## Contents

**Apache Cassandra 1.0 Documentation**  

- Introduction to Apache Cassandra and DataStax Community Edition .................................. 1  
  - Installing the DataStax Community Binaries on Linux or Mac OS X as Root ............................ 1  
    - Steps to Install Cassandra ........................................................................................................ 1  
    - Next Steps .................................................................................................................................. 2  
  - Installing the DataStax Community Binaries as User ................................................................. 2  
    - Steps to Install Cassandra ............................................................................................................ 2  
    - Steps to Configure and Start the Cluster ..................................................................................... 3  
    - Next Steps .................................................................................................................................... 4  
  - Installing the DataStax Community Binaries on Windows ......................................................... 4  
    - Prerequisites ................................................................................................................................. 4  
    - Starting Services .......................................................................................................................... 4  
    - Launching Cassandra Server and OpsCenter ................................................................................ 4  
    - Troubleshooting OpsCenter Windows Installations ...................................................................... 5  
  - Running the Portfolio Demo Sample Application .............................................................................. 5  
    - About the Portfolio Demo Use Case ............................................................................................... 5  
    - Running the Demo Web Application ............................................................................................ 6  
    - Exploring the Sample Data Model .................................................................................................. 7  
      - Viewing the Schema Definitions in Cassandra-CLI ...................................................................... 7  
  - DataStax Community Release Notes ............................................................................................... 8  
    - What’s New ..................................................................................................................................... 8  
    - Prerequisites ................................................................................................................................. 8  
  - The Install Types .............................................................................................................................. 8  
    - Learning about Cassandra .............................................................................................................. 8  
  - Understanding the Cassandra Architecture ..................................................................................... 8  
    - About Internode Communications (Gossip) .................................................................................... 8  
      - About Cluster Membership and Seed Nodes .............................................................................. 9  
    - About Failure Detection and Recovery .......................................................................................... 9  
  - About Data Partitioning in Cassandra ............................................................................................ 10  
    - Understanding the Partitioner Types .............................................................................................. 10  
      - About the RandomPartitioner ..................................................................................................... 11  
      - About the ByteOrderedPartitioner ............................................................................................ 11  
    - About Partitioning in Multiple Data Center Clusters .................................................................. 11  
  - About Replication in Cassandra ...................................................................................................... 13  
    - Replication Strategy ...................................................................................................................... 13  
      - SimpleStrategy ............................................................................................................................. 13  
      - NetworkTopologyStrategy ............................................................................................................ 14  
  - Prerequisites ...................................................................................................................................... 2  
  - Steps to Configure and Start the Cluster .......................................................................................... 4  
  - Steps to Install Cassandra ................................................................................................................. 7  
  - Next Steps ....................................................................................................................................... 13  
  - Troubleshooting OpsCenter Windows Installations ......................................................................... 13  
  - Starting Services ............................................................................................................................. 14
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculating Tokens for a Single Data Center</td>
<td>44</td>
</tr>
<tr>
<td>Calculating Tokens for Multiple Racks in a Single Data Center</td>
<td>44</td>
</tr>
<tr>
<td>Calculating Tokens for a Multiple Data Center Cluster</td>
<td>45</td>
</tr>
<tr>
<td>About Writes in Cassandra</td>
<td>45</td>
</tr>
<tr>
<td>Offsetting Token Assignments</td>
<td>46</td>
</tr>
<tr>
<td>About Deletes</td>
<td>47</td>
</tr>
<tr>
<td>About Inserts and Updates</td>
<td>47</td>
</tr>
<tr>
<td>About Transactions and Concurrency Control</td>
<td>47</td>
</tr>
<tr>
<td>About Compaction</td>
<td>47</td>
</tr>
<tr>
<td>Understanding the Cassandra Data Model</td>
<td>47</td>
</tr>
<tr>
<td>The Cassandra Data Model</td>
<td>47</td>
</tr>
<tr>
<td>Comparing the Cassandra Data Model to a Relational Database</td>
<td>47</td>
</tr>
<tr>
<td>About Keyspaces</td>
<td>49</td>
</tr>
<tr>
<td>Defining Keyspaces</td>
<td>49</td>
</tr>
<tr>
<td>About Column Families</td>
<td>50</td>
</tr>
<tr>
<td>About Columns</td>
<td>51</td>
</tr>
<tr>
<td>About Special Columns (Counter, Expiring, Super)</td>
<td>51</td>
</tr>
<tr>
<td>About Expiring Columns</td>
<td>51</td>
</tr>
<tr>
<td>About Counter Columns</td>
<td>52</td>
</tr>
<tr>
<td>About Super Columns</td>
<td>52</td>
</tr>
<tr>
<td>About Data Types (Comparators and Validators)</td>
<td>53</td>
</tr>
<tr>
<td>About Validators</td>
<td>53</td>
</tr>
<tr>
<td>About Comparators</td>
<td>54</td>
</tr>
<tr>
<td>About Column Family Compression</td>
<td>54</td>
</tr>
<tr>
<td>When to Use Compression</td>
<td>54</td>
</tr>
<tr>
<td>Configuring Compression on a Column Family</td>
<td>54</td>
</tr>
<tr>
<td>About Indexes in Cassandra</td>
<td>55</td>
</tr>
<tr>
<td>About Primary Indexes</td>
<td>55</td>
</tr>
<tr>
<td>About Secondary Indexes</td>
<td>55</td>
</tr>
<tr>
<td>When to Use Secondary Indexes</td>
<td>55</td>
</tr>
<tr>
<td>Building and Using Secondary Indexes</td>
<td>55</td>
</tr>
<tr>
<td>Planning Your Data Model</td>
<td>56</td>
</tr>
<tr>
<td>Start with Queries</td>
<td>56</td>
</tr>
<tr>
<td>Denormalize to Optimize</td>
<td>56</td>
</tr>
<tr>
<td>Planning for Concurrent Writes</td>
<td>56</td>
</tr>
<tr>
<td>Using Natural or Surrogate Row Keys</td>
<td>57</td>
</tr>
<tr>
<td>UUID Types for Column Names</td>
<td>57</td>
</tr>
<tr>
<td>Managing and Accessing Data in Cassandra</td>
<td>57</td>
</tr>
<tr>
<td>About Writes in Cassandra</td>
<td>57</td>
</tr>
<tr>
<td>About Compaction</td>
<td>57</td>
</tr>
<tr>
<td>About Transactions and Concurrency Control</td>
<td>58</td>
</tr>
<tr>
<td>About Inserts and Updates</td>
<td>58</td>
</tr>
<tr>
<td>About Deletes</td>
<td>58</td>
</tr>
</tbody>
</table>
Getting Started with CQL

Getting Started Using the Cassandra CLI

Cassandra Client APIs

About Data Consistency in Cassandra

Tunable Consistency for Client Requests

About Write Consistency

About Read Consistency

Choosing Client Consistency Levels

Consistency Levels for Multi-Data Center Clusters

Specifying Client Consistency Levels

About Cassandra's Built-in Consistency Repair Features

Cassandra Client APIs

About Cassandra CLI

About CQL

Other High-Level Clients

Java: Hector Client API

Python: Pycassa Client API

PHP: Phpcassa Client API

Getting Started Using the Cassandra CLI

Creating a Keyspace

Creating a Column Family

Creating a Counter Column Family

Inserting Rows and Columns

Reading Rows and Columns

Setting an Expiring Column

Indexing a Column

Deleting Rows and Columns

Dropping Column Families and Keyspaces

Getting Started with CQL

Starting the CQL Command-Line Program (cqlsh)

Running CQL Commands with cqlsh

Creating a Keyspace

Creating a Column Family

Inserting and Retrieving Columns

Adding Columns with ALTER COLUMNFAMILY

Altering Column Metadata

Specifying Column Expiration with TTL

Dropping Column Metadata

Indexing a Column

Deleting Columns and Rows
Configuration ................................................................. 70
Node and Cluster Configuration (cassandra.yaml) ............... 70

Node and Cluster Initialization Properties ....................... 72
  broadcast_address ...................................................... 72
  cluster_name .......................................................... 72
  commitlog_directory .................................................. 72
  data_file_directories ............................................... 72
  initial_token .......................................................... 73
  listen_address ........................................................ 73
  partitioner ............................................................. 73
  rpc_address ........................................................... 73
  rpc_port ............................................................... 73
  saved_caches_directory .............................................. 73
  seed_provider ........................................................ 73
  seeds .................................................................. 73
  storage_port ............................................................ 73
  endpoint_snitch ......................................................... 74

Performance Tuning Properties ......................................... 74
  column_index_size_in_kb ............................................. 74
  commitlog_sync ......................................................... 74
  commitlog_sync_period_in_ms ...................................... 74
  commitlog_total_space_in_mb ...................................... 74
  compaction_preheat_key_cache .................................... 74
  compaction_throughput_mb_per_sec ............................... 74
  concurrent_compactors ............................................... 74
  concurrent_reads ...................................................... 75
  concurrent_writes ..................................................... 75
  flush_largest_memtables_at ........................................... 75
  in_memory_compaction_limit_in_mb ............................... 75
  index_interval ......................................................... 75
  memtable_flush_queue_size ......................................... 75
  memtable_flush_writers ............................................... 75
  memtable_total_space_in_mb ........................................ 75
  reduce_cache_capacity_to .......................................... 76
  reduce_cache_sizes_at ............................................... 76
  sliced_buffer_size_in_kb ........................................... 76
  stream_throughput_outbound_megabits_per_sec .................. 76

Remote Procedure Call Tuning Properties .......................... 75
Keyspace and Column Family Storage Configuration

Keyspace Attributes

Security Properties

Automatic Backup Properties

Internode Communication and Fault Detection Properties

placement_strategy
name
request_scheduler
request_scheduler_id
request_scheduler_options
    throttle_limit
default_weight
weights
rpc_keepalive
rpc_max_threads
rpc_min_threads
rpc_recv_buff_size_in_bytes
rpc_send_buff_size_in_bytes
rpc_timeout_in_ms
rpc_server_type
thrift_framed_transport_size_in_mb
thrift_max_message_length_in_mb

Internode Communication and Fault Detection Properties

dynamic_snitch
dynamic_snitch_badness_threshold
dynamic_snitch_reset_interval_in_ms
dynamic_snitch_update_interval_in_ms
hinted_handoff_enabled
hinted_handoff_throttle_delay_in_ms
max_hint_window_in_ms
phi_convict_threshold

Automatic Backup Properties

incremental_backups
snapshot_before_compaction

Security Properties

authenticator
authority
internode_encryption
keystore
keystore_password
truststore
truststore_password

Keyspace and Column Family Storage Configuration

Keyspace Attributes

name
placement_strategy
strategy_options

Column Family Attributes

column_metadata

column_type

comment

compaction_strategy

compaction_strategy_options

comparator

compare_subcolumns_with

compression_options

dc_local_read_repair_chance

default_validation_class

gc_grace_seconds

keys_cached

key_validation_class

name

read_repair_chance

replicate_on_write

max_compaction_threshold

min_compaction_threshold

memtable_flush_after_mins

memtable_operations_in_millions

memtable_throughput_in_mb

rows_cached

Java and System Environment Settings Configuration

Heap Sizing Options

JMX Options

Further Reading on JVM Tuning

Authentication and Authorization Configuration

access.properties

passwd.properties

Logging Configuration

Logging Levels via the Properties File

Logging Levels via JMX

Operations

Monitoring a Cassandra Cluster

Monitoring Using DataStax OpsCenter

Monitoring Using nodetool

Monitoring Using JConsole
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compaction Metrics</td>
<td>91</td>
</tr>
<tr>
<td>Thread Pool Statistics</td>
<td>92</td>
</tr>
<tr>
<td>Read/Write Latency Metrics</td>
<td>92</td>
</tr>
<tr>
<td>ColumnFamily Statistics</td>
<td>92</td>
</tr>
<tr>
<td>Monitoring and Adjusting Cache Performance</td>
<td>93</td>
</tr>
<tr>
<td>Backing Up and Restoring Data</td>
<td>94</td>
</tr>
<tr>
<td>Managing a Cassandra Cluster</td>
<td>95</td>
</tr>
<tr>
<td>Tuning Cassandra</td>
<td>96</td>
</tr>
<tr>
<td>Tuning the Cache</td>
<td>97</td>
</tr>
<tr>
<td>How Caching Works</td>
<td>98</td>
</tr>
<tr>
<td>Configuring the Column Family Key Cache</td>
<td>98</td>
</tr>
<tr>
<td>Configuring the Column Family Row Cache</td>
<td>99</td>
</tr>
<tr>
<td>Data Modeling Considerations for Cache Tuning</td>
<td>99</td>
</tr>
<tr>
<td>Hardware and OS Considerations for Cache Tuning</td>
<td>100</td>
</tr>
<tr>
<td>Tuning the Cache</td>
<td>100</td>
</tr>
<tr>
<td>Estimating Cache Sizes</td>
<td>101</td>
</tr>
<tr>
<td>Tuning Write Performance (Memtables)</td>
<td>101</td>
</tr>
<tr>
<td>Tuning Java Heap Size</td>
<td>102</td>
</tr>
<tr>
<td>Tuning Java Garbage Collection</td>
<td>102</td>
</tr>
<tr>
<td>Tuning Compaction</td>
<td>103</td>
</tr>
<tr>
<td>Choosing a Column Family Compaction Strategy</td>
<td>103</td>
</tr>
<tr>
<td>Setting the Compaction Strategy on a Column Family</td>
<td>104</td>
</tr>
<tr>
<td>Tuning Options for Size-Tiered Compaction</td>
<td>104</td>
</tr>
<tr>
<td>Adding Capacity to an Existing Cluster</td>
<td>105</td>
</tr>
<tr>
<td>Calculating Tokens For the New Nodes</td>
<td>105</td>
</tr>
<tr>
<td>Adding Nodes to a Cluster</td>
<td>106</td>
</tr>
<tr>
<td>Adding a Data Center to a Cluster</td>
<td>106</td>
</tr>
<tr>
<td>Changing the Replication Factor</td>
<td>107</td>
</tr>
<tr>
<td>Replacing a Dead Node</td>
<td>107</td>
</tr>
<tr>
<td>Backing Up and Restoring Data</td>
<td>108</td>
</tr>
<tr>
<td>Restoring from a Snapshot</td>
<td>108</td>
</tr>
<tr>
<td>References</td>
<td>109</td>
</tr>
<tr>
<td>CQL Language Reference</td>
<td>110</td>
</tr>
<tr>
<td>CQL Lexical Structure</td>
<td>110</td>
</tr>
<tr>
<td>CQL Identifiers and Keywords</td>
<td>110</td>
</tr>
<tr>
<td>CQL Data Types</td>
<td>111</td>
</tr>
<tr>
<td>Working with Dates and Times</td>
<td>111</td>
</tr>
</tbody>
</table>
CQL Commands ................................................................. 104
Specifying Consistency Level ......................................... 104
CQL Storage Parameters .................................................. 104
CQL Keyspace Storage Parameters .................................... 104
CQL Column Family Storage Parameters ............................ 105
CQL Commands ................................................................. 105
ALTER TABLE ................................................................. 105
  Synopsis ..................................................................... 105
  Description ................................................................. 105
  Examples ..................................................................... 106
BATCH ................................................................................ 106
  Synopsis ..................................................................... 106
  Description ................................................................. 107
  Example ....................................................................... 107
CREATE TABLE ................................................................. 107
  Synopsis ..................................................................... 107
  Description ................................................................. 108
  Examples ..................................................................... 108
CREATE INDEX ................................................................. 109
  Synopsis ..................................................................... 109
  Description ................................................................. 109
  Examples ..................................................................... 109
CREATE KEYSPACE ........................................................... 109
  Synopsis ..................................................................... 109
  Description ................................................................. 109
  Examples ..................................................................... 110
DELETE ............................................................................. 110
  Synopsis ..................................................................... 110
  Description ................................................................. 110
  Example ....................................................................... 111
DROP TABLE ................................................................. 111
  Synopsis ..................................................................... 111
  Description ................................................................. 111
  Example ....................................................................... 111
DROP INDEX ................................................................. 112
  Synopsis ..................................................................... 112
  Description ................................................................. 112
  Example ....................................................................... 112
DROP KEYSIZE ................................................................. 112
Troubleshooting Guide

