILE> -D <NODESFILE>	File containing host nodes (one per line).
--no-replicate-on-write -W	Set replicate_on_write to false for counters. Only for <i>counters</i> with a consistency level of ONE (CL=ONE).
--num-different-keys <NUM-DIFFERENT-KEYS> -F <NUM-DIFFERENT-KEYS>	Number of different keys. If less than NUM-KEYS, the same key is re-used multiple times. Default is NUM-KEYS.
--num-keys <NUMKEYS> -n <NUMKEYS>	Number of keys to write or read. Default is 1,000,000.
--operation <OPERATION> -o <OPERATION>	Operation to perform: INSERT, READ, INDEXED_RANGE_SLICE, MULTI_GET, COUNTER_ADD, COUNTER_GET. Default is INSERT.

--port <PORT> -p <PORT>	Thrift port. Default is 9160.
--progress-interval <PROGRESS-INTERVAL> -i <PROGRESS-INTERVAL>	The interval, in seconds, at which progress is output. Default is 10 seconds.
--query-names <QUERY-NAMES> -Q <QUERY-NAMES>	Comma-separated list of column names to retrieve from each row.
--random -r	Use random key generator. When used --stdev has no effect. Default is false.
--replication-factor <REPLICATION-FACTOR> -I <REPLICATION-FACTOR>	Replication Factor to use when creating column families. Default is 1.
--replication-strategy <REPLICATION-STRATEGY> -R <REPLICATION-STRATEGY>	Replication strategy to use (only on insert and when a keyspace does not exist.) Default is: <i>SimpleStrategy</i> .
--send-to <SEND-TO> -T <SEND-TO>	Sends the command as a request to the cassandra-stressd daemon at the specified IP address. The daemon must already be running at that address.
--skip-keys <SKIP-KEYS> -N <SKIP-KEYS>	Fraction of keys to skip initially. Default is 0.
--stdev <STDEV> -s <STDEV>	Standard deviation. Default is 0.1.
--strategy-properties <STRATEGY-PROPERTIES> --O <STRATEGY-PROPERTIES>	Replication strategy properties in the following format: <dc_name>:<num>,<dc_name>:<num>,... For use with <i>NetworkTopologyStrategy</i> .
--threads <THREADS> -t <THREADS>	Number of threads to use. Default is 50.
--unframed -m	Use unframed transport. Default is false.
--use-prepared-statements -P	(CQL only) Perform queries using prepared statements.

Using the Daemon Mode

Usage for the daemon mode in binary installs:

```
<install_location>/tools/bin/cassandra-stressd start|stop|status [-h <host>]
```

During stress testing, you can keep the daemon running and send it commands through it using the `--send-to` option.

Examples

- Inserts 1,000,000 rows to given host:

```
/tools/bin/cassandra-stress -d 192.168.1.101
```

When the number of rows is not specified, one million rows are inserted.

- Read 1,000,000 rows from given host:

```
tools/bin/cassandra-stress -d 192.168.1.101 -o read
```

- Insert 10,000,000 rows across two nodes:

```
/tools/bin/cassandra-stress -d 192.168.1.101,192.168.1.102 -n 10000000
```

- Insert 10,000,000 rows across two nodes using the daemon mode:

```
/tools/bin/cassandra-stress -d 192.168.1.101,192.168.1.102 -n 10000000 --send-to 54.0.0.0
```

Interpreting the output of *cassandra-stress*

The *cassandra-stress* tool periodically outputs information about the running tests. For example:

```
7251,725,725,56.1,95.1,191.8,10
19523,1227,1227,41.6,86.1,189.1,21
41348,2182,2182,22.5,75.7,176.0,31
...
```

Each line reports data for the interval between the last elapsed time and current elapsed time, which is set by the `--progress-interval` option (default 10 seconds). The following explains this information:

- **total**: the total number of operations since the start of the test.
- **interval_op_rate**: the number of operations performed during the interval.
- **interval_key_rate**: the number of keys/rows read or written during the interval (normally be the same as `interval_op_rate` unless doing range slices).
- **latency**: the average latency for each operation during that interval.
- **elapsed**: the number of seconds elapsed since the beginning of the test.

sstable2json / json2sstable

The utility *sstable2json* converts the on-disk SSTable representation of a column family into a JSON formatted document. Its counterpart, *json2sstable*, does exactly the opposite: it converts a JSON representation of a column family to a Cassandra usable SSTable format. Converting SSTables this way can be useful for testing and debugging.

