Apache Cassandra™ 1.1 Documentation

July 06, 2012

© 2012 DataStax. All rights reserved.

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

Contents

Apache Cassandra 1.1 Documentation

Apache Cassandra is a free, open-source, distributed database system for managing large amounts of structured, semi-structured, and unstructured data. Cassandra is designed to scale to a very large size across many commodity servers with no single point of failure. Cassandra provides a powerful dynamic schema data model designed to allow for maximum flexibility and performance at scale.

DataStax Community Edition is a smart bundle comprised of the most up-to-date and stable version of Apache Cassandra, DataStax OpsCenter Community Edition, and the CQL command line utility.

Introduction to Apache Cassandra and DataStax Community Edition

The best way to get up and running quickly with Cassandra is to install the binary tarball distribution and start a single-node cluster. Cassandra is intended to be run on multiple nodes. However, installing a single-node cluster is a great way to get started. To install on multiple nodes, see *Installing a Cassandra Cluster*.

There are three ways to get going on a single node cluster:

- Install the binary tarball packages using root permissions or sudo.

  This method requires the fewest steps, but is not fully contained within your home directory; some files and directories are created outside of the install location. If you use this method, you will need to start DataStax Enterprise and run all the demos using sudo.

- Install the binary tarball packages in a single location and without root permissions or sudo.

  This method requires more steps and are files and directories are fully contained within a single directory. Use it if you do not have or want to use root privileges. If you use this method, you can start DataStax Enterprise and run the demos without using sudo.

- Install on Windows using the installation wizard.

  This method installs and automatically starts a single node instance of Cassandra and the OpsCenter.

After installing and starting Cassandra, you can get started using Cassandra with an *introduction* to CQL (Cassandra Query Language).

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.

Key Improvements

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 AID 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 to 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.

Installing the DataStax Community Binaries on Linux or Mac OSX as Root

The quickest way to get going on a single node with Cassandra is to install the DataStax Community Edition binary tarball packages using root permissions (or sudo). This installation also creates files and directories in `/var/lib/cassandra` and `/var/log/cassandra`. If you need to install everything in a single location, such as your home directory, and without root permissions, see *Installing the DataStax Community Binaries as User*.

Note

The instructions in the following sections for a quick start tutorial, not production installations. See *Planning a Cassandra Cluster Deployment*, *Installing a Cassandra Cluster*, and *Initializing a Multiple Node Cluster in a Single Data Center* for production cluster setup best practices.

Prerequisites

Cassandra is a Java program and requires that a Java Virtual Machine (JVM) is installed before starting the server. For production deployments, you will need the Oracle Java Runtime Environment 1.6.0_19 or later, but if you are installing only an evaluation instance, any JVM is fine.

To check for Java, run the following command in a terminal window:

```
$ java -version
```

If you do not have Java installed, see *Installing Oracle JRE* for instructions.

Steps to Install Cassandra

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 website.

1. Download the Cassandra package (required), and the optional OpsCenter:

   To get the *latest* versions of DataStax and OpsCenter Community Editions and the Portfolio Demo:

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

2. Unpack the distributions:

```
$ tar -xzvf dsc.tar.gz
$ tar -xzvf opscenter.tar.gz
$ rm *.tar.gz
```

3. Start the Cassandra server in the background from the directory where the package was installed. For example, if `dsc-cassandra-1.1.0` is installed in your home directory:

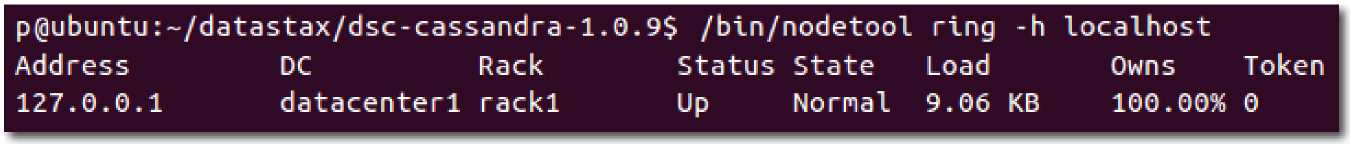
```
$ cd ~/dsc-cassandra-1.1.0
$ sudo bin/cassandra
```

Note

When Cassandra loads, you may notice a message that MX4J will not load and that `mx4j-tools.jar` is not in the classpath. You can ignore this message. MX4j provides an HTML and HTTP interface to JMX and is not necessary to run Cassandra. DataStax recommends using OpsCenter. It has more monitoring capabilities than MX4J.

4. Check that Cassandra is running by invoking the nodetool utility from the installation home directory:

```
bin/nodetool ring –h localhost
```



```
p@ubuntu:~/datastax/dsc-cassandra-1.0.9$ /bin/nodetool ring -h localhost
Address         DC          Rack        Status State   Load        Owns    Token
127.0.0.1       datacenter1 rack1       Up     Normal  9.06 KB     100.00% 0
```

Next Steps

- *Using the Database - CQL Introduction* - Basic commands for creating a keyspace; creating a column family; and inserting, updating, deleting, and reading data.
- Install the DataStax OpsCenter. The OpsCenter is a browser-based application for managing and monitoring a Cassandra cluster. See the Installing the OpsCenter.
- To stop the Cassandra server, see *Starting and Stopping a Cassandra Cluster*.

Installing the DataStax Community Binaries as User

This section provides instructions for installing and setting up a self-contained, single-node cluster of Cassandra in your home directory that does not require root permissions using the binary tarball packages.

Prerequisites

Cassandra is a Java program and requires that a Java Virtual Machine (JVM) is installed before starting the server. For production deployments, you will need the Oracle Java Runtime Environment 1.6.0_19 or later, but if you are installing only an evaluation cluster, any JVM is fine.

To check for Java, run the following command in a terminal window:

```
$ java -version
```

If you do not have Java installed, see *Installing Oracle JRE* for instructions.

Steps to Install Cassandra

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.

1. From your home directory, download the Cassandra package (required), and the OpsCenter package(optional):

   For example, on Linux to get the latest versions of DataStax and OpsCenter Community Editions and the Portfolio Demo:

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

2. Unpack the distributions:

   ```
   $ tar -xzvf dsc.tar.gz
   $ tar -xzvf opscenter.tar.gz
   $ rm *.tar.gz
   ```

3. Rename the downloaded directory to datastax:

   ```
   $ mv dsc-cassandra-1.1.0 datastax
   ```

4. In the `datastax` directory, create the data and logging directory for Cassandra.

   ```
   $ cd datastax
   $ mkdir cassandra-data
   ```

5. In `cassandra-data`, create the following directories: `saved_caches` and `commitlog`.

   ```
   $ cd cassandra-data
   $ mkdir data
   $ mkdir saved_caches
   $ mkdir commitlog
   ```

Steps to Configure and Start the Cluster

After installing a single-node Cassandra cluster in your home directory, you must set some configuration properties. These properties are specified in the `cassandra.yaml` and `log4j-server.properties` files.

1. Go the directory containing the `cassandra.yaml` file:

   ```
   $ cd ~/datastax/conf
   ```

2. Edit the following lines in `cassandra.yaml`:

   ```
   initial_token: 0
   data_file_directories: - ~/datastax/cassandra-data/data
   commitlog_directory: ~/datastax/cassandra-data/commitlog
   saved_caches_directory: ~/datastax/cassandra-data/saved_caches
   ```

3. In the `conf` directory, change the `log4j-server.properties` file:

   ```
   log4j.appender.R.File= ~/datastax/cassandra-data/system.log
   ```

4. Start the Cassandra server in the background.

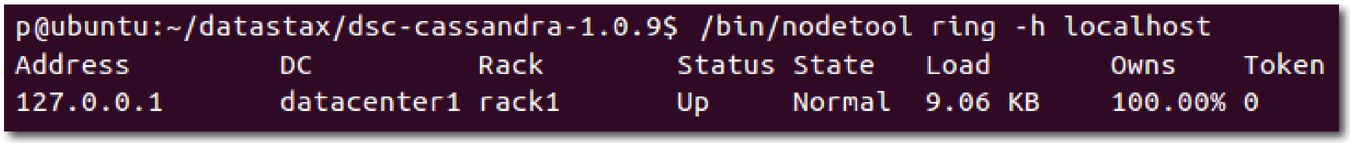
```
$ cd ~/datastax
$ bin/cassandra
```

Note

When Cassandra loads, you may notice a message that MX4J will not load and that `mx4j-tools.jar` is not in the classpath. You can ignore this message. MX4j provides an HTML and HTTP interface to JMX and is not necessary to run Cassandra. DataStax recommends using OpsCenter. It has more monitoring capabilities than MX4J.

5. Check that Cassandra is running by invoking the nodetool utility from the installation home directory:

```
$ bin/nodetool ring -h localhost
```



```
p@ubuntu:~/datastax/dsc-cassandra-1.0.9$ /bin/nodetool ring -h localhost
Address         DC          Rack        Status State   Load        Owns    Token
127.0.0.1       datacenter1 rack1       Up     Normal  9.06 KB     100.00% 0
```

Next Steps

- *Using the Database - CQL Introduction* - Basic commands for creating a keyspace; creating a column family; and inserting, updating, deleting, and reading data.
- Install the DataStax OpsCenter. The OpsCenter is a browser-based application for managing and monitoring a Cassandra cluster. See the Installing the OpsCenter.
- To stop the Cassandra server, see *Starting and Stopping a Cassandra Cluster*.

Installing the DataStax Community Binaries on Windows

DataStax provides a GUI installer for installing both Cassandra and OpsCenter on Windows. Download the Windows installer for your chosen platform (32- or 64-bit Windows 7 or Windows Server 2008) from DataStax Downloads. Then follow the installation wizard to install Cassandra, the sample applications, and OpsCenter.

Prerequisites

Before installing OpsCenter on a single node make sure you have met the following prerequisite:

- On 32-bit systems, there is a dependency on the Visual C++ 2008 runtime. If needed, download it from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=29.
- OpenSSL: 0.9.8. SSL is disabled by default. To enable, see Enabling SSL – Windows.

Starting Services

During installation, accept the options to start the DataStax Cassandra Server and OpsCenter services automatically.

When you select to start services, the Cassandra server, the OpsCenter server, and the OpsCenter agent start automatically when the installation completes (and whenever the computer reboots).

Launching Cassandra Server and OpsCenter

To finish installing DataStax Community Edition, accept the option to launch immediately.

To launch OpsCenter at a later time, enter the following URL in a browser:

```
http://localhost:8888/opscenter
```

Troubleshooting OpsCenter Windows Installations

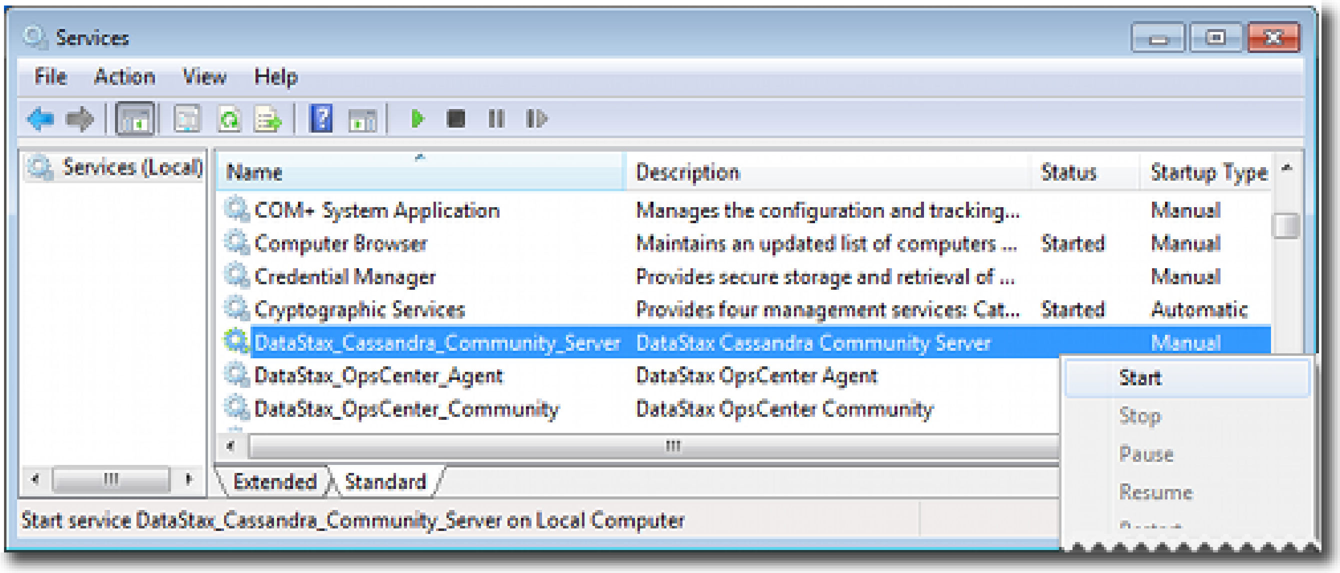
Problems starting up OpsCenter and delays in stopping the OpsCenter Agent service are easily solved.

OpsCenter does not start up:

If OpsCenter does not start up when you enter the URL to launch it, the most likely reasons are:

- DataStax services are not running:

  Solution: Check that the Cassandra and OpsCenter services are running. If not, start them in the Control Panel.



Name	Description	Status	Startup Type
COM+ System Application	Manages the configuration and tracking...		Manual
Computer Browser	Maintains an updated list of computers ...	Started	Manual
Credential Manager	Provides secure storage and retrieval of ...		Manual
Cryptographic Services	Provides four management services: Cat...	Started	Automatic
DataStax_Cassandra_Community_Server	DataStax Cassandra Community Server		Manual
DataStax_OpsCenter_Agent	DataStax OpsCenter Agent		
DataStax_OpsCenter_Community	DataStax OpsCenter Community		

Start service DataStax_Cassandra_Community_Server on Local Computer

- Microsoft Visual C++ 2008 Redistributable Package is not installed:

  Solution: Check that this package is installed. If not, download and install the package from Microsoft for 32- and 64-bit systems.

Stopping the OpsCenter Agent service takes a long time:

Be patient. Manually stopping the OpsCenter Agent service takes time.

Using the Database - CQL Introduction

Cassandra has several interfaces for querying the database from the command line:

- Cassandra Query Language shell (CQLsh) 2
- *CQLsh 3*
- *Cassandra Command Line Interface (CLI)*

Getting Started Using CQLsh

CQLsh 2 is the default query language and CQLsh 3 is the new, recommended version. After starting the Cassandra Server, follow these steps to get started using CQLsh 3 quickly:

1. Start CQLsh 3 on Linux or Mac:

   From the `bin` directory of the Cassandra installation, run the `cqlsh` script.

   ```
   cd <install_location>/bin

   ./cqlsh --cql3
   ```
2. Create a keyspace to work with:

   ```
   cqlsh> CREATE KEYSPACE demodb
            WITH strategy_class = 'org.apache.cassandra.locator.SimpleStrategy'
            AND strategy_options:replication_factor='1';
   ```

   ```
   cqlsh> USE demodb;
   ```
3. Create a column family (the counterpart to a table in relational database world):

   ```
   cqlsh> CREATE TABLE users (
            user_name varchar,
            password varchar,
            state varchar,
            PRIMARY KEY (user_name)
          );
   ```
4. Enter and read data from Cassandra:

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

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

   The output is:

   ```
    user_name | password  | state
   -----------+-----------+-------
       jsmith | ch@ngem3a |  null
   ```
5. Exit CQLsh:

   ```
   cqlsh> exit
   ```

DataStax Community Release Notes

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.

If you just want to learn more about Cassandra and how it works, see the following conceptual topics:

- *Understanding the Cassandra Architecture*
- *Understanding the Cassandra Data Model*
- *Managing and Accessing Data in Cassandra*

Understanding the 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 certain aspects of its configuration. This section explains those 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.

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 if *hinted handoff* is enabled (for a period of time, anyways). However, it is possible that some writes were missed between the interval of a node actually going down and when it is detected as down. Or if a node is down for longer than *max_hint_window_in_ms* (one hour by default), hints will no longer be saved. For that reason, it is best practice to routinely run *nodetool repair* on all nodes to ensure they have consistent data, and to also run repair after recovering a node that has been down for an extended period.

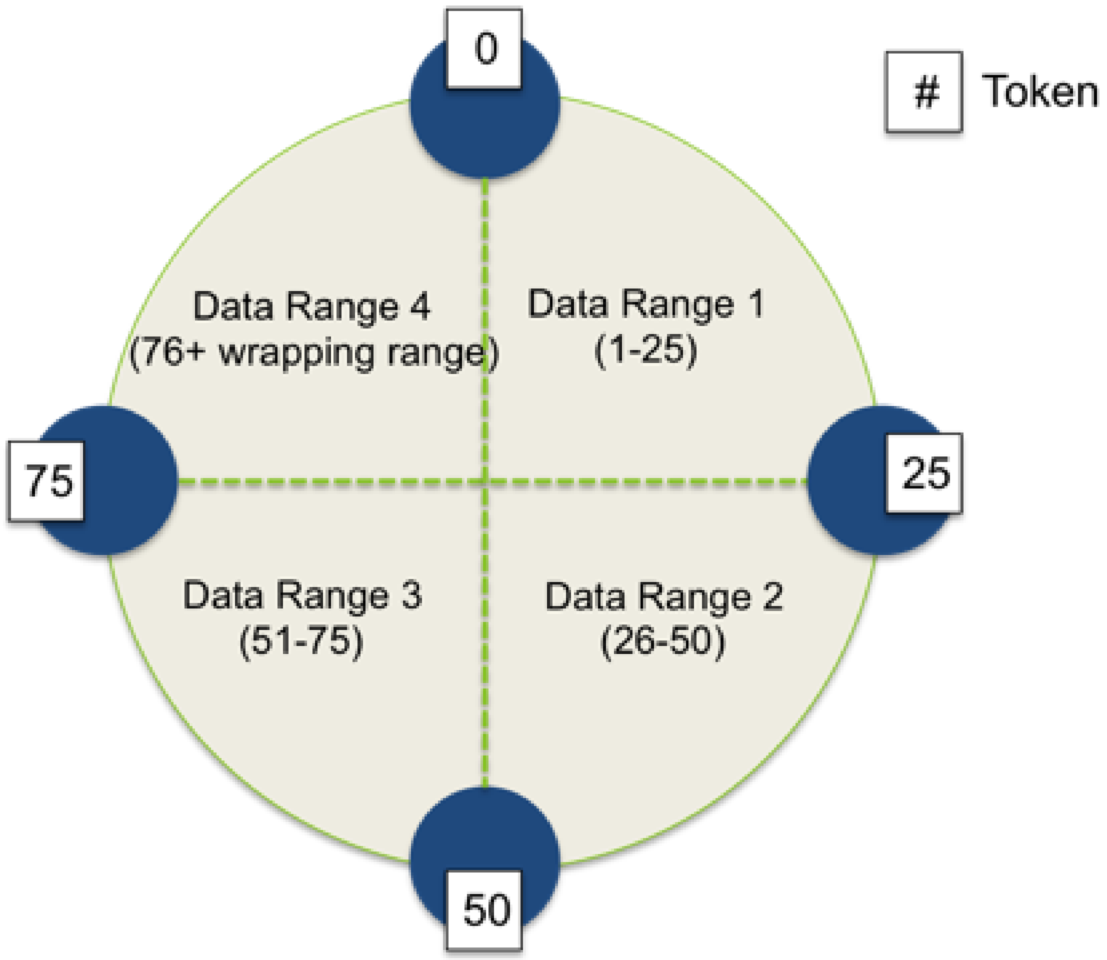
About Data Partitioning in Cassandra

When you start a Cassandra cluster, you must choose how the data will be divided across the nodes in the cluster. This is done by choosing a *partitioner* for the cluster.

In Cassandra, the total data managed by the cluster is represented as a circular space or *ring*. The ring is divided up into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the overall data. Before a node can join the ring, it must be assigned a token. The token determines the node's position on the ring and the range of data it is responsible for.

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.

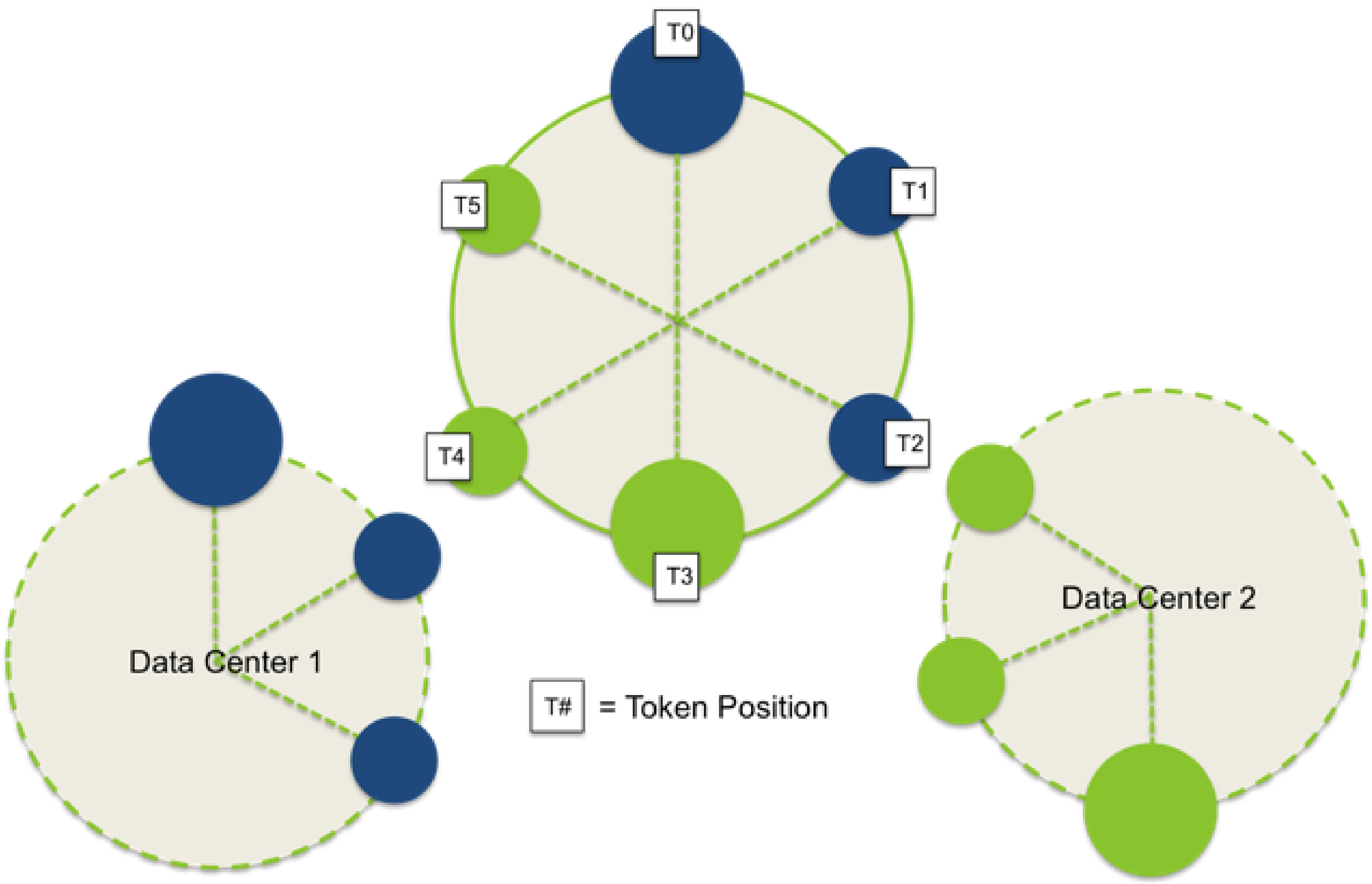
For example, consider a simple 4 node cluster where all of the row keys managed by the cluster were numbers in the range of 0 to 100. Each node is assigned a token that represents a point in this range. In this simple example, the token values are 0, 25, 50, and 75. The first node, the one with token 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.



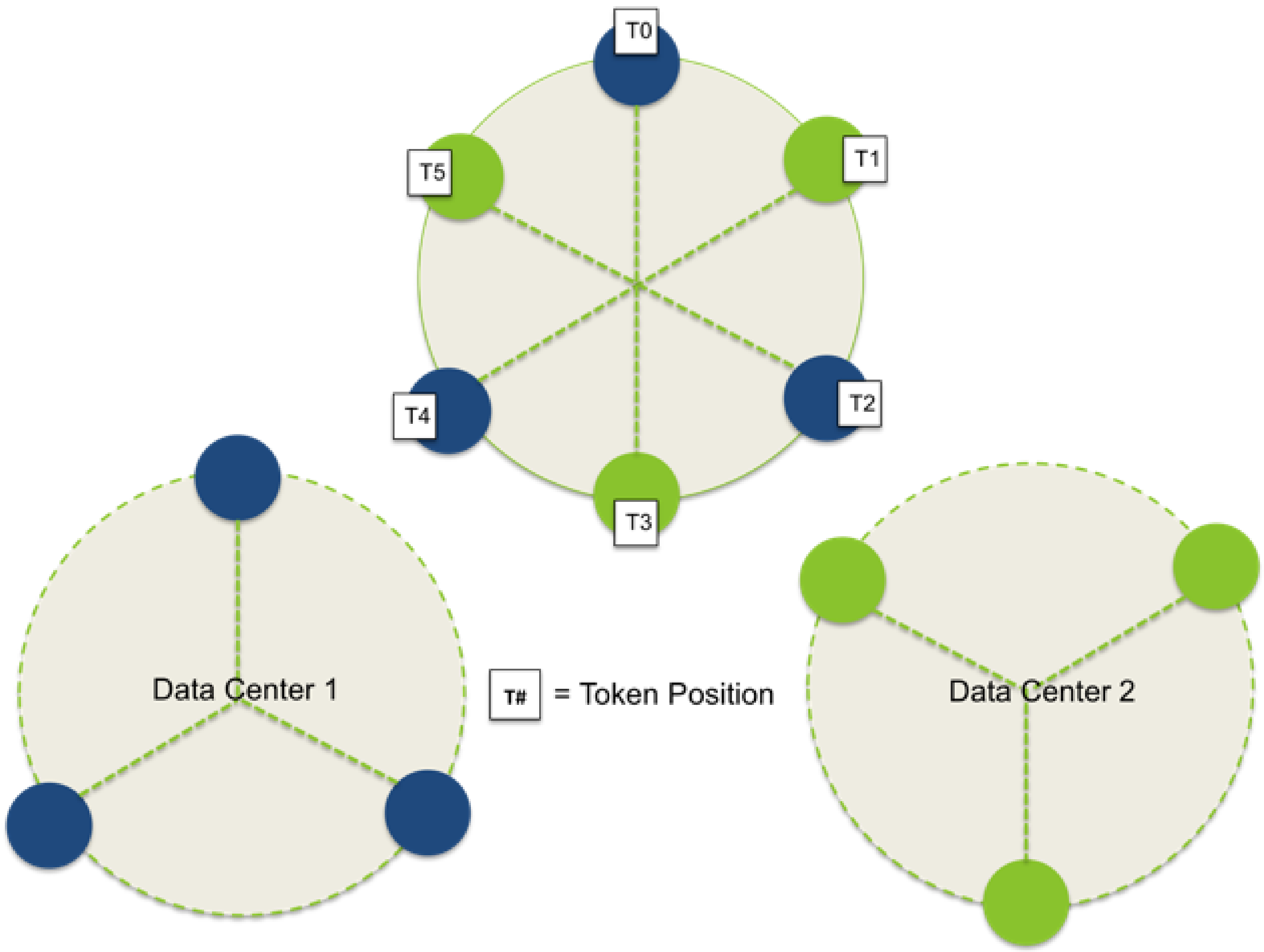
About Partitioning in Multi-Data Center Clusters

In multi-data center deployments, replica placement is calculated per data center when using the `NetworkTopologyStrategy` replica placement strategy. In each data center (or replication group) the first replica for a particular row is determined by the token value assigned to a node. Additional replicas in the same data center are placed by walking the ring clockwise until it reaches the first node in another rack.