Java reports an error saying there are too many open files

View of ring differs between some nodes

Nodes are dying with OOM errors

Nodes seem to freeze after some period of time

Reads are getting slower while writes are still fast

Schema Disagreement Exception Occurs

Configuring Firewall Port Access

Install Locations

sstable2json / json2sstable

sstable2json

Usage

Environment

Options

Examples

stress

Setting up the Stress Utility

Usage

Using the Daemon Mode ($stressd)

Examples

Starting Cassandra as a Service

Starting Cassandra as a Stand-Alone Process

Stopping Cassandra as a Service

Stopping Cassandra as a Stand-Alone Process

Binary Tarball Install Directories

Locations of the Configuration Files

Packaged Installs Directories

Binary Tarball Install Directories

Configuring Firewall Port Access

Starting and Stopping a Cassandra Cluster

Starting Cassandra as a Stand-Alone Process

Starting Cassandra as a Service

Stopping Cassandra as a Stand-Alone Process

Stopping Cassandra as a Service

Troubleshooting Guide

Schema Disagreement Exception Occurs

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
Apache Cassandra 1.0 Documentation

Apache Cassandra is a free, open-source, highly-scalable, distributed database system for managing large amounts of data.

Introduction to Apache Cassandra and DataStax Community Edition

This section contains information for installing a single node evaluation cluster of Apache Cassandra. For production or running Cassandra on multiple nodes, see Installing a Multiple Node Cluster.

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

These instructions tell you how to install an older version of Cassandra. DataStax recommends installing the latest version of Cassandra. See the Download page.

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. Check that Java is installed by running the following command in a terminal window:

   java -version

   Note

   Any version of Java is fine for installing an evaluation instance. For production deployments, see Installing Oracle JRE.

2. 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-cassandra-1.0.9-bin.tar.gz
   $ curl -OL http://downloads.datastax.com/community/opscenter.tar.gz

3. Unpack the distributions:

   $ tar -xzf dsc-cassandra-1.0.9-bin.tar.gz
   $ tar -xzf opscenter.tar.gz
   $ rm *.tar.gz
4. Start the Cassandra server in the background from the directory where the package was installed. For example, if `dsc-cassandra-1.0.9` is installed in your home directory:

   ```
   $ cd ~/dsc-cassandra-1.0.9
   $ 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.

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

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

---

**Next Steps**

- Install the DataStax OpsCenter. The OpsCenter is a browser-based application for managing and monitoring a Cassandra cluster. See the Installing the OpsCenter.
- Run the Portfolio Demo example application.
- 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. The root permissions install creates files in the `var/lib/cassandra` directory.

**Note**

These instructions tell you how to install an older version of Cassandra. DataStax recommends installing the latest version of Cassandra. See the Download page.

**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. Check that Java is installed by running the following command in a terminal window:

   java -version

   **Note**
   Any version of Java is fine for installing an evaluation instance. For production deployments, see *Installing Oracle JRE*.

2. 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-cassandra-1.0.9-bin.tar.gz
   $ curl -OL http://downloads.datastax.com/community/opscenter.tar.gz

3. Unpack the distributions:

   $ dsc-cassandra-1.0.9-bin.tar.gz
   $ tar -xzvf opscenter.tar.gz
   $ rm *.tar.gz

4. Rename the downloaded directory to datastax:

   $ mv dsc-cassandra-1.0.9 datastax

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

   $ cd datastax
   $ mkdir cassandra-data

6. 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 to use the new directory locations. 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, edit the following line in 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

Next Steps

- Install the DataStax OpsCenter. The OpsCenter is a browser-based application for managing and monitoring a Cassandra cluster. See the Installing the OpsCenter.
- Run the Portfolio Demo example application.

### 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:


- 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:
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.

- 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.

Running the Portfolio Demo Sample Application

Your DataStax Community (DSC) installation can run the Portfolio Demo Sample Application. This demo showcases using Apache Cassandra in a real-time web application developed in Java and using the Cassandra Query Language (CQL) JDBC driver.

The demo application is located in:

- **Binary Tarball Installs:** `<install_location>/demos/portfolio_manager`
- **RHEL or Debian Packaged Installs:** `/usr/share/dse-demos/portfolio_manager`

About the Portfolio Demo Use Case

The Portfolio demo is a financial application where users can actively create and manage a portfolio of stocks. Each portfolio contains a list of stocks, the number of shares purchased, and the price at which the shares were purchased. An overall value is maintained for each stock portfolio as well as the percentage of gain or loss compared to the original stock purchase prices for a portfolio.
The demo has a pricer utility that simulates an active feed of live stock market data. For each stock ticker symbol, the application tracks the current stock price and the historical market data (end-of-day price) for each stock for 100 days.

**Running the Demo Web Application**

**Prerequisites**

- The DataStax Community Edition package is installed using either the binary install or a packaged install (see [Introduction to Apache Cassandra and DataStax Community Edition](#) or [Installing a Multiple Node Cluster](#)).
- Your Cassandra cluster is configured and running.

1. You might need to configure the demo application to point to the correct Cassandra installation location. For example, if DataStax Community Edition is installed in a directory called `datastax` in your home directory:

   ```sh
   $ sed -i -e "s,/usr/share/cassandra,~/datastax,"
   ~/datastax/demos/portfolio_manager/bin/pricer
   
   *If you copy the above text, be sure to delete any extra spaces.*
   ```

2. Go to the Portfolio Manager demo directory.

   - **Binary Installs:**
     ```sh
     $ cd <install_location>/demos/portfolio_manager
     
     **Packaged Installs:**
     $ cd /usr/share/dse-demos/portfolio_manager
   ```

   **Note**
   You must run the `pricer` utility from a directory where you have write permissions (such as your home directory), or else run it as root or using `sudo`.

3. Run the `pricer` utility to generate stock data for the application. To see the available options:

   ```sh
   $ bin/pricer --help
   ```

4. To generate 100 days worth of historical data:

   If running on a single node cluster on localhost:

   ```sh
   bin/pricer -o INSERT_PRICES
   bin/pricer -o UPDATE_PORTFOLIOS
   bin/pricer -o INSERT_HISTORICAL_PRICES -n 100
   ```

4. Start the web service:

   ```sh
   $ cd <install_location>/demos/portfolio_manager/website
   $ java -jar start.jar &
   ```

5. Open a browser and go to:

   - http://localhost:8983/portfolio (if running on the local machine)
   - http://<webhost_ip>:8983/portfolio (if running remotely - specify the correct IP address of the remote server).

This open the Portfolio Manager demo web application home page.
Exploring the Sample Data Model

The sample data for the demo application is contained in a Cassandra keyspace called `PortfolioDemo`.

In that keyspace are four column families:

- **Portfolio** - One row per portfolio where the column names are the stock ticker symbols and the column values are the current stock price.
- **StockHist** - One row per stock ticker symbol with (time-ordered) dates for the column names and column values are the end-of-day price for a particular day.
- **Stocks** - One row per stock ticker symbol with a static column name `price` and the column value is the current stock value.
- **HistLoss** - One row per stock ticker symbol where the column name is the worst date in the stock's history in the form of YYYY-MM-DD and the column value is the loss dollar amount.

Viewing the Schema Definitions in Cassandra-CLI

The `cassandra-cli` program is a command-line interface for Cassandra, which you can use to explore the `PortfolioDemo` keyspace and data model.

1. Start `cassandra-cli` and specify a Cassandra node to connect to. For example:
   
   ```
   $ cd <install_location>
   $ bin/cassandra-cli -h localhost
   ```

2. Specify the keyspace you want to connect to:
   
   ```
   [default@unknown] USE PortfolioDemo;
   ```
3. To see the keyspace and column family schema definitions:
   
   [default@unknown] SHOW SCHEMA;

4. To select a row from the Stocks column family (by specifying the row key value of a stock ticker symbol):
   
   [default@unknown] GET Stocks[GCO];

5. To exit cassandra-cli:
   
   [default@unknown] exit;

---

**DataStax Community Release Notes**

**What's New**

As of Cassandra 1.0.9, the install now includes the Portfolio Manager sample application. Added new platform support (Windows 7 and Windows 2008 Server, both 32- and 64-bit) for Cassandra development. The Windows MSI installer provides a full install, including OpsCenter, sets all the WIN services, creates a Windows program group, and quickly starts the new version of DataStax OpsCenter (2.0).

**Prerequisites**


**The Install Types**

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

- If you have root permissions or sudo, see *Installing the DataStax Community Binaries on Linux or Mac OSX as Root*.
- If you do not have root permissions or sudo, see *Installing the DataStax Community Binaries as User*.
- If you want to install on Windows, see *Installing the DataStax Community Binaries on Windows*.

After installing and starting Cassandra, run the *demo application* to see Cassandra in action.

**Learning about Cassandra**

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

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

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

**About Data Partitioning in Cassandra**

Data partitioning determines how data is distributed across the nodes in the cluster. Three factors are involved with data distribution:

- A partitioner that determines which node to store the data on.
- The number of copies of data, which is determined by the *replica placement strategy*.
- The topology of the cluster, which is the number of nodes, the distribution of the nodes on racks, and the number of data centers.

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.

**Understanding the Partitioner Types**

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

Cassandra offers the following partitioners:
About the RandomPartitioner

The RandomPartitioner (org.apache.cassandra.dht.RandomPartitioner) is the default partitioning strategy for a Cassandra cluster, and in almost all cases is the right choice.

The RandomPartitioner uses consistent hashing to determine which node stores which 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 effected.

To distribute the data evenly across the number of nodes, a hashing algorithm creates an MD5 hash value of the row key. The maximum possible range of hash values is 0 to $2^{127} - 1$. Each node in the cluster is assigned a token that represents a hash value within this range and then owns the rows with a hash value less than its token number. 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 ensures that read and write requests to the cluster are evenly distributed. Another benefit of using random partitioning is that it simplifies load balancing a cluster. Because each part of the hash range receives an equal number of rows on average, it is easier to correctly assign tokens to new nodes.

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

About the ByteOrderedPartitioner

Cassandra provides the ByteOrderedPartitioner (org.apache.cassandra.dht.ByteOrderedPartitioner) for ordered partitioning. (The OrderPreservingPartitioner and CollatingOrderPreservingPartitioner are deprecated as of Cassandra 0.7.) This partitioner orders rows lexically by key bytes. You calculate tokens by looking at the actual values of your row key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you could assign an A token using its hexadecimal representation of 41.

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

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

- **Sequential writes can cause hot spots**: If your application tends to write or update a sequential block of rows at a time, then these writes are not distributed across the cluster; they all go to one node. This is frequently a problem for applications dealing with timestamped data.

- **More administrative overhead to load balance the cluster**: An ordered partitioner requires administrators to manually calculate token ranges based on their estimates of the row key distribution. In practice, this requires actively moving node tokens around to accommodate the actual distribution of data once it is loaded.

- **Uneven load balancing for multiple column families**: If your application has multiple column families, chances are that those column families have different row keys and different distributions of data. An ordered partitioner than is balanced for one column family may cause hot spots and uneven distribution for another column family in the same cluster.

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

\textbf{Uneven Data Distribution}

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

\textbf{Even Data Distribution}
**About Replication in Cassandra**

Replication is the process of storing copies of data on multiple nodes to ensure reliability and fault tolerance. Cassandra stores copies, called replicas, of each row based on the row key. You set the number of replicas when you create a keyspace using the replica placement strategy. In addition to setting the number of replicas, this strategy sets the distribution of the replicas across the nodes in the cluster depending on the cluster's topology.

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

To determine the physical location of nodes and their proximity to each other, the replication strategy also relies on the cluster-configured snitch, which is described below.

**Replication Strategy**

The available strategies are:

- `SimpleStrategy`
- `NetworkTopologyStrategy`

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

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

---

**NetworkTopologyStrategy**

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

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

- **Two replicas in each data center.** This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of **ONE**.

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

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

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

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

- Additional replicas are placed by walking the ring clockwise until a node in a different rack is found. If no such node exists, additional replicas are placed in different nodes in the same rack.
NetworkTopologyStrategy attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) can fail at the same time due to power, cooling, or network issues.

Below is an example of how NetworkTopologyStrategy places replicas spanning two data centers with a total replication factor of 4. When using NetworkTopologyStrategy, you set the number of replicas per data center.

In the following graphic, notice the tokens are assigned to alternating racks. For more information, see Calculating Tokens for a Multiple Data Center Cluster.
NetworkTopologyStrategy relies on a properly configured snitch to place replicas correctly across data centers and racks. It is important to configure your cluster to use the type of snitch that correctly determines the locations of nodes in your network.