Note

Starting with version 0.7, *json2sstable* and *sstable2json* must be run in such a way that the schema can be loaded from system tables. This means that *cassandra.yaml* must be found in the classpath and refer to valid storage directories.

See also: The Import/Export section of <http://wiki.apache.org/cassandra/Operations>.

sstable2json

This converts the on-disk SSTable representation of a column family into a JSON formatted document.

Usage

```
bin/sstable2json SSTABLE
[-k KEY [-k KEY [...]]] [-x KEY [-x KEY [...]]] [-e]
```

SSTABLE should be a full path to a column-family-name-Data.db file in Cassandra's data directory. For example, `/var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db`.

`-k` allows you to include a specific set of keys. The *KEY* must be in HEX format. Limited to 500 keys.

`-x` allows you to exclude a specific set of keys. Limited to 500 keys.

-e causes keys to only be enumerated

Output Format

The output of `sstable2json` for standard column families is:

```
{
  ROW_KEY:
  {
    [
      [ COLUMN_NAME, COLUMN_VALUE, COLUMN_TIMESTAMP, IS_MARKED_FOR_DELETE ],
      [ COLUMN_NAME, ... ],
      ...
    ]
  },
  ROW_KEY:
  {
    ...
  },
  ...
}
```

The output for super column families is:

```
{
  ROW_KEY:
  {
    SUPERCOLUMN_NAME:
    {
      deletedAt: DELETION_TIME,
      subcolumns:
      [
        [ COLUMN_NAME, COLUMN_VALUE, COLUMN_TIMESTAMP, IS_MARKED_FOR_DELETE ],
        [ COLUMN_NAME, ... ],
        ...
      ]
    },
    SUPERCOLUMN_NAME:
    {
      ...
    },
    ...
  },
  ROW_KEY:
  {
    ...
  },
  ...
}
```

Row keys, column names and values are written in as the hex representation of their byte arrays. Line breaks are only in between row keys in the actual output.

json2sstable

This converts a JSON representation of a column family to a Cassandra usable SSTable format.

Usage

```
bin/json2sstable -K KEYSPACE -c COLUMN_FAMILY JSON SSTABLE
```

JSON should be a path to the JSON file

SSTABLE should be a full path to a column-family-name-Data.db file in Cassandra's data directory. For example, /var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db.

sstablekeys

The sstablekeys utility is shorthand for sstable2json with the -e option. Instead of dumping all of a column family's data, it dumps only the keys.

Usage

```
bin/sstablekeys SSTABLE
```

SSTABLE should be a full path to a column-family-name-Data.db file in Cassandra's data directory. For example, /var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db.

CQL Commands Quick Reference

The following table provides links to the CQL commands and CQL shell commands:

CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	ASSUME
ALTER KEYSPACE	CAPTURE
BATCH	COPY
CREATE TABLE	DESCRIBE
CREATE INDEX	EXIT
CREATE KEYSPACE	SHOW
DELETE	SOURCE
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

Install Locations

Locations of the Configuration Files

The configuration files, such as *cassandra.yaml*, are located in the following directories:

- Cassandra packaged installs: `/etc/cassandra/conf`
- Cassandra binary installs: `<install_location>/conf`

For DataStax Enterprise installs, see [Configuration Files Locations](#).

Packaged Installs Directories

The packaged releases install into the following directories.