If you do not calculate partitioner tokens so that the data ranges are evenly distributed for each data center, you could end up with uneven data distribution within a data center.



The goal is to ensure that the nodes for each data center have token assignments that evenly divide the overall range. Otherwise, you could end up with nodes in each data center that own a disproportionate number of row keys. Each data center should be partitioned as if it were its own distinct ring, however token assignments within the entire cluster cannot conflict with each other (each node must have a unique token). See *Calculating Tokens for a Multiple Data Center Cluster* for strategies on how to generate tokens for multi-data center clusters.



Understanding the Partitioner Types

Unlike almost every other configuration choice in Cassandra, the partitioner may not be changed without reloading all of your data. It is important to choose and configure the correct partitioner before initializing your cluster.

Cassandra offers a number of partitioners out-of-the-box, but the random partitioner is the best choice for most Cassandra deployments.

About the Random Partitioner

The `RandomPartitioner` is the default partitioning strategy for a Cassandra cluster, and in almost all cases is the right choice.

Random partitioning uses *consistent hashing* to determine which node will store a particular row. Unlike naive modulus-by-node-count, consistent hashing ensures that when nodes are added to the cluster, the minimum possible set of data is affected.

To distribute the data evenly across the number of nodes, a hashing algorithm creates an MD5 hash value of the row key. The possible range of hash values is from 0 to 2**127. Each node in the cluster is assigned a *token* that represents a hash value within this range. A node then owns the rows with a hash value less than its token number. For single data center deployments, tokens are calculated by dividing the hash range by the number of nodes in the cluster. For multi data center deployments, tokens are calculated per data center (the hash range should be evenly divided for the nodes in each replication group).

The primary benefit of this approach is that once your tokens are set appropriately, data from all of your column families is evenly distributed across the cluster with no further effort. For example, one column family could be using user names as the row key and another column family timestamps, but the row keys from each individual column family are still spread evenly. This also means that read and write requests to the cluster will also be evenly distributed.

Another benefit of using random partitioning is the simplification of load balancing a cluster. Because each part of the hash range will receive an equal number of rows on average, it is easier to correctly assign tokens to new nodes.

About Ordered Partitioners

Using an ordered partitioner ensures that row keys are stored in sorted order. Unless absolutely required by your application, DataStax strongly recommends choosing the random partitioner over an ordered partitioner.

Using an ordered partitioner allows range scans over rows, meaning 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 would not be possible with randomly partitioned row keys, since the keys are stored in the order of their MD5 hash (not sequentially).

Although having the ability to do range scans on rows sounds like a desirable feature of ordered partitioners, there are ways to 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.

Using an ordered partitioner is not recommended 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 the writes will not be distributed across the cluster; they will 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 than is balanced for one column family may cause hot spots and uneven distribution for another column family in the same cluster.

There are three choices of built-in ordered partitioners that come with Cassandra. Note that the `OrderPreservingPartitioner` and `CollatingOrderPreservingPartitioner` are deprecated as of Cassandra 0.7 in favor of the `ByteOrderedPartitioner`:

- **ByteOrderedPartitioner** - Row keys are stored in order of their raw bytes rather than converting them to encoded strings. Tokens are calculated 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.
- **OrderPreservingPartitioner** - Row keys are stored in order based on the UTF-8 encoded value of the row keys. Requires row keys to be UTF-8 encoded strings.
- **CollatingOrderPreservingPartitioner** - Row keys are stored in order based on the United States English locale (EN_US). Also requires row keys to be UTF-8 encoded strings.

About Replication in Cassandra

Replication is the process of storing copies of data on multiple nodes to ensure reliability and fault tolerance. When you create a keyspace in Cassandra, you must decide the *replica placement strategy*, that is, the number of replicas and how those replicas are distributed across nodes in the cluster. The replication strategy relies on the cluster-configured *snitch* to help it determine the physical location of nodes and their proximity to each other.

The total number of replicas across the cluster is often referred to as the *replication factor*. A replication factor of 1 means that there is only one copy of each row. A replication factor of 2 means two copies of each row. All replicas are equally important; there is no *primary* or *master* replica in terms of how read and write requests are handled.

As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, it is possible to increase 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.

About Replica Placement Strategy

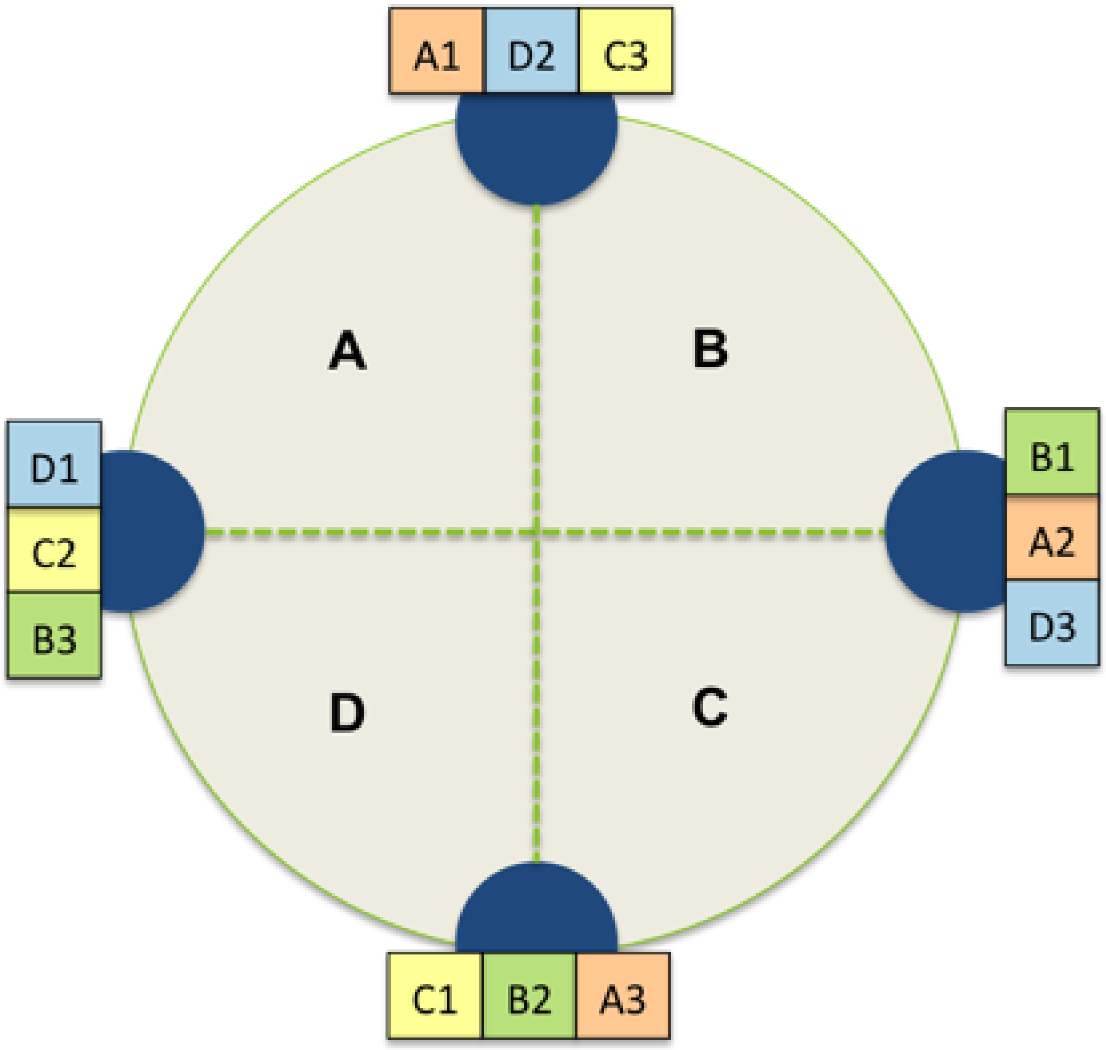
The replica placement strategy determines how replicas for a keyspace are distributed across the cluster. The replica placement strategy is set when you create a keyspace.

You can choose from several strategies based on your goals and the information you have about where nodes are located.

SimpleStrategy

SimpleStrategy is the default replica placement strategy when creating a keyspace using the Cassandra CLI. Other interfaces, such as the CQL utility, require you to explicitly specify a strategy.

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.



NetworkTopologyStrategy

`NetworkTopologyStrategy` is the preferred replication placement strategy when you have information about how nodes are grouped in your data center, or when you have (or plan to have) your cluster deployed across multiple data centers. This strategy allows you to specify how many replicas you want in each data center.

Note

In Cassandra, the term *data center* is synonymous with *replication group*. It is a group of related nodes configured together within a cluster for replication purposes. It is not necessarily a *physical* data center.

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

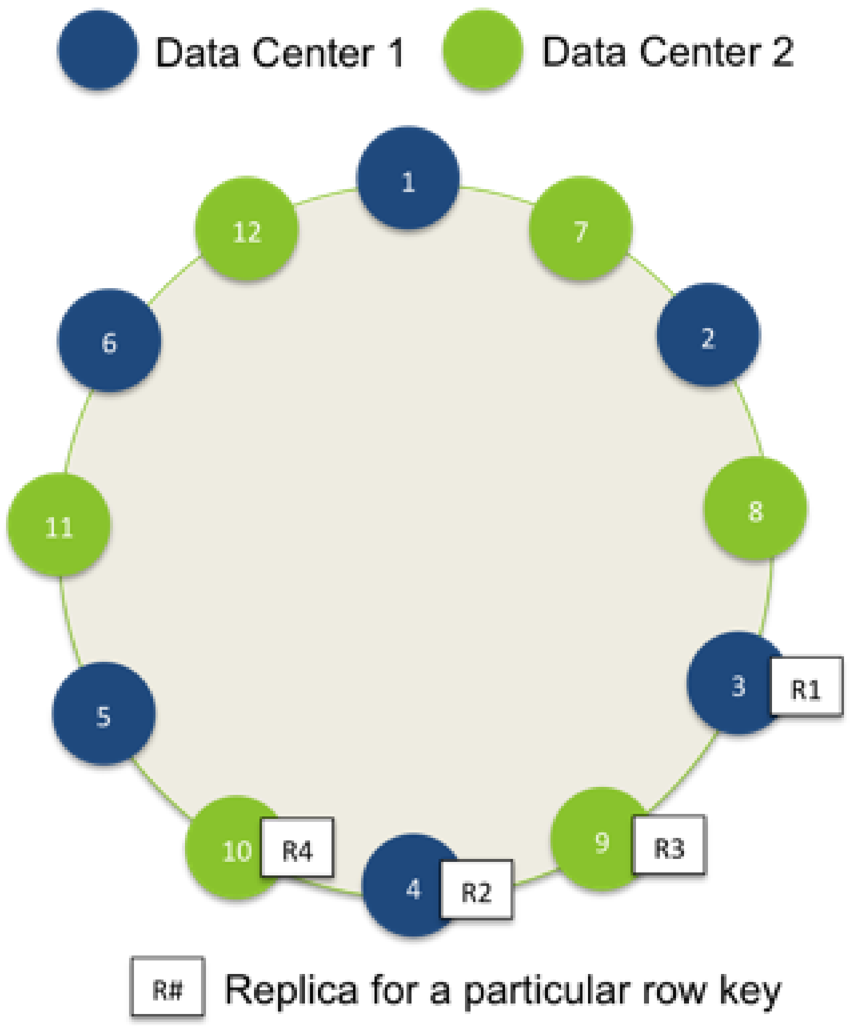
The two most common ways to configure multiple data center clusters are:

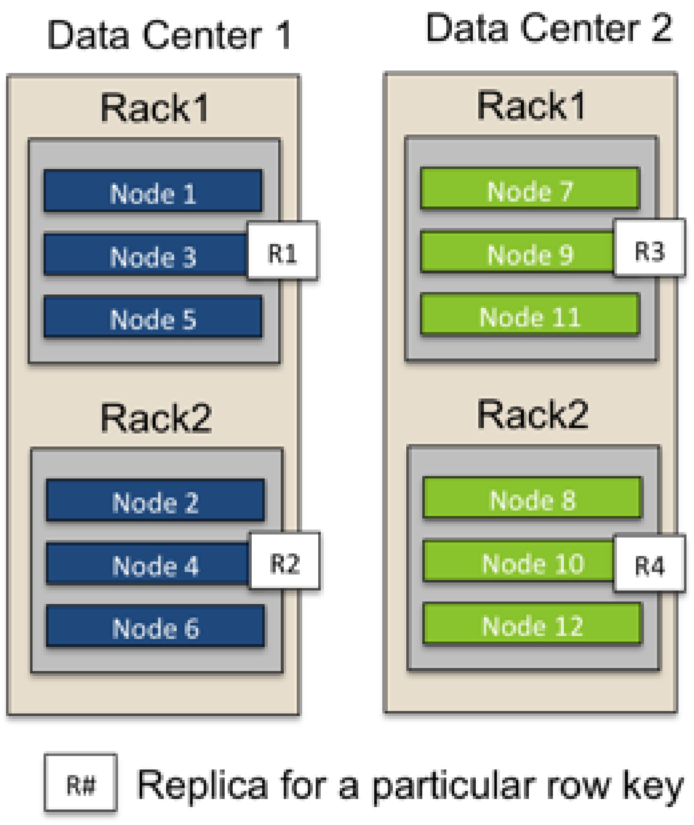
- Two replicas in each data center. This configuration tolerates the failure of a single node per replication group and still allows local reads at a *consistency level* of `ONE`.
- Three replicas in each data center. This configuration tolerates the failure of a one node per replication group at a strong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

Asymmetrical replication groupings are also possible depending on your use case. For example, you may want to have three replicas per data center to serve real-time application requests, and then have a single replica in a separate data center designated to running analytics.

With `NetworkTopologyStrategy`, replica placement is determined independently within each data center (or replication group). The first replica per data center is placed according to the *partitioner* (same as with `SimpleStrategy`). Additional replicas in the same data center are then determined by walking the ring clockwise until a node in a different rack from the previous replica is found. If there is no such node, additional replicas are placed in the same rack. `NetworkTopologyStrategy` places replicas on distinct racks if possible. Nodes in the same rack (or similar physical grouping) can easily fail at the same time due to power, cooling, or network issues.

Here is an example of how `NetworkTopologyStrategy` places replicas spanning two data centers with a total replication factor of 4 (two replicas in Data Center 1 and two replicas in Data Center 2):


Data Center 1
Data Center 2
1
2
3
4
5
6
7
8
9
10
11
12
R1
R2
R3
R4
R#
Replica for a particular row key



Notice how tokens are assigned to alternating racks.

`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.

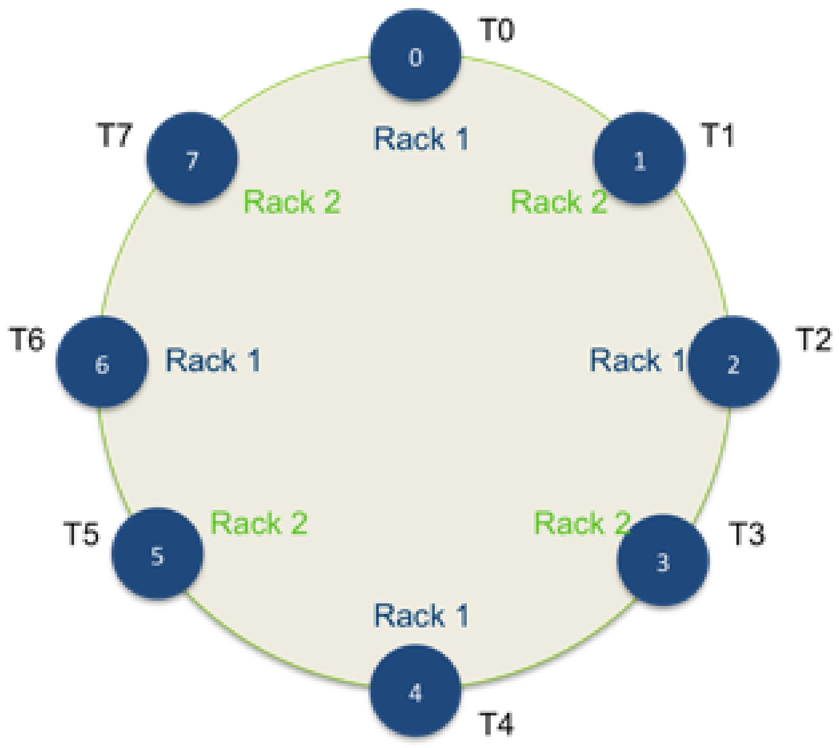
Note

`NetworkTopologyStrategy` should be used in place 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 is is a configurable component of a Cassandra cluster that 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 within the confines of the replica placement strategy. The snitch does not affect requests between the client application and Cassandra and it does not control which node a client connects to.

Snitches are configured for a Cassandra cluster in the *cassandra.yaml* configuration file. All nodes in a cluster must use the same snitch configuration. When assigning tokens, assign them to alternating racks. For example: rack1, rack2, rack3, rack1, rack2, rack3, and so on.



Assign tokens to nodes in alternating racks

The following snitches are available:

SimpleSnitch

The SimpleSnitch (the default) is appropriate if you have no rack or data center information available. Single-data center deployments (or single-zone in public clouds) usually fall into this category.

If using this snitch, use `replication_factor=<#>` when defining your keyspace *strategy_options*. This snitch does not recognize data center or rack information.

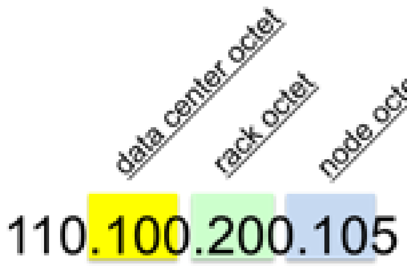
DseSimpleSnitch

DseSimpleSnitch is used in DataStax Enterprise (DSE) deployments only. It logically configures Hadoop analytics nodes in a separate data center from pure Cassandra nodes in order to segregate analytic and real-time workloads. It can be used for mixed-workload DSE clusters located in one physical data center. It can also be used for multi-data center DSE clusters that have exactly 2 data centers, with all analytic nodes in one data center and all Cassandra real-time nodes in the other data center.

If using this snitch, use `Analytics` or `Cassandra` as your data center names when defining your keyspace *strategy_options*.

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.



If using this snitch, use the second octet number of your node IPs as your data center names when defining your keyspace *strategy_options*. For example, `100` would be 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. See *Configuring the PropertyFileSnitch* for more information.

When using this snitch, you can define your data center names to be whatever you want. Make sure that the data center names you define in the `cassandra-topology.properties` file correlates to the name of your data centers in your keyspace *strategy_options*.

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.

If using this snitch, use the EC2 region name (for example,``us-east``) as your data center name when defining your keyspace *strategy_options*.

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 multi-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, open `storage_port` or `ssl_storage_port` on the public IP firewall.

If using this snitch, use the EC2 region name (for example,``us-east``) as your data center names when defining your keyspace *strategy_options*.

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, and is recommended for use in most deployments.

Dynamic snitch thresholds can be configured in the *cassandra.yaml* configuration file for a node.

About Client Requests in Cassandra

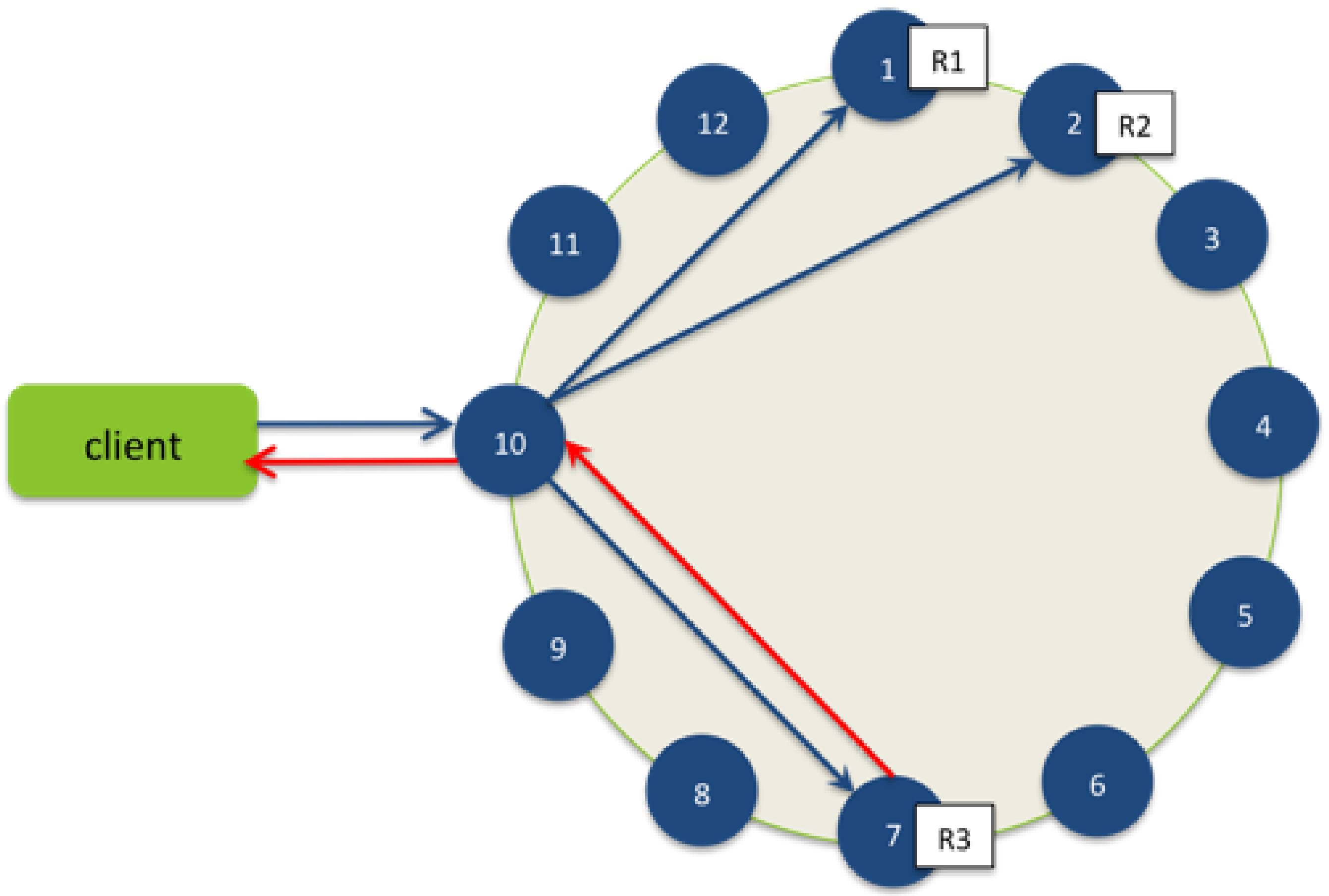
All nodes in Cassandra are peers. A client read or write request can go to any node in the cluster. When a client connects to a node and issues a read or write request, that node serves as the *coordinator* for that particular client operation.

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.

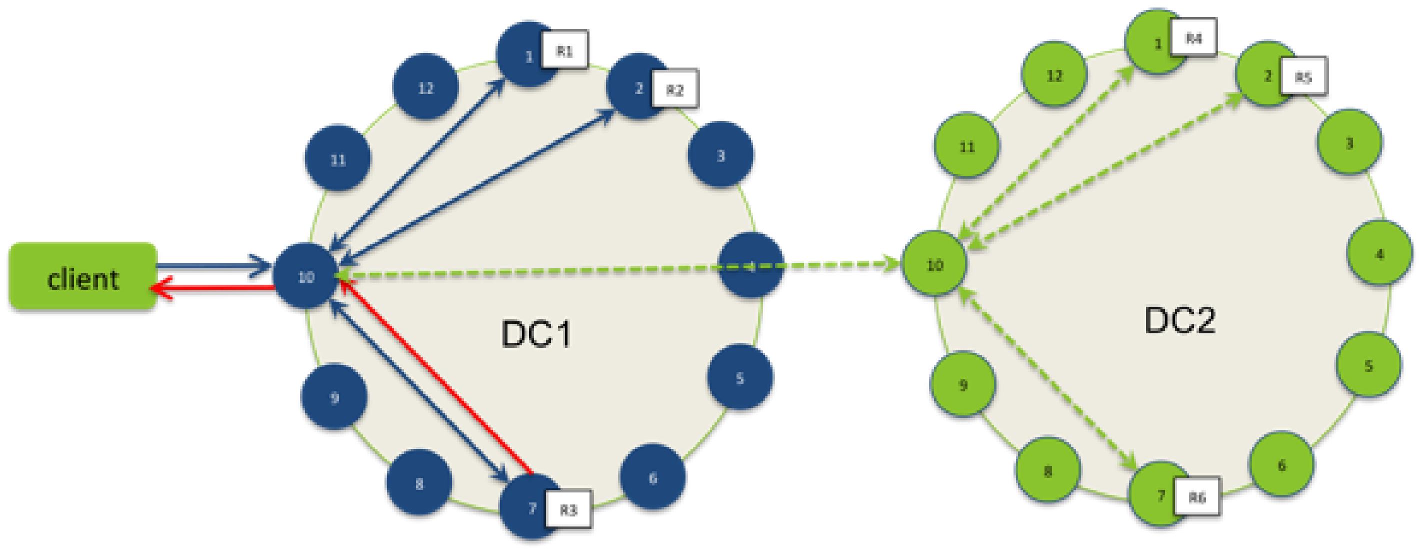


Also see *About Writes in Cassandra* for more information about how Cassandra processes writes locally at the node level.