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

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

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

The following snitches are available:

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

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

**DseSimpleSnitch**
For information about this snitch, see DataStax Enterprise 2.1 documentation.

**RackInferringSnitch**

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

![Image showing IP address and octets]

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

**PropertyFileSnitch**

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

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

**EC2Snitch**

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

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

**EC2MultiRegionSnitch**

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

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

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

```yaml
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.

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

**About Dynamic Snitching**

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

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

**About Client Requests in Cassandra**

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

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

**About Write Requests**

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

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

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

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

The process is 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.

**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 bottlenecks.

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

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

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

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

**Planning an Amazon EC2 Cluster**
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:

\[
raw\_capacity = disk\_size \times number\_of\_disks
\]

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:

\[
\frac{(raw\_capacity \times 0.9)}{2} = formatted\_disk\_space
\]

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%):

\[
formatted\_disk\_space \times 0.5 = usable\_disk\_space
\]

### 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):

\[
primary\_key\_index = number\_of\_rows \times (32 + average\_key\_size)
\]

- **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.

\[
replication\_overhead = total\_data\_size \times (replication\_factor - 1)
\]

### 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 node’s participation in a cluster and how the node is known to the cluster.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster_name</td>
<td>Name of the cluster that this node is joining. Should be the same for every node in the cluster.</td>
</tr>
<tr>
<td>listen_address</td>
<td>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.</td>
</tr>
<tr>
<td>seeds</td>
<td>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.</td>
</tr>
<tr>
<td>storage_port</td>
<td>The intra-node communication port (default is 7000). Should be the same for every node in the cluster.</td>
</tr>
<tr>
<td>initial_token</td>
<td>The initial token is used to determine the range of data this node is responsible for.</td>
</tr>
</tbody>
</table>

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. The location of this file depends on the type of installation; see Cassandra Configuration Files Locations or DataStax Enterprise Configuration Files Locations.

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*):

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

Or for a multi-data center cluster:

```bash
[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 Multiple Node Cluster

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

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:
   ```
   ```

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 dsc-1.0.10
```

This installs the DataStax Community distribution of Cassandra, DataStax Community demos, 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. Add the DataStax Community repository to the aptitude repository source list file (`/etc/apt/sources.list`).

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

3. (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.

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

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

5. If needed, install the Python CQL driver.

   ```
   $ sudo apt-get update
   $ sudo apt-get install python-cql=1.0.10-1
   ```

6. Install the package and Python CQL driver.

   ```
   $ sudo apt-get update
   $ sudo apt-get install dsc=1.0.10 cassandra=1.0.10
   ```

   This installs the Cassandra, DataStax Community demos, and OpsCenter packages. 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. 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 Oracle JRE.

2. Download the Cassandra DataStax Community tarball:
   
   `$ curl -OL http://downloads.datastax.com/community/dsc-cassandra-1.0.9-bin.tar.gz`

3. Unpack the distribution:
   
   `$ tar -xvzf dsc-cassandra-1.0.9-bin.tar.gz`
   `$ rm *.tar.gz`

4. By default, Cassandra installs files into the /var/lib/cassandra and /var/log/cassandra directories. If you do not have root access to the default directories, ensure you have write access as follows:
   
   `$ sudo mkdir /var/lib/cassandra`
   `$ sudo mkdir /var/log/cassandra`
   `$ sudo chown -R $USER:$GROUP /var/lib/cassandra`
   `$ sudo chown -R $USER:$GROUP /var/log/cassandra`

Next Steps

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

Recommended Settings for Production Installations

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

File Descriptors

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

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

**User Resource Limits**

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

- `* soft nofile 32768`
- `* hard nofile 32768`
- `root soft nofile 32768`
- `root hard nofile 32768`
- `* soft memlock unlimited`
- `* hard memlock unlimited`
- `root soft memlock unlimited`
- `root hard memlock unlimited`
- `* soft as unlimited`
- `* hard as unlimited`
- `root soft as unlimited`
- `root hard as unlimited`

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

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

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

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

- `* soft nproc 10240`

For more information, see Insufficient User Resource Limits Errors.

**Disable Swap**

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

```
sudo swapoff --all
```

For more information, see Nodes seem to freeze after some period of time.

**Synchronize Clocks**

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

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

**Optimum blockdev --setra Settings for RAID**

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

Check to ensure setra is not set to 65536:
Installing a Multiple Node Cluster

sudo blockdev --report /dev/<device>

To set setra:

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

**Installing the JRE and JNA**

Cassandra is a 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'.
```

<table>
<thead>
<tr>
<th>Selection</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/usr/lib/jvm/jre-1.6.0-openjdk.x86_64/bin/java</td>
</tr>
<tr>
<td>*+ 2</td>
<td>/usr/java/jre1.6.0_32/bin/java</td>
</tr>
</tbody>
</table>

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](https://www.oracle.com/technetwork/java/javase/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**

---

33
Installing a Multiple Node Cluster

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](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 Installions

1. Download `jna.jar` from the [GitHub](https://github.com).
2. Add `jna.jar` to `<install_location>/resources/dse/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**

For instructions on installing the DataStax AMI (Amazon Machine Image), see the latest [AMI documentation](https://docs.datastax.com/en/amis/).

**Expanding a Cassandra AMI Cluster**

For instructions on expanding the DataStax AMI (Amazon Machine Image), see the latest [AMI documentation](https://docs.datastax.com/en/amis/).

**Upgrading Cassandra**

This section describes how to upgrade Cassandra 0.8.x to 1.0.x and how to upgrade between minor releases of Cassandra 1.0.x. The procedures also apply to DataStax Community Edition.

**Best Practices for Upgrading Cassandra**

The following steps are recommended before upgrading Cassandra:

- Take a snapshot before the upgrade. This allows you to rollback to the previous version if necessary. Cassandra is able to read data files created by the previous version, but the inverse is not always true.

  Taking a snapshot is fast, especially if you have JNA installed, and takes effectively zero disk space until you start compacting the live data files again.
Upgrading Cassandra: 0.8.x to 1.0.x

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 installation from 0.8.x to 1.0.x:

1. Save the cassandra.yaml file from the old installation to a safe place.
2. On each node, download and unpack the 1.0 binary tarball package from the downloads section of the Cassandra website.
3. In the new installation, open the cassandra.yaml for writing.
4. In the old installation, open the cassandra.yaml.
5. Diff the new and old cassandra.yaml files.
6. Merge the diffs by hand from the old file into the new one.
7. Follow steps for completing the upgrade.

To upgrade a CentOS/RHEL packaged release installation from 0.8.x to 1.0.x:

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

To upgrade a Debian/Ubuntu packaged release installation from 0.8.x to 1.0.x:

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

Completing the Upgrade

To complete the upgrade, perform the following steps:

1. Account for New and Changed Parameters between 0.8 and 1.0 in cassandra.yaml.
2. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
3. Run `nodetool drain` on the node to flush the commit log.
4. Stop the old Cassandra process, then start the new binary process.
5. Monitor the log files for any issues.
6. After upgrading and restarting all Cassandra processes, restart client applications.
7. After upgrading, run `nodetool upgradesstables` against each node before running repair, moving nodes, or adding new ones. If you are using Cassandra 1.0.3 and earlier, use `nodetool scrub` instead of nodetool upgradesstables.

Upgrading Between Minor Releases of Cassandra 1.0.x
The upgrade procedure between minor releases of Cassandra 1.1.x is identical to the upgrade procedure between major releases with one exception: Do not perform the last step of *Completing the Upgrade* to run nodetool upgradesstables or nodetool scrub after upgrading.

**New and Changed Parameters between 0.8 and 1.0**

This table lists cassandra.yaml parameters that have changed between 0.8 and 1.0. See the cassandra.yaml reference for details about these parameters.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.0 Release</strong></td>
<td></td>
</tr>
<tr>
<td>broadcast_address</td>
<td>Same as listen_address - set to the public IP in multi-region EC2 clusters</td>
</tr>
<tr>
<td>compaction_thread_priority</td>
<td>Removed (use compaction_throughput_mb_per_sec instead)</td>
</tr>
<tr>
<td>commitlog_rotation_threshold_in_mb</td>
<td>Removed</td>
</tr>
<tr>
<td>commitlog_total_space_in_mb</td>
<td>4096 (replaces column family storage property memtable_flush_after_mins)</td>
</tr>
<tr>
<td>multithreaded_compaction</td>
<td>false</td>
</tr>
<tr>
<td>memtable_total_space_in_mb</td>
<td>1/3 of heap (replaces column family storage properties memtable_operations_in_millions and memtable_throughput_in_mb)</td>
</tr>
<tr>
<td><strong>0.8 Release</strong></td>
<td></td>
</tr>
<tr>
<td>seed_provider</td>
<td>SimpleSeedProvider</td>
</tr>
<tr>
<td>seeds</td>
<td>Now a comma-delimited list in double quotes.</td>
</tr>
<tr>
<td>memtable_total_space_in_mb</td>
<td>1/3 of heap</td>
</tr>
<tr>
<td>compaction_throughput_mb_per_sec</td>
<td>16</td>
</tr>
<tr>
<td>concurrent_compactors</td>
<td>One per CPU</td>
</tr>
<tr>
<td>internode_encryption</td>
<td>none</td>
</tr>
<tr>
<td>keystore</td>
<td>conf/.keystore</td>
</tr>
<tr>
<td>keystore_password</td>
<td>cassandra</td>
</tr>
<tr>
<td>truststore</td>
<td>conf/.truststore</td>
</tr>
<tr>
<td>truststore_password</td>
<td>cassandra</td>
</tr>
</tbody>
</table>

**Initializing a Cassandra Cluster**

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

**Initializing a Multiple Node Cluster in a Single Data Center**

In this scenario, data replication is distributed across a single data center. Data replicates across the data centers automatically and transparently; no ETL work is necessary to move data between different systems or servers. You can configure the number of copies of the data in each data center and Cassandra handles the rest, replicating the data for you. To configure a multiple data center cluster, see Initializing Multiple Data Center Clusters on Cassandra.
Note
In Cassandra, the term data center is a grouping of nodes. Data center is synonymous with replication group, that is, a grouping of nodes configured together for replication purposes. The data replication protects against hardware failure and other problems that cause data loss in a single cluster.

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

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

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

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

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

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

To configure a mixed-workload cluster:
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*.

<table>
<thead>
<tr>
<th>Node</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>0</td>
</tr>
<tr>
<td>node1</td>
<td>28356863910078205288614550619314017621</td>
</tr>
<tr>
<td>node2</td>
<td>56713727820156410577229101238628035242</td>
</tr>
<tr>
<td>node3</td>
<td>85070591730234615865843651857942052864</td>
</tr>
<tr>
<td>node4</td>
<td>113427455640312821154458202477256070485</td>
</tr>
<tr>
<td>node5</td>
<td>141784319550391026443072753096570088106</td>
</tr>
</tbody>
</table>

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

4. Stop the nodes and clear the data.
   - For packaged installs, run the following commands:
     - `$ sudo service cassandra stop` *(stops the service)*
     - `$ sudo rm -rf /var/lib/cassandra/*` *(clears the data from the default directories)*
   - For binary installs, run the following commands from the install directory:
     - `$ ps auwx | grep cassandra` *(finds the Cassandra Java process ID [PID])*
     - `$ sudo kill <pid>` *(stops the process)*
     - `$ sudo rm -rf /var/lib/cassandra/*` *(clears the data from the default directories)*
5. Modify the following property settings in the cassandra.yaml file for each node:

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

**node0**

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

**node1 to node5**

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

**node1**

initial_token: 28356863910078205288614550619314017621
listen_address: 110.82.155.1

**node2**

initial_token: 56713727820156410577229101238628035242
listen_address: 110.82.155.2

**node3**

initial_token: 85070591730234615865843651857942052864
listen_address: 110.82.155.3

**node4**

initial_token: 113427455640312821154458202477256070485
listen_address: 110.82.155.4

**node5**

initial_token: 141784319550391026443072753096570088106
listen_address: 110.82.155.5
6. After you have installed and configured Cassandra on all nodes, start the seed nodes one at a time, and then start
the rest of the nodes.

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

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

7. Check that your ring is up and running:

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

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

### Initializing Multiple Data Center Clusters on Cassandra

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

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

### Prerequisites

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

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

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

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

**Configuration Example**

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

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

**Note**

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

To configure a cluster with multiple data centers:

1. Suppose you install Cassandra on these nodes:
   
   10.168.66.41 (seed1)  
   10.176.43.66  
   10.168.247.41  
   10.176.170.59 (seed2)  
   10.169.61.170  
   10.169.30.138

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

<table>
<thead>
<tr>
<th>Node</th>
<th>IP Address</th>
<th>Token</th>
<th>Offset</th>
<th>Data Center</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>10.168.66.41</td>
<td>0</td>
<td>NA</td>
<td>DC1</td>
</tr>
<tr>
<td>node1</td>
<td>10.176.43.66</td>
<td>56713727820156410577229101238628035242</td>
<td>NA</td>
<td>DC1</td>
</tr>
<tr>
<td>node2</td>
<td>10.168.247.41</td>
<td>113427455640312821154458202477256070485</td>
<td>NA</td>
<td>DC1</td>
</tr>
<tr>
<td>node3</td>
<td>10.176.170.59</td>
<td>10</td>
<td>10</td>
<td>DC2</td>
</tr>
<tr>
<td>node4</td>
<td>10.169.61.170</td>
<td>56713727820156410577229101238628035252</td>
<td>10</td>
<td>DC2</td>
</tr>
<tr>
<td>node5</td>
<td>10.169.30.138</td>
<td>113427455640312821154458202477256070495</td>
<td>10</td>
<td>DC2</td>
</tr>
</tbody>
</table>

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

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

   **node1 to node5**
   
   The properties for the rest of the nodes are the same as **Node0** except for the initial_token and listen_address:

5. Determine a naming convention for each data center and rack, for example: DC1, DC2 or 100, 200 and RAC1, RAC2 or R101, R102.

6. In the cassandra-topology.properties file, assign data center and rack names to the IP addresses of each node. For example:

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

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

   # default for unknown nodes
   default=DC1:RAC1
8. After you have installed and configured Cassandra on all nodes, start the seed nodes one at a time, and then start the rest of the nodes.

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

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

9. Check that your ring is up and running:

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

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

### Balancing the Data Center Nodes

When you deploy a Cassandra cluster, you need to use the *partitioner* to distribute roughly an equal amount of data to nodes. You also use identifiers for each data center (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: 

\[
\text{token} = \left( \frac{2^{127}}{\text{num_nodes_in_dc}} \times n + \text{DC_ID} \right)
\]

where \( n \) is the node for which the token is being calculated and \( \text{DC_ID} \) is the numerical name.

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

Token Generating Tool
DataStax provides a Python program for generating tokens using the maximum possible range (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": 2835686391007820528861455061931407621,
    "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. As a best practice, each rack should have the same number of nodes so you can alternate the rack assignments, for example: rack1, rack2, rack3, rack1, rack2, rack3, and so on.

   ./tokengentool 8

The tool displays the token for each node. The graphic shows the rack assignments:
Calculating Tokens for a Multiple Data Center Cluster

In multiple data center deployments, use NetworkTopologyStrategy for replica placement. This strategy determines replica placement independently within each data center. For more detailed information, see NetworkTopologyStrategy.

You can use when different methods for calculating the tokens in 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. Two manual methods are recommended:

- **Alternate token assignments.** This method works best with data centers that have equal numbers of nodes in each data center.
- **Offset token values.** This method works with data centers that have different number of nodes in each data center (and data centers of the same size).

**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. In the tool, enter the number of nodes for each data center.

`.tokengentool 3 3`

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

```
{
    "0": {  
        "0": 0,
        "1": 56713727820156410577229101238628035242,
    },
    "1": {
        "0": 56713727820156410577229101238628035242,
    },
    "2": {
        "0": 56713727820156410577229101238628035242,
    }
}
```
The following image shows the token position and data center assignments:

**Offsetting Token Assignments**

To avoid token collisions offset the values use an offset of +100; this allows room to replace a dead node.

The following graphic shows a cluster with two 3 node data centers and one 2 node data center:

```
./tokengentool 3
{
  "0": { 
    "0": 0,
    "1": 567137278201564105577229101238628035242,
    "2": 1134274556403128202477256070485
  }
}
./tokentool 2
```
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, and so on.). 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 categoryid 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".