- `/var/lib/cassandra` (data directories)
- `/var/log/cassandra` (log directory)
- `/var/run/cassandra` (runtime files)
- `/usr/share/cassandra` (environment settings)
- `/usr/share/cassandra/lib` (JAR files)
- `/usr/bin` (binary files)
- `/usr/sbin`
- `/etc/cassandra` (configuration files)
- `/etc/init.d` (service startup script)
- `/etc/security/limits.d` (cassandra user limits)
- `/etc/default`

Binary Tarball Install Directories

The following directories are installed in the installation home directory.

- `bin` (utilities and start scripts)
- `conf` (configuration files and environment settings)
- `interface` (Thrift and Avro client APIs)
- `javadoc` (Cassandra Java API documentation)
- `lib` (JAR and license files)

Configuring Firewall Port Access

If you have a firewall running on the nodes in your Cassandra cluster, you must open up the following ports to allow communication between the nodes, including certain Cassandra ports. If this isn't done, when you start Cassandra on a node, the node acts as a standalone database server rather than joining the database cluster.

Port	Description
Public Facing Ports	
22	SSH port.
8888	OpsCenter website port.
Cassandra Inter-node Ports	
1024+	JMX reconnection/loopback ports. See description for port 7199.

7000	Cassandra inter-node cluster communication.
7199	Cassandra JMX monitoring port. After the initial handshake, the JMX protocol requires that the client reconnects on a randomly chosen port (1024+).
9160	Cassandra client port (Thrift).
OpsCenter ports	
61620	OpsCenter monitoring port. The opscenterd daemon listens on this port for TCP traffic coming from the agent.
61621	OpsCenter agent port. The agents listen on this port for SSL traffic initiated by OpsCenter.

Starting and Stopping a Cassandra Cluster

On initial start-up, each node must be started one at a time, starting with your seed nodes.

Starting Cassandra as a Stand-Alone Process

Start the Cassandra Java server process starting with the seed nodes:

```
$ cd <install_location>
$ bin/cassandra
```

`-f` starts Cassandra in the foreground.

Starting Cassandra as a Service

Packaged installations provide startup scripts in `/etc/init.d` for starting Cassandra as a service. The service runs as the `cassandra` user.

To start the Cassandra service, you must have root or sudo permissions:

```
$ sudo service cassandra start
```

Note

On Enterprise Linux systems, the Cassandra service runs as a `java` process. On Debian systems, the Cassandra service runs as a `jsvc` process.

Stopping Cassandra as a Stand-Alone Process

To stop the Cassandra process, find the Cassandra Java process ID (PID), and then `kill` the process using its PID number:

```
$ ps auwx | grep cassandra
$ sudo kill <pid>
```

Stopping Cassandra as a Service

To stop the Cassandra service, you must have root or sudo permissions:

```
$ sudo service cassandra stop
```

Troubleshooting Guide

This page contains recommended fixes and workarounds for issues commonly encountered with Cassandra:

- *Reads are getting slower while writes are still fast*
- *Nodes seem to freeze after some period of time*
- *Nodes are dying with OOM errors*
- *Nodetool or JMX connections failing on remote nodes*
- *View of ring differs between some nodes*
- *Java reports an error saying there are too many open files*
- *Insufficient user resource limits errors*
- *Cannot initialize class org.xerial.snappy.Snappy*

Reads are getting slower while writes are still fast

Check the SSTable counts in *cfstats*. If the count is continually growing, the cluster's IO capacity is not enough to handle the write load it is receiving. Reads have slowed down because the data is fragmented across many SSTables and compaction is continually running trying to reduce them. Adding more IO capacity, either via more machines in the cluster, or faster drives such as SSDs, will be necessary to solve this.

If the SSTable count is relatively low (32 or less) then the amount of file cache available per machine compared to the amount of data per machine needs to be considered, as well as the application's read pattern. The amount of file cache can be formulated as $(\text{TotalMemory} - \text{JVMHeapSize})$ and if the amount of data is greater and the read pattern is approximately random, an equal ratio of reads to the cache:data ratio will need to seek the disk. With spinning media, this is a slow operation. You may be able to mitigate many of the seeks by using a key cache of 100%, and a small amount of row cache (10000-20000) if you have some 'hot' rows and they are not extremely large.