About Multi-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.



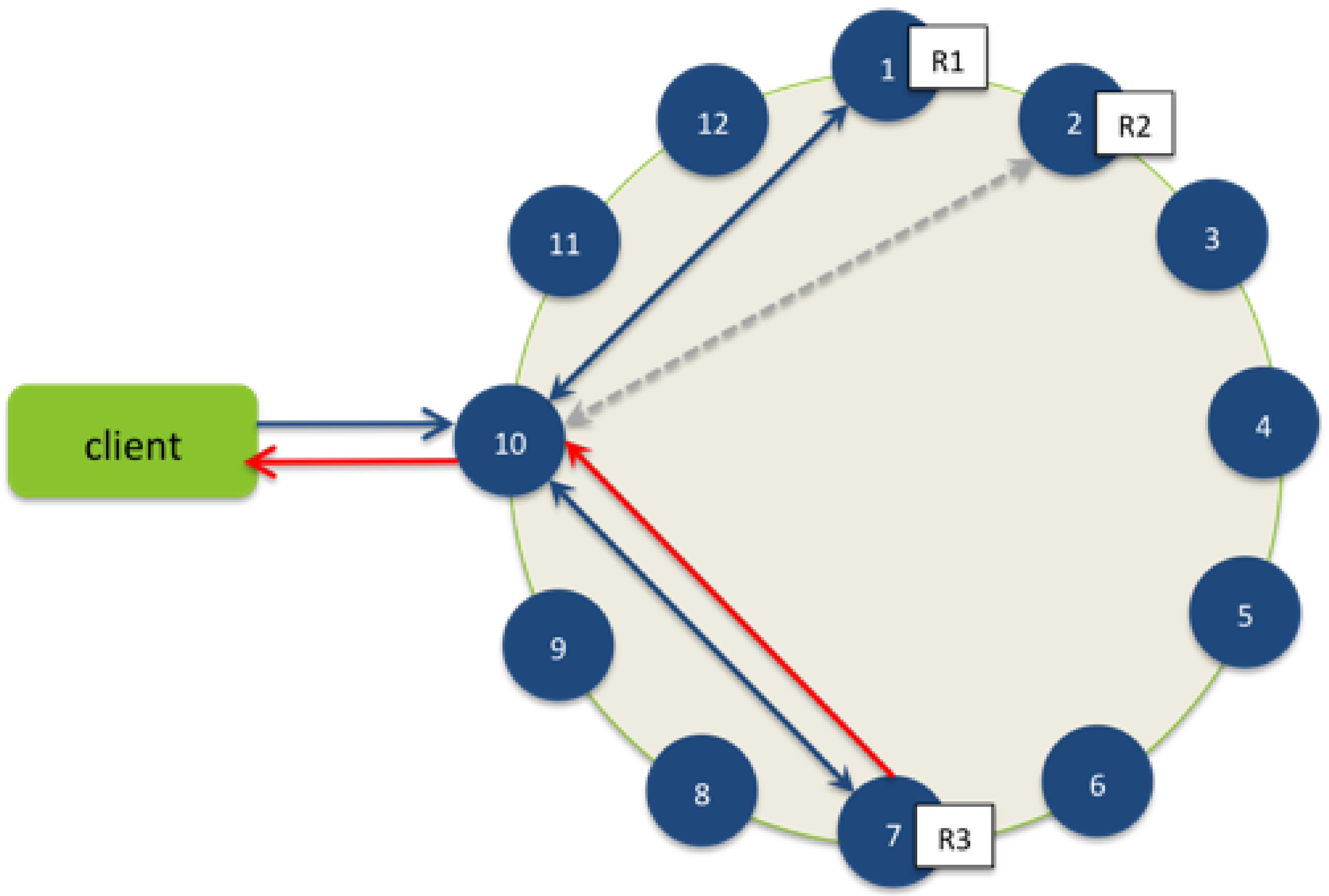
About Read Requests

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 will send these requests to the replicas that are currently responding most promptly. The nodes contacted will respond with the requested data; if multiple nodes are contacted, the rows from each replica are compared in memory to see if they are consistent. If they are not, then the replica that has the most recent data (based on the timestamp) is used by the coordinator to forward the result back to the client.

To ensure that all replicas have the most recent version of frequently-read data, the coordinator also contacts and 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. This process 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. These 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,
    ByteBufferUitl.bytes(0)
  );

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

The Cassandra WordCount example has been revised to demonstrate secondary index support. It is availabe 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 your typical application workload.

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, a minimum of than 8GB of RAM is needed. DataStax recommends 16GB - 32GB.
- 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.)
- For a virtual environment use a minimum of 4GB, such as Amazon EC2 Large instances. For production clusters with a healthy amount of traffic, 8GB is more common.

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).

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 also a valid alternative for Cassandra. Cassandra's sequential, streaming write patterns minimize the undesirable effects of write amplification associated with SSDs.
- Ideally Cassandra needs at least two disks, 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.
  - Because data is stored in the memtable, generally RAID is not needed for the commit log disk, but 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 bottlenecksT.

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

Cassandra clusters can be deployed on cloud infrastructures such as Amazon EC2.

For production Cassandra clusters on EC2, use Large or Extra Large instances with local storage. RAID0 the ephemeral disks, and put both the data directory and the commit log on that volume. This has proved to be better in practice than putting the commit log on the root volume (which is also a shared resource). For data redundancy, consider deploying your Cassandra cluster across multiple availability zones or using EBS volumes to store your Cassandra backup files.

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.

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, initialize your EC2 instances for each Cassandra node and then configure the cluster as a multi data center cluster.

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. This section explains the various configuration decisions that need to be made before deploying a Cassandra cluster, be it a single-node, multi-node, or multi-data center cluster.

These properties mentioned in this section 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 `/var/lib/cassandra`. In a production cluster deployment, you should change the *commitlog_directory* so it is on a different disk device than the *data_file_directories*.

Gossip Settings

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

Property	Description
cluster_name	Name of the cluster that this node is joining. Should be the same for every node in the cluster.
listen_address	The IP address or hostname that other Cassandra nodes will use to connect to this node. Should be changed from `localhost` to the public address for the host.
seeds	A comma-delimited list of node IP addresses used to bootstrap the gossip process. Every node should have the same list of seeds. In multi data center clusters, the seed list should include a node from *each* data center.
storage_port	The intra-node communication port (default is 7000). Should be the same for every node in the cluster.
initial_token	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
```

Partitioner Settings

When you deploy a Cassandra cluster, you need to make sure that each node is responsible for roughly an equal amount of data. This is also known as load balancing. This is done by configuring the *partitioner* for each node, and correctly assigning the node an *initial_token* value.

DataStax strongly recommends using the *RandomPartitioner* (the default) for all cluster deployments. Assuming use of this partitioner, each node in the cluster is assigned a *token* that represents a hash value within the range of 0 to 2**127.

For clusters where all nodes are in a single data center, you can calculate tokens by dividing the range by the total number of nodes in the cluster. In multi data-center deployments, tokens should be calculated such that each data center is individually load balanced as well. See *Generating Tokens* for the different approaches to generating tokens for nodes in single and multi-data center clusters.

Snitch Settings

The snitch is responsible for knowing the location of nodes within your network topology. This affects where replicas are placed as well as how requests are routed between replicas. The *endpoint_snitch* property configures the snitch for a node. All nodes should have the exact same snitch configuration.

For a single data center (or single node) cluster, using the default *SimpleSnitch* is usually sufficient. However, if you plan to expand your cluster at a later time to multiple racks and data centers, it will be easier if you choose a rack and data center aware snitch from the start. All *snitches* are compatible with all *replica placement strategies*.

Configuring the PropertyFileSnitch

The PropertyFileSnitch allows you to define your data center and rack names to be whatever you want. Using this snitch requires you to define network details for each node in the cluster in a `cassandra-topology.properties` configuration file. This file is located in `/etc/cassandra/conf/cassandra.yaml` in packaged installations or `<install_location>/conf/cassandra.yaml` in binary installations.

Every node in the cluster should be described in this file, and this file should be exactly the same on every node in the cluster.

For example, supposing you had non-uniform IPs and two physical data centers with two racks in each, and a third logical data center for replicating analytics data:

```
# 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
```

Make sure the data center names you define in the `cassandra-topology.properties` file correlates to what you name your data centers in your keyspace *strategy_options*.

Choosing Keyspace Replication Options

When you create a keyspace, you must define the *replica placement strategy* and the number of replicas you want. DataStax recommends always choosing `NetworkTopologyStrategy` for both single and multi-data center clusters. It is as easy to use as `SimpleStrategy` and allows for expansion to multiple data centers in the future, should that become useful. It is much easier to configure the most flexible replication strategy up front, than to reconfigure replication after you have already loaded data into your cluster.

`NetworkTopologyStrategy` takes as options the number of replicas you want *per data center*. Even for single data center (or single node) clusters, you can use this replica placement strategy and just define the number of replicas for one data center. For example (using `cassandra-cli`):

```
[default@unknown] CREATE KEYSPACE test
      WITH placement_strategy = 'NetworkTopologyStrategy'
      AND strategy_options=[{us-east:6}];
```

Or for a multi-data center cluster:

```
[default@unknown] CREATE KEYSPACE test
      WITH placement_strategy = 'NetworkTopologyStrategy'
      AND strategy_options=[{DC1:6,DC2:6,DC3:3}];
```

When declaring the keyspace *strategy_options*, what you name your data centers depends on the *snitch* you have chosen for your cluster. The data center names must correlate to the snitch you are using in order for replicas to be placed in the correct location.

As a general rule, the number of replicas should not exceed the number of nodes in a replication group. However, it is possible to increase the number of replicas, and then add the desired number of nodes afterwards. When the replication factor exceeds the number of nodes, writes will be rejected, but reads will still be served as long as the desired *consistency level* can be met.

Installing a Cassandra Cluster

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

For information on installing on Windows, see the *Installing the DataStax Community Binaries on Windows*.

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.

- Oracle Java SE Runtime Environment (JRE) **6**. Java 7 is not recommended.
- Java Native Access (JNA) is required for production installations. See *Installing JNA*.

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
   ```

   DataStax recommends using the most recently released version of Oracle Java SE Runtime Environment (JRE) 6 on all DSE nodes. Versions earlier than 1.6.0_19 should not be used. 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.
- Oracle Java SE Runtime Environment (JRE) **6**. Java 7 is not recommended.
- Java Native Access (JNA) is required for production installations. See *Installing JNA*.

Note

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*.

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
   ```

   DataStax recommends using the most recently released version of Oracle Java SE Runtime Environment (JRE) 6 on all DSE nodes. Versions earlier than 1.6.0_19 should not be used. Java 7 is not recommended. If you need help installing Java, see *Installing the JRE on Debian or Ubuntu Systems*.
2. Edit the aptitude repository source list file (`/etc/apt/sources.list`).

   ```
   $ sudo vi /etc/apt/sources.list
   ```
3. In this file, add the DataStax Community repository.

   ```
   deb http://debian.datastax.com/community stable main
   ```
4. (Debian Systems Only) 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.
5. Add the DataStax repository key to your aptitude trusted keys.

   ```
   $ curl -L http://debian.datastax.com/debian/repo_key | sudo apt-key add -
   ```
6. Install the package and Python CQL driver.

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

   This installs the DataStax Community distribution of Cassandra and the OpsCenter Community Edition. By default, the Debian packages start the Cassandra service automatically.
7. To stop the service and clear the initial gossip history that gets populated by this initial start:

   ```
   $ sudo service cassandra stop
   $ sudo bash -c 'rm /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:

- Oracle Java SE Runtime Environment (JRE) **6**. Java 7 is not recommended.
- Java Native Access (JNA) is required for production installations. See *Installing JNA*.

  #### *Note*

  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*.

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
   ```

   DataStax recommends using the most recently released version of Oracle Java SE Runtime Environment (JRE) 6 on all DSE nodes. Versions earlier than 1.6.0_19 should not be used. 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*

Installing the JRE and JNA

Cassandra is Java program. It requires that the Java Runtime Environment (JRE) 1.6.0_19 or later from Oracle is installed on Linux systems; Java Native Access (JNA) is needed for production installations.

Installing Oracle JRE

Cassandra is a Java program and requires that the Java Runtime Environment (JRE) 1.6.0_19 or later from Oracle is installed on Linux systems.

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

Installing the JRE on RHEL or CentOS Systems

The RPM packages install the OpenJDK Java Runtime Environment instead of the Oracle JRE. After installing using the RPM packaged releases, configure your operating system to use the Oracle JRE instead of 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).
3. Go to the directory where you downloaded the JRE package, and change the permissions so the file is executable. For example:

   ```
   $ cd /tmp
   $ chmod a+x jre-6u32-linux-x64-rpm.bin
   ```

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

   ```
   $ sudo ./jre-6u32-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_32/bin/java 20000
   ```

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

   ```
   $ java -version
     java version "1.6.0_32"
     Java(TM) SE Runtime Environment (build 1.6.0_32-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_32/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-6u32-linux-i586.bin`, for your system and place it in `/opt/java/<32 or 64>`.
3. Make the file executable:

```
sudo chmod 755 /opt/java/32/jre-6u32-linux-i586.bin
```

4. Go to the new folder:

```
cd /opt/java
```

5. Execute the file:

```
sudo ./jre-6u32-linux-i586.bin
```

6. If needed, accept the license terms to continue installing the JRE.

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

```
sudo update-alternatives --install "/usr/bin/java" "java" "/opt/java/32/jre1.6.0_32/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.

8. Set the new JRE as the default:

```
sudo update-alternatives --set java /opt/java/32/jre1.6.0_32/bin/java
```

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

```
$ java -version

java version "1.6.0_32"
Java(TM) SE Runtime Environment (build 1.6.0_32-b05)
Java HotSpot(TM) 64-Bit Server VM (build 20.7-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.

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
```

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
   ```

Initializing a Cassandra Cluster on Amazon EC2 Using the DataStax AMI

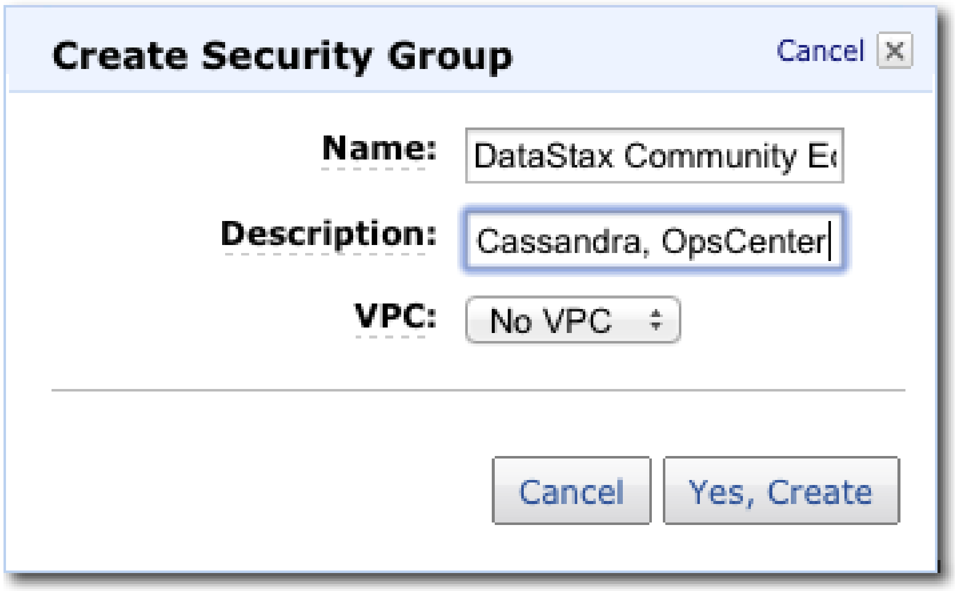
This is a step-by-step guide to using the Amazon Web Services EC2 Management Console to set up a simple Cassandra cluster using the DataStax Community Edition AMI (Amazon Machine Image). Installing via the AMI allows you to quickly deploy a Cassandra cluster within a single availability zone. When you launch the AMI, you can specify the total number of nodes in your cluster.

The DataStax Cassandra AMI does the following:

- Installs Cassandra on a Ubuntu 10.10 (Maverick Meercat) image on an AMD64-server-20111001.
- Uses RAID0 ephemeral disks for data storage and commit logs.
- Uses the private interface for intra-cluster communication.
- Configures a Cassandra cluster using the RandomPartitioner .
- Configures the Cassandra replication strategy using the `EC2Snitch`
- Sets the seed node cluster-wide.
- Starts Cassandra on all the nodes.
- Installs DataStax OpsCenter on the first node in the cluster (by default).

Creating an EC2 Security Group for DataStax Community Edition

1. In your Amazon EC2 Console Dashboard, select **Security Groups** in the **Network & Security** section.
2. Click **Create Security Group**. Fill out the name and description and then click **Yes, Create**.



Name: DataStax Community E

Description: Cassandra, OpsCenter

VPC: No VPC

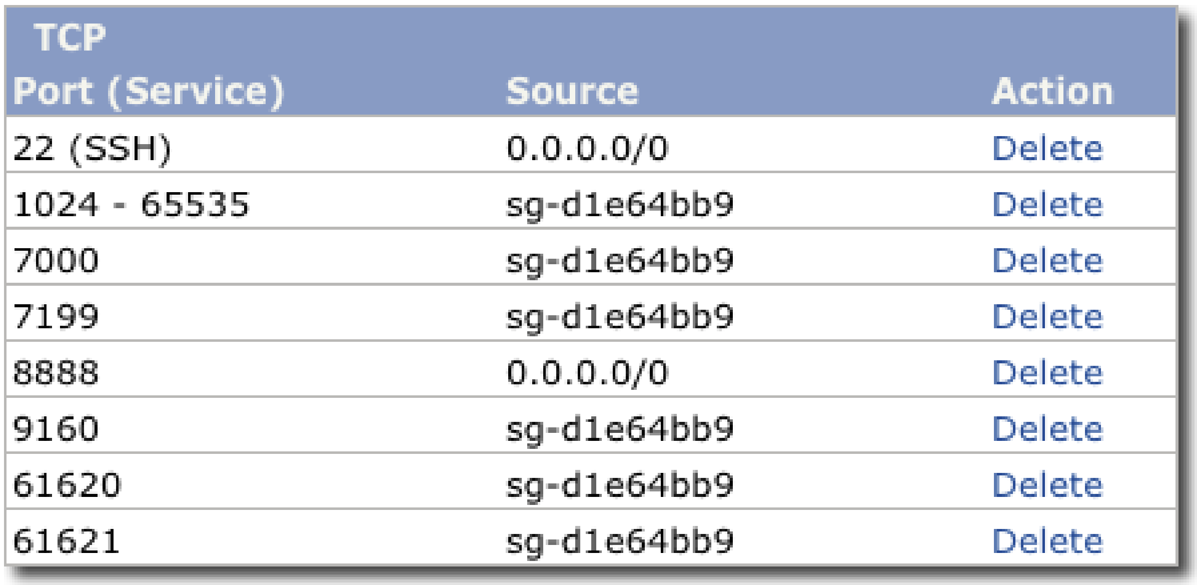
3. Click **Inbound** and add rules (using the **Create a new rule** drop-down list) for the following ports:

Port	Rule Type	Description
Public Facing Ports		
22	SSH	Default SSH port
OpsCenter Specific		
8888	Custom TCP Rule	OpsCenter website port
Intranode Ports		
1024+	Custom TCP Rule (current security group)	JMX reconnection/loopback ports
7000	Custom TCP Rule (current security group)	Cassandra intra-node port
7199	Custom TCP Rule (current security group)	Cassandra JMX monitoring port
9160	Custom TCP Rule (current security group)	Cassandra client port
OpsCenter Specific		
61620	Custom TCP Rule (current security group)	OpsCenter intra-node monitoring port
61621	Custom TCP Rule (current security group)	OpsCenter agent ports

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.

4. After you are done adding the above port rules, click **Apply Rule Changes**. Your completed port rules should look similar to this:



TCP Port (Service)	Source	Action
22 (SSH)	0.0.0.0/0	Delete
1024 - 65535	sg-d1e64bb9	Delete
7000	sg-d1e64bb9	Delete
7199	sg-d1e64bb9	Delete
8888	0.0.0.0/0	Delete
9160	sg-d1e64bb9	Delete
61620	sg-d1e64bb9	Delete
61621	sg-d1e64bb9	Delete

Warning

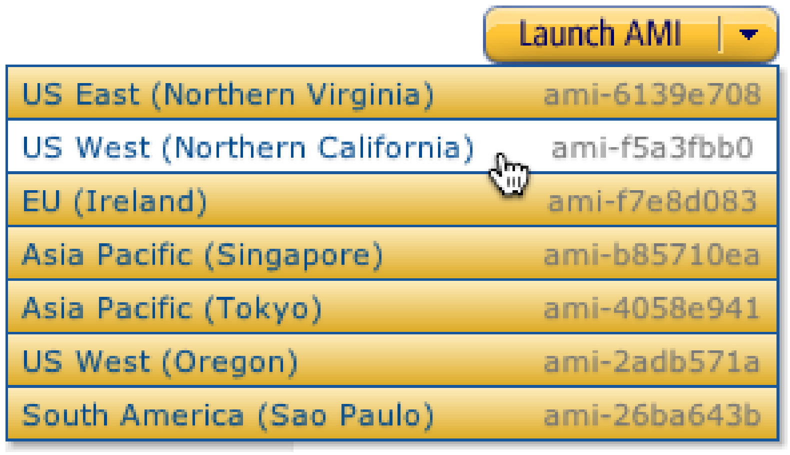
This security configuration shown in the above example opens up all externally accessible ports to incoming traffic from any IP address (0.0.0.0/0). The risk of data loss is high. If you desire a more secure configuration, see the Amazon EC2 help on Security Groups.

Launching the DataStax Community AMI

After you have created your security group, you are ready to launch an instance of DataStax Enterprise using the DataStax AMI.

1. Right-click the following link to open the **DataStax Amazon Machine Image** page in a new window:

   https://aws.amazon.com/amis/datastax-auto-clustering-ami-2-2
2. Click **Launch AMI**, then select the region where you want to launch the AMI.

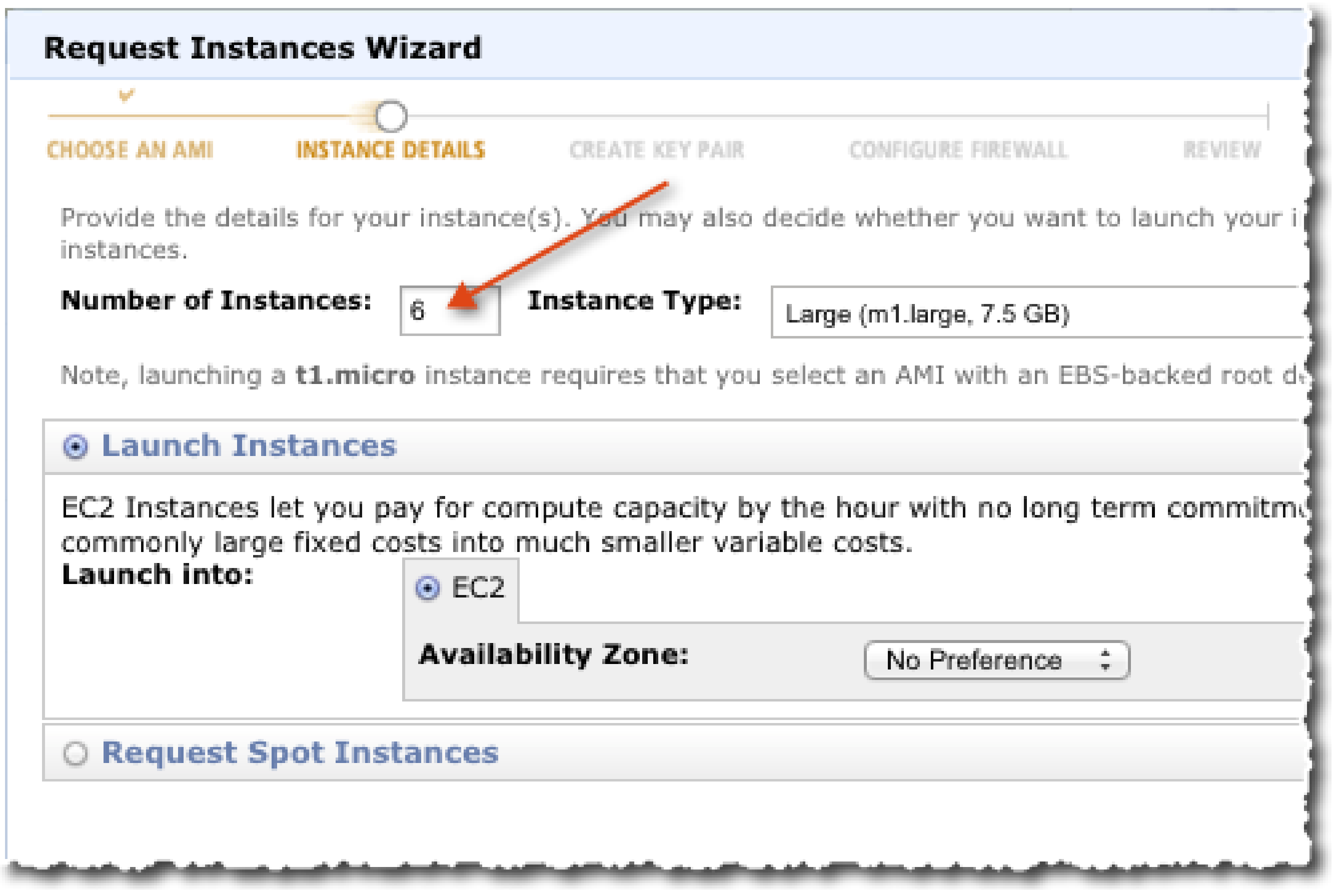


US East (Northern Virginia)	ami-6139e708
US West (Northern California)	ami-f5a3fbb0
EU (Ireland)	ami-f7e8d083
Asia Pacific (Singapore)	ami-b85710ea
Asia Pacific (Tokyo)	ami-4058e941
US West (Oregon)	ami-2adb571a
South America (Sao Paulo)	ami-26ba643b

3. On the **Request Instances Wizard** page, verify the settings and then click **Continue**.

4. On the **Instance Details** page, enter the total number of nodes that you want in your cluster, select the **Instance Type** (minimum Large), and then click **Continue**.