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?
- Show me all of the blog entries about fashion.
- Show me the most recent entries for the blogs I subscribe to.

You 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:

```cql
CREATE KEYSPACE keyspace_name
WITH strategy_class = 'SimpleStrategy'
AND strategy_options:replication_factor='2';
```
Or in Cassandra CLI:

```
CREATE KEYSPACE keyspace_name WITH
placement_strategy = 'SimpleStrategy'
AND strategy_options = {replication_factor:2};
```

See *Getting Started Using the Cassandra CLI* and *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.

Although column families are very flexible, in practice a column family is not entirely schema-less. Each column family should be designed to contain a single type of data. There are two typical column family design patterns in Cassandra; the static and dynamic column families.

A static column family uses a relatively static set of column names and is more 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 will 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.

![Diagram of column families](image)

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.
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.

For all column families, each row is uniquely identified by its row key, similar to the primary key in a relational table. A column family is always partitioned on its row key, and the row key is always implicitly indexed. Empty row keys are not allowed.

**About 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.

**About Special Columns (Counter, Expiring, Super)**

Cassandra has three special types of columns, described below:

**About 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.

You can use either CLI or CQL to set the TTL for a column. See Setting an Expiring Column and Specifying Column Expiration with TTL.

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.

About 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.

About 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).

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

**AboutValidators**

For all column families, it is best practice to define a default row key validator using the *key_validation_class* property.

For static column families, you should define each column and its associated type when you define the column family using the *column_metadata* property.

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.
Understanding the Cassandra Data Model

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 Comparators

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.

About Column Family Compression

Data compression can be configured on a per-column family basis. 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.

Besides reducing data size, compression typically improves both read and write performance. Cassandra is able to quickly find the location of rows in the SSTable index, and only decompresses the relevant row chunks. This means compression improves read performance not just by allowing a larger data set to fit in memory, but it also benefits workloads where the hot data set does not fit into memory.

Unlike in traditional databases, write performance is not negatively impacted by compression in Cassandra. Writes on compressed tables can in fact show up to a 10 percent performance improvement. In traditional relational databases, writes require overwrites to existing data files on disk. This means that the database has to locate the relevant pages on disk, decompress them, overwrite the relevant data, and then compress them again (an expensive operation in both 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 only compressed once, when they are written to disk.

Enabling compression can yield the following benefits, depending on the data characteristics of the column family:

- 2x-4x reduction in data size
- 25-35% performance improvement on reads
- 5-10% performance improvement on writes

When to Use Compression

Compression is best suited for column families where there are many rows, with each row having the same columns, or at least many columns in common. For example, a column family containing user data such as username, email, etc., would be a good candidate for compression. The more similar the data across rows, the greater the compression ratio will be, and the larger the gain in read performance.

Compression is not as a good fit for column families where each row has a different set of columns, or where there are just a few very wide rows. Dynamic column families such as this will not yield good compression ratios.

Configuring Compression on a Column Family

When you create or update a column family, you can choose to make it a compressed column family by setting the compression_options attributes.

You can enable compression when you create a new column family, or update an existing column family to add compression later on. When you add compression to an existing column family, existing SSTables on disk are not
compressed immediately. Any new SSTables that are created will be compressed, and any existing SSTables will be compressed during the normal Cassandra compaction process. If necessary, you can force existing SSTables to be rewritten and compressed by using \texttt{nodetool upgradesstables} (Cassandra 1.0.4 or later) or \texttt{nodetool scrub}.

For example, to create a new column family with compression enabled using the Cassandra CLI, you would do the following:

\begin{verbatim}
[default@demo]$ CREATE COLUMN FAMILY users WITH key_validation_class=UTF8Type AND column_metadata = [
{column_name: 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}
] AND compression_options={sstable_compression:SnappyCompressor, chunk_length_kb:64};
\end{verbatim}

### 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 \texttt{partitioner} and the keyspace-configured \texttt{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 \texttt{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 \texttt{KEYS} (similar to a hash index).

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

#### When to Use Secondary Indexes

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.

#### Building and Using Secondary Indexes

An 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.

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):

```cql
[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:

```cql
[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:

```cql
[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 way data is structured 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.

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. The bloom filter is used to check 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, also see *About Client Requests in Cassandra.*

**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.

**About Transactions and Concurrency Control**

Unlike relational databases, Cassandra does not offer fully ACID-compliant transactions. There is no locking or transactional dependencies when concurrently updating multiple rows or column families.

ACID is an acronym used to describe transactional behavior in a relational database systems, which stands for:

- **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.

Cassandra trades transactional isolation and atomicity for high availability and fast write performance. 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.

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 before the write is forwarded to it, or if it fails to acknowledge the write for any reason, the coordinator will store a hint for it. The hint consists of the target replica, as well as the mutation to be replayed.

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

**Note**

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

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

When a replica that is storing hints detects via gossip that the failed node is alive again, it will begin streaming the missed writes to catch up the out-of-date replica.
Note
Hinted handoff does not completely replace the need for regular node repair operations. In addition to the time set by `max_hint_window_in_ms`, the coordinator node storing hints could fail before replay. You should always run a full repair after losing a node or disk.

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 is used to check 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). See Tuning the Cache for more information about optimizing read performance using the built-in caching features.

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.

Tunable Consistency for Client Requests

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

About Write Consistency

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

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

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

\[
\text{quorum} = \left\lfloor \frac{\text{replication_factor}}{2} \right\rfloor + 1
\]

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).

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>Least consistent but highest availability.</td>
</tr>
<tr>
<td>ALL</td>
<td>Highest consistent but lowest availability.</td>
</tr>
<tr>
<td>QUORUM</td>
<td>Good middle-ground ensuring strong consistency, yet still tolerating some level of failure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMARY</td>
<td>Default consistency level when no consistency level is specified.</td>
</tr>
<tr>
<td>LOCAL</td>
<td>All writes to a single node.</td>
</tr>
<tr>
<td>LOCAL_QUORUM</td>
<td>All writes to a quorum of nodes.</td>
</tr>
<tr>
<td>LOCAL_ONE</td>
<td>All writes to a single node and a quorum of nodes.</td>
</tr>
<tr>
<td>QUORUM</td>
<td>Consistency level that is a good middle-ground ensuring strong consistency, yet still tolerating some level of failure.</td>
</tr>
</tbody>
</table>

60
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.

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):

\[
\left\lfloor \frac{\text{replication_factor}}{2} \right\rfloor + 1
\]

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).

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>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.</td>
</tr>
<tr>
<td>QUORUM</td>
<td>Returns the record with the most recent timestamp once a quorum of replicas has responded.</td>
</tr>
<tr>
<td>LOCAL_QUORUM</td>
<td>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.</td>
</tr>
<tr>
<td>EACH_QUORUM</td>
<td>Returns the record with the most recent timestamp once a quorum of replicas in each data center of the cluster has responded.</td>
</tr>
<tr>
<td>ALL</td>
<td>Returns the record with the most recent timestamp once all replicas have responded. The read operation will fail if a replica does not respond.</td>
</tr>
</tbody>
</table>

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:

\[(\text{nodes_written} + \text{nodes_read}) > \text{replication_factor}\]

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:

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

**About Cassandra’s Built-In Consistency Repair Features**

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

- **Read Repair** - For reads, there are two types of read requests that a coordinator can send to a replica; a direct read request and a background read repair request. The number of replicas contacted by a direct read request is determined by the consistency level specified by the client. Background read repair requests are sent to any additional replicas that did not receive a direct request. To ensure that frequently-read data remains consistent, the coordinator compares the data from all the remaining replicas that own the row in the background, and if they are inconsistent, issues writes to the out-of-date replicas to update the row to reflect the most recently written values. Read repair can be configured per column family (using `read_repair_chance`), and is enabled by default.

- **Anti-Entropy Node Repair** - For data that is not read frequently, or to update data on a node that has been down for a while, the `nodetool repair` process (also referred to as anti-entropy repair) ensures that all data on a replica is made consistent. Node repair should be run routinely as part of regular cluster maintenance operations.
- **Hinted Handoff** - Writes are always sent to all replicas for the specified row regardless of the consistency level specified by the client. If a node happens to be down at the time of write, its corresponding replicas will save hints about the missed writes, and then handoff the affected rows once the node comes back online. Hinted handoff ensures data consistency due to short, transient node outages. The hinted handoff feature is configurable at the node-level in the `cassandra.yaml` file. 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.
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.

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.

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:

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

To increment the counter:

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:

```
[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 KEYSSPACE demo;
```
Getting Started with CQL

Developers can access CQL commands in a variety of ways. Drivers are available for Python, Twisted Python, and JDBC-based client programs.

For the purposes of administrators, the most direct way to run simple CQL commands is via the Python-based cqlsh command-line client.

Starting the CQL Command-Line Program (cqlsh)

As of DataStax Community version 1.0.1, the cqlsh client is installed with Cassandra in <install_location>/bin for tarball installations, or /usr/bin for packaged installations.

When you start cqlsh, you must provide the IP of a Cassandra node to connect to (default is localhost) and the RPC connection port (default is 9160). For example:

```
$ cqlsh 103.263.89.126 9160
```

to exit cqlsh type exit at the command prompt.

```
cqlsh> exit
```

Running CQL Commands with cqlsh

Commands in cqlsh combine SQL-like syntax that maps to Cassandra concepts and operations. If you are just getting started with CQL, make sure to refer to the CQL Reference.

As of CQL version 2.0, cqlsh has the following limitations in support for Cassandra operations and data objects:

- Super Columns are not supported; column_type and subcomparator arguments are not valid
- Composite columns are not supported
- Only a subset of all the available column family storage properties can be set using CQL.

The rest of this section provides some guidance with simple CQL commands using cqlsh. This is a similar (but not identical) set of commands as the set described in Using the Cassandra Client.

Creating a Keyspace

You can use the cqlsh commands described in this section to create a keyspace. In creating an example keyspace for Twissandra, we will assume a desired replication factor of 1 and implementation of the SimpleStrategy replica placement strategy. For more information on these keyspace options, see About Replication in Cassandra.

Note the single quotes around the string value of strategy_class:

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

Creating a Column Family

For this example, we use cqlsh to create a users column family in the newly created keyspace. Note the USE command to connect to the twissandra keyspace.

```
cqlsh> USE twissandra;

cqlsh> CREATE COLUMNFAMILY users (
          ... KEY varchar PRIMARY KEY,
          ... password varchar,
          ... gender varchar,
```
Inserting and Retrieving Columns

Though in production scenarios it is more practical to insert columns and column values programatically, it is possible to use cqlsh for these operations. The example in this section illustrates using the INSERT and SELECT commands to insert and retrieve some columns in the users column family.

The following commands create and then get a user record for "jsmith." The record includes a value for the password column we created when we created the column family, as well as an expiration time for the password column. Note that the user name "jsmith" is the row key, or in CQL terms, the primary key.

```cqlsh
INSERT INTO users (KEY, password) VALUES ('jsmith', 'ch@ngem3a') USING TTL 86400;
SELECT * FROM users WHERE KEY='jsmith';
```

Adding Columns with ALTER COLUMNFAMILY

The ALTER COLUMNFAMILY command lets you add 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

With ALTER COLUMNFAMILY, you can change the type of a column any time after it is defined or added to a column family. For example, if we decided the coupon_code column should store coupon codes in the form of integers, we could change the validation type as follows:

```cqlsh
ALTER TABLE users ALTER coupon_code TYPE int;
```

Note that existing coupon codes will not be validated against the new type, only newly inserted values.

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
UPDATE users USING TTL 432000 SET 'password' = 'ch@ngem3a' WHERE KEY = 'jsmith';
```

Dropping Column Metadata

If your aim is to remove a column’s metadata entirely, including the column name and validation type, you can 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 they will not be validated until you explicitly add a type again.

Indexing a Column
cqlsh can be used to create secondary indexes, or indexes on column values. In this example, we will create an index on the state and birth_year columns in the users column family.

```cqlsh
CREATE INDEX state_key ON users (state);
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. In this example we will delete user jsmith's session token column, and then delete jsmith's row entirely.

```cqlsh
DELETE session_token FROM users where KEY = 'jsmith';
DELETE FROM users where KEY = 'jsmith';
```

Note, however, that the phenomena called "range ghosts" in Cassandra may mean that keys for deleted rows are still retrieved by SELECT statements and other "get" operations. Deleted values, including range ghosts, are removed completely by the first compaction following deletion.

Dropping Column Families and Keyspaces

With 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 twissandra keyspace altogether:

```cqlsh
DROP COLUMNFAMILY users;
DROP KEYSPACE twissandra;
```

Configuration

Like any modern server-based software, Cassandra has a number of configuration options to tune the system towards specific workloads and environments. Substantial efforts have been made to provide meaningful default configuration values, but given the inherently complex nature of distributed systems coupled with the wide variety of possible workloads, most production deployments will require some modifications of the default configuration.

Node and Cluster Configuration (cassandra.yaml)

The cassandra.yaml file is the main configuration file for Cassandra. It is located in the following directories:

- Cassandra packaged installs: /etc/cassandra/conf
- Cassandra binary installs: <install_location>/conf
- DataStax Enterprise packaged installs: /etc/dse/cassandra
- DataStax Enterprise binary installs: <install_location>/resources/cassandra/conf

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticator</td>
<td>org.apache.cassandra.auth.AllowAllAuthenticator</td>
</tr>
<tr>
<td>authority</td>
<td>org.apache.cassandra.auth.AllowAllAuthority</td>
</tr>
<tr>
<td><strong>broadcast_address</strong></td>
<td>same as <strong>listen_address</strong></td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td><strong>cluster_name</strong></td>
<td><strong>Test Cluster</strong></td>
</tr>
<tr>
<td><strong>column_index_size_in_kb</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>commitlog_directory</strong></td>
<td>/var/lib/cassandra/commitlog</td>
</tr>
<tr>
<td><strong>commitlog_sync</strong></td>
<td>periodic</td>
</tr>
<tr>
<td><strong>commitlog_sync_period_in_ms</strong></td>
<td>10000 (ten seconds)</td>
</tr>
<tr>
<td><strong>commitlog_total_space_in_mb</strong></td>
<td>4096</td>
</tr>
<tr>
<td><strong>compaction_preheat_key_cache</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>compaction_throughput_mb_per_sec</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>concurrent_compactors</strong></td>
<td>One per CPU core</td>
</tr>
<tr>
<td><strong>concurrent_reads</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>concurrent_writes</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>data_file_directories</strong></td>
<td>/var/lib/cassandra/data</td>
</tr>
<tr>
<td><strong>dynamic_snitch</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>dynamic_snitch_badness_threshold</strong></td>
<td>0.0</td>
</tr>
<tr>
<td><strong>dynamic_snitch_reset_interval_in_ms</strong></td>
<td>600000</td>
</tr>
<tr>
<td><strong>dynamic_snitch_update_interval_in_ms</strong></td>
<td>100</td>
</tr>
<tr>
<td><strong>endpoint_snitch</strong></td>
<td>org.apache.cassandra.locator.SimpleSnitch</td>
</tr>
<tr>
<td><strong>flush_largest_memtables_at</strong></td>
<td>0.75</td>
</tr>
<tr>
<td><strong>hinted_handoff_enabled</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>hinted_handoff_throttle_delay_in_ms</strong></td>
<td>50</td>
</tr>
<tr>
<td><strong>in_memory_compaction_limit_in_mb</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>incremental_backups</strong></td>
<td>false</td>
</tr>
<tr>
<td><strong>index_interval</strong></td>
<td>128</td>
</tr>
<tr>
<td><strong>initial_token</strong></td>
<td>n/a</td>
</tr>
<tr>
<td><strong>internode_encryption</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>keystore</strong></td>
<td>conf/.keystore</td>
</tr>
<tr>
<td><strong>keystore_password</strong></td>
<td>cassandra</td>
</tr>
<tr>
<td><strong>listen_address</strong></td>
<td>localhost</td>
</tr>
<tr>
<td><strong>max_hint_window_in_ms</strong></td>
<td>3600000 (one hour)</td>
</tr>
<tr>
<td><strong>memtable_flush_queue_size</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>memtable_flush_writers</strong></td>
<td>One per data directory</td>
</tr>
<tr>
<td><strong>memtable_total_space_in_mb</strong></td>
<td>1/3 of the heap</td>
</tr>
<tr>
<td><strong>partitioner</strong></td>
<td>org.apache.cassandra.dht.RandomPartitioner</td>
</tr>
<tr>
<td><strong>phi_convict_threshold</strong></td>
<td>8</td>
</tr>
<tr>
<td><strong>reduce_cache_capacity_to</strong></td>
<td>0.6</td>
</tr>
<tr>
<td><strong>reduce_cache_sizes_at</strong></td>
<td>0.85</td>
</tr>
</tbody>
</table>
### 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**

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**

The name of the cluster. All nodes participating in a cluster must have the same value.

**commitlog_directory**

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**

The path to the data file directories.

---

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_scheduler</td>
<td>org.apache.cassandra.scheduler.NoScheduler</td>
</tr>
<tr>
<td>request_scheduler_id</td>
<td>keyspace</td>
</tr>
<tr>
<td>rpc_address</td>
<td>localhost</td>
</tr>
<tr>
<td>rpc_keepalive</td>
<td>true</td>
</tr>
<tr>
<td>rpc_max_threads</td>
<td>Unlimited</td>
</tr>
<tr>
<td>rpc_min_threads</td>
<td>16</td>
</tr>
<tr>
<td>rpc_port</td>
<td>9160</td>
</tr>
<tr>
<td>rpc_recv_buff_size_in_bytes</td>
<td>n/a</td>
</tr>
<tr>
<td>rpc_send_buff_size_in_bytes</td>
<td>n/a</td>
</tr>
<tr>
<td>rpc_server_type</td>
<td>sync</td>
</tr>
<tr>
<td>rpc_timeout_in_ms</td>
<td>10000</td>
</tr>
<tr>
<td>saved_caches_directory</td>
<td>/var/lib/cassandra/saved_caches</td>
</tr>
<tr>
<td>seeds</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>seed_provider</td>
<td>org.apache.cassandra.locator.SimpleSeedProvider</td>
</tr>
<tr>
<td>sliced_buffer_size_in_kb</td>
<td>64</td>
</tr>
<tr>
<td>snapshot_before_compaction</td>
<td>false</td>
</tr>
<tr>
<td>storage_port</td>
<td>700</td>
</tr>
<tr>
<td>stream_throughput_outbound_megabits_per_sec</td>
<td>400</td>
</tr>
<tr>
<td>thrift_framed_transport_size_in_mb</td>
<td>15</td>
</tr>
<tr>
<td>thrift_max_message_length_in_mb</td>
<td>16</td>
</tr>
<tr>
<td>truststore</td>
<td>conf/.truststore</td>
</tr>
<tr>
<td>truststore_password</td>
<td>cassandra</td>
</tr>
</tbody>
</table>
The directory location where column family data (SSTables) will be stored.

**initial_token**

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 are using. With the random partitioner, this value will be a number between 0 and \(2^{127}\). With the byte order preserving partitioner, this value will be a byte array of hex values based on your actual row key values. With the order preserving and collated order preserving partitioners, this value will be a UTF-8 string based on your actual row key values. See *Generating Tokens* for more information.

**listen_address**

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**

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` (default)
- `org.apache.cassandra.dht.ByteOrderedPartitioner`
- `org.apache.cassandra.dht.OrderPreservingPartitioner` (deprecated)
- `org.apache.cassandra.dht.CollatingOrderPreservingPartitioner` (deprecated)

**rpc_address**

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). Default Value: localhost

Allowed Values: An IP address, hostname, or leave unset to resolve the address using the hostname configuration of the node.

**rpc_port**

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

**saved_caches_directory**

The directory location where column family key and row caches will be stored.

**seed_provider**

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**

When a node joins a cluster, it contacts the seed node(s) 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**
The port for inter-node communication. Default port is 7000.

**endpoint_snitch**

Sets the snitch to use 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

**Performance Tuning Properties**

The following properties are used to tune performance and system resource utilization (memory, disk I/O, CPU, etc.) for reads and writes.

**column_index_size_in_kb**

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**

The method that Cassandra will use 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. It will wait the configured number of milliseconds for other writes before performing a sync. Allowed Values are periodic (default) or batch.

**commitlog_sync_period_in_ms**

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**

When the commitlog size on a node exceeds this threshold, Cassandra will flush 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 keeping commit log segments around indefinitely. This replaces the per-column family storage setting memtable_flush_after_mins.

**compaction_preheat_key_cache**

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 the keys_cached attribute set on a column family).

**compaction_throughput_mb_per_sec**

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 16-32 times the rate of write throughput (in MBs/second). Setting to 0 disables compaction throttling.

**concurrent_compactors**
Configuration

Sets the number of concurrent compaction processes allowed to run simultaneously on a node. Defaults to one compaction process per CPU core.

**concurrent_reads**
For workloads with more data than can fit in memory, the bottleneck will be reads that need to fetch data from disk. Setting to \((16 \times \text{number_of_drives})\) allows operations to queue low enough in the stack so that the OS and drives can reorder them.

**concurrent_writes**
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 \((8 \times \text{number_of_cpu_cores})\).

**flush_largest_memtables_at**
When Java heap usage after a full concurrent mark sweep (CMS) garbage collection is higher than this percentage, the largest memtables will be 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. The default value of .75 means flush memtables when Java heap usage is above 75 percent total heap size. 1.0 disables this feature.

**in_memory_compaction_limit_in_mb**
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**
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**
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**
Sets the number of memtable flush writer threads. These will be blocked by disk I/O, and each one will hold a memtable in memory while blocked. If you have a large Java heap size and many data directories (see data_file_directories), you can increase this value for better flush performance. By default this is set to the number of data directories defined (which is 1).

**memtable_total_space_in_mb**
Specifies total memory used for all column family memtables on a node. Defaults to a third of your JVM heap size. This replaces the old per-column family storage settings memtable_operations_in_millions and memtable_throughput_in_mb.
reduce_cache_capacity_to
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
When Java heap usage after a full concurrent mark sweep (CMS) garbage collection is higher than this percentage, Cassandra will reduce the cache capacity to the fraction of the current size as specified by reduce_cache_capacity_to. The default is 85 percent (0.85). 1.0 disables this feature.

sliced_buffer_size_in_kb
The buffer size (in kilobytes) 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
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. The default is 400 Mb/s or 50 MB/s.

Remote Procedure Call Tuning Properties
The following properties are used to configure and tune remote procedure calls (client connections).

request_scheduler
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 (default)
- 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
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
The number of active requests per client. Requests beyond this limit are queued up until running requests complete. The default is 80. Recommended value is \((concurrent_{reads} + concurrent_{writes}) \times 2\).

default_weight
The default weight controls how many requests are handled during each turn of the RoundRobin. The default is 1.
Configuration

**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**
Enable or disable keepalive on client connections.

**rpc_max_threads**
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**
Sets the minimum thread pool size for remote procedure calls.

**rpc_recv_buff_size_in_bytes**
Sets the receiving socket buffer size for remote procedure calls.

**rpc_send_buff_size_in_bytes**
Sets the sending socket buffer size in bytes for remote procedure calls.

**rpc_timeout_in_ms**
The time in milliseconds that a node will wait on a reply from other nodes before the command is failed.

**rpc_server_type**
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 and will be removed in the next major release. Do not use.

**thrift_framed_transport_size_in_mb**
Specifies the frame size in megabytes (maximum field length) for Thrift. 0 disables framing. This option is deprecated in favor of thrift_max_message_length_in_mb.

**thrift_max_message_length_in_mb**
The maximum length of a Thrift message in megabytes, including all fields and internal Thrift overhead.

**Internode Communication and Fault Detection Properties**

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

**dynamic_snitch_badness_threshold**
Sets a 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**
Time interval in milliseconds to reset all node scores (allowing a bad node to recover).

**dynamic_snitch_update_interval_in_ms**
The time interval in milliseconds for calculating read latency.

**hinted_handoff_enabled**
Enables or disables hinted handoff.

**hinted_handoff_throttle_delay_in_ms**
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**
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. The default is one hour (3600000 ms).

**phi_convict_threshold**
The Phi convict threshold 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 will prevent false failures. Values higher than 12 and lower than 5 are not recommended. The default is 8.

**Automatic Backup Properties**

**incremental_backups**
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**
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

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

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

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

Description: 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

Password for the keystore.

truststore

The location of a truststore containing the trusted certificate used to authenticate remote servers.

truststore_password

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 can be manipulated programmatically, but in most cases the practical method for defining keyspace and column family attributes is to use the Cassandra CLI or CQL interfaces.

Prior to release 0.7.3, keyspace and column family attributes could be specified in cassandra.yaml, but that is no longer true in 0.7.4 and later. These attributes are now stored in the system keyspace within Cassandra.
**Note**
The attribute names documented in this section are the names as they are stored in the system keyspace within Cassandra. Most of these attributes can be set in the various client applications, such as Cassandra CLI or CQL. 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ks_name</td>
<td>n/a (A user-defined value is required)</td>
</tr>
<tr>
<td>placement_strategy</td>
<td>org.apache.cassandra.locator.SimpleStrategy</td>
</tr>
<tr>
<td>strategy_options</td>
<td>n/a (container attribute)</td>
</tr>
</tbody>
</table>

**name**
Required. The name for the keyspace.

**placement_strategy**
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`
- `org.apache.cassandra.locator.OldNetworkTopologyStrategy` (deprecated)

These options are described in detail in the replication section.

**Note**
`NetworkTopologyStrategy` and `OldNetworkTopologyStrategy` require 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.

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

Setting and updating strategy options with the Cassandra CLI requires a slightly different command syntax than other attributes; note the brackets and curly braces in this example:
Configuration

```
[default@unknown] CREATE KEYSPACE test
WITH placement_strategy = 'NetworkTopologyStrategy'
AND strategy_options={us-east:6,us-west:3};
```

**Column Family Attributes**

The following attributes can be declared per column family.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>column_metadata</td>
<td>n/a (container attribute)</td>
</tr>
<tr>
<td>column_type</td>
<td>Standard</td>
</tr>
<tr>
<td>comment</td>
<td>n/a</td>
</tr>
<tr>
<td>compaction_strategy</td>
<td>SizeTieredCompactionStrategy</td>
</tr>
<tr>
<td>compaction_strategy_options</td>
<td>n/a (container attribute)</td>
</tr>
<tr>
<td>comparator</td>
<td>BytesType</td>
</tr>
<tr>
<td>compare_subcolumns_with</td>
<td>BytesType</td>
</tr>
<tr>
<td>compression_options</td>
<td>n/a (container attribute)</td>
</tr>
<tr>
<td>default_validation_class</td>
<td>n/a</td>
</tr>
<tr>
<td>dc_local_read_repair_chance</td>
<td>0.0</td>
</tr>
<tr>
<td>gc_grace_seconds</td>
<td>864000 (10 days)</td>
</tr>
<tr>
<td>key_validation_class</td>
<td>n/a</td>
</tr>
<tr>
<td>keys_cached</td>
<td>200000</td>
</tr>
<tr>
<td>max_compaction_threshold</td>
<td>32</td>
</tr>
<tr>
<td>min_compaction_threshold</td>
<td>4</td>
</tr>
<tr>
<td>memtable_flush_after_mins</td>
<td>ignored in 1.0 and later releases</td>
</tr>
<tr>
<td>memtable_operations_in_millions</td>
<td>ignored in 1.0 and later releases</td>
</tr>
<tr>
<td>memtable_throughput_in_mb</td>
<td>ignored in 1.0 and later releases</td>
</tr>
<tr>
<td>cf_name</td>
<td>n/a (A user-defined value is required.)</td>
</tr>
<tr>
<td>read_repair_chance</td>
<td>0.1 or 1 (See description below.)</td>
</tr>
<tr>
<td>replicate_on_write</td>
<td>true</td>
</tr>
<tr>
<td>rows_cached</td>
<td>0 (disabled by default)</td>
</tr>
</tbody>
</table>

**column_metadata**

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Binds a validation_class and (optionally) an index to a column.</td>
</tr>
<tr>
<td>validation_class</td>
<td>Type used to check the column value.</td>
</tr>
<tr>
<td>index_name</td>
<td>Name for the secondary index.</td>
</tr>
<tr>
<td>index_type</td>
<td>Type of index. Currently the only supported value is KEYS.</td>
</tr>
</tbody>
</table>
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**

Defaults to **Standard** for regular column families. For super column families, use **Super**.

**comment**

A human readable comment describing the column family.

**compaction_strategy**

Sets the compaction strategy for the column family. The available strategies are:

- **SizeTieredCompactionStrategy** - This is the default compaction strategy and the only compaction strategy available in pre-1.0 releases. This strategy triggers a minor compaction whenever there are a number of similar sized SSTables on disk (as configured by `min_compaction_threshold`). This strategy causes bursts in I/O activity while a compaction is in process, followed by longer and longer lulls in compaction activity as SSTable files grow larger in size. These I/O bursts can negatively effect read-heavy workloads, but typically do not impact write performance. Watching disk capacity is also important when using this strategy, as compactions can temporarily double the size of SSTables for a column family while a compaction is in progress.

- **LeveledCompactionStrategy** - The leveled compaction strategy creates SSTables of a fixed, relatively small size (5 MB by default) that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables. This can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after Google's leveldb implementation.

**compaction_strategy_options**

Sets options related to the chosen `compaction_strategy`. Currently only `LeveledCompactionStrategy` has options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstable_size_in_mb</td>
<td>5</td>
<td>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.</td>
</tr>
</tbody>
</table>

Setting and updating compaction strategy options 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 compaction_strategy=LeveledCompactionStrategy
AND compaction_strategy_options={sstable_size_in_mb: 10};
```

**comparator**

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**

Required when `column_type` is "Super". Same as `comparator` but for sub-columns of a SuperColumn.
For attributes of columns, see column_metadata.