Nodes seem to freeze after some period of time

Check your system.log for messages from the GCInspector. If the GCInspector is indicating that either the ParNew or ConcurrentMarkSweep collectors took longer than 15 seconds, there is a very high probability that some portion of the JVM is being swapped out by the OS. One way this might happen is if the mmap DiskAccessMode is used without JNA support. The address space will be exhausted by mmap, and the OS will decide to swap out some portion of the JVM that isn't in use, but eventually the JVM will try to GC this space. Adding the JNA libraries will solve this (they cannot be shipped with Cassandra due to carrying a GPL license, but are freely available) or the DiskAccessMode can be switched to mmap_index_only, which as the name implies will only mmap the indices, using much less address space. DataStax recommends that Cassandra nodes disable swap entirely (`sudo swapoff --all`), since it is better to have the OS OutOfMemory (OOM) killer kill the Java process entirely than it is to have the JVM buried in swap and responding poorly.

If the GCInspector isn't reporting very long GC times, but is reporting moderate times frequently (ConcurrentMarkSweep taking a few seconds very often) then it is likely that the JVM is experiencing extreme GC pressure and will eventually OOM. See the section below on OOM errors.

Nodes are dying with OOM errors

If nodes are dying with OutOfMemory exceptions, check for these typical causes:

- Row cache is too large, or is caching large rows
 - Row cache is generally a high-end optimization. Try disabling it and see if the OOM problems continue.

- The memtable sizes are too large for the amount of heap allocated to the JVM
 - You can expect $N + 2$ memtables resident in memory, where N is the number of column families. Adding another 1GB on top of that for Cassandra itself is a good estimate of total heap usage.

If none of these seem to apply to your situation, try loading the heap dump in **MAT** and see which class is consuming the bulk of the heap for clues.

Nodetool or JMX connections failing on remote nodes

If you can run nodetool commands locally but not on other nodes in the ring, you may have a common JMX connection problem that is resolved by adding an entry like the following in `<install_location>/conf/cassandra-env.sh` on each node:

```
JVM_OPTS="$JVM_OPTS -Djava.rmi.server.hostname=<public name>"
```

If you still cannot run nodetool commands remotely after making this configuration change, do a full evaluation of your firewall and network security. The nodetool utility communicates through JMX on port 7199.

View of ring differs between some nodes

This is an indication that the ring is in a bad state. This can happen when there are token conflicts (for instance, when bootstrapping two nodes simultaneously with automatic token selection.) Unfortunately, the only way to resolve this is to do a full cluster restart; a rolling restart is insufficient since gossip from nodes with the bad state will repopulate it on newly booted nodes.

Java reports an error saying there are too many open files

Java is not allowed to open enough file descriptors. Cassandra generally needs more than the default (1024) amount. To increase the number of file descriptors, change the security limits on your Cassandra nodes. For example, using the following commands:

```
echo "* soft nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "* hard nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "root soft nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "root hard nofile 32768" | sudo tee -a /etc/security/limits.conf
```

Another, much less likely possibility, is a file descriptor leak in Cassandra. Run `lsof -n | grep java` to check that the number of file descriptors opened by Java is reasonable and reports the error if the number is greater than a few thousand.