Note
Small and Medium instances are not supported.



Provide the details for your instance(s). You may also decide whether you want to launch your i
instances.

Number of Instances: 6 **Instance Type:** Large (m1.large, 7.5 GB)

Note, launching a **t1.micro** instance requires that you select an AMI with an EBS-backed root d

◉ Launch Instances

EC2 Instances let you pay for compute capacity by the hour with no long term commitm
commonly large fixed costs into much smaller variable costs.

Launch into: ◉ EC2

Availability Zone: No Preference

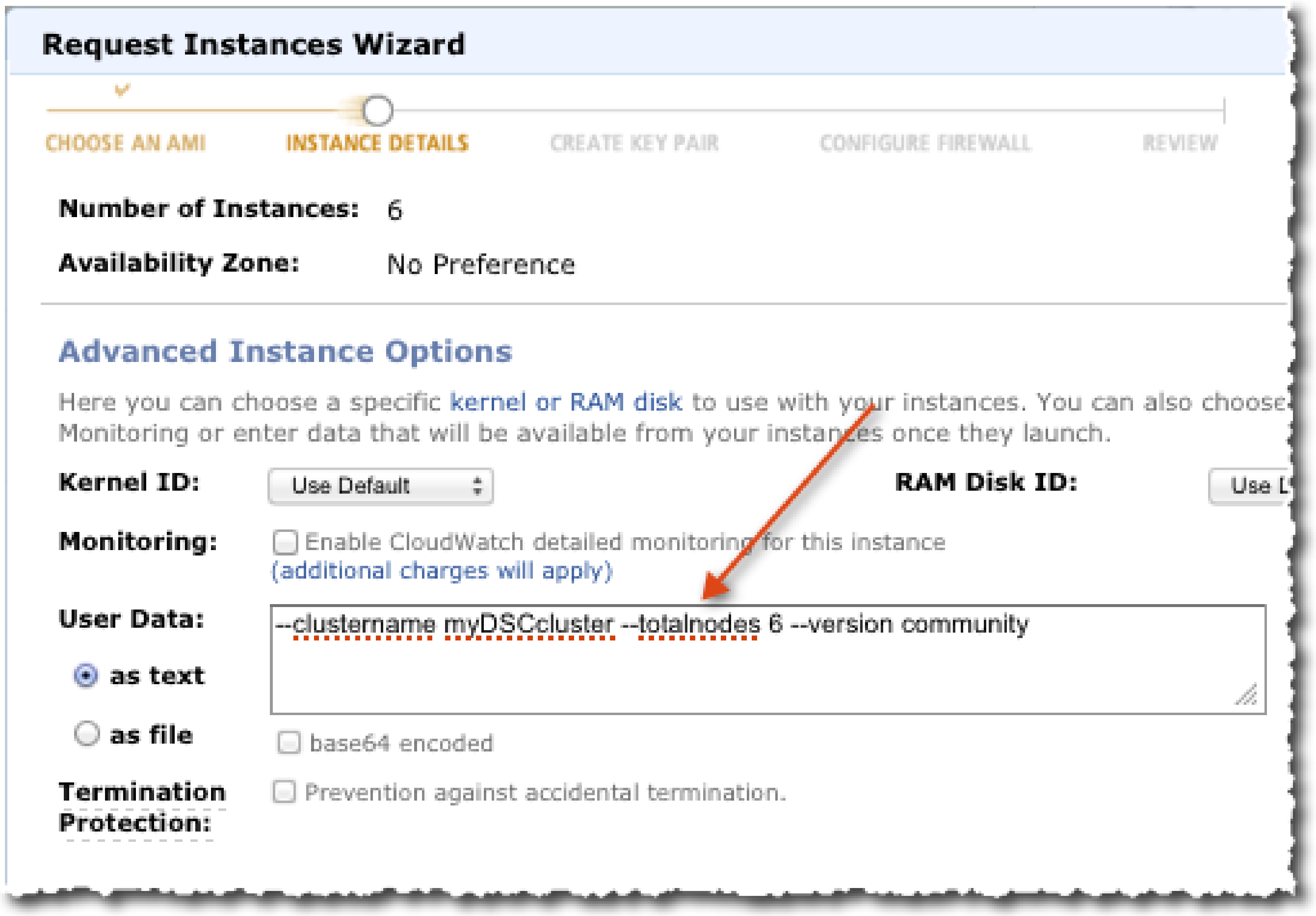
○ Request Spot Instances

5. On the next page, under **Advanced Instance Options**, add the following options to the **User Data** section according to the type of cluster you want, and then click **Continue**.

   For new DataStax Enterprise clusters the available options are:

Option	Description
`--clustername <name>`	Required. The name of the cluster.
`--totalnodes <#_nodes>`	Required. The total number of nodes in the cluster.
`--version community`	Required. The version of the cluster. Use `community` to install the latest version of DataStax Community.
`--opscenter [no]`	Optional. By default, DataStax OpsCenter is installed on the first instance. Specify no to disable.
`--reflector <url>`	Optional. Allows you to use your own reflector. Default: `http://reflector2.datastax.com/reflector2.php`

   For example, `--clustername myDSCcluster --totalnodes 6 --version community`



Number of Instances: 6

Availability Zone: No Preference

Advanced Instance Options

Here you can choose a specific kernel or RAM disk to use with your instances. You can also choose Monitoring or enter data that will be available from your instances once they launch.

Kernel ID: Use Default

RAM Disk ID: Use

Monitoring: Enable CloudWatch detailed monitoring for this instance (additional charges will apply)

User Data: [✓ as text] as file

```
--clustername myDSCcluster --totalnodes 6 --version community
```

base64 encoded

Termination Protection: Prevention against accidental termination.

6. On the **Tags** page, give a name to your DataStax Community instance, such as `cassandra-node`, and then click **Continue**.
7. On the **Create Key Pair** page, create a new key pair or select an existing key pair, and then click **Continue**. Save this key (`.pem` file) to your local machine; you will need it to log in to your DataStax Enterprise instance.
8. On the **Configure Firewall** page, select the security group that you created earlier and click **Continue**.
9. On the **Review** page, review your cluster configuration and then click **Launch**.

10. Close the **Launch Install Wizard** and go to the **My Instances** page to see the status of your Cassandra instance. Once a node has a status of **running**, you can connect to it.

Connecting to Your DataStax Community EC2 Instance

You can connect to your new Datastax Community EC2 instance using any SSH client, such as PuTTY or from a Terminal. To connect, you will need the private key (`.pem` file you created earlier and the public DNS name of a node.

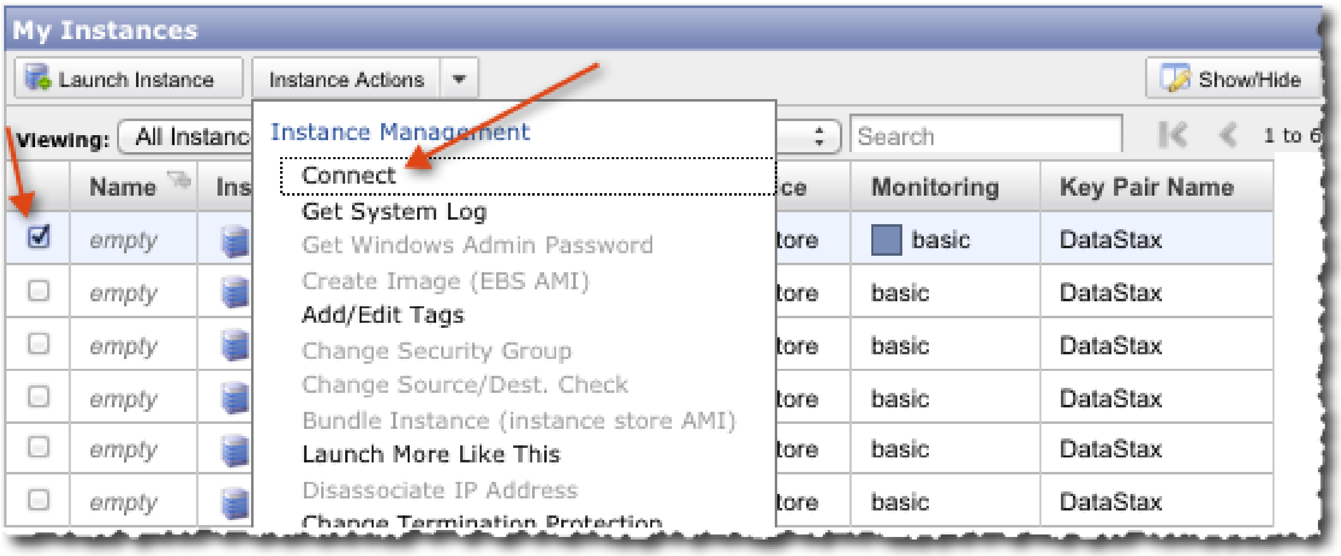
Connect as user `ubuntu` rather than as `root`.

If this is the first time you are connecting, copy your private key file (<keyname>.pem) you downloaded earlier to your home directory, and change the permissions so it is not publicly viewable. For example:

```
chmod 400 datastax-key.pem
```

1. From the **My Instances** page in your AWS EC2 Dashboard, select the node that you want to connect to.

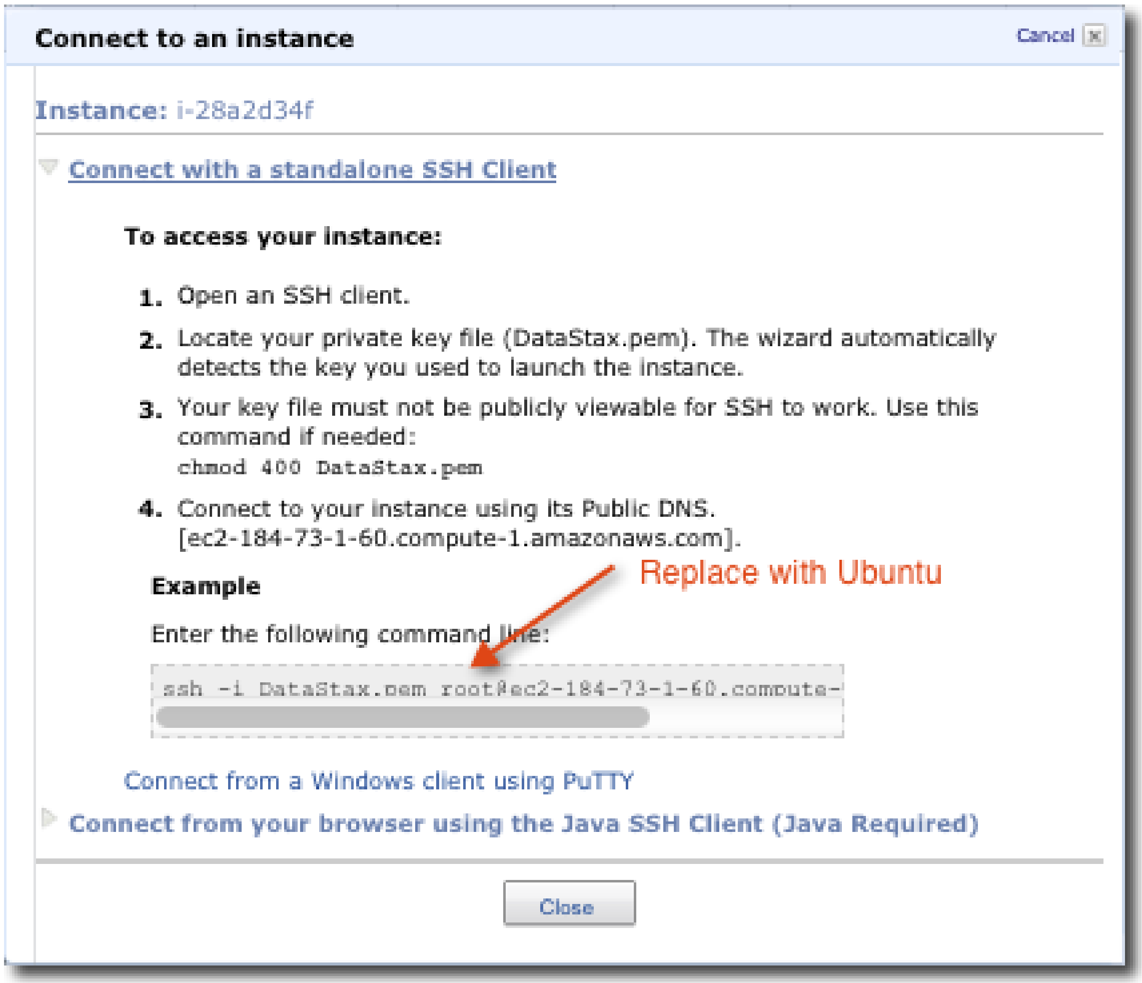
   Because all nodes are peers in Cassandra, you can connect using any node in the cluster. However, the first node generally runs OpsCenter and is the Cassandra seed node.



	Name	Ins	ce	Monitoring	Key Pair Name
☑	empty		tore	🟦 basic	DataStax
☐	empty		tore	basic	DataStax
☐	empty		tore	basic	DataStax
☐	empty		tore	basic	DataStax
☐	empty		tore	basic	DataStax
☐	empty		tore	basic	DataStax

2. To get the public DNS name of a node, select **Instance Actions > Connect**.

3. In the **Connect Help - Secure Shell (SSH)** page, copy the command line and change the connection user from **root** to **ubuntu**, then paste it into your SSH client.



Connect to an instance

Instance: i-28a2d34f

Connect with a standalone SSH Client

To access your instance:

1. Open an SSH client.
2. Locate your private key file (DataStax.pem). The wizard automatically detects the key you used to launch the instance.
3. Your key file must not be publicly viewable for SSH to work. Use this command if needed:
   `chmod 400 DataStax.pem`
4. Connect to your instance using its Public DNS. [ec2-184-73-1-60.compute-1.amazonaws.com].

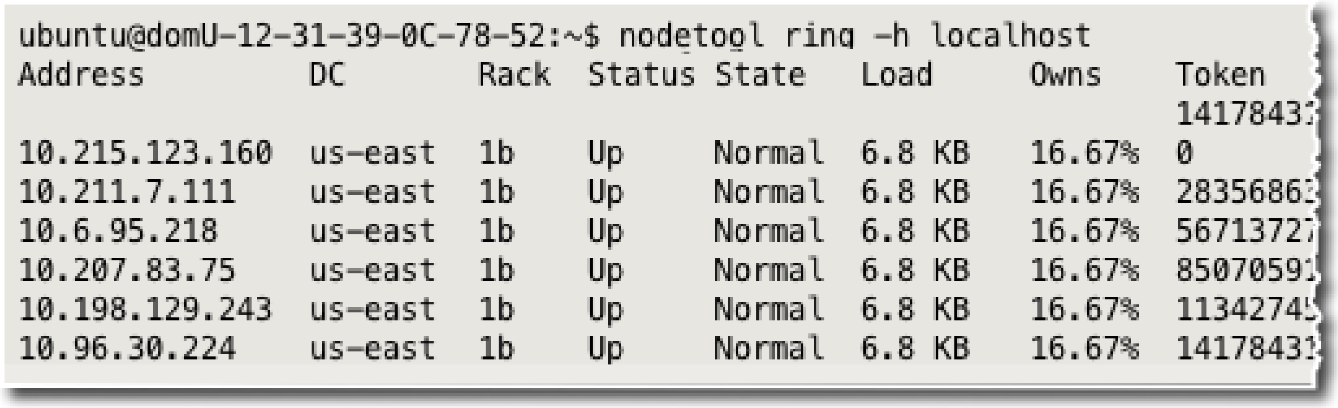
Example

Enter the following command line:

```
ssh -i DataStax.pem root@ec2-184-73-1-60.compute-
```

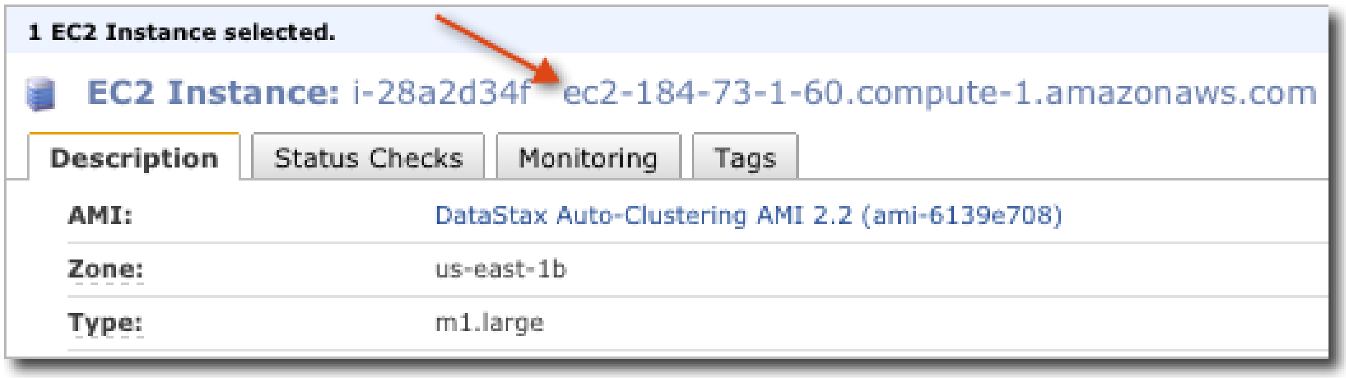
Cancel
Replace with Ubuntu
Connect from a Windows client using PuTTY
Connect from your browser using the Java SSH Client (Java Required)
Close

4. The AMI image configures your cluster and starts the Cassandra services. After you have logged into a node, run the *nodetool ring -h localhost* command (*nodetool*) to make sure your cluster is running.



```
ubuntu@domU-12-31-39-0C-78-52:~$ nodetool ring -h localhost
Address         DC          Rack  Status State   Load       Owns    Token
                                                                    14178431
10.215.123.160  us-east     1b    Up     Normal  6.8 KB     16.67%  0
10.211.7.111    us-east     1b    Up     Normal  6.8 KB     16.67%  28356863
10.6.95.218     us-east     1b    Up     Normal  6.8 KB     16.67%  56713727
10.207.83.75    us-east     1b    Up     Normal  6.8 KB     16.67%  85070591
10.198.129.243  us-east     1b    Up     Normal  6.8 KB     16.67%  11342745
10.96.30.224    us-east     1b    Up     Normal  6.8 KB     16.67%  14178431
```

5. If you installed the OpsCenter with your Cassandra cluster, allow about 60 to 90 seconds after the cluster has finished initializing for OpsCenter to start. You can launch OpsCenter using the URL: `http://<public-dns-of-first-instance>:8888`.

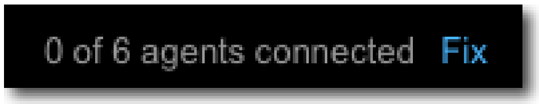


1 EC2 Instance selected.

EC2 Instance: i-28a2d34f ec2-184-73-1-60.compute-1.amazonaws.com

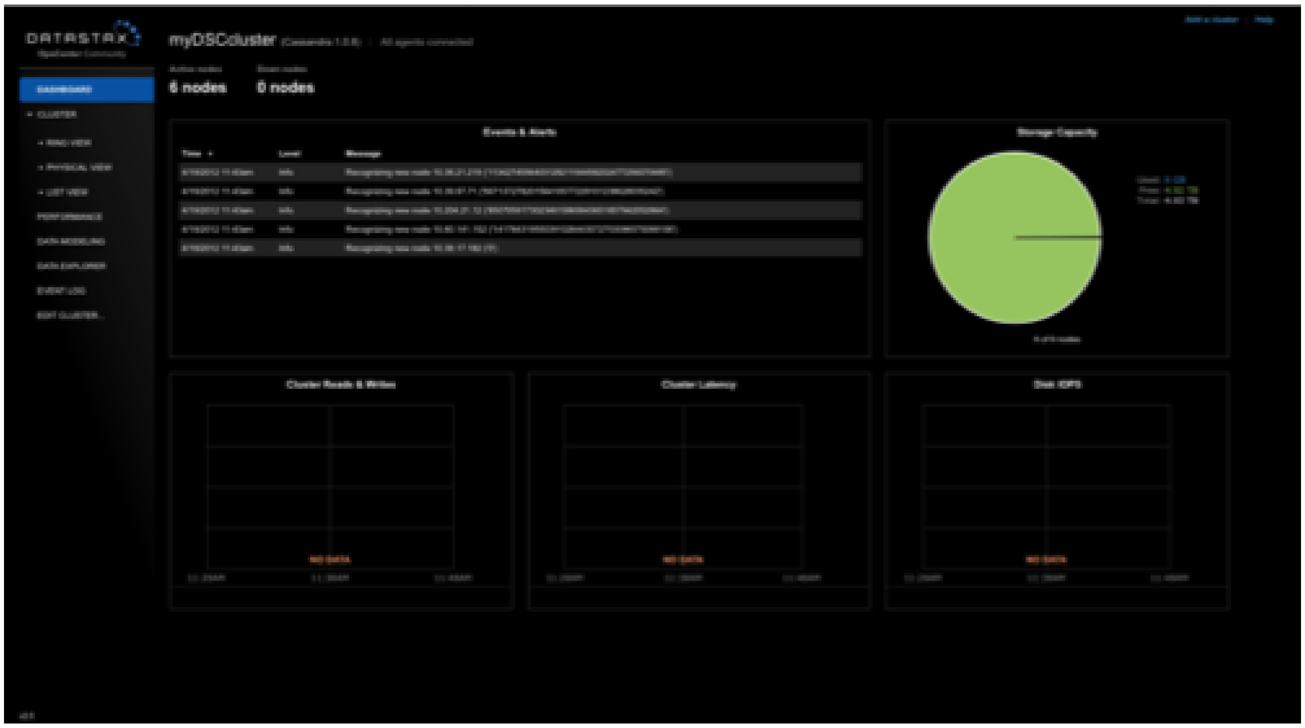
AMI:	DataStax Auto-Clustering AMI 2.2 (ami-6139e708)
Zone:	us-east-1b
Type:	m1.large

6. After the OpsCenter loads, you must install the OpsCenter agents to see the cluster performance data.
   a. Click the **Fix** link located near the top of the Dashboard in the left navigation pane to install the agents.



0 of 6 agents connected Fix

   b. When prompted for credentials for the agent nodes, use the username **ubuntu** and copy and paste the entire contents from your private key (`.pem`) file that you downloaded earlier.