**compression_options**

This is a container attribute for setting compression options on a column family. It contains the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstable_compression</td>
<td>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.</td>
</tr>
<tr>
<td>chunk_length_kb</td>
<td>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.</td>
</tr>
</tbody>
</table>

Setting and updating compression options 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 compression_options={sstable_compression:SnappyCompressor, chunk_length_kb:64};
```

**dc_local_read_repair_chance**

Specifies the probability with which read repairs should be invoked over all replicas in the current data center. Contrast read_repair_chance.

**default_validation_class**

Defines the data type used to validate column values. There are several built-in column validators available.

**gc_grace_seconds**

Specifies the time to wait before garbage collecting tombstones (deletion markers). Defaults to 864000, or 10 days, which 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.

**keys_cached**

Defines how many key locations will be kept in memory per SSTable (see rows_cached for details on caching actual row values). This can be a fixed number of keys or a fraction (for example 0.5 means 50 percent).
DataStax recommends a fixed sized cache over a relative sized cache. Only use relative cache sizing when you are confident that the data in the column family will not continue to grow over time. Otherwise, your cache will grow as your data set does, potentially causing unplanned memory pressure.

**key_validation_class**

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.

**name**

Required. The user-defined name of the column family.

**read_repair_chance**

Specifies the probability with which read repairs should be invoked on non-quorum reads. The value must be between 0 and 1. For column families created in versions of Cassandra before 1.0, it defaults to 1.0. For column families created in versions of Cassandra 1.0 and higher, it defaults to 0.1. However, for Cassandra 1.0, the default is 1.0 if you use CLI or any Thrift client, such as Hector or py cassa, and is 0.1 if you use CQL.

A value of .01 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**

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

**max_compaction_threshold**

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).

Setting this to 0 disables minor compactions. Defaults to 32.

**min_compaction_threshold**

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. Defaults to 4.

**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.
rows_cached

Specifies how many rows to cache in memory. This can be a fixed number of rows or a fraction (for example 0.5 means 50 percent).

Using a row cache means that the entire row is cached in memory. This can be detrimental to performance in cases where rows are large, or where rows are frequently modified or removed.

Java and System Environment Settings Configuration

There are two files that control environment settings for Cassandra:

- `conf/cassandra-env.sh` - Java Virtual Machine (JVM) configuration settings
- `bin/cassandra-in.sh` - Sets up Cassandra environment variables such as CLASSPATH and JAVA_HOME.

Heap Sizing Options

If you decide to change the Java heap sizing, both MAX_HEAP_SIZE and HEAP_NEWSIZE should be set together in `conf/cassandra-env.sh` (if you set one, set the other as well). See the section on Tuning Java Heap Size for more information on choosing the right Java heap size.

- `MAX_HEAP_SIZE` - Sets the maximum heap size for the JVM. The same value is also used for the minimum heap size. This allows the heap to be locked in memory at process start to keep it from being swapped out by the OS. Defaults to half of available physical memory.
- `HEAP_NEWSIZE` - The size of the young generation. The larger this is, the longer GC pause times will be. The shorter it is, the more expensive GC will be (usually). A good guideline is 100 MB per CPU core.

JMX Options

Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX. JConsole, nodetool and DataStax OpsCenter are examples of JMX-compliant management tools.

By default, the `conf/cassandra-env.sh` file configures JMX to listen on port 7199 without authentication. See the table below for more information on commonly changed JMX configuration properties.

- `com.sun.management.jmxremote.port` - The port on which Cassandra listens from JMX connections.
- `com.sun.management.jmxremote.ssl` - Enable/disable SSL for JMX.
- `com.sun.management.jmxremote.authenticate` - Enable/disable remote authentication for JMX.
- `-Djava.rmi.server.hostname` - Sets the interface hostname or IP that JMX should use to connect. Uncomment and set if you are having trouble connecting.

Further Reading on JVM Tuning

The remaining options are optimal across a wide variety of workloads and environments and are not frequently changed. See the Sun JVM options list for more information on JVM tuning parameters.

Authentication and Authorization Configuration
Note

As of release 1.0, the SimpleAuthenticator and SimpleAuthority classes have been moved to the example directory of the Apache Cassandra project repository. They are no longer available in the packaged and binary distributions. They are only examples and do not provide actual security in their current state. DataStax does not officially support them and do not recommend their use.

Using authentication and authorization requires configuration changes in cassandra.yaml and two additional files: one for assigning users and their permissions to keyspaces and column families, and the other for assigning passwords to those users. These files are named access.properties and passwd.properties, respectively, and are located in the examples directory of the Apache Cassandra project repository. To test simple authentication, you can move these files to the conf directory.

The location of cassandra.yaml file depends on the type of installation; see Cassandra Configuration Files Locations or DataStax Enterprise Configuration Files Locations.

To set up simple authentication and authorization

1. Edit cassandra.yaml, setting org.apache.cassandra.auth.SimpleAuthenticator as the authenticator value. The default value of AllowAllAuthenticator is equivalent to no authentication.

2. Edit access.properties, adding entries for users and their permissions to read and write to specified keyspaces and column families. See access.properties below for details on the correct format.

3. Make sure that users specified in access.properties have corresponding entries in passwd.properties. See passwd.properties below for details and examples.

4. After making the required configuration changes, you must specify the properties files when starting Cassandra with the flags -Dpasswd.properties and -Daccess.properties. For example:

   cd <install_location>
   sh bin/cassandra -f -Dpasswd.properties=conf/passwd.properties -Daccess.properties=conf/access.properties

access.properties

This file contains entries in the format KEYSPACE[.COLUMNFAMILY].PERMISSION=USERS where

- KEYSPACE is the keyspace name.
- COLUMNFAMILY is the column family name.
- PERMISSION is one of <ro> or <rw> for read-only or read-write respectively.
- USERS is a comma delimited list of users from passwd.properties.

For example, to control access to Keyspace1 and give jsmith and Elvis read-only permissions while allowing dilbert full read-write access to add and remove column families, you would create the following entries:

   Keyspace1.<ro>=jsmith,Elvis Presley
   Keyspace1.<rw>=dilbert

To provide a finer level of access control to the Standard1 column family in Keyspace1, you would create the following entry to allow the specified users read-write access:

   Keyspace1.Standard1.<rw>=jsmith,Elvis Presley,dilbert

The access.properties file also contains a simple list of users who have permissions to modify the list of keyspaces:

   <modify-keyspaces>=jsmith

passwd.properties
This file contains name/value pairs in which the names match users defined in `access.properties` and the values are user passwords. Passwords are in clear text unless the `passwd.mode=MD5` system property is provided.

```
jsmith=havebadpass
Elvis Presley=graceland4ever
dilbert=nomoovetime
```

**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 Brisk cluster are available in OpsCenter. OpsCenter metrics are captured and stored within Cassandra.

The performance metrics viewed within OpsCenter can be customized according to your monitoring needs. Administrators can also perform routine node administration tasks from OpsCenter. Metrics within OpsCenter are divided into three general categories: column family metrics, cluster metrics, and OS metrics. For many of the available metrics, you can choose to view aggregated cluster-wide information, or view information on a per-node basis.
Monitoring Using nodetool

The **nodetool** utility is a command-line interface for monitoring Cassandra and performing routine database operations. It is included in the Cassandra distribution and is typically run directly from an operational Cassandra node.

The **nodetool** utility supports the most important JMX metrics and operations, and includes other useful commands for Cassandra administration. This utility is commonly used to output a quick summary of the ring and its current state of general health with the `ring` command. For example:

```
# nodetool -h localhost -p 7199 ring
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Status</th>
<th>State</th>
<th>Load</th>
<th>Owns</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.194.171.160</td>
<td>Down</td>
<td>Normal</td>
<td>?</td>
<td>39.98</td>
<td>95315431979199388464207182617231204396</td>
</tr>
<tr>
<td>10.196.14.48</td>
<td>Up</td>
<td>Normal</td>
<td>3.16 KB</td>
<td>30.01</td>
<td>61078635599166706937511052402724559481</td>
</tr>
<tr>
<td>10.196.14.239</td>
<td>Up</td>
<td>Normal</td>
<td>3.16 KB</td>
<td>30.01</td>
<td>78197033789183047700859117509977881938</td>
</tr>
</tbody>
</table>

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.
Operations

The **Overview** and **Memory** tabs contain information that is very useful for Cassandra developers. The Memory tab allows you to compare heap and non-heap memory usage, and provides a control to immediately perform Java garbage collection.

For specific Cassandra metrics and operations, the most important area of JConsole is the **MBeans** tab. This tab lists the following Cassandra MBeans:

- `org.apache.cassandra.db` - Includes caching, column family metrics, and compaction.
- `org.apache.cassandra.internal` - Internal server operations such as gossip and hinted handoff.
- `org.apache.cassandra.net` - Inter-node communication including FailureDetector, MessagingService and StreamingService.
- `org.apache.cassandra.request` - Tasks related to read, write, and replication operations.

When you select an MBean in the tree, its MBeanInfo and MBean Descriptor are both displayed on the right, and any attributes, operations or notifications appear in the tree below it. For example, selecting and expanding the `org.apache.cassandra.db` MBean to view available actions for a column family results in a display like the following:
If you choose to monitor Cassandra using JConsole, keep in mind that JConsole consumes a significant amount of system resources. For this reason, DataStax recommends running JConsole on a remote machine rather than on the same host as a Cassandra node.

**Compaction Metrics**

Monitoring compaction performance is an important aspect of knowing when to add capacity to your cluster. The following attributes are exposed through `CompactionManagerMBean`:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompletedTasks</td>
<td>Number of completed compactions since the last start of this Cassandra instance</td>
</tr>
<tr>
<td>PendingTasks</td>
<td>Number of estimated tasks remaining to perform</td>
</tr>
<tr>
<td>ColumnFamilyInProgress</td>
<td>ColumnFamily currently being compacted. <code>null</code> if no compactions are in progress.</td>
</tr>
<tr>
<td>BytesTotalInProgress</td>
<td>Total number of data bytes (index and filter are not included) being compacted. <code>null</code> if no compactions are in progress.</td>
</tr>
</tbody>
</table>
Operations

| 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`).

<table>
<thead>
<tr>
<th>Thread Pool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE_SERVICE_STAGE</td>
<td>Shows anti-entropy tasks</td>
</tr>
<tr>
<td>CONSISTENCY-MANAGER</td>
<td>Handles the background consistency checks if they were triggered from the client's consistency level <code>&lt;consistency&gt;</code></td>
</tr>
<tr>
<td>FLUSH-SORTER-POOL</td>
<td>Sorts flushes that have been submitted</td>
</tr>
<tr>
<td>FLUSH-WRITER-POOL</td>
<td>Writes the sorted flushes</td>
</tr>
<tr>
<td>GOSSIP_STAGE</td>
<td>Activity of the Gossip protocol on the ring</td>
</tr>
<tr>
<td>LB-OPERATIONS</td>
<td>The number of load balancing operations</td>
</tr>
<tr>
<td>LB-TARGET</td>
<td>Used by nodes leaving the ring</td>
</tr>
<tr>
<td>MEMTABLE-POST-FLUSHER</td>
<td>Memtable flushes that are waiting to be written to the commit log.</td>
</tr>
<tr>
<td>MESSAGE-STREAMING-POOL</td>
<td>Streaming operations. Usually triggered by bootstrapping or decommissioning nodes.</td>
</tr>
<tr>
<td>MIGRATION_STAGE</td>
<td>Tasks resulting from the call of <code>system_*</code> methods in the API that have modified the schema</td>
</tr>
<tr>
<td>MISC_STAGE</td>
<td></td>
</tr>
<tr>
<td>MUTATION_STAGE</td>
<td>API calls that are modifying data</td>
</tr>
<tr>
<td>READ_STAGE</td>
<td>API calls that have read data</td>
</tr>
<tr>
<td>RESPONSE_STAGE</td>
<td>Response tasks from other nodes to message streaming from this node</td>
</tr>
<tr>
<td>STREAM_STAGE</td>
<td>Stream tasks from this node</td>
</tr>
</tbody>
</table>

### Read/Write Latency Metrics

Cassandra keeps track 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemtableDataSize</td>
<td>The total size consumed by this column family's data (not including meta data)</td>
</tr>
<tr>
<td>MemtableColumnsCount</td>
<td>Returns the total number of columns present in the memtable (across all keys)</td>
</tr>
<tr>
<td>MemtableSwitchCount</td>
<td>How many times the memtable has been flushed out</td>
</tr>
<tr>
<td>RecentReadLatencyMicros</td>
<td>The average read latency since the last call to this bean</td>
</tr>
<tr>
<td>RecentWriterLatencyMicros</td>
<td>The average write latency since the last call to this bean</td>
</tr>
</tbody>
</table>
LiveSSTableCount | The number of live SSTables for this ColumnFamily

The first three Memtable attributes are discussed in detail on the *Tuning Cassandra* page.

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

Careful, incremental monitoring of cache changes is the best way to maximize benefit from Cassandra’s built-in caching features. Adjustments that increase cache hit rate are likely to use more system resources, such as memory. After making changes to the cache configuration, it is best to monitor Cassandra as a whole for unintended impact on the system.

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:

![MBeans tab screenshot](image)

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.

Row cache is disabled by default. Caching large rows can very quickly consume memory. Row cache rates should be increased carefully in small increments. If row cache hit rates cannot be tuned to above 30%, it may make more sense to leave row caching disabled.

### Tuning Cassandra

Effective tuning depends not only on the types of operations your cluster performs most frequently, but also on the shape of the data itself. For example, 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 columns is small compared to the number of columns and rows themselves (sometimes called *skinny rows*), this overhead can be substantial. Thus, the optimal tuning for this type of data is quite different than the optimal tuning for a small numbers of columns with more data (*fat rows*).
Tuning the Cache

Cassandra's built-in key and row caches can provide very efficient data caching. Some Cassandra production deployments have leveraged Cassandra's caching features to the point where dedicated caching tools such as memcached could be completely replaced. Such deployments not only remove a redundant layer from the stack, but they also achieve the fundamental efficiency of strengthening caching functionality in the lower tier where the data is already being stored. Among other advantages, this means that caching never needs to be restarted in a completely cold state.

Cache tuning should be done using small, incremental adjustments and then monitoring the effects of each change. See Monitoring and Adjusting Cache Performance for more information about monitoring tuning changes to a column family cache. With proper tuning, key cache hit rates of 85% or better are possible. Row caching, when feasible, can save the system from performing any disk seeks at all when fetching a cached row. Whenever growth in the read load begins to impact your cache hit rates, you can add capacity to quickly restore optimal levels of caching.

How Caching Works

If both row and key caches are configured, the row cache will return 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, the entire requested row is returned without a disk seek. If a 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 will populate the key cache after accessing the row on disk so subsequent reads of the row can benefit. Each hit on a key cache can save one disk seek per SSTable.

Configuring the Column Family Key Cache

The key cache holds the location of row keys in memory on a per-column family basis. High levels of key caching are recommended for most production scenarios. Turning this level up can optimize reads (after the cache warms) when there is a large number of rows that are accessed frequently.