Insufficient user resource limits errors

Insufficient resource limits may result in a number of errors in Cassandra and OpsCenter, including the following:

Cassandra errors

Insufficient as (address space) or memlock setting:

```
ERROR [SSTableBatchOpen:1] 2012-07-25 15:46:02,913 AbstractCassandraDaemon.java (line 139)
Fatal exception in thread Thread[SSTableBatchOpen:1,5,main]
java.io.IOException: java.io.IOException: Map failed at ...
```

Insufficient memlock settings:

Cannot initialize class org.xerial.snappy.Snappy

```
WARN [main] 2011-06-15 09:58:56,861 CLibrary.java (line 118) Unable to lock JVM memory (ENOMEM).  
This can result in part of the JVM being swapped out, especially with mmap'd I/O enabled.  
Increase RLIMIT_MEMLOCK or run Cassandra as root.
```

Insufficient nofiles setting:

```
WARN 05:13:43,644 Transport error occurred during acceptance of message.  
org.apache.thrift.transport.TTransportException: java.net.SocketException:  
Too many open files ...
```

Insufficient nofiles setting:

```
ERROR [MutationStage:11] 2012-04-30 09:46:08,102 AbstractCassandraDaemon.java (line 139)  
Fatal exception in thread Thread[MutationStage:11,5,main]  
java.lang.OutOfMemoryError: unable to create new native thread
```

OpsCenter errors

Insufficient nofiles setting:

```
2012-08-13 11:22:51-0400 [] INFO: Could not accept new connection (EMFILE)
```

Recommended settings

You can view the current limits using the `ulimit -a` command. Although limits can also be temporarily set using this command, DataStax recommends permanently changing the settings by adding the following entries to your `/etc/security/limits.conf` file:

```
* soft nofile 32768  
* hard nofile 32768  
root soft nofile 32768  
root hard nofile 32768  
* soft memlock unlimited  
* hard memlock unlimited  
root soft memlock unlimited  
root hard memlock unlimited  
* soft as unlimited  
* hard as unlimited  
root soft as unlimited  
root hard as unlimited
```

In addition, you may need to be run the following command:

```
sysctl -w vm.max_map_count=131072
```

The command enables more mapping. It is not in the `limits.conf` file.

On CentOS, RHEL, OEL Sysems, change the system limits from 1024 to 10240 in `/etc/security/limits.d/90-nproc.conf` and then start a new shell for these changes to take effect.

```
* soft nproc 10240
```

Cannot initialize class org.xerial.snappy.Snappy

The following error may occur when Snappy compression/decompression is enabled although its library is available from the classpath:

Cannot initialize class org.xerial.snappy.Snappy

```
java.util.concurrent.ExecutionException: java.lang.NoClassDefFoundError:
  Could not initialize class org.xerial.snappy.Snappy
...
Caused by: java.lang.NoClassDefFoundError: Could not initialize class org.xerial.snappy.Snappy
  at org.apache.cassandra.io.compress.SnappyCompressor.initialCompressedBufferLength
    (SnappyCompressor.java:39)
```

The native library `snappy-1.0.4.1-libsnapappyjava.so` for Snappy compression is included in the `snappy-java-1.0.4.1.jar` file. When the JVM initializes the JAR, the library is added to the default temp directory. If the default temp directory is mounted with a `noexec` option, it results in the above exception.

One solution is to specify a different temp directory that has already been mounted without the `noexec` option, as follows:

- If you use the DSE/Cassandra command `$_BIN/dse cassandra` or `$_BIN/cassandra`, simply append the command line:

```
DSE: bin/dse cassandra -t -Dorg.xerial.snappy.tmpdir=/path/to/newtmp
```

```
Cassandra: bin/cassandra -Dorg.xerial.snappy.tmpdir=/path/to/newtmp
```

- If starting from a package using `service dse start` or `service cassandra start`, add a system environment variable `JVM_OPTS` with the value:

```
JVM_OPTS=-Dorg.xerial.snappy.tmpdir=/path/to/newtmp
```

The default `cassandra-env.sh` looks for the variable and appends to it when starting the JVM.

DataStax Community Release Notes

Current DataStax Community version: Cassandra 1.1.9

Issues

Setting `bloom_filter_fp_chance` to 0 produces a compaction error. This error is harmless and will be corrected in a future version.

Fixes and New Features in Cassandra 1.1

For a list of fixes and new features in Cassandra 1.1, see <https://github.com/apache/cassandra/blob/trunk/CHANGES.txt>.