Stopping or Starting a Node

To stop the service:

```
sudo service cassandra stop
```

To start the service:

```
sudo service cassandra start
```

What's Next

Expanding a Cassandra AMI Cluster

Expanding a Cassandra AMI Cluster

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

1. In the AWS Management Console, create a cluster with the number of nodes you want add, as described above.
2. After the nodes have initialized, login to each node and stop the service:

   ```
   sudo service cassandra stop
   ```

3. Integrate each node into the cluster by modifying the property settings in each node's `/etc/dse/cassandra/cassandra.yaml` file as necessary. 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"
   ```

   Be sure apply the correct token to the node; it determines the node's placement within the ring.
4. Set `auto_bootstrap` if required:
   - If adding nodes to an *existing* data center, set `auto_bootstrap: true` in the `cassandra.yaml` file.
   - If adding nodes to a new data center, keep this disabled (default), then run a rolling nodetool repair after all nodes have joined the ring.
5. If the *new* node has existing data, create a backup folder and move the data into it:

   ```
   sudo mkdir /raid0/cassandra.bak
   sudo mv /raid0/cassandra/* /raid0/cassandra.bak
   ```

   #### Note
   The asterisk preserves the `cassandra` folder and permissions.

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

   ```
   sudo service cassandra start
   ```

7. Check that the Cassandra ring is up and running:

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

Upgrading Cassandra

This section includes information on upgrading Cassandra between releases major and minor releases of Cassandra. It contains the following:

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

Note
This information also applies to DataStax Community Edition.

Best Practices for Upgrading Cassandra

The following best practices are recommended when upgrading Cassandra:

- Always take a snapshot before any 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.

  Note
  Snapshotting is fast, especially if you have JNA installed, and takes effectively zero disk space until you start compacting the live data files again.

- Be sure to check https://github.com/apache/cassandra/blob/trunk/NEWS.txt for any new information on upgrading.
- For a list of fixes and new features, see https://github.com/apache/cassandra/blob/trunk/CHANGES.txt.

About Upgrading Cassandra to 1.1

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 ColumnFamilies (and unchanged for ColumnFamilies 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. Versions 0.8.8 and 1.0.3-1.0.5 generate cross-datacenter forwarding that is incompatible with 1.1.

  Cross-datacenter forwarding means optimizing cross-datacenter replication. If DC1 needs to replicate a write to three replicas in DC2, only one message is sent across datacenters; 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 returns custom Cassandra objects. Any such 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 `KEY ``as the primary key name in some select statements, even if the ``PK` was actually given a different name. In Cassandra 1.1+ you must use the defined `PK` name.
- 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	
key_cache_size_in_mb	empty
key_cache_save_period	14400 (4 hours)
row_cache_size_in_mb	0 (disabled)
row_cache_save_period	0 (disabled)
1.0 Release (Column Family Attributes)	
key_cache_size	2MB (ignored in 1.1)
key_cache_save_period_in_seconds	na
row_cache_size	0 (ignored in 1.1)
row_cache_save_period_in_seconds	na

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. On each node, download and unpack the binary tarball package from the downloads section of the Cassandra website.
2. Account for *New Parameters between 1.0 and 1.1* in `cassandra.yaml`. You can copy your existing configuration file into the upgraded Cassandra instance and manually update it with new content.

3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Run `nodetool drain` on the node to flush the commit log.
5. Stop the old Cassandra process, then start the new binary process.
6. Monitoring the log files for any issues.
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 using Cassandra 1.0.3 and earlier, use *nodetool scrub* instead.)

To upgrade a RHEL or CentOS Installation

1. On each of your Cassandra nodes, run `sudo yum install apache-cassandra1`.
2. Account for *New Parameters between 1.0 and 1.1* in `cassandra.yaml`. The installer creates the file `cassandra.yaml.rpmnew` in `/etc/cassandra/default.conf/`. You can diff this file with your existing configuration and add new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Run `nodetool drain` on the node to flush the commit log.
5. Restart the Cassandra process.
6. Monitor the log files for any issues.
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 using Cassandra 1.0.3 and earlier, use *nodetool scrub* instead.)

To Upgrade a Debian or Ubuntu Installation

1. On each of your Cassandra nodes, run `sudo apt-get install cassandra1`.
2. Account for *New Parameters between 1.0 and 1.1* in `cassandra.yaml`. The installer creates the file `cassandra.yaml.rpmnew` in `/etc/cassandra/default.conf/`. You can diff this file with your existing configuration and add new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Run `nodetool drain` on the node to flush the commit log.
5. Restart the Cassandra process.
6. Monitor the log files for any issues.
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 using Cassandra 1.0.3 and earlier, use *nodetool scrub* instead.)

Upgrading Between Minor Releases of Cassandra 1.1.x

Upgrading minor releases 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 package installation:

1. On each node, download and unpack the binary tarball package from the downloads section of the Cassandra website .

2. Account for *New Parameters between 1.0 and 1.1* in `cassandra.yaml`. You can copy your existing configuration file into the upgraded Cassandra instance and manually update it with new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Flush the commit log on the upgraded node by running `nodetool drain`.
5. Stop the old Cassandra process, then start the new binary process.
6. Monitor the log files for any issues.

To upgrade a Debian/Ubuntu or RHEL/CentOS package installation:

1. On each node, download and install the package from the downloads section of the Cassandra website .
2. Account for *New Parameters between 1.0 and 1.1* in `cassandra.yaml`. You can copy your existing configuration file into the upgraded Cassandra instance and manually update it with new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Flush the commit log on the upgraded node by running `nodetool drain`.
5. Restart the Cassandra process.
6. Monitor the log files for any issues.

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.

Location of the property file:

You set properties for each node in the `cassandra.yaml` file. This file is located in different places depending on the type of installation:

- Packaged installations: `/etc/cassandra/conf/cassandra.yaml`
- Binary installations: `<install_location>/conf/cassandra.yaml`

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:

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 Cassandra or DataStax Enterprise 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)

ur ring is up and running:

ed installs: `nodetool ring -h localhost`

nstalls:

```
install_directory>
```

```
/nodetool ring -h localhost
```

ddress	DC	Rack	Status	State	Load	Owns	
0	DC1	rack1	Up	Normal	81.67KB	16.67%	0
1	DC2	rack1	Up	Normal	72.58KB	16.67%	2835
2	DC1	rack1	Up	Normal	67.97KB	16.67%	5671
3	DC2	rack2	Up	Normal	68.19KB	16.67%	8507
4	DC1	rack2	Up	Normal	72.58KB	16.67%	1134
5	DC2	rack2	Up	Normal	2.58KB	16.67%	1417

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. The steps for configuring multiple data centers on binary and packaged installations are the same except the configuration files are located in different directories.

Location of the property files in packaged installations:

- `/etc/cassandra/conf/cassandra.yaml`
- `/etc/cassandra/conf/cassandra-topology.properties`

Location of the property files in binary installations:

- `<install_location>/resources/cassandra/conf/cassandra.yaml`
- `<install_location>/resources/cassandra/conf/cassandra-topology.properties`

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
10.176.43.66
10.168.247.41
10.176.170.59
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 *Calculating Tokens for a Multiple Data Center Cluster*.

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 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)
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: 56713727820156410577229101238628035242
   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 node from *each* data center. It is a best practice to have at more than one seed node per data center.

   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)

our ring is up and running:

ed installs: `nodetool ring -h localhost`

nstalls:

```
install_directory>
/nodetool ring -h localhost
```

Address	DC	Rack	Status	State	Load	Owns	
1	DC1	RAC1	Up	Normal	81.67KB	16.67%	0
6	DC2	RAC1	Up	Normal	72.58KB	16.67%	10
41	DC1	RAC1	Up	Normal	67.97KB	16.67%	567
59	DC2	RAC1	Up	Normal	68.19KB	16.67%	567
70	DC1	RAC1	Up	Normal	72.58KB	16.67%	113
38	DC2	RAC1	Up	Normal	2.58KB	16.67%	113

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 (see step 5) 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: token = (2^127 / num_nodes_in_dc * n + DC_ID) where n is the node for which the token is being calculated and DC_ID is the numerical name.

Frequently Asked Questions

1. Can all the application data be 100% owned by Cassandra nodes?

There is no ownership. You set a replication factor (RF) for each data center, including the virtual analytics one. So, you might have one copy of the data in each of C1, C2, C3, and Analytics (AN), for example. Regardless of what data center or nodes you write to, the data is replicated to all four data centers. Replication is configured per keyspace.

For example, one keyspace with RF = {C1:1, C2:1, C3:1, AN:0}, and a different keyspace with RF = {C1:0, C2:0, C3:0, AN:1}. In such a configuration, if you write into the first keyspace, the analytics nodes do not have any copies of the data. If you write into the second keyspace only the analytics nodes have copies. If you write data, such as flat files, directly into CFS, by default only the AN nodes have copies of the data. The assumption is only the AN nodes need access to the flat files because their only use is for analytics. This is actually accomplished by having a Cassandra File System (CFS) keyspace, where AN has a RF > 0 and the others have RF=0.

2. Through replication, can you give the analytics nodes all of the application data?

Yes, as exemplified in the previous answer: RF = {C1: 1, C2:1, ... AN:1}

3. If all the analytics data is written to column families (CF), how can application nodes get a copy of the data?

The destination CFs used for the output of the analytics jobs are in a keyspace where only the Cassandra data centers have a RF > 0 (that is, the output of the analytics jobs do not need to be stored on the analytics nodes. There are common exceptions to this. If the output does not really belong in Cassandra, for some reason, such as the output is for analysis and not part of the operational data set, you can write the output into a keyspace where only the analytics DC had RF > 0. If you want both Cassandra and analytic nodes to have copies of the data, it is just a matter of setting the RF correctly on the keyspace you write to.

4. If you add or remove a node from a ring, do all nodes in the data centers need to be rebalanced?

You need to rebalance a data center after adding or removing a node. Nodes in other data centers do not have to be rebalanced. You need to stagger tokens between data centers for maximum data availability.

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, you must choose how the data (column family rows) is divided across the nodes in the cluster. A partitioner determines what each node stores by row (key). A token is a partitioner-dependent element of the cluster. Each node in a cluster is assigned a token and that token determines the node's position in the ring and what data the node is responsible for in the cluster. The tokens assigned to your nodes need to be distributed throughout the entire possible range of tokens. Each node is responsible for the region of the ring between itself (inclusive) and its predecessor (exclusive). As a simple example, if the range of possible tokens was 0 to 100 and you had four nodes, you would want the tokens for your nodes to be: 0, 25, 50, 75. This approach ensures that each node is responsible for an equal range of data. *Each data center should be partitioned as if it were its own distinct ring.* For more detailed information, see *About Data Partitioning in Cassandra*.

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

DataStax provides a Python program for generating tokens. Tokens are integers ranging from 0 to 2^{127} -1.

To set up the Token Generating Tool:

1. Using a text editor, create a new file named `tokengentool` for your token generator program.
2. Go to https://raw.github.com/riptano/ComboAMI/2.2/tokentoolv2.py.
3. Copy and paste the program into the `tokengentool` file.
4. Save and close the file.
5. Make it executable:

```
chmod +x tokengentool
```

6. Run the program:

   ```
   ./tokengentool <nodes_in_dc1> <nodes_in_dc2> ...
   ```

   The Token Generating Tool calculates the token values.
7. Enter the corresponding value for each node in the `initial_token` property of the node's `cassandra.yaml` file.

Calculating Tokens for a Single Data Center

For a single data center, using the *RandomPartitioner*, enter the number of nodes in Token Generating Tool. For example, for 6 nodes in a single data center, you enter:

```
./tokengentool 6
```

The tool displays the token for each node:

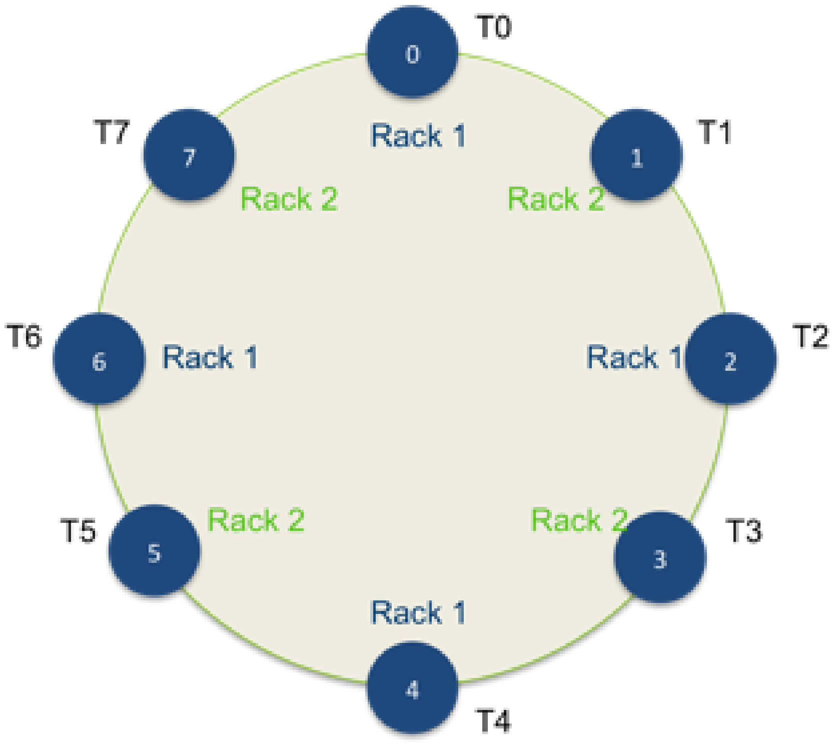
```
{
  "0": {
        "0": 0,
        "1": 28356863910078205288614550619314017621,
        "2": 56713727820156410577229101238628035242,
        "3": 85070591730234615865843651857942052864,
        "4": 113427455640312821154458202477256070485,
        "5": 141784319550391026443072753096570088106
        }
}
```

Calculating Tokens for Multiple Racks in a Single Data Center

If you have multiple racks in single data center, enter the number of nodes in the Token Generating Tool and then assign the tokens to nodes to alternating racks. For example: rack1, rack2, rack3, rack1, rack2, rack3, and so on. As a best practice, each rack should have the same number of nodes so you can alternate the rack assignments. For example:

```
./tokengentool 8
```

The tool displays the token for each node. The image shows the rack assignments:



Assign tokens to nodes in alternating racks

Calculating Tokens for a Multiple Data Center Cluster

In multiple data center deployments, replica placement is calculated per data center using the `NetworkTopologyStrategy`. This strategy`` determines replica placement independently within each data center. The first replica is placed according to the partitioner. Additional replicas in the same data center are determined by walking the ring clockwise until a node in a different rack from the previous replica is found. If no such node exists, additional replicas are placed in the same rack. For more detailed information, see *NetworkTopologyStrategy*.

There are different methods you can use when calculating multiple data center clusters. The important point is that the nodes within each data center manage an equal amount of data. The distribution of the nodes within the cluster is not as important. DataStax recommends using the DataStax Enterprise OpsCenter to rebalance a cluster.

Alternating Token Assignments

Calculate tokens for each data center using the Token Generating Tool and then alternate the token assignments so that the nodes for each data center are evenly dispersed around the ring.

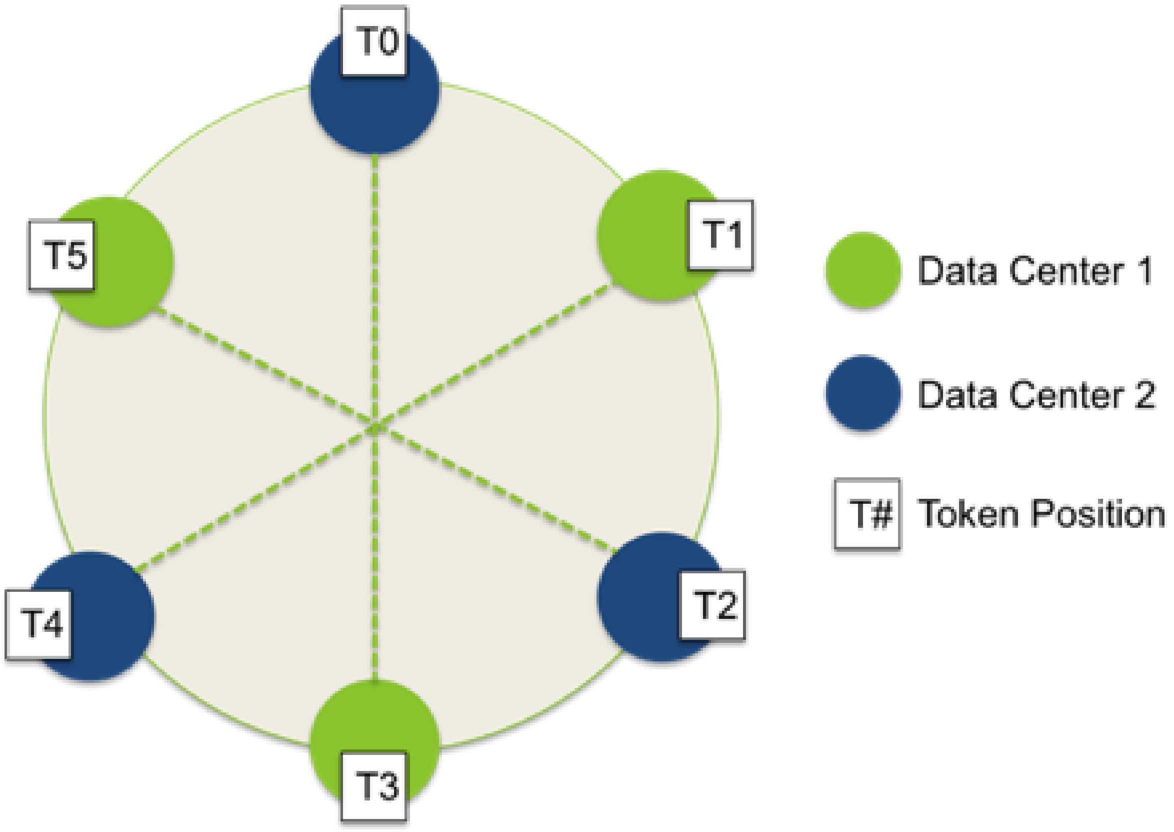
```
./tokengentool 3 3
```

The tool displays the token for each node in each data center:

```
{
  "0": {
      "0": 0,
      "1": 56713727820156410577229101238628035242,
      "2": 113427455640312821154458202477256070485
       },
  "1": {
      "0": 28356863910078205288614550619314017621,
      "1": 85070591730234615865843651857942052863,
      "2": 141784319550391026443072753096570088106
```

```
     }
}
```

The following image shows the token position and data center assignments:



Avoiding Token Collisions

To avoid token collisions offset the values. Although you can increment in values of 1, it is better to use a larger offset value, such as 100, to allow room to replace a dead node.

The following shows an example of a cluster with two 3 node data centers and one 2 node data center.

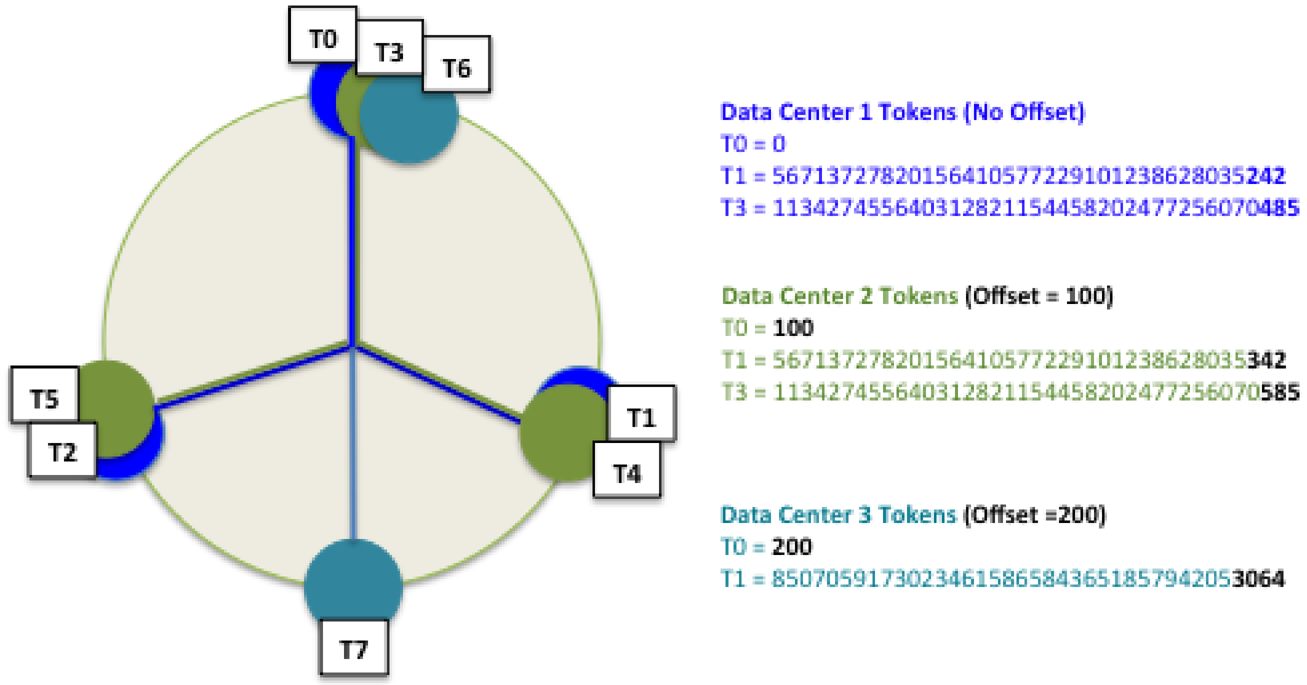
```
./tokengentool 3

   {
    "0": {
        "0": 0,
        "1": 56713727820156410577229101238628035242,
        "2": 113427455640312821154458202477256070485
         }
 }

./tokentool 2

  {
      "0": {
        "0": 0,
          "1": 85070591730234615865843651857942052864
           }
   }
```

The graphic shows the distribution of the nodes with the associated offsets.


T0
T3
T6
T5
T2
T1
T4
T7
Data Center 1 Tokens (No Offset)
T0 = 0
T1 = 56713727820156410577229101238628035242
T3 = 113427455640312821154458202477256070485
Data Center 2 Tokens (Offset = 100)
T0 = 100
T1 = 56713727820156410577229101238628035342
T3 = 113427455640312821154458202477256070585
Data Center 3 Tokens (Offset =200)
T0 = 200
T1 = 85070591730234615865843651857942053064

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.

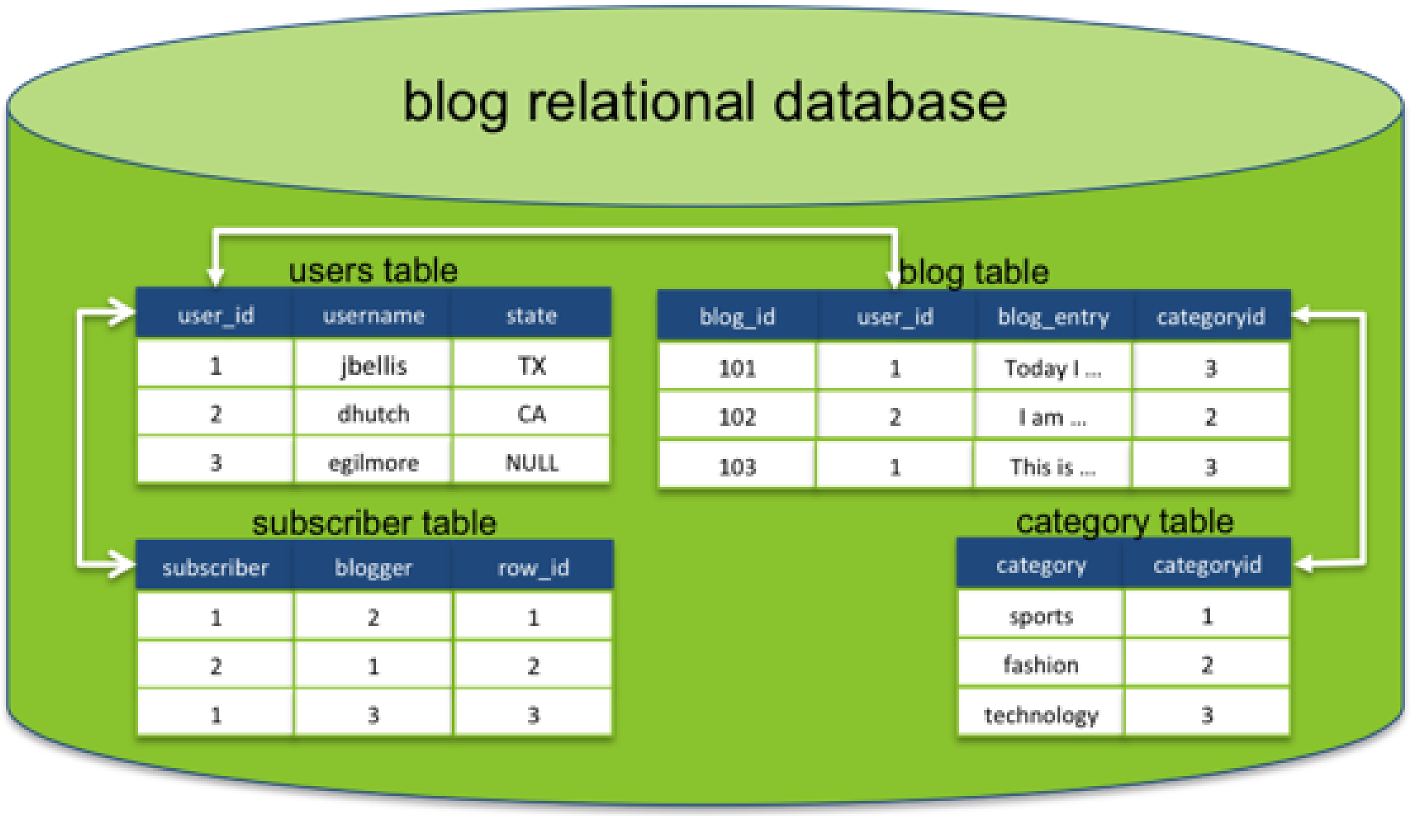
The Cassandra Data Model

For developers new to Cassandra and coming from a relational database background, the data model can be a bit confusing. The following section provides a comparison of the two.

Comparing the Cassandra Data Model to a Relational Database

The Cassandra data model is designed for distributed data on a very large scale. Although it is natural to want to compare the Cassandra data model to a relational database, they are really quite different. In a relational database, data is stored in tables and the tables comprising an application are typically related to each other. Data is usually normalized to reduce redundant entries, and tables are joined on common keys to satisfy a given query.

For example, consider a simple application that allows users to create blog entries. In this application, blog entries are categorized by subject area (sports, fashion, etc.). Users can also choose to subscribe to the blogs of other users. In this example, the user id is the primary key in the *users* table and the foreign key in the *blog* and *subscriber* tables. Likewise, the category id is the primary key of the *category* table and the foreign key in the *blog_entry* table. Using this relational model, SQL queries can perform joins on the various tables to answer questions such as "what users subscribe to my blog" or "show me all of the blog entries about fashion" or "show me the most recent entries for the blogs I subscribe to".



users table

user_id	username	state
1	jbellis	TX
2	dhutch	CA
3	egilmore	NULL

blog table

blog_id	user_id	blog_entry	categoryid
101	1	Today I ...	3
102	2	I am ...	2
103	1	This is ...	3

subscriber table

subscriber	blogger	row_id
1	2	1
2	1	2
1	3	3

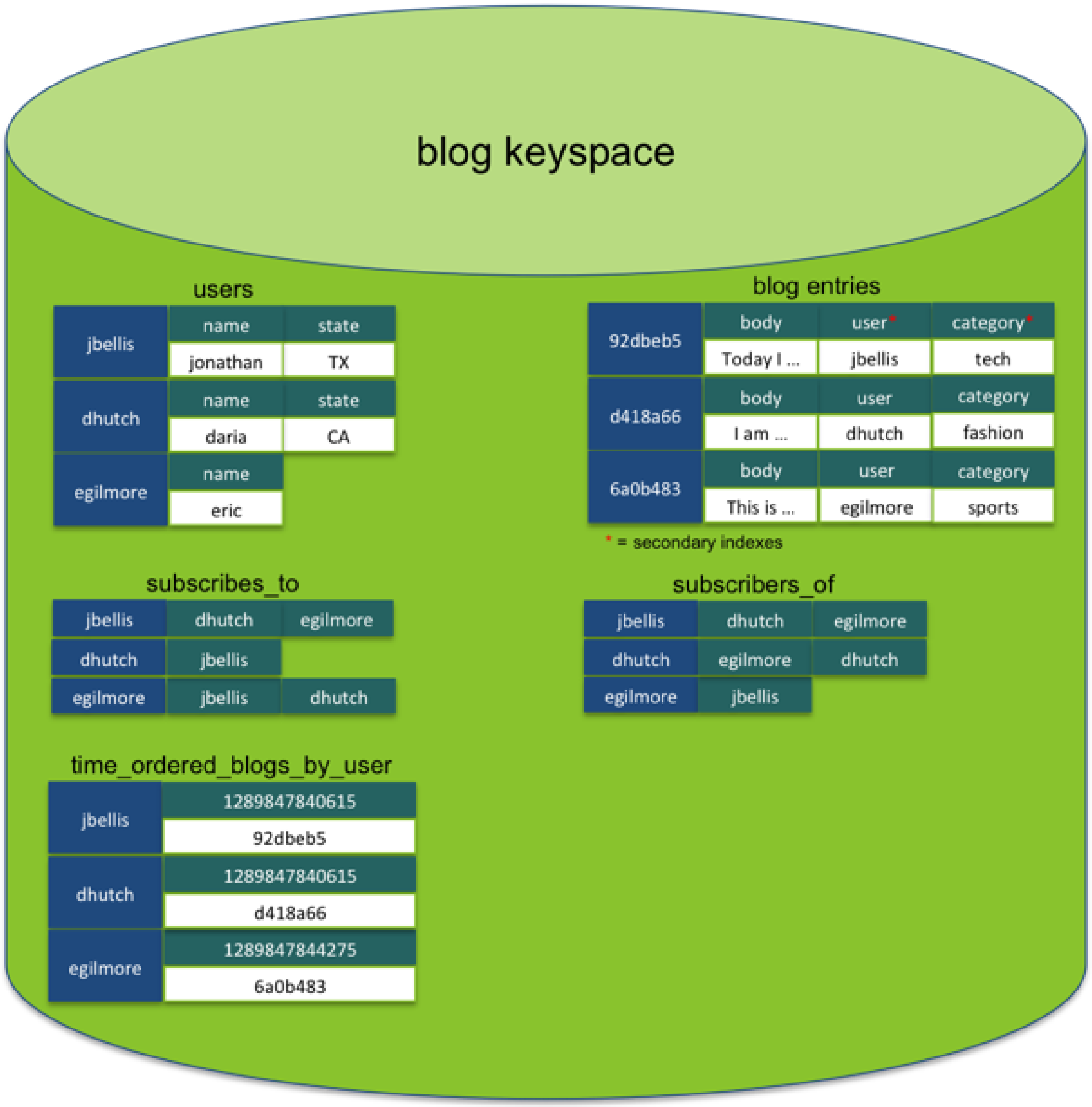
category table

category	categoryid
sports	1
fashion	2
technology	3

In Cassandra, the *keyspace* is the container for your application data, similar to a database or schema in a relational database. Inside the keyspace are one or more *column family* objects, which are analogous to tables. Column families contain *columns*, and a set of related columns is identified by an application-supplied row *key*. Each row in a column family is *not* required to have the same set of columns.

Cassandra does not enforce relationships between column families the way that relational databases do between tables: there are no formal foreign keys in Cassandra, and joining column families at query time is not supported. Each column family has a self-contained set of columns that are intended to be accessed together to satisfy specific queries from your application.

For example, using the blog application example, you might have a column family for user data and blog entries similar to the relational model. Other column families (or secondary indexes) could then be added to support the queries your application needs to perform. For example, to answer the queries "what users subscribe to my blog" or "show me all of the blog entries about fashion" or "show me the most recent entries for the blogs I subscribe to", you would need to design additional column families (or add secondary indexes) to support those queries. Keep in mind that some denormalization of data is usually required.



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 keyspaces. 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.

Defining Keyspaces

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