The caching of 200,000 row keys is enabled by default. This can be adjusted by setting keys_cached on a column family. For example, using Cassandra CLI:

```
[default@demo] UPDATE COLUMN FAMILY users WITH keys_cached=205000;
```

Key cache performance can be monitored by using nodetool cfstats and examining the reported 'Key cache hit rate'. See also Monitoring and Adjusting Cache Performance for more information about monitoring tuning changes to a column family key cache.

Configuring the Column Family Row Cache

The row cache holds the entire contents of the row in memory. In cases where rows are large or frequently modified/removed, row caching can actually be detrimental to performance. For this reason, row caching is disabled by default.

Row cache should remain disabled for column families with large rows or high write:read ratios. In these situations, row cache can very quickly consume a large amount of available memory. Note also that, when a row cache is operating efficiently, it keeps Java garbage compaction processes very active.

Row caching is best for workloads that access a small subset of the overall rows, and within those rows, all or most of the columns are returned. For this use case a row cache keeps the most accessed rows hot in memory, and can have substantial performance benefits.

To enable row cache on a column family, set rows_cached to the desired number of rows.

Row cache performance can be monitored by using nodetool cfstats and examining the reported 'Row cache hit rate'. See also Monitoring and Adjusting Cache Performance for more information about monitoring tuning changes to a column family key cache.

Data Modeling Considerations for Cache Tuning
If your requirements permit it, a data model that logically separates heavily-read data into discrete column families can help optimize caching. Column families with relatively small, narrow rows lend themselves to highly efficient row caching. By the same token, it can make sense to separately store lower-demand data, or data with extremely long rows, in a column family with minimal caching, if any.

Row caching in such contexts brings the most benefit when access patterns follow a normal (Gaussian) distribution. When the keys most frequently requested follow such patterns, cache hit rates tend to increase. If you have particularly hot rows in your data model, row caching can bring significant performance improvements.

**Hardware and OS Considerations for Cache Tuning**

Deploying a large number of Cassandra nodes under a relatively light load per node will maximize the fundamental benefit from key and row caches.

A less obvious but very important consideration is the OS page cache. Modern operating systems maintain page caches for frequently accessed data and are very efficient at keeping this data in memory. Even after a row is released in the Java Virtual Machine 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 your requirements allow you to lower JVM heap size and memtable sizes to leave memory for OS page caching, then do so. Ultimately, through gradual adjustments, you should achieve the desired balance between these three demands on available memory: heap, memtables, and caching.

**Estimating Cache Sizes**

`nodetool cfstats` can be used to get the necessary information for estimating actual cache sizes.

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

\[
\text{key cache size} = \left( \text{average('Key size in bytes')} + 64 \right) \times \text{Key cache size} \times 10^{-12}
\]

To estimate the row cache size:

\[
\text{row cache size} = \left( \text{average 'Row cache size in bytes'} + 64 \right) \times \text{Row cache size} \times 10^{-12}
\]

The row cache for a column family is stored in native memory by default rather than using the JVM heap.

**Tuning Write Performance (Memtables)**

A memtable is a column family specific, in memory data structure that can be easily described as a write-back cache. Memtables are flushed to disk, creating SSTables whenever one of the configurable thresholds has been exceeded.

Effectively tuning memtable thresholds depends on your data as much as your write load. Memtable thresholds are configured per node using the `cassandra.yaml` properties: `memtable_throughput_in_mb` and `commitlog_total_space_in_mb`.

You should increase memtable throughput if:

1. Your write load includes a high volume of updates on a smaller set of data
2. You have steady stream of continuous writes (this will lead to more efficient compaction)

Note that increasing memory allocations for memtables takes memory away from caching and other internal Cassandra structures, so tune carefully and in small increments.

**Tuning Java Heap Size**

Because Cassandra is a database, it spends significant time interacting with the operating system's I/O infrastructure through the JVM, so a well-tuned Java heap size is important. Cassandra's default configuration opens the JVM with a heap size that is based on the total amount of system memory:

<table>
<thead>
<tr>
<th>System Memory</th>
<th>Heap Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 2GB</td>
<td>1/2 of system memory</td>
</tr>
</tbody>
</table>
### General Guidelines

Many users new to Cassandra are tempted to turn up Java heap size too high, which consumes the majority of the underlying system’s RAM. In most cases, increasing the Java heap size is actually detrimental for these reasons:

- In most cases, the capability of Java 6 to gracefully handle garbage collection above 8GB quickly diminishes.
- Modern operating systems maintain the OS page cache for frequently accessed data and are very good at keeping this data in memory, but can be prevented from doing its job by an elevated Java heap size.

If you have more than 2GB of system memory, which is typical, keep the size of the Java heap relatively small to allow more memory for the page cache. To change a JVM setting, modify the cassandra-env.sh file.

Because MapReduce runs outside the JVM, changes to the JVM do not affect Hadoop operations directly.

#### Tuning Java Garbage Collection

Cassandra's GCInspector class will log information about garbage collection whenever a garbage collection takes longer than 200ms. If garbage collections are occurring frequently and are taking a moderate length of time to complete (such as ConcurrentMarkSweep taking a few seconds), this is an indication that there is a lot of garbage collection pressure on the JVM; this needs to be addressed by adding nodes, lowering cache sizes, or adjusting the JVM options regarding garbage collection.

#### Tuning Compaction

During normal operations, numerous SSTables may be created on disk for a given column family. Compaction is the process of merging multiple SSTables into one consolidated SSTable. Additionally, the compaction process merges keys, combines columns, discards tombstones and creates a new index in the merged SSTable.

##### Choosing a Column Family Compaction Strategy

Tuning compaction involves first choosing the right compaction strategy for each column family based on its access patterns. As of Cassandra 1.0, there are two choices of compaction strategies:

- **SizeTieredCompactionStrategy** - This is the default compaction strategy for a column family, and prior to Cassandra 1.0, the only compaction strategy available. This strategy is best suited for column families with insert-mostly workloads that are not read as frequently. This strategy also requires closer monitoring of disk utilization because (as a worst case scenario) a column family can temporarily double in size while a compaction is in progress.

- **LeveledCompactionStrategy** - This is a new compaction strategy introduced in Cassandra 1.0. This compaction strategy is based on (but not an exact implementation of) Google's leveldb. This strategy is best suited for column families with read-heavy workloads that also have frequent updates to existing rows. When using this strategy, you want to keep an eye on read latency performance for the column family. If a node cannot keep up with the write workload and pending compactions are piling up, then read performance will degrade for a longer period of time.

##### Setting the Compaction Strategy on a Column Family

You can set the compaction strategy on a column family by setting the `compaction_strategy` attribute. For example, to update a column family to use the leveled compaction strategy using Cassandra CLI:

```
[default@demo] UPDATE COLUMN FAMILY users WITH compaction_strategy=LeveledCompactionStrategy AND compaction_strategy_options={sstable_size_in_mb: 10};
```

#### Tuning Options for Size-Tiered Compaction

<table>
<thead>
<tr>
<th>2GB to 4GB</th>
<th>1GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than 4GB</td>
<td>1/4 system memory, but not more than 8GB</td>
</tr>
</tbody>
</table>
For column families that use size-tiered compaction (the default), the frequency and scope of minor compactions is controlled by the following column family attributes:

- min_compaction_threshold
- max_compaction_threshold

These parameters set thresholds for the number of similar-sized SSTables that can accumulate before a minor compaction is triggered. With the default values, a minor compaction may begin any time after four SSTables are created on disk for a column family, and must begin before 32 SSTables accumulate.

You can tune these values per column family. For example, using Cassandra CLI:

```
=default@demo] UPDATE COLUMN FAMILY users WITH max_compaction_threshold = 20;
```

**Note**

Administrators can also initiate a major compaction through nodetool compact, which 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. Also, once you run a major compaction, automatic minor compactions are no longer triggered frequently forcing you to manually run major compactions on a routine basis. So while read performance will be good immediately following a major compaction, it will continually degrade until the next major compaction is manually invoked. For this reason, major compaction is NOT recommended by DataStax.

### 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 and by adding an entire data center. When a new node joins an existing cluster, it needs to know:

- Its position in the ring and the range of data it is responsible for. 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. The location of this file depends on the type of installation; see Cassandra Configuration Files Locations or DataStax Enterprise Configuration Files Locations.

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 several 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.

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 configuration 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.

Adding a Data Center to a Cluster

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

1. Ensure that you are using `NetworkTopologyStrategy` for all of your custom keyspaces.
2. For each new node, edit the configuration properties in the cassandra.yaml file:
   - Set auto_bootstrap to False.
   - Set the initial_token. Be sure to offset the tokens in the new data center; see Generating Tokens.
   - Set the cluster name.
   - Set any other non-default settings.
   - Set the seed lists. Every node in the cluster must have the same list of seeds and include at least one node from each data center. Typically one to three seeds are used per data center.

3. If using the PropertyFileSnitch, update the cassandra-topology.properties file on all servers to include the new nodes. You do not need to restart.
   The location of this file depends on the type of installation; see Cassandra Configuration Files Locations or DataStax Enterprise Configuration Files Locations.

4. Ensure that your client does not autodetect the new nodes so that they aren’t contacted by the client until explicitly directed. For example in Hector, set hostConfig.setAutoDiscoverHosts(false);

5. If using a QUORUM consistency level for reads or writes, check the LOCAL QUORUM or EACH QUORUM consistency level to see if the level meets your requirements for multiple data centers.

6. Start the new nodes.

7. After all nodes are running in the cluster:
   a. Change the strategy_options for your keyspace to the desired replication factor for the new data center. For example: strategy_options={DC1:2,DC2:2}
   b. On each new node, run nodetool repair without the -pr option one at a time.

**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), bring up a new node using the token of the dead node as described in the next procedure. This token used must already be part of the ring.

To replace a dead node:
1. Confirm that the node is dead using the `nodetool ring` command on any live node in the cluster.

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

   ```
   $ nodetool ring -h localhost
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | Address | DC   | Rock | Status | State | Load | Owns | Token |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.11      | dotcenter1 | rock1 | Up    | Normal | 179.58 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.12      | dotcenter1 | rock1 | Down  | Normal | 315.21 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.13      | dotcenter1 | rock1 | Up    | Normal | 267.71 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.14      | dotcenter1 | rock1 | Up    | Normal | 315.21 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.15      | dotcenter1 | rock1 | Up    | Normal | 292.36 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.16      | dotcenter1 | rock1 | Up    | Normal | 300.02 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.17      | dotcenter1 | rock1 | Up    | Normal | 147.84 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.18      | dotcenter1 | rock1 | Down  | Normal | 134.27 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.19      | dotcenter1 | rock1 | Up    | Normal | 205.28 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.20      | dotcenter1 | rock1 | Up    | Normal | 174.09 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.21      | dotcenter1 | rock1 | Up    | Normal | 205.28 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.22      | dotcenter1 | rock1 | Up    | Normal | 205.28 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.23      | dotcenter1 | rock1 | Up    | Normal | 205.28 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.24      | dotcenter1 | rock1 | Up    | Normal | 205.28 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   | 10.46.123.25      | dotcenter1 | rock1 | Up    | Normal | 205.28 KB | 16.67% | 0       |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   |                  |      |     |        |       |        |       |        |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   |                  |      |     |        |       |        |       |        |
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   +-------------------+-------+---------------+-----+---------+---------------------+---+--------+------------------------+
   ```

2. Install Cassandra on the replacement node.

3. Remove any pre-existing Cassandra data on the replacement node:

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

4. Configure any non-default settings in the node’s `cassandra.yaml` to match your existing cluster.

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

   ```yaml
   initial_token: 28356863910078205288614550619314017620
   ```

6. Start the new node.

7. After the new node is up, run `nodetool repair` on each keyspace to ensure the node is fully consistent. For example:

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

8. Remove the dead node.

**Backing Up and Restoring Data**

Cassandra backs up data by taking a snapshot of all on-disk data files (SSTable files) stored in the data directory. Snapshots are taken per keyspace and while the system is online. However, nodes must be taken offline in order to restore a snapshot.

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. The snapshot files are stored in the `/var/lib/cassandra/data` (by default) in the `snapshots` directory of each keyspace.

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>/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 keyspace, 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. Clear all `*.db` files in `<data_directory_location>/<keyspace_name>`, but **DO NOT** delete the `/snapshots` and `/backups` subdirectories.
4. Locate the most recent snapshot folder in
<data_directory_location>/<keyspace_name>/snapshots/<snapshot_name>, and copy its contents
into <data_directory_location>/<keyspace_name>.

5. If using incremental backups as well, copy all contents of
<data_directory_location>/<keyspace_name>/backups
into <data_directory_location>/<keyspace_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 Language Reference

Cassandra Query Language (CQL) is based on SQL (Structured Query Language), the standard for relational database
manipulation. Although CQL has many similarities to SQL, there are some fundamental differences. For example, CQL
is adapted to the Cassandra data model and architecture so there is still no allowance for SQL-like operations such as
JOINs or range queries over rows on clusters that use the random partitioner. This reference describes CQL 2.0.0.

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. All statements end in a semicolon (;).

For example, the following is valid CQL syntax:

```
SELECT * FROM MyColumnFamily;
```

```
UPDATE MyColumnFamily SET 'SomeColumn' = 'SomeValue' WHERE KEY = 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.

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).

<table>
<thead>
<tr>
<th>CQL Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>US-ASCII character string</td>
</tr>
<tr>
<td>bigint</td>
<td>64-bit signed long</td>
</tr>
<tr>
<td>blob</td>
<td>Arbitrary bytes (no validation), expressed as hexadecimal</td>
</tr>
<tr>
<td>boolean</td>
<td>true or false</td>
</tr>
<tr>
<td>counter</td>
<td>Distributed counter value (64-bit long)</td>
</tr>
<tr>
<td>decimal</td>
<td>Variable-precision decimal</td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE-754 floating point</td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE-754 floating point</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>text</td>
<td>UTF-8 encoded string</td>
</tr>
<tr>
<td>timestamp</td>
<td>Date plus time, encoded as 8 bytes since epoch</td>
</tr>
<tr>
<td>uuid</td>
<td>Type 1 or type 4 UUID</td>
</tr>
<tr>
<td>varchar</td>
<td>UTF-8 encoded string</td>
</tr>
<tr>
<td>varint</td>
<td>Arbitrary-precision integer</td>
</tr>
</tbody>
</table>

In addition to the CQL types listed in the previous 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`
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)
- 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. CQL does not currently offer support for defining all of the possible properties, 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 Column Family Storage Parameters

CQL supports setting the following column family properties, which in a few cases have slightly different names than their corresponding *column family attributes*.

<table>
<thead>
<tr>
<th>CQL Parameter Name</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>compaction_strategy_class</td>
<td>SizeTieredCompactionStrategy</td>
</tr>
<tr>
<td>compaction_strategy_options</td>
<td>none</td>
</tr>
<tr>
<td>compression_options</td>
<td>none</td>
</tr>
<tr>
<td>comparator</td>
<td>text</td>
</tr>
<tr>
<td>comment</td>
<td>&quot;(an empty string)&quot;</td>
</tr>
<tr>
<td>default_validation</td>
<td>text</td>
</tr>
<tr>
<td>gc_grace_seconds</td>
<td>864000</td>
</tr>
<tr>
<td>min_compaction_threshold</td>
<td>4</td>
</tr>
<tr>
<td>max_compaction_threshold</td>
<td>32</td>
</tr>
<tr>
<td>read_repair_chance</td>
<td>1.0</td>
</tr>
<tr>
<td>replicate_on_write</td>
<td>false</td>
</tr>
</tbody>
</table>

compaction_strategy_class in CQL corresponds to the compaction_strategy attribute. default_validation in CQL corresponds to the default_validation_class attribute.

### CQL Commands

The CQL language is comprised of the following commands:

**ALTER TABLE**

Manipulates the column metadata of a column family.

**Synopsis**

```
ALTER TABLE <name>

  (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 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.