```
CREATE KEYSPACE keyspace_name WITH
strategy_class = 'SimpleStrategy'
AND strategy_options:replication_factor=2;
```

See *Getting Started with CQL* for more information on DDL commands for Cassandra.

About Column Families

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* -- The typical Cassandra column family design
- *Dynamic column family* -- For use with a custom data type

Column families consists 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.

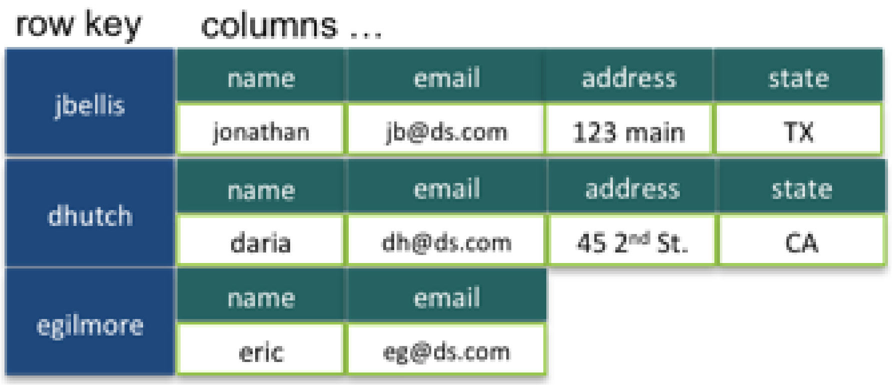
Although column families are very flexible, in practice a column family is not entirely schema-less.

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.

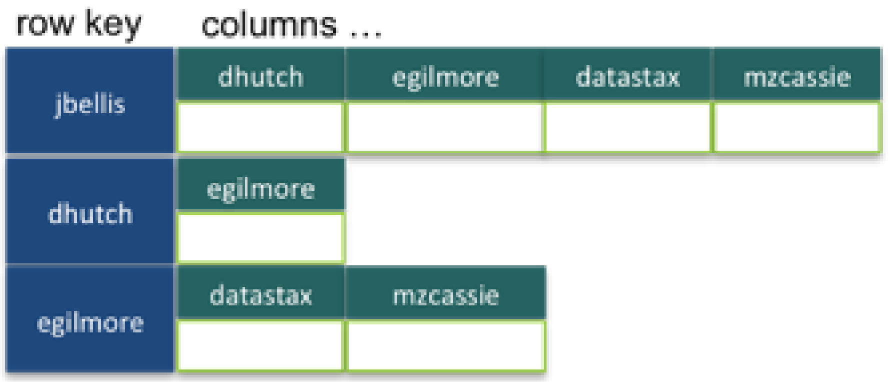
Static Column Families

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.



Dynamic Column Families

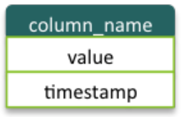
A dynamic column family takes advantage of Cassandra's ability to use arbitrary application-supplied column names to store data. A dynamic column family allows you to pre-compute result sets and store them in a single row for efficient data retrieval. Each row is a snapshot of data meant to satisfy a 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.



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.



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.

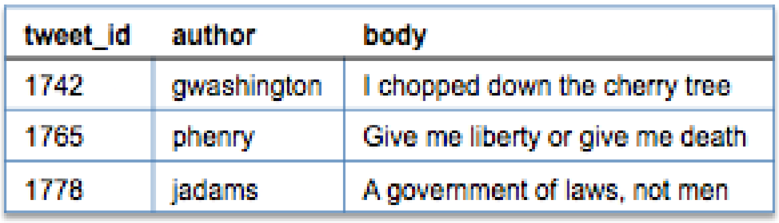
- The timeline table denormalizes the tweets, setting up composite columns by virtue of the composite primary key.

```
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.

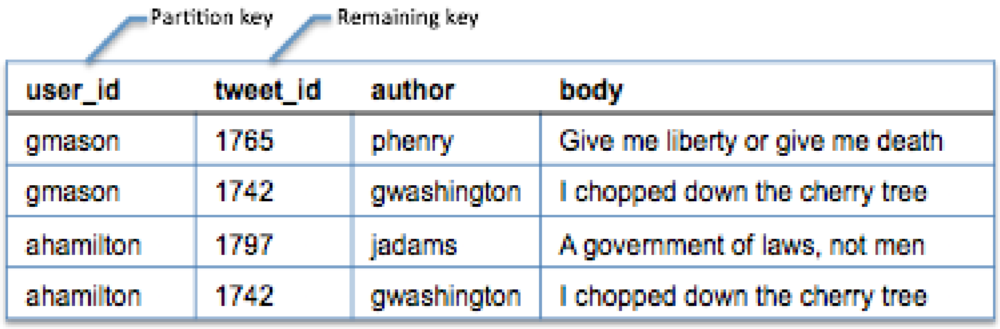
Tweets Table



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

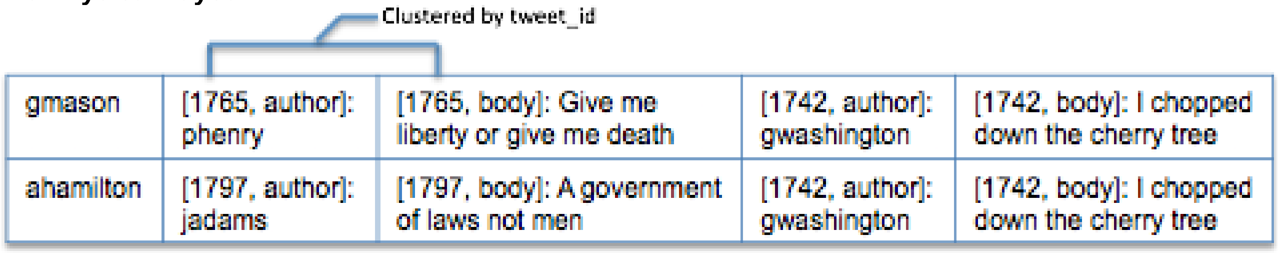

Partition key
Remaining key

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 partition key (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


Clustered by tweet_id

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 ahamiliton 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.

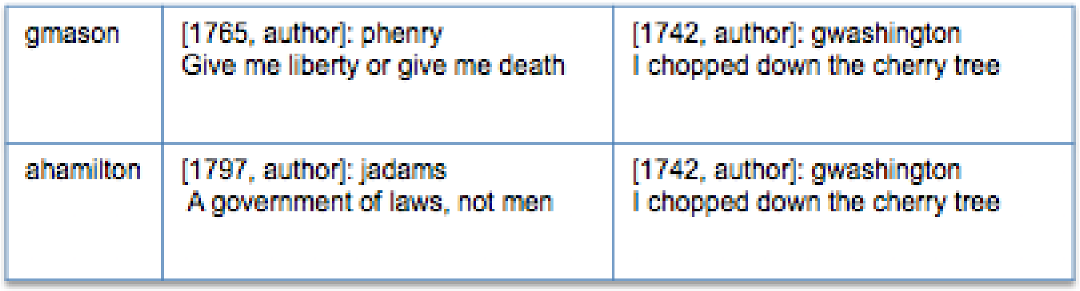
Using the CQL 3 model, you can query a single sequential set of data on disk to get the tweets of a user's followers.

```
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.

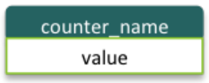
An expiring column has an additional overhead of 8 bytes in memory and on disk (to record the TTL and expiration time) compared to standard columns.

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; they will be allowed to mix with normal columns in a future release.

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.

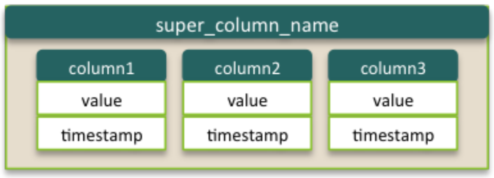


Internally, the structure of a counter column is a bit more complex. Cassandra tracks the distributed state of the counter as well as a server-generated timestamp upon deletion of a counter column. For this reason, it is important that all nodes in your cluster have their clocks synchronized using network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

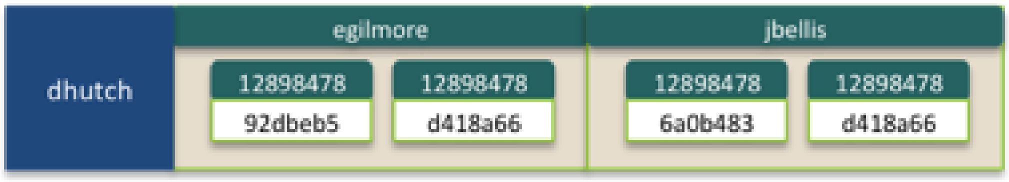
A counter can be read or written at any of the available *consistency levels*. However, it's important to understand that unlike normal columns, a write to a counter requires a read in the background to ensure that distributed counter values remain consistent across replicas. If you write at a consistency 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

A Cassandra column family can contain either regular columns or *super columns*, which adds another level of nesting to the regular column family structure. Super columns are comprised of a (super) column name and an ordered map of sub-columns. A super column can specify a comparator on both the super column name as well as on the sub-column names.



A super column is a way to group multiple columns based on a common lookup value. The primary use case for super columns is to denormalize multiple rows from other column families into a single row, allowing for materialized view data retrieval. For example, suppose you wanted to create a materialized view of blog entries for the bloggers that a user follows.



One limitation of super columns is that all sub-columns of a super column must be deserialized in order to read a single sub-column value, and you cannot create secondary indexes on the sub-columns of a super column. Therefore, the use of super columns is best suited for use cases where the number of sub-columns is a relatively small number.

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 (`BytesType`). 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
BytesType	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
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 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. Using an ordered partitioner does allow for range queries over rows, but is not recommended because of the difficulty in maintaining even data distribution across nodes. See *About Data Partitioning in Cassandra* for more information.

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 supports secondary indexes of the type `KEYS` (similar to a hash index).

Secondary indexes allow for efficient querying by specific values using equality predicates (where column *x* = value *y*). Also, queries on indexed values can apply additional filters to the result set for values of other columns.

Cassandra's built-in secondary indexes are best for cases when many rows 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. On the other hand, if you wanted to look up users by their email address (a value that is typically unique for each user), it may be more efficient to manually maintain a dynamic column family as a form of an "index". Even for columns containing unique data, it is often 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.

Another advantage of secondary indexes is the operational ease of populating and maintaining the index. When you create a secondary index on an existing column, it indexes the existing data in the background. 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.

Building and Using Secondary Indexes

You can specify the `KEYS` index type when creating a column definition, or you can add it later to index an existing column. Secondary indexes are built in the background automatically, without blocking reads or writes.

For example, in the Cassandra CLI, you can create a secondary index on a column when defining a column family (note the `index_type:KEYS` specification for the `state` and `birth_year` columns):

```
[default@demo] create column family users with comparator=UTF8Type
... and column_metadata=[{column_name: full_name, validation_class: UTF8Type},
... {column_name: email, validation_class: UTF8Type},
... {column_name: birth_year, validation_class: LongType, index_type: KEYS},
... {column_name: state, validation_class:  UTF8Type, index_type: KEYS}];
```

Or you can add an index to an existing column family:

```
[default@demo] update column family users with comparator=UTF8Type
... and column_metadata=[{column_name: full_name, validation_class: UTF8Type},
... {column_name: email, validation_class: UTF8Type},
... {column_name: birth_year, validation_class: LongType, index_type: KEYS},
... {column_name: state, validation_class:  UTF8Type, index_type: KEYS}];
```

Because of the secondary index created for `state`, its values can then be queried directly for users who live in a given state. For example:

```
[default@demo] get users where state = 'TX';
```

Planning Your Data Model

Planning a data model in Cassandra has different design considerations than one may be used to from 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 backwards 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.

For example, start with listing the all of 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, if you need events in chronological order, or if you only care about the last 6 months worth of data, those would be factors in your data model design for Cassandra.

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 Types for Column Names

The UUID comparator type (universally unique id) is used to avoid collisions in column names. 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 and Accessing Data in Cassandra

This section provides information about accessing and managing data in Cassandra via a client application. Cassandra offers a number of client utilities and application programming interfaces (APIs) that can be used for developing applications that utilize Cassandra for data storage and retrieval.

About Writes in Cassandra

Cassandra is optimized for very fast and highly available data writing. 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 usually are not performant on writes.

In contrast, Cassandra is optimized for write throughput. 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 row or 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).
3. **The row key for a deleted row may still appear in range query results.** When you delete a row in Cassandra, it marks all columns for that row key with a tombstone. Until those tombstones are cleared by compaction, you have an empty row key (a row that contains no columns). These deleted keys can show up in results of `get_range_slices()` calls. If your client application performs range queries on rows, you may want to have if filter out row keys that return empty column lists.

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 at the time the write occurs, a corresponding live replica will store a hint. The hint consists of location information (the replica node and row key that require a replay), as well as the actual data being written. There is minimal overhead to storing hints on replica nodes that already own the written row, since the data being written is already accounted for by the usual write process. The hint data itself is relatively small in comparison to most data rows.

If all replicas for the affected row key are down, it is still possible for a write to succeed if using a *write consistency* level of ANY. Under this scenario, the hint and written data are stored on the coordinator node, but will not be available to reads until the hint gets written to the actual replicas that own the row. The ANY consistency level provides absolute write availability at the cost of consistency, as there is no guarantee as to when written data will be available to reads (depending how long the replicas are down). Using the ANY consistency level can also potentially increase load on the cluster, as coordinator nodes must temporarily store extra rows whenever a replica is not available to accept a write.

Note

By default, hints are only saved for one hour before they are dropped. If all replicas are down at the time of write, and they all remain down for longer than the configured time of *max_hint_window_in_ms*, you could potentially lose a write made at consistency level ANY.

Hinted handoff does not count towards any other consistency level besides ANY. For example, if using a consistency level of ONE and all replicas for the written row are down, the write will fail regardless of whether a hint is written or not.

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.

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, large numbers of schema changes can simultaneously take place 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 another 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 instanteous 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):

```
(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 *hinted handoff* 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 *all* 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 once a quorum of replicas has responded.
LOCAL_QUORUM	Returns the record with the most recent timestamp once 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 once a quorum of replicas in each data center of the cluster has responded.
ALL	Returns the record with the most recent timestamp once all replicas have responded. The read operation will fail if a replica does not respond.

Note

LOCAL_QUORUM and EACH_QUORUM are designed for use in multi-data center clusters using a rack-aware replica placement strategy (such as `NetworkTopologyStrategy`) and a properly configured snitch.

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 Multi-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. The consistency level `LOCAL_QUORUM` is specifically designed for doing quorum reads and writes in multi data center clusters.

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 WHERE state='TX' USING CONSISTENCY QUORUM;
```

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* See *About Hinted Handoff Writes* for more information on how hinted handoff works.

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

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.

Note

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

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

The Python driver includes a command-line interface, `cql.sh`. See *Getting Started with CQL*.

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, SuperColumn support, 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 = 'org.apache.cassandra.locator.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

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;
```

Getting Started with CQL

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.

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

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

```
./cqlsh
```

Starting CQLsh Using the CQL 3 Specification

To start CQLsh on the using the CQL 3 Specification from the Cassandra bin directory on Linux, for example:

```
./cqlsh --cql3
```

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

```
./cqlsh 1.2.3.4 9160 --cql3
```

To exit CQLsh:

```
cqlsh> exit
```

Using CQL Commands

Commands in CQL combine SQL-like syntax that maps to Cassandra concepts and operations. CQL 3 supports tab completion.

This section presents an overview of the CQL 3 commands. These commands are described in detail in the *CQL Reference*. For a description of CQL 2, see the CQL reference for Cassandra 1.0.

Creating a Keyspace

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

```
cqlsh> CREATE KEYSPACE demodb
         WITH strategy_class = 'org.apache.cassandra.locator.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.

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.

```
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*.

Creating a Column Family

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

```
cqlsh> 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.

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

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

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> 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 CQL column family:

```
cqlsh> 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> 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> 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> 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> 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 a shown:

```
cqlsh> INSERT INTO users
         (user_name, password)
         VALUES ('cbrown', 'ch@ngem4a') USING TTL 86400;

cqlsh> 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> 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> CREATE INDEX state_key ON users (state);
cqlsh> 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> 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> DELETE session_token FROM users where pk = 'jsmith';
cqlsh> DELETE FROM users where pk = 'jsmith';
```

You can retrieve keys for deleted rows using `SELECT` statements and other "get" operations if "range ghosts" are present. Deleted values, including range ghosts, are removed completely by the first compaction following deletion.

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> DROP TABLE users;
cqlsh> DROP KEYSPACE demodb;
```

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. This file is located in the following directories:

- **Packaged installations:** `/etc/cassandra/conf`
- **binary installations:** `<install_location>/conf`

Note

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

Option	Option
authenticator	memtable_flush_queue_size
authority	memtable_flush_writers
broadcast_address	memtable_total_space_in_mb
cluster_name	multithreaded_compaction
column_index_size_in_kb	partitioner
commitlog_directory	phi_convict_threshold
commitlog_sync	reduce_cache_capacity_to
commitlog_sync_period_in_ms	reduce_cache_sizes_at
commitlog_total_space_in_mb	request_scheduler
compaction_preheat_key_cache	request_scheduler_id
compaction_throughput_mb_per_sec	row_cache_provider
concurrent_compactors	row_cache_size_in_mb
concurrent_reads	row_cache_save_period
concurrent_writes	rpc_address
data_file_directories	rpc_keepalive
dynamic_snitch	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
endpoint_snitch	rpc_send_buff_size_in_bytes
flush_largest_memtables_at	rpc_server_type
hinted_handoff_enabled	rpc_timeout_in_ms
hinted_handoff_throttle_delay_in_ms	saved_caches_directory
in_memory_compaction_limit_in_mb	seeds
incremental_backups	seed_provider
index_interval	sliced_buffer_size_in_kb
initial_token	snapshot_before_compaction
internode_encryption	storage_port
key_cache_size_in_mb	stream_throughput_outbound_megabits_per_sec
key_cache_save_period	thrift_framed_transport_size_in_mb
keystore	thrift_max_message_length_in_mb

keystore_password	truststore
listen_address	truststore_password
max_hint_window_in_ms	

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, and should be evaluated and changed as needed before starting a node for the first time. These properties control how a node is configured within a cluster in regards to inter-node communication, data partitioning, and replica placement.

broadcast_address

(Default: Same as listen_address) If your Cassandra cluster is deployed across multiple Amazon EC2 regions (and you are using the *EC2MultiRegionSnitch*), you should set broadcast_address to *public* IP address of the node (and listen_address to the *private* IP). If not declared, defaults to the same address as specified for listen_address.

cluster_name

(Default: `Test Cluster`) The name of the cluster. All nodes participating in a cluster must have the same value.

commitlog_directory

(Default: `/var/lib/cassandra/commitlog`) The directory where the commit log will be 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) will be 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. The initial token can be left unset when introducing a new node to an established cluster. Otherwise, the token value depends on the partitioner you use. With the random partitioner, this value will be a number between 0 and 2^{127} -1. With the byte order preserving partitioner, this value is byte array of hex values based on your actual row key values. With the order preserving and collated order preserving partitioners, this value is a UTF-8 string based on your actual row key values. See *Generating Tokens* for more information.

listen_address

(Default: `localhost`) The IP address or hostname that other Cassandra nodes will use to connect to this node. If left blank, you must have hostname resolution correctly configured on all nodes in your cluster so that the hostname resolves to the correct IP address for this node (using `/etc/hostname`, `/etc/hosts` or DNS).

partitioner

(Default: `org.apache.cassandra.dht.RandomPartitioner`) Sets the partitioning method used when assigning a row key to a particular node (also see initial_token). Allowed values are:

- 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 remote procedure calls (client connections). To listen on all configured interfaces, set to 0.0.0.0. If left blank, you must have hostname resolution correctly configured on all nodes in your cluster so that the hostname resolves to the correct IP address for this node (using `/etc/hostname`, `/etc/hosts` or DNS). Allowed Values: An IP address, hostname, or leave unset to resolve the address using the hostname configuration of the node.

rpc_port

(Default: `9160`) The port for remote procedure calls (client connections) and the Thrift service.

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`) The seed provider is a pluggable interface for providing a list of seed nodes. The default seed provider requires a comma-delimited list of seeds.

seeds

(Default: `127.0.0.1`) When a node joins a cluster, it contacts the seed nodes to determine the ring topology and obtain gossip information about the other nodes in the cluster. Every node in the cluster should have the same list of seeds, specified as a comma-delimited list of IP addresses. In multiple data center clusters, the seed 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`) The `endpoint_snitch` has two functions: It informs Cassandra about your network topology so that requests are routed efficiently and allows Cassandra to distribute replicas in the cluster to prevent correlated failures. It does this by grouping machines into data centers and racks. Cassandra does its best to avoid having more than one replica on the same rack (which may not actually be a physical location).

The `endpoint_snitch` sets which snitch is used for locating nodes and routing requests. In deployments with rack-aware replication placement strategies, use either `RackInferringSnitch`, `PropertyFileSnitch`, or `EC2Snitch` (if on Amazon EC2). Has a dependency on the replica *placement_strategy*, which is defined on a keyspace. The `PropertyFileSnitch` also requires a `cassandra-topology.properties` configuration file. Snitches included with Cassandra are:

- org.apache.cassandra.locator.SimpleSnitch
- org.apache.cassandra.locator.RackInferringSnitch
- org.apache.cassandra.locator.PropertyFileSnitch
- org.apache.cassandra.locator.Ec2Snitch

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.

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. The recommended value is 10% of heap.

key_cache_save_period

(Default: `14400`- 4 hours) Duration in seconds that keys are saved in cache. Caches are saved to *saved_caches_directory*.

row_cache_size_in_mb

(Default: `0`) A global cache setting for column families. The recommended value is 10% of heap.

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.

Option	Description
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.IRowCacheProvider`.
ConcurrentLinkedHashCacheProvider	Rows are cached using the JVM heap, providing the same row cache behavior as Cassandra versions prior to 0.8.

`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`) Column indexes are added to a row after the data reaches this size. This usually happens if there are a large number of columns in a row or the column values themselves are large. If you consistently read only a few columns from each row, this should be kept small as it denotes how much of the row data must be deserialized to read the column.

commitlog_sync

(Default: `periodic`) The method that Cassandra uses to acknowledge writes. The default mode of periodic is used in conjunction with commitlog_sync_period_in_ms to control how often the commit log is synchronized to disk. Periodic syncs are acknowledged immediately. In batch mode, writes are not acknowledged until fsynced to disk. Cassandra waits the configured number of milliseconds for other writes before performing a sync. Allowed Values are `periodic` or `batch`.

commitlog_sync_period_in_ms

(Default: `10000 - 10` seconds) Determines how often (in milliseconds) to send the commit log to disk when commitlog_sync is set to `periodic` mode.

commitlog_total_space_in_mb

(Default: `4096`) When the commitlog size on a node exceeds this threshold, Cassandra flushes memtables to disk for the oldest commitlog segments, thus allowing those log segments to be removed. This reduces the amount of data to replay on startup, and prevents infrequently-updated column families from indefinitely keeping commit log segments. This replaces the per-column family storage setting `memtable_flush_after_mins`.

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.

concurrent_reads

(Default: `32`) For workloads with more data than can fit in memory, the bottleneck will be reads that need to fetch 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 almost never 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 is exceed this percentage, the largest memtables are flushed to disk in order to free memory. This parameter serves as more of an emergency measure for preventing sudden out-of-memory (OOM) errors rather than a strategic tuning mechanism. It is most effective under light to moderate load, or read-heavy workloads. A value of `0.75` means flush memtables when Java heap usage is above 75 percent total heap size. Set to `1.0` to disable this feature.

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`) Each SSTable has an index file containing row keys and the position at which that row starts in the data file. At startup, Cassandra reads a sample of that index into memory. By default 1 row key out of every 128 is sampled. To find a row, Cassandra performs a binary search on the sample, then does just one disk read of the index block corresponding to the closest sampled entry. The larger the sampling, the more effective the index is (at the cost of memory usage). A smaller value for this property results in a larger, more effective index. Generally, a value between 128 and 512 in combination with a large column family key cache offers the best trade off between memory usage and performance. You may want to increase the sample size if you have small rows, thus decreasing the index size and memory usage. 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, this should be 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 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 this value for better flush performance (see data_file_directories).

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`.

multithreaded_compaction

(Default: `false`) When set to `true`, each compaction operation will use one thread per SSTable being merged in addition to one thread per core. This is typically only useful on nodes with SSD hardware. With regular disks, the goal is to limit the disk I/O for compaction (see compaction_throughput_mb_per_sec).

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`.

sliced_buffer_size_in_kb

(Default: `64`) The buffer size to use for reading contiguous columns. This should match the size of the columns typically retrieved using query operations involving a slice predicate.

stream_throughput_outbound_megabits_per_sec

(Default: `400`) Throttles all outbound streaming file transfers on a node to the specified throughput in Mb per second. Cassandra does mostly sequential I/O when streaming data during bootstrap or repair, which can lead to saturating the network connection and degrading client performance.

Remote Procedure Call 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 only applies to client requests, not inter-node communication. Useful for throttling client requests in implementations that have multiple keyspaces. Allowed Values are:

- `org.apache.cassandra.scheduler.NoScheduler`
- `org.apache.cassandra.scheduler.RoundRobinScheduler`
- A Java class that implements the `RequestScheduler` interface

  If using the `RoundRobinScheduler`, there are additional request_scheduler_options properties.

request_scheduler_id

(Default: `keyspace`) An identifier on which to perform request scheduling. Currently the only valid option is `keyspace`.

request_scheduler_options

Contains a list of additional properties that define configuration options for request_scheduler. `NoScheduler` does not have any options. `RoundRobinScheduler` has the following additional configuration properties: throttle_limit, default_weight, weights.

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 ((concurrent_reads + concurrent_writes) * 2).

default_weight

(Default: `1`) The default weight controls how many requests are handled during each turn of the RoundRobin.

weights

Allows control of weight per keyspace during each turn of the RoundRobin. If not set, each keyspace uses the default_weight. Takes a list of 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 in bytes 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.

rpc_server_type

(Default: `sync`) Cassandra provides three options for the RPC server. The default is `sync` because `hsha` is about 30% slower on Windows. On Linux, `sync` and `hsha` performance is about the same with `hsha` using less memory.

- `sync` - (default) One connection per thread in the RPC pool. For a very large number of clients, memory will be your limiting factor; on a 64 bit JVM, 128KB is the minimum stack size per thread. Connection pooling is very, very 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. Do not use.

thrift_framed_transport_size_in_mb

Deprecated, use thrift_max_message_length_in_mb.

thrift_max_message_length_in_mb

(Default: `16`) The maximum length of a Thrift message in megabytes, including all fields and internal Thrift overhead.

Internode Communication and Fault Detection Properties

dynamic_snitch

(Default: `true`) When set to true, enables the dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes.

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 would continue to prefer the static snitch values until the node response time was 20 percent 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 configured snitch). Having requests consistently routed to a given replica can help keep a working set of data *hot* when *read repair* is less than 100% or disabled.

dynamic_snitch_reset_interval_in_ms

(Default: `600000`) Time interval in milliseconds to reset all node scores (allowing 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.

hinted_handoff_throttle_delay_in_ms

(Default: `50`) When a node detects that a node for which it is holding hints has recovered, it begins sending the hints to that node. This specifies a sleep interval (in milliseconds) after delivering each row or row fragment in an effort to throttle traffic to the recovered 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, hints are dropped. This 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.

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

incremental_backups

(Default: `false`) Backs up data updated since the last snapshot was taken. When enabled, each time an SSTable is flushed, a hard link is copied into a `/backups` subdirectory of the keyspace data directory.

snapshot_before_compaction

(Default: `false`) Defines whether or not to take a snapshot before each compaction. Be careful using this option, since Cassandra does not clean up older snapshots automatically. This can be useful to back up data when there is a data format change.

Security Properties

authenticator

(Default: `org.apache.cassandra.auth.AllowAllAuthenticator`) The default value disables authentication. Basic authentication is provided using the SimpleAuthenticator, which uses the `access.properties` and `password.properties` configuration files to configure authentication privileges. Allowed values are: * org.apache.cassandra.auth.AllowAllAuthenticator * org.apache.cassandra.auth.SimpleAuthenticator * A Java class that implements the IAuthenticator interface

Note

The `SimpleAuthenticator` and `SimpleAuthority` classes have been moved to the example directory of the Apache Cassandra project repository as of release 1.0. They are no longer available in the packaged and binary distributions. They never provided actual security, and in their current state are only meant as examples.

authority

(Default: `org.apache.cassandra.auth.AllowAllAuthority`) The default value disables user access control (all users can access all resources). To control read/write permissions to keyspaces and column families, use the `SimpleAuthority`, which uses the `access.properties` configuration file to define per-user access. Allowed values are: * org.apache.cassandra.auth.AllowAllAuthority * org.apache.cassandra.auth.SimpleAuthority * A Java class that implements the IAuthority interface

internode_encryption

(Default: `none`) Enables or disables encryption of inter-node communication using `TLS_RSA_WITH_AES_128_CBC_SHA` as the cipher suite for authentication, key exchange and encryption of the actual data transfers. To encrypt all inter-node communications, set to `all`. You must also generate keys and provide the appropriate key and trust store locations and passwords.

keystore

(Default: `conf/.keystore`) The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), 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 a truststore containing the trusted certificate used to authenticate remote servers.

truststore_password

(Default: `cassandra`) Password for the truststore.

Keyspace and Column Family Storage Configuration

Many aspects of storage configuration are set on a per-keyspace or per-column family basis. These attributes are stored in the `system` keyspace within Cassandra can be manipulated programmatically, but in most cases the practical method for defining keyspace and column family attributes is to use the Cassandra CQL interface.

Note

The attribute names documented in this section are the names as they are stored in the `system` keyspace within Cassandra. You can use a client application, such as Cassandra CQL, to set the keyspace attributes and some of the column family storage attributes. There may be slight differences in how these attributes are named depending on how they are implemented in the client.

Keyspace Attributes

A keyspace must have a user-defined name and a replica placement strategy. It also has replication strategy options, which is a container attribute for replication factor or the number of replicas per data center.

Option	Default Value
name	N/A (A user-defined value is required)
placement_strategy	org.apache.cassandra.locator.SimpleStrategy
strategy_options	N/A (container attribute)

name

Required. The name for the keyspace.

placement_strategy

strategy_options

Required. Determines how replicas for a keyspace will be distributed among nodes in the ring.

Allowed values are:

- `org.apache.cassandra.locator.SimpleStrategy`
- `org.apache.cassandra.locator.NetworkTopologyStrategy`

These options are described in detail in the *replication* section.

Note

NetworkTopologyStrategy requires a properly configured snitch to be able to determine rack and data center locations of a node (see *endpoint_snitch*).

strategy_options

Specifies configuration options for the chosen replication strategy.

For `SimpleStrategy`, it specifies `replication_factor` in the format of `replication_factor`:*number_of_replicas*.

For `NetworkTopologyStrategy`, it specifies the number of replicas per data center in a comma separated list of *datacenter_name*:*number_of_replicas*. Note that what you specify for *datacenter_name* depends on the cluster-configured *snitch* you are using. There is a correlation between the data center name defined in the keyspace `strategy_options` and the data center name as recognized by the snitch you are using. The *nodetool ring* command prints out data center names and rack locations of your nodes if you are not sure what they are.

Choosing Keyspace Replication Options describes how to configure replication strategy and strategy options for a cluster. Set strategy options using CQL:

```
cqlsh> CREATE KEYSPACE test WITH
         strategy_class = 'NetworkTopologyStrategy'
         AND strategy_options:replication_factor=2;
```

Column Family Attributes

The following attributes can be declared per column family.

Option	Option
bloom_filter_fp_chance	dc_local_read_repair_chance
caching	gc_grace_seconds
column_metadata	key_validation_class
column_type	max_compaction_threshold [1]
comment	min_compaction_threshold
compaction_strategy	memtable_flush_after_mins [1]
compaction_strategy_options	memtable_operations_in_millions [1]
comparator	memtable_throughput_in_mb [1]
compare_subcolumns_with	name
compression_options	read_repair_chance
default_validation_class	replicate_on_write

bloom_filter_fp_chance

(Default: ~ `0.0007`) Desired false-positive probability for SSTable Bloom filters. Valid values are 0.0001 to 1.0. At 1.0 the Bloom filter is effectively disabled; this is reasonable if you are using LeveledCompactionStrategy and not querying for non-existent rows. If you have many keys per node and are worried about Bloom filter memory usage, a reasonable first step is to try 0.01, if that is still too large then 0.1. For information about how Cassandra uses Bloom filters, see *About Reads in Cassandra* and *About Writes in Cassandra*.

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.

1 Ignored in Cassandra 1.1 but can still be declared (for backward compatibility).

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
name	Binds a validation_class and (optionally) an index to a column.
validation_class	Type used to check the column value.
index_name	Name for the secondary index.
index_type	Type of index. Currently the only supported value is KEYS.

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.

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 limits the size of SSTables to a small file size (5 MB by default) so that SSTables do not continue to grow in size with each successive compaction. 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.

Option	Default Value	Description
sstable_size_in_mb	5	Sets the file size for leveled SSTables. A compaction is triggered when unleveled SSTables (newly flushed SSTable files in Level 0) exceeds 4 `* sstable_size_in_mb`.

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: `BytesType`) 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: `BytesType`) Required when column_type is "Super". Same as comparator but for the sub-columns of a SuperColumn.

For attributes of columns, see column_metadata.

compression_options

(Default: N/A - container attribute) This is a container attribute for setting compression options on a column family. It contains the following options:

Option	Description
sstable_compression	Specifies the compression algorithm to use when compressing SSTable files. Cassandra supports two built-in compression classes: `SnappyCompressor` (Snappy compression library) and `DeflateCompressor` (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 `org.apache.cassandra.io.compress.ICompressor` interface.
chunk_length_kb	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.

Setting and updating compression options using CQL requires a slightly different command syntax than other attributes and a different name. Instead of compression_options, use compression_parameters. For example:

```
ALTER TABLE addamsFamily WITH
  compression_parameters:sstable_compression = 'DeflateCompressor'
  AND compression_parameters:chunk_length_kb = 64;
```

default_validation_class

(Default: N/A) Defines the data type used to validate column values. There are several built-in *column validators* available.

dc_local_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

Ignored in Cassandra 1.1 and later. In earlier versions, sets the maximum number of SSTables to allow in a minor compaction when `compaction_strategy=SizeTieredCompactionStrategy`. Obsolete as of Cassandra 0.8 with the addition of compaction throttling (see `cassandra.yaml` parameter *compaction_throughput_mb_per_sec*).

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. Setting this to 0 disables minor compactions.

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) Specifies the probability with which read repairs should be invoked on non-quorum reads. The value must be between 0 and 1. A value of 0.1 means that a read repair is performed 10% of the time and a value of 1 means that a read repair is performed 100% of the time. Lower values improve read throughput, but increase the chances of stale values when not using a strong *consistency level*.

replicate_on_write

(Default: `true`) When set to `true`, replicates writes to all affected replicas regardless of the consistency level specified by the client for a write request. For counter column families, this should always be set to `true`.

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 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

The `SimpleAuthenticator` and `SimpleAuthority` classes have been moved to the example directory of the Apache Cassandra project repository as of release 1.0. They are no longer available in the packaged and binary distributions. They never provided actual security, and in their current state are only meant as examples.

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.

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

In some situations, the output provided by Cassandra's JMX MBeans and the *nodetool* utility are not enough to diagnose issues. If you find yourself in a place where you need more information about the runtime behavior of a specific Cassandra node, you can increase the logging levels to get more diagnostic information on specific portions of the system.

Cassandra uses a SLF4J to provide logging functionality; by default a log4j backend is used. Many of the core Cassandra classes have varying levels of logging output available which can be increased in one of two ways:

1. Updating the `log4j-server.properties` file
2. Through JMX

Logging Levels via the Properties File

To update the via properties file, edit `conf/log4j-server.properties` to include the following two lines (warning - this will generate *a lot* of logging output on even a moderately trafficked cluster):

```
log4j.logger.org.apache.cassandra.db=DEBUG
log4j.logger.org.apache.cassandra.service.StorageProxy=DEBUG
```

This will apply the `DEBUG` log level to all the classes in `org.apache.cassandra.db` package and below as well as to the StorageProxy class directly. Cassandra checks the log4j configuration file every ten seconds, and applies changes without needing a restart.

Note that by default, logging will go to `/var/log/cassandra/system.log`; the location of this log file can be changed as well - just update the `log4j.appender.R.File` path to where you would like the log file to exist, and ensure that the directory exists and is writable by the process running Cassandra.

Additionally, the default configuration will roll the log file once the size exceeds 20MB and will keep up to 50 backups. These values may be changed as well to meet your requirements.

Logging Levels via JMX

To change logging levels via JMX, bring up the JConsole tool and attach it to the CassandraDaemon process. Locate the StorageService MBean and find `setLog4jLevel` under the Operations list.

This operation takes two arguments - a class qualifier and a logging level. The class qualifier can either be a full class name or any portion of a package name, similar to the `log4j-server.properties` configuration above except without the initial `log4j.logger` appender assignment. The level must be one of the standard logging levels in use by Log4j.

In keeping with our initial example, to adjust the logging output of StorageProxy to `DEBUG`, the first argument would be `org.apache.cassandra.service.StorageProxy`, and the second one `DEBUG`. On a system with traffic, you should see the effects of this change immediately.

Operations

Monitoring a Cassandra Cluster

Understanding the performance characteristics of your Cassandra cluster is critical to diagnosing issues and planning capacity.

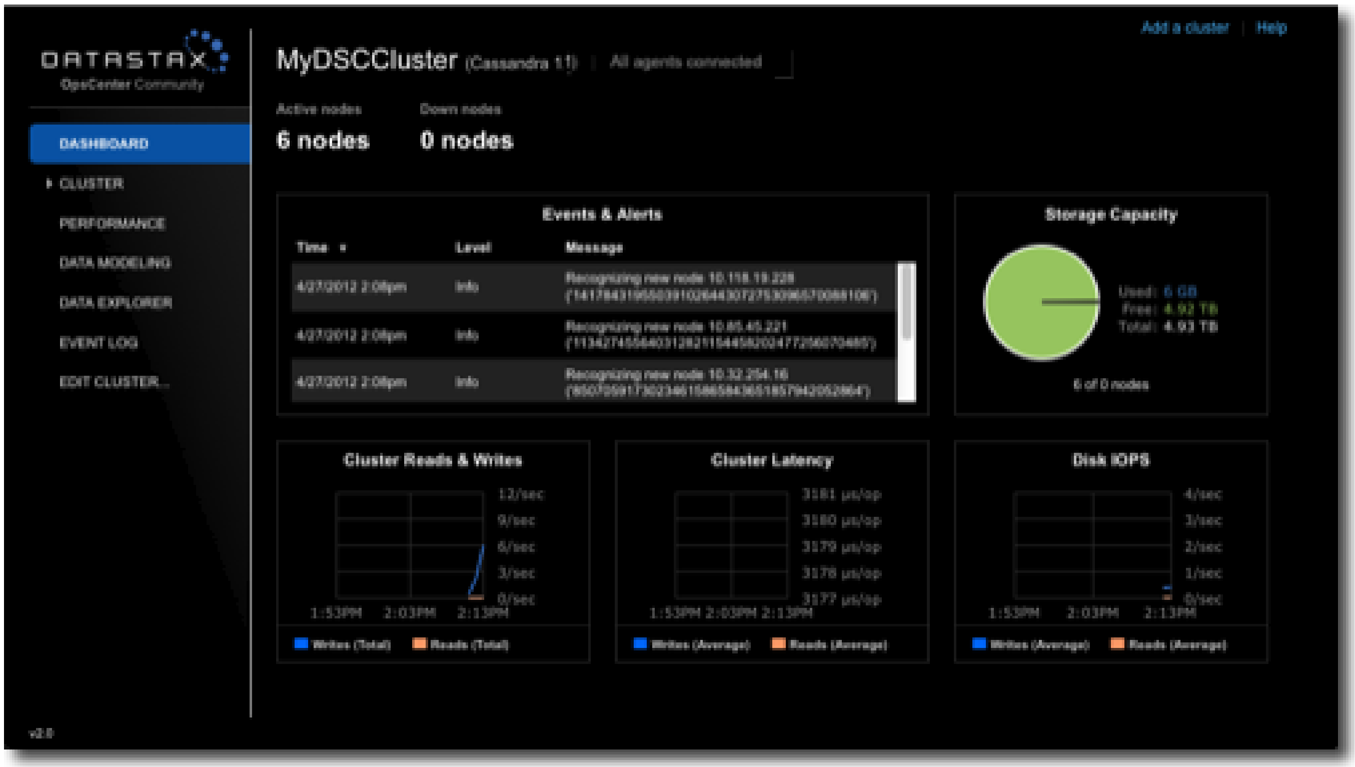
Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed 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

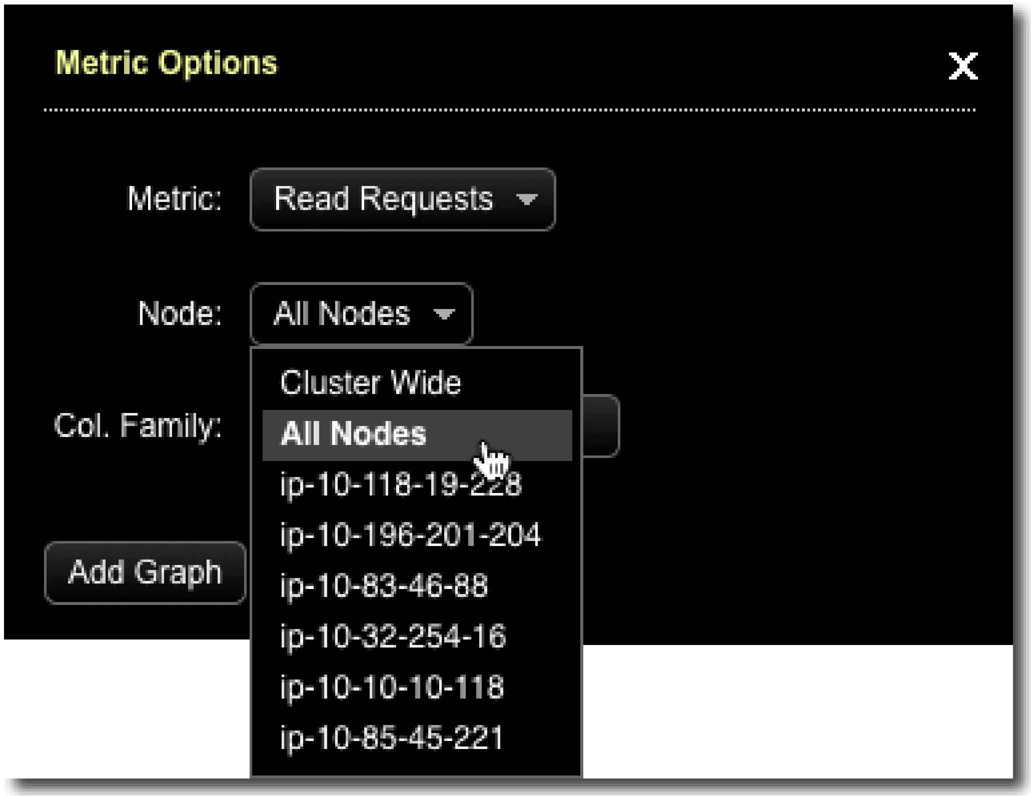
DataStax OpsCenter is a graphical user interface for monitoring and administering all nodes in a Cassandra cluster from one centralized console. DataStax OpsCenter is bundled with DataStax support offerings, or you can register for a free version licensed for development or non-production use.

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 Cassandra or DataStax Enterprise cluster are available in OpsCenter. OpsCenter metrics are captured and stored within Cassandra.



Time	Level	Message
4/27/2012 2:08pm	Info	Recognizing new node 10.118.19.228 ('141784319550391026443072753096570088106')
4/27/2012 2:08pm	Info	Recognizing new node 10.85.45.221 ('113427455640312821154458202477256070485')
4/27/2012 2:08pm	Info	Recognizing new node 10.32.254.16 ('85070591730234615865843651857942052864')

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
                                                      95315431979199388464207182617231204396
10.194.171.160  Down     Normal   ?           39.98   61078635599166706937511052402724559481   |<--|
10.196.14.48    Up       Normal   3.16 KB     30.01   78197033789183047700859117509977881938   |   |
10.196.14.239   Up       Normal   3.16 KB     30.01   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:

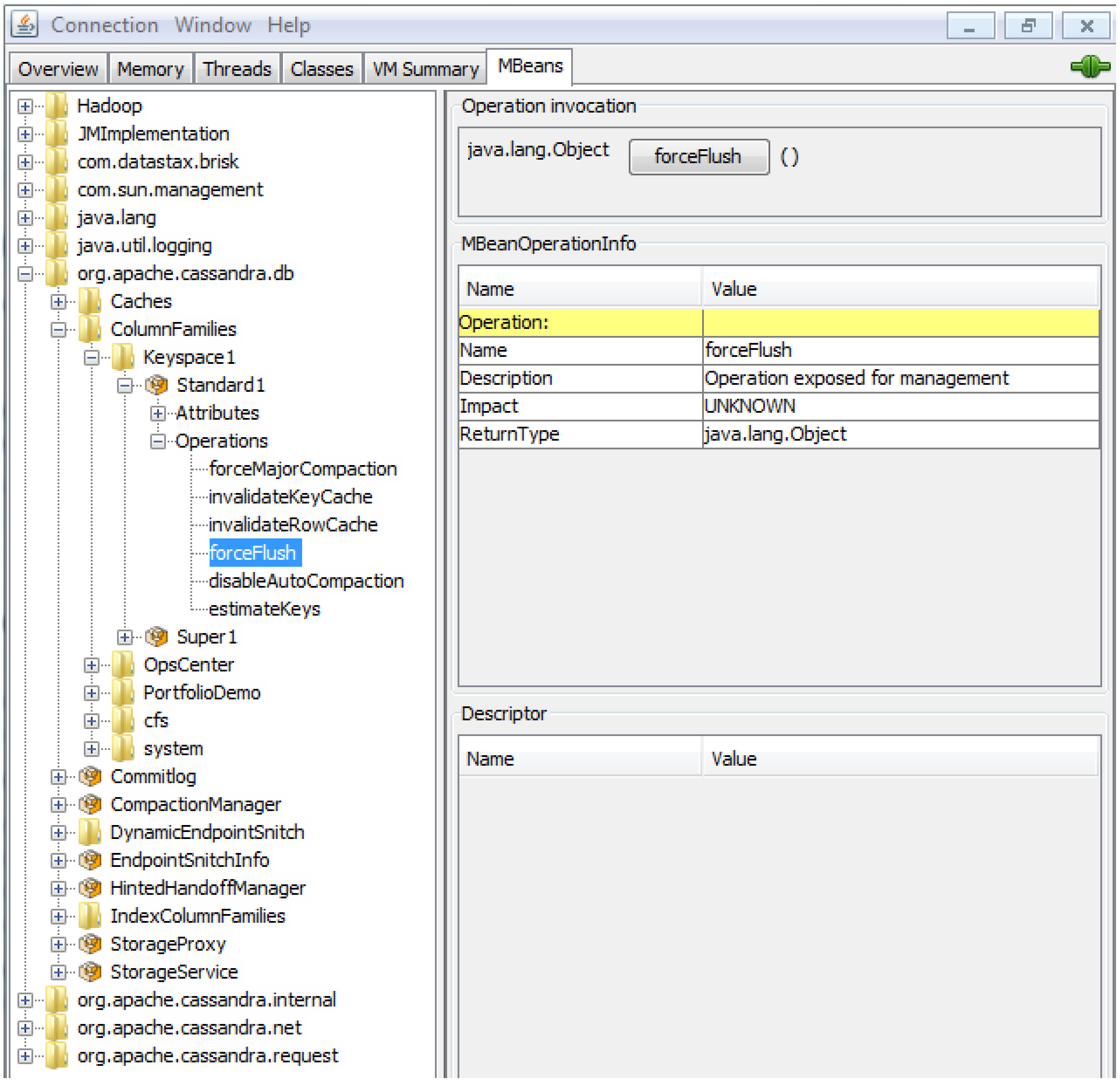
- **Overview** - Displays overview information about the Java VM and monitored values.
- **Memory** - Displays information about memory use.Threads - Displays information about thread use.
- **Classes** - Displays information about class loading.
- **VM Summary** - Displays information about the Java 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:



Name	Value
Operation:	
Name	forceFlush
Description	Operation exposed for management
Impact	UNKNOWN
ReturnType	java.lang.Object

Name	Value

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. `null` if no compactions are in progress.
BytesTotalInProgress	Total number of data bytes (index and filter are not included) being compacted. `null` if no compactions are in progress.

BytesCompacted	The progress of the current compaction. `null` 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 *consistency level <consistency>*.
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 `system_*` 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.

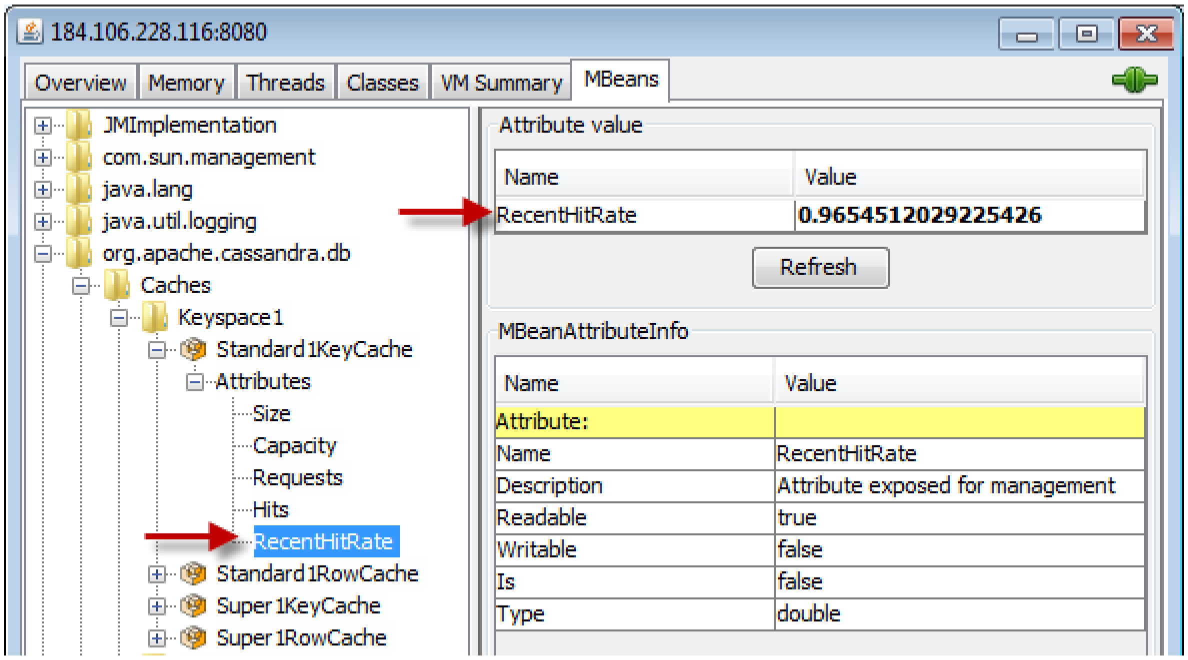
The recent read latency and write latency counters are important in making sure that operations are happening in a consistent manner. If these counters start to increase after a period of staying flat, it is probably an indication of a need to add cluster capacity.

`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:



Name	Value
RecentHitRate	0.9654512029225426

Name	Value
Attribute:	
Name	RecentHitRate
Description	Attribute exposed for management
Readable	true
Writable	false
Is	false
Type	double

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 Data Caches*
- *Tuning Java Garbage Collection*
- *Tuning the Java Heap*
- *Tuning Write Performance by Increasing Memtable Throughput*
- *Tuning Compaction*

- *Tuning Column Family Compression*
- Modifying *performance-related properties* in the cassandra.yaml file.

Tuning Data Caches

There are two types of data caches:

- Key cache: a cache of the *primary key index* for a Cassandra table.
- 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.

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.

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*.

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, skinny rows), this overhead can be substantial. Short, skinny 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 CQL. Set the *caching* parameter to enable or disable caching on the keys or rows, or both of a column family. 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 key 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*

Key cache hit rates of 85% or better are possible. 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.

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 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 one of the following heap sizes:

- One quarter (1/4) of the available system memory
- A minimum of 1GB and maximum of 8GB for systems with a very low or very high amount of RAM

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.

Changes to Cassandra's default should keep heap space to a minimum of 1/2 of RAM, but a maximum of 8GB.

Taking Advantage of OS Caching

In Cassandra 1.0 and later, column family row caches are stored in native memory outside the Java heap by default. Storing the caches in this location results in a smaller per-row memory footprint and reduced JVM heap requirements. Even after a row is released from the JVM memory, it can be kept in the OS page cache, especially if the data is requested repeatedly, or no other requested data replaces it.

If possible, lower *JVM heap size* and *memtable sizes* to free memory for OS page caching. Ultimately, through gradual adjustments, proper tuning balances the demands of heap, memtables, and caching on available memory.

Constraining Java Heap Size

Regardless of how much RAM the hardware has, keep the Java heap size constrained by the following formulas:

Cassandra

(*memtable_total_space_in_mb*) + 1GB + (*key_cache_size_estimate*) = min(1/2 RAM, 8GB)

DataStax Enterprise

(*memtable_total_space_in_mb*) + 1GB + (*key_cache_size_estimate*) + `$DSE_ROLE` = min(1/2 RAM, 8GB)

where `$DSE_ROLE` adds 1 GB for Analytics (Hadoop) and 2 GB for Search (Solr) nodes.

Estimating Cache Sizes

Use the *nodetool cfstats* to get the necessary information for estimating actual cache sizes.

To estimate the key cache size for a single column family:

```
key cache size = (average(‘Key size in bytes’) + 64)) * ‘Key cache size’ * 10-12
```

To estimate the row cache size:

```
row cache size = (average ‘Row size in bytes’ + 64) * ‘Row cache size’ * 10-12
```

How Caching Works

With proper tuning, each key cache hit can save one disk seek per SSTable and each row cache hit can save at least two seeks. Using only the key cache, disk (OS page cache) activity continues when Cassandra reads requested data rows. If both row and key caches are configured, the row cache returns results whenever possible.

In the case of a row cache miss, the key cache may still provide a hit, assuming that it holds a larger number of keys than the row cache. If a read operation hits the row cache, and the row is not in the row cache, but is present in the key cache, the key cache is used to find the exact location of the row on disk in the SSTable. If a row is not in the key cache, the read operation populates the key cache after accessing the row on disk so subsequent reads of the row can benefit.

Tuning Write Performance by Increasing Memtable Throughput

Cassandra flushes memtables to disk, creating SSTables when *the commit log space threshold* has been exceeded. Configure this threshold per node in the `cassandra.yaml`. How you tune memtable thresholds depends on your data and write load. Increase memtable throughput under either of these conditions:

- The write load includes a high volume of updates on a smaller set of data.
- A steady stream of continuous writes occurs. This action leads to more efficient compaction.

Note

Allocating memory for memtables reduces the memory available for caching and other internal Cassandra structures, so tune carefully and in small increments.

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 the *compaction_strategy* for each column family based on its access patterns. 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;
```

Initiate a major compaction through *nodetool compact*. A major compaction merges all SSTables into one. Though major compaction can free disk space used by accumulated SSTables, during runtime it temporarily doubles disk space usage and is I/O and CPU intensive. After running a major compaction, automatic minor compactions are no longer triggered, frequently requiring you to manually run major compactions on a routine basis. Expect read performance to improve immediately following a major compaction, and then to continually degrade until you invoke the next major compaction. DataStax does *not* recommend major compaction.

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 Set and Tune Compression

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.

Managing a Cassandra Cluster

This section discusses routine management and maintenance tasks.

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 for 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. 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. This is determined by the settings of *initial_token* when the node first starts up.
- The nodes it should contact to learn about the cluster and establish the gossip process. This is determined by the setting the *seeds* when the node first starts up, that is, the nodes it needs to contact to get ring and gossip information about the other nodes in the cluster.
- 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 as well before it is started.

You set the *Node and Cluster Initialization Properties* in *cassandra.yaml* file.

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 operationally 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 doubling the cluster size is not feasible, and you need to increase capacity by a non-uniform number of nodes, you will have to recalculate tokens for the entire cluster. Existing nodes will have to have their new tokens assigned using *nodetool move*. After all nodes have been restarted with their new token assignments, run a *nodetool cleanup* in order 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. Note that this approach will probably not result in a perfectly balanced ring, but it will alleviate hot spots.

Note

If you have DataStax OpsCenter Enterprise Edition, you can quickly add nodes to the cluster using this approach and then use its rebalance feature to automatically calculate balanced token ranges, move tokens accordingly, and then perform cleanup on the nodes after the moves are complete.

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. *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 a 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.

Note

The rebalance feature in DataStax OpsCenter Enterprise Edition automatically calculates balanced token ranges and perform steps 6 and 7 on each node in the cluster in the correct order.

Changing the Replication Factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster.

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.

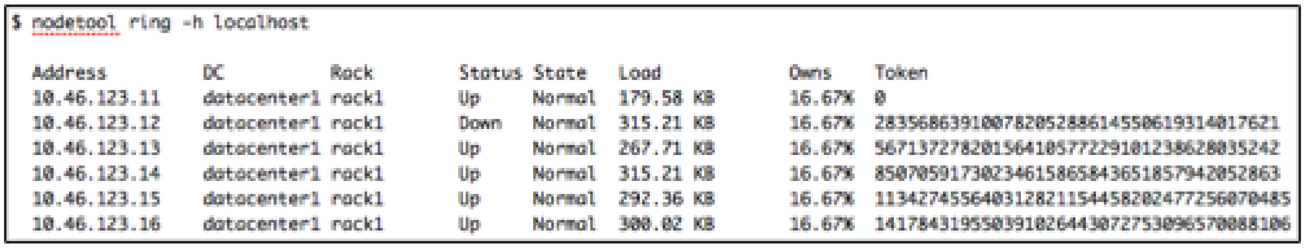
Replacing a Dead Node

To replace a node that has died (due to hardware failure, for example), you can bring up a new node in its place by starting the new node with the `-Dcassandra.replace_token=<token>` parameter and having the new node assume the token position of the node that has died. To replace a dead node in this way:

- The token that is used has to be from a node that is down - trying to replace a node using a token from a live node will result in an exception.
- The token that is used must already be part of the ring.
- The new node that is joining the cluster cannot have any preexisting Cassandra data on it (empty the data directory if you want to force a node replacement).

To replace a dead node:

1. Confirm the dead node using the *nodetool ring* command on any live node in the cluster (note the *Down* status and the token value of the dead node). For example:



```
$ 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. Prepare the replacement node by *installing Cassandra* and correctly configuring its *cassandra.yaml* file.
3. Start Cassandra on the new node using the startup property `-Dcassandra.replace_token=<token>` and pass in the same token that was used by the dead node. For example:

   ```
   $ cassandra -Dcassandra.replace_token=28356863910078205288614550619314017621
   ```

4. The new node will start in a hibernate state and begin to bootstrap data from its associated replica nodes. During this time, the node will not accept writes and is seen as down to other nodes in the cluster. When the bootstrap is complete, the node will be marked as up and any missed writes that occurred during bootstrap will be replayed using hinted handoff.
5. Once the new node is up, it is strongly recommended to run *nodetool repair* on each keyspace to ensure the node is fully consistent. For example:

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

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 and 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 can resume consistency using Cassandra's built-in consistency mechanisms.

After a system-wide snapshot has been taken, you can enable incremental backups on each node (disabled by default) to backup data that has changed since the last snapshot was taken. Each time an SSTable is flushed, a hard link is copied into a `/backups` subdirectory of the data directory.

Taking a Snapshot

Snapshots are taken per node using the *nodetool snapshot* command. If you want to take a global snapshot (capture all nodes in the cluster at the same time), 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 will 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 off 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 snapshot name. For example:

```
$ nodetool -h localhost -p 7199 snapshot 12022011
```

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.

Clearing Snapshot Files

When taking a snapshot, previous snapshot files are not automatically deleted. To maintain the snapshot directories, old snapshots that are no longer needed should be removed.

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

If you want to clear snapshots on all nodes at once, run the `nodetool clearsnapshot` command using a parallel ssh utility, such as `pssh`.

To clear all snapshots for a node

Run the `nodetool clearsnapshot` command. For example:

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

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 you to store 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

To restore a keyspace from a snapshot, you will need all of the snapshot files for the column family, and if using incremental backups, any incremental backup files created after the snapshot was taken.

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 to be restored.
2. Clear all files the `/var/lib/cassandra/commitlog` (by default).
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 as well, 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, keeping in mind that a temporary burst of I/O activity will consume a large amount of CPU resources.

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 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 Identifiers and Keywords

String literals and identifiers, such as keyspace and column family names, are case-sensitive. For example, identifier `MyColumnFamily` and `mycolumnfamily` are not equivalent. CQL keywords are case-insensitive. For example, the keywords `SELECT` and `select` are equivalent, although this document shows keywords in uppercase.

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
ascii	US-ASCII character string
bigint	64-bit signed long
blob	Arbitrary bytes (no validation), expressed as hexadecimal
boolean	true or false
counter	Distributed counter value (64-bit long)
decimal	Variable-precision decimal
double	64-bit IEEE-754 floating point
float	32-bit IEEE-754 floating point
int	32-bit signed integer
text	UTF-8 encoded string

timestamp	Date plus time, encoded as 8 bytes since epoch
uuid	Type 1 or type 4 UUID
varchar	UTF-8 encoded string
varint	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.marshal package.

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 handing 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 WHERE state='TX' USING CONSISTENCY QUORUM;
```

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

CQL supports 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`

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
compaction_strategy_class	SizeTieredCompactionStrategy
compaction_strategy_options	none
compression_parameters	none
comment	''(an empty string)

dc_local_read_repair_chance	0.0
gc_grace_seconds	864000
min_compaction_threshold	4
max_compaction_threshold	32
read_repair_chance	0.1
replicate_on_write	false

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, use `ALTER TABLE` and the `ALTER` and `TYPE` keywords in the following way:

```
ALTER TABLE addamsFamily ALTER lastKnownLocation TYPE uuid;
```

The column must 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. 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:sstable_compression = 'DeflateCompressor'
  AND compression_parameters:chunk_length_kb = 64;

ALTER TABLE users
   WITH compaction_strategy_class='SizeTieredCompactionStrategy'
   AND min_compaction_threshold = 6;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*

DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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* for the written columns.

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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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.

```
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 adding new 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*.

At this time, updates to data in a column family created with compact storage is not allowed.

Unless you specify WITH COMPACT STORAGE, CQL creates a column family with non-compact storage.

Specifying Column Types (optional)

You can assign columns a type during column family creation. These columns are validated when a write occurs, and intelligent CQL drivers and interfaces can decode the column values correctly when receiving them. 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;
```

```
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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	

INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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

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);
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
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> [...] ];
```

Description

`CREATE KEYSPACE` creates a top-level namespace and sets the replica placement strategy (and associated replication options) for the keyspace. Valid keyspace names are strings of alpha-numeric characters and underscores, and must begin with a letter. Properties such as replication strategy and count are specified during creation using the following accepted keyword arguments:

Keyword	Description
strategy_class	Required. The name of the *replica placement strategy* for the new keyspace, such as SimpleStrategy and NetworkTopologyStrategy.
strategy_options	Optional. Additional arguments appended to the option name.

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`.

See *Choosing Keyspace Replication Options* for guidance on how to best configure replication strategy and strategy options for your cluster.

```
CREATE KEYSPACE Excelsior WITH strategy_class = 'SimpleStrategy'
  AND strategy_options:replication_factor = 1;
CREATE KEYSPACE Excalibur WITH strategy_class = 'NetworkTopologyStrategy'
  AND strategy_options:DC1 = 1 AND strategy_options:DC2 = 3;
```

For `NetworkTopologyStrategy`, you specify the number of replicas per data center in the format of `strategy_options:<datacenter_name>=<number>`. Note that what you specify for <datacenter_name> depends on the cluster-configured *snitch* you are using. There is a correlation between the data center name defined in the keyspace `strategy_options` and the data center name as recognized by the snitch you are using. The *nodetool ring* command prints out data center names and rack locations of your nodes if you are not sure what they are.

Examples

Define a new keyspace using the simple replication strategy:

```
CREATE KEYSPACE History WITH strategy_class = 'SimpleStrategy'
 AND strategy_options:replication_factor = 1;
```

Define a new keyspace using a network-aware replication strategy and snitch. This example assumes you are using the *PropertyFileSnitch* and your data centers are named `DC1` and `DC2` in the `cassandra-topology.properties` file:

```
CREATE KEYSPACE MyKeyspace WITH strategy_class = 'NetworkTopologyStrategy'
 AND strategy_options:DC1 = 3 AND strategy_options:DC2 = 3;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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.

Example

```
DELETE col1, col2, col3 FROM Planeteers 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* for the written columns.

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 col1 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 col1 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');
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	

TRUNCATE	
UPDATE	
USE	

DROP TABLE

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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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

```
DROP INDEX user_state;

DROP INDEX users_zip_idx;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*

BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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*, an integer, for the written columns.

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, gwashington, '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');
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

SELECT

Retrieves data, including Solr data, from a Cassandra column family.

Synopsis

```
SELECT <select expression>
 FROM <column family>
 [USING CONSISTENCY <level>]
 [WHERE (<clause>)] [LIMIT <n>]
 [ORDER BY <composite key 2>] [ASC, DESC]
```

`<clause>` syntax is:

```
<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;
```

If you do not specify a limit, a maximum of 10,000 rows are returned by default. Using the *LIMIT option*, you can specify that the query return a greater or fewer number of rows.

```
SELECT COUNT(*) FROM big_columnfamily;

 count
-------
 10000

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

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.

Note: 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 <=.

WHERE clauses can include greater-than and less-than 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
```

```
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;
```

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
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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 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* for the written columns.

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-b1ed-4e96-bf79-4280797cba80,
                         06a8913c-c0d6-477c-937d-6c1b69a95d43,
                         bc108776-7cb5-477f-917d-869c12dfffa8);
```

Update several columns in a single row:

```
UPDATE users USING CONSISTENCY QUORUM
  SET 'name' = 'John Smith', 'email' = 'jsmith@cassie.com'
  WHERE user_uuid = 88b8fd18-b1ed-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';
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*

CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	

USE	

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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	

DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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.

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	

USE	

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 various forms of the `DESCRIBE` or *DESC* command yield information about the currently connected Cassandra cluster and the data objects (keyspaces and column families) stored in the cluster:

- 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.

Examples

```
DESCRIBE CLUSTER;

DESCRIBE KEYSPACE PortfolioDemo;

DESCRIBE COLUMNFAMILIES;

DESCRIBE COLUMNFAMILY Stocks;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	

DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

EXIT

Terminates the CQL command-line client.

Synopsis

```
EXIT | QUIT;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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:

- 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;
```

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

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 '~/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.

Other CQL Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

nodetool

Under the bin directory you will find the nodetool utility. This can be used to help manage a cluster. Usage:

```
bin/nodetool -h HOSTNAME [-p JMX_PORT ] COMMAND...
```

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:

```
bin/nodetool -h HOSTNAME [-p JMX_PORT -u JMX_USERNAME -p JMX_PASSWORD ] COMMAND...
```

Command List

The available commands are:

Command List		
cfstats	*info*	*scrub*
cfhistograms	*invalidatekeycache*	*setcachecapacity*
cleanup	*invalidaterowcache*	*setcompactionthreshold*
clearsnapshot	*join*	*snapshot*
compact	*loadbalance*	*stop*
decomission	*move*	*streams*
drain	*removetoken*	*tpstats*
flush	*repair*	*upgradesstables*

getcompactionthreshold	*ring*	*version*

Command Details

Details for each command are listed below:

cfstats

Prints statistics for every keyspace and column family.

cfhistograms *keyspace cf_name*

Prints 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.

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

Deletes all existing snapshots.

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.

decommission

Tells a live node to decommission itself (streaming its data to the next node on the ring) See also:

> http://wiki.apache.org/cassandra/NodeProbe#Decommission
>
> http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely

drain

Flushes all memtables for a node and causes the node to stop accepting write operations. Read operations will continue to work. This is typically used before upgrading a node to a new version of Cassandra.

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

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.

loadbalance

Moves the node to a new token so that it will split the range of whatever token currently has the highest load (this is the same heuristic used for bootstrap). This is rarely called for, as it does not balance the ring in a meaningful way. See *Adding Capacity to an Existing Cluster*.

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

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 a quick idea of how balanced the load 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.

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 *keyspace cf_name key_cache_capacity row_cache_capacity*

Sets the size of the key cache and row cache. These may be either an absolute number or a percentage (in the form of a floating point number).

setcompactionthreshold *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

snapshot [*snapshot-name*]

Takes an online snapshot of Cassandra's data. Before taking the snapshot, the node is flushed. The results can be found in Cassandra's data directory (typically `/var/lib/cassandra/data`) under the `snapshots` directory of each keyspace. See also:

> http://wiki.apache.org/cassandra/Operations#Backing_up_data

stop [*operation type*]

Stops an operation from continuing to run. Options are COMPACTION, 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.

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.

tpstats

Prints 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. For more details, 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

Prints the Cassandra release version for the node being queried.

cassandra

The `cassandra` utility starts the Cassandra Java server process.

Usage

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
- ~/.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
`access.properties=<filename>`	The file location of the access.properties file.
`cassandra-pidfile=<filename>`	Log the Cassandra server process ID in the named file. Useful for stopping Cassandra by killing its PID.
`cassandra.config=<directory>`	The directory location of the Cassandra configuration files.
`cassandra.initial_token=<token>`	Sets the initial partitioner token for a node the first time the node is started.
`cassandra.join_ring=<true|false>`	Set to false to start Cassandra on a node but not have the node join the cluster.
`cassandra.load_ring_state=` `<true|false>`	Set to false to clear all gossip state for the node on restart. Use if youhave changed node information in `cassandra.yaml` (such as `listen_address`).
`cassandra.renew_counter_id=` `<true|false>`	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 `-Dcassandra.renew_counter_id=true`, then run `nodetool repair` once the node is up again.
`cassandra.replace_token=<token>`	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.
`cassandra.write_survey=true`	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
```

stress

The `/tools/stress` directory contains the Java-based stress testing utilities that can help in benchmarking and load testing a Cassandra cluster: `stress.java` and the daemon `stressd`. The daemon mode, which keeps the JVM warm more efficiently, may be useful for large-scale benchmarking.

Setting up the Stress Utility

Use Apache ant to to build the stress testing tool:

1. Run `ant` from the Cassandra source directory.
2. Run `ant` from the /tools/stress directory.

Usage

There are three different modes of operation:

- inserting (loading test data)
- reading
- range slicing (only works with the OrderPreservingPartioner)
- indexed range slicing (works with RandomParitioner on indexed ColumnFamiles).

You can use these modes with or without the `stressd` daemon running. For larger-scale testing, the daemon can yield better performance by keeping the JVM warm and preventing potential skew in test results.

If no specific operation is specified, `stress` will insert 1M rows.

The options available are:

-o <operation>, --operation <operation>

Sets the operation mode, one of 'insert', 'read', 'rangeslice', or 'indexedrangeslice'

-T <IP>, --send-to <IP>

Sends the command as a request to the stress daemon at the specified IP address. The daemon must already be running at that address.

-n <NUMKEYS>, --num-keys <NUMKEYS>

Number of keys to write or read. Default is 1,000,000.

-l <RF>, --replication-factor <RF>

Replication Factor to use when creating needed column families. Defaults to 1.

-R <strategy>, --replication-strategy <strategy>

Replication strategy to use (only on insert when keyspace does not exist. Default is:org.apache.cassandra.locator.SimpleStrategy.

-O <properties>, --strategy-properties <properties>

Replication strategy properties in the following format <dc_name>:<num>,<dc_name>:<num>,... Use with network topology strategy.

-W, --no-replicate-on-write

Set replicate_on_write to false for counters. Only for counters add with CL=ONE.

-e <CL>, --consistency-level <CL>

Consistency Level to use (ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL, ANY). Default is ONE.

-c <COLUMNS>, --columns <COLUMNS>

Number of columns per key. Default is 5.

-d <NODES>, --nodes <NODES>

Nodes to perform the test against.(comma separated, no spaces). Default is “localhost”.

-y <TYPE>, --family-type <TYPE>

Sets the ColumnFamily type. One of 'Standard' or 'Super'. If using super, set the -u option also.

-V, --average-size-values

Generate column values of average rather than specific size.

-u <SUPERCOLUMNS>, --supercolumns <SUPERCOLUMNS>

Use the number of supercolumns specified. You must set the -y option appropriately, or this option has no effect.

-g <COUNT>, --get-range-slice-count <COUNT>

Sets the number of rows to slice at a time and defaults to 1000. This is only used for the rangeslice operation and will *NOT* work with the RandomPartioner. You must set the OrderPreservingPartioner in your storage configuration (note that you will need to wipe all existing data when switching partioners.)

-g <KEYS>, --keys-per-call <KEYS>

Number of keys to get_range_slices or multiget per call. Default is 1000.

-r, --random

Only used for reads. By default, stress will perform reads on rows with a Guassian distribution, which will cause some repeats. Setting this option makes the reads completely random instead.

-i, --progress-interval

The interval, in seconds, at which progress will be output.

Using the Daemon Mode (`stressd`)

Usage for the daemon mode is:

```
/tools/stress/bin/stressd start|stop|status [-h <host>]
```

During stress testing, you can keep the daemon running and send `stress.java` commands through it using the `-T` or `--send-to` option flag.

Examples

1M inserts to given host:

```
/tools/stress/bin/stress -d 192.168.1.101
```

1M reads from given host:

```
tools/stress/bin/stress -d 192.168.1.101 -o read
```

10M inserts spread across two nodes:

```
/tools/stress/bin/stress -d 192.168.1.101,192.168.1.102 -n 10000000
```

10M inserts spread across two nodes using the daemon mode:

```
/tools/stress/bin/stress -d 192.168.1.101,192.168.1.102 -n 10000000 -T 54.0.0.1
```

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. 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 and CQL Shell Commands

CQL Commands	CQL Shell Commands
ALTER TABLE	*ASSUME*
BATCH	*CAPTURE*
CREATE TABLE	*DESCRIBE*
CREATE INDEX	*EXIT*
CREATE KEYSPACE	*SHOW*
DELETE	*SOURCE*
DROP TABLE	
DROP INDEX	
DROP KEYSPACE	
INSERT	
SELECT	
TRUNCATE	
UPDATE	
USE	

Install 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 (or Hadoop in DataStax Enterprise) on a node, the node will act as a standalone database server rather than joining the database cluster.

Port	Description
Public Facing Ports	
22	SSH (default)
OpsCenter Specific	
8888	OpsCenter website port
Intranode Ports	
Cassandra Specific	
1024+	JMX reconnection/loopback ports
7000	Cassandra intra-node port
7199	Cassandra JMX monitoring port
9160	Cassandra client port
OpsCenter Specific	
50031	OpsCenter HTTP proxy for Job Tracker
61620	OpsCenter intra-node monitoring port
61621	OpsCenter agent ports

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` (in the background - default) `$ bin/cassandra -f` (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*
- *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 (TotalMemory – 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 indicies, using much less address space. DataStax recommends that Cassandra nodes disable swap entirely, 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 communciates 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

One possibility is that Java is not allowed to open enough file descriptors. Cassandra generally needs more than the default (1024) amount. This can be adjusted by increasing 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. See if the number of file descriptors opened by java seems reasonable when running `lsof -n | grep java` and report the error if the number is greater than a few thousand.

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:

```
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-libsnappyjava.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.tempdir=/path/to/newtmp`

  Cassandra: `bin/cassandra -Dorg.xerial.snappy.tempdir=/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.tempdir=/path/to/newtmp`

  The default `cassandra-env.sh` looks for the variable and appends to it when starting the JVM.