**Changing the Type of a Typed Column**

To change the storage type for a column of type ascii to type text, use ALTER TABLE and the ALTER and TYPE keywords in the following way:

```sql
ALTER TABLE addamsFamily ALTER lastKnownLocation TYPE text;
```

The column must already have a type in the column family metadata and the old type must be compatible with the new type. The column may or may not already exist in current rows. No validation of existing data occurs. The bytes stored in values for that column remain unchanged, and if existing data is not deserializable according to the new type, your CQL driver or interface might report errors.

**Adding a Typed Column**

To add a typed column to a column family, use ALTER TABLE and the ADD keyword in the following way:

```sql
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:

```sql
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:

```sql
ALTER TABLE addamsFamily WITH comment = 'A most excellent and useful column family'
    AND read_repair_chance = 0.2;
```

See [CQL Column Family Storage Parameters](#) for the column family options you can define.

Setting `compaction_strategy_options` or `compression_options` erases all previous `compaction_strategy_options` or `compression_options` settings, respectively.

**Examples**

```sql
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;
```

**BATCH**

Sets a global consistency level and client-supplied timestamp for all columns written by the statements in the batch.

**Synopsis**
BEGIN BATCH

[ USING <write_option> [ AND <write_option> [...] ] ];

<dml_statement>
<dml_statement>
[...]

APPLY BATCH;

<write_option> is:

USING CONSISTENCY <consistency_level>
TIMESTAMP <integer>

Description

A BATCH statement combines multiple data modification (DML) statements into a single logical operation. BATCH supports setting a client-supplied, global consistency level and timestamp that is used for each of the operations included in the batch.

You can specify these global options in the USING clause:

- Consistency level
- Timestamp (current time)

Batched statements default to a consistency level of ONE when unspecified.

After the USING clause, you can add only these DML statements:

- INSERT
- UPDATE
- DELETE

Individual DML statements inside a BATCH cannot specify a consistency level or timestamp. These individual statements can specify a TTL (time to live). TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

Close the batch statement with APPLY BATCH.

BATCH is not an analogue for SQL ACID transactions. Column updates are considered atomic within a given record (row) only.

Example

BEGIN BATCH USING CONSISTENCY QUORUM
  INSERT INTO users (KEY, password, name) VALUES ('user2', 'ch@ngem3b', 'second user')
  UPDATE users SET password = 'ps22dhds' WHERE KEY = 'user2'
  INSERT INTO users (KEY, password) VALUES ('user3', 'ch@ngem3c')
  DELETE name FROM users WHERE key = 'user2'
  INSERT INTO users (KEY, password, name) VALUES ('user4', 'ch@ngem3c', 'Andrew')
APPLY BATCH;

CREATE TABLE

Define a new column family.

Synopsis
CREATE TABLE <cf_name> (  
<key_column_name> <data_type> PRIMARY KEY  
[, <column_name> <data_type> [, ...] ] )  
[ WITH <storage_parameter> = <value>  
[AND <storage_parameter> = <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 Fish (KEY blob PRIMARY KEY);  
CREATE TABLE FastFoodEatings (user text PRIMARY KEY)  
  WITH comparator=timestamp AND default_validation=int;

Specifying the Key Type

When creating a new column family, specify the key type. The list of possible types is identical to column comparators/validators (see CQL Data Types), except 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 require UTF-8 keys. If you use an identifier for the primary key name, instead of the KEY keyword, a key alias is set automatically.

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)

A number of optional keyword arguments can be supplied to control the configuration of a new column family. See CQL Column Family Storage Parameters for the column family options you can define.

Define a default data type for column names using this syntax: WITH comparator=<data_type>. Define values using this syntax: WITH default_validation=<data_type>.

CREATE TABLE MonkeyTypes (  
  KEY uuid PRIMARY KEY,  
  species text,  
  alias text,  
  population varint  
) WITH comment='Important biological records'  
  AND read_repair_chance = 1.0;

Examples

Dynamic column family definition:

CREATE TABLE user_events (user text PRIMARY KEY)  
  WITH comparator=timestamp AND default_validation=int;

Static column family definition:

CREATE TABLE users (  
  KEY uuid PRIMARY KEY,
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 (
  KEY uuid PRIMARY KEY,
  firstname text,
  lastname text,
  email text,
  address text,
  zip int,
  state text);

CREATE INDEX user_state
  ON users (state);

CREATE INDEX ON users (zip);
```

CREATE KEYSSPACE

Define a new keyspace and its replica placement strategy.

**Synopsis**

```
CREATE KEYSSPACE <ks_name> WITH strategy_class = <value> [ AND strategy_options:<option> = <value> [...] ];
```

**Description**

`CREATE KEYSSPACE` 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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy_class</td>
<td>Required. The name of the replica placement strategy for the new keyspace, such as SimpleStrategy and NetworkTopologyStrategy.</td>
</tr>
<tr>
<td>strategy_options</td>
<td>Optional. Additional arguments appended to the option name.</td>
</tr>
</tbody>
</table>

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 MyKeyspace 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;
```

**DELETE**

Removes one or more columns from the named row(s).

**Synopsis**

```
DELETE [<column_name> [, ...]]
FROM <column_family>
[USING CONSISTENCY <consistency_level> [AND TIMESTAMP <integer>]]
WHERE <row_specification>;
```

A DELETE statement removes one or more columns from one or more rows in the named column family.

References

110
Specifying Columns

After the `DELETE` keyword, optionally list column names, separated by commas.

```
DELETE col1, col2, col3 FROM Planeteers USING CONSISTENCY ONE WHERE KEY = 'Captain';
```

When no column names are specified, the entire row(s) specified in the WHERE clause are deleted.

```
DELETE FROM MastersOfTheUniverse WHERE KEY IN ('Man-At-Arms', 'Teela');
```

Specifying the Column Family

The column family name follows the list of column names and the keyword `FROM`.

Specifying Options

You can specify these options:

- *Consistency level*
- *Timestamp (current time)*

When a column is deleted, it is not removed from disk immediately. The deleted column is marked with a tombstone and then removed after the configured grace period has expired. The optional timestamp defines the new tombstone record. See About Deletes for more information about how Cassandra handles deleted columns and rows.

Specifying Rows

The WHERE clause specifies which row or rows to delete from the column family. This form allows the specification of a single keyname using the `KEY` keyword (or key alias) and the `=` operator.

```
DELETE col1 FROM SomeColumnFamily WHERE KEY = '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 keyalias IN (key1, key2);
```

Example

```
DELETE email, phone
FROM users
USING CONSISTENCY QUORUM AND TIMESTAMP 1318452291034
WHERE user_name = 'jsmith';

DELETE phone FROM users WHERE KEY IN ('jdoe', 'jsmith');
```

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;

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;

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;

INSERT
Adds or updates one or more columns in the identified row of a column family.

Synopsis
INSERT INTO <column_family> (<key_name>, <column_name> [, ...])
VALUES (<key_value>, <column_value> [, ...])
[USING <write_option> [AND <write_option> [...] ] ];

<write_option> is:
CONSISTENCY <consistency_level>
TTL <seconds>
TIMESTAMP <integer>

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 (either the **KEY** keyword or the row key alias defined on the column family). 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.

**Specifying Options**

You can specify these options:

- **Consistency level**
- Time-to-live (TTL)
- Timestamp (current time)

TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

**Example**

```
INSERT INTO NerdMovies (KEY, 11924) VALUES ('cfd66ccc-d857-4e90-b1e5-df98a3d40cd6', 'johndoe')
USING CONSISTENCY LOCAL_QUORUM AND TTL 86400;
```

**SELECT**

Retrieves data from a Cassandra column family.

**Synopsis**

```sql
SELECT [FIRST <n>] [REVERSED] <select expression>
FROM <column family>
[USING <consistency>]
[WHERE <clause>] [LIMIT <n>]
```

<select expression> syntax is:

```
{ <start_of_range> .. <end_of_range> | * }
| COUNT(* | 1)
```

<clause> syntax is:

```
KEY | <key_alias> { = | < | > | <= | >= } <key_value>
KEY | <key_alias> 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 keys 199, 200, or 207:

```
SELECT Name, Occupation FROM People WHERE key 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.

Specifying a Range of Columns

To specify a range of columns, specify the start and end column names separated by two periods (..). The set of columns returned for a range is start and end inclusive. Use the asterisk (*) to specify a range representing all columns.

Select a range of columns from all rows, but limit the number of columns to 3 per row starting with the end of the range:

```
SELECT FIRST 3 REVERSED 'time199'..'time100' FROM Events;
```

When requesting a range of columns, it may be useful to limit the number of columns that can be returned from each row using the FIRST clause. This sets an upper limit on the number of columns returned per row. The default is 10,000.

The REVERSED keyword sorts columns in reverse order. Using a FIRST clause, the query returns the columns at the end instead of at the beginning of the range.

Counting Returned Rows

A SELECT expression using COUNT(*) returns the number of rows that matched the query. Alternatively, you can use COUNT(1) to get the same result.

Count the number of rows in the users column family:

```
SELECT COUNT(*) FROM users;
```

Using the LIMIT option, you can specify that the query limit the number of rows returned. For example, the output of these statements if you had 105,291 rows in the database would be: 50000 and 105291

```
SELECT COUNT(*) FROM big_columnfamily LIMIT 50000;
```

```
<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
</tr>
</tbody>
</table>
```

```
SELECT COUNT(*) FROM big_columnfamily LIMIT 200000;
```

```
<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>105291</td>
</tr>
</tbody>
</table>
```

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 <=.

**Specifyng 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 Artists WHERE favoriteArtist = 'Adele' LIMIT 90000;
```

**TRUNCATE**

Removes all data from a column family.

**Synopsis**

```
TRUNCATE <column_family>;
```

**Description**

A TRUNCATE statement results in the immediate, irreversible removal of all data in the named column family.

**Example**

```
TRUNCATE user_activity;
```

**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:

```
KEY | <key_alias> = <key_value>
KEY | <key_alias> 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 and the **KEY** keyword or key alias.

```plaintext
UPDATE Movies SET col1 = val1, col2 = val2 WHERE KEY = key1;
UPDATE Movies SET col3 = val3 WHERE KEY IN (key1, key2, key3);
UPDATE Movies SET col4 = 22 WHERE keyalias = key4;
```

**UPDATE** NerdMovies **USING** CONSISTENCY ALL AND TTL 400

```plaintext
SET 'A 1194' = 'The Empire Strikes Back',
'B 1194' = 'Han Solo'
WHERE KEY = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

```plaintext
UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
```

You can specify these options:

- **Consistency level**
- Time-to-live (TTL)
- Timestamp (client-side time)

TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

**Examples**

Update a column in several rows at once:

```plaintext
UPDATE users USING CONSISTENCY QUORUM
SET 'state' = 'TX'
WHERE KEY IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
06a8913c-c0d6-477c-937d-6c1b69a95d43,
bc108776-7cb5-477f-917d-869c12dfff8a);
```

Update several columns in a single row:

```plaintext
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:

```plaintext
UPDATE page_views USING CONSISTENCY QUORUM AND TIMESTAMP=1318452291034
SET 'index.html' = 'index.html' + 1
WHERE KEY = 'www.datastax.com';
```

**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;

The CQL Shell Program

Using the CQL client, 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, plus the following commands:

ASSUME

Sets the client-side encoding for a cqlsh session.

Synopsis

ASSUME [<keyspace_name>].<columnfamily_name>
<storage_type_definition>
[, ...] ;

<storage_type_definition> is:

(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 (and optionally 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 or alias. Alternatively, you can use the KEY keyword to denote the row key if the row key (or any other column) does not have a name or alias defined in the column family schema.

Examples

ASSUME users NAMES ARE text, VALUES are text;

ASSUME users(KEY) VALUES are uuid;
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. Release 1.0.8 and later.

- **COLUMNFAMILY**: Yields all the information that CQL is capable of representing about the column family.

Examples

Describe a cluster

DESCRIBE CLUSTER;

Sample output is:

**Cluster**: Test Cluster
Partitioner: RandomPartitioner
Snitch: com.datastax.bdp.snitch.DseDelegateSnitch

Describe a keyspace

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

DESCRIBE KEYSPACE Excelsior;

Sample output is:

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

Describe a column family
References

Use Excelsior;

```
CREATE COLUMNFAMILY users ( 
   KEY uuid PRIMARY KEY, 
   username text, 
   email text )
   WITH comment='user information'
   AND read_repair_chance = 1.0;

DESCRIBE COLUMNFAMILY users;
```

Sample output is:

```
CREATE COLUMNFAMILY users ( 
   KEY uuid PRIMARY KEY, 
   email text, 
   username text 
 ) WITH 
   comment='user information' AND 
   comparator=text AND 
   read_repair_chance=1.000000 AND 
   gc_grace_seconds=864000 AND 
   default_validation=text AND 
   min_compaction_threshold=4 AND 
   max_compaction_threshold=32 AND 
   replicate_on_write=True;
```

EXIT

Terminates the CQL command-line client.

**Synopsis**

```
EXIT | QUIT;
```

**SHOW**

Shows the Cassandra version, host, or data type assumptions for the current cqlsh client session.

**Synopsis**

```
SHOW
```

**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**
nodetool

The nodetool utility is a command line interface for Cassandra. You can use it to help manage a cluster.

In binary installations, nodetool is located in the <install_location>/bin directory. Square brackets indicate optional parameters.

**Standard usage:**

`nodetool -h HOSTNAME [-p JMX_PORT] COMMAND`

**RMI usage:**

If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials:

`nodetool -h HOSTNAME [-p JMX_PORT -u JMX_USERNAME -pw JMX_PASSWORD] COMMAND`

**Options**

The available options are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>--host arg</td>
<td>Hostname of node or IP address.</td>
</tr>
<tr>
<td>-p</td>
<td>--port arg</td>
<td>Remote JMX agent port number.</td>
</tr>
<tr>
<td>-pr</td>
<td>--partitioner-range</td>
<td>Repair only the first range returned by the partitioner for the node.</td>
</tr>
<tr>
<td>-pw</td>
<td>--password arg</td>
<td>Remote JMX agent password.</td>
</tr>
<tr>
<td>-t</td>
<td>--tag arg</td>
<td>Optional name to give a snapshot. (snapshot option only)</td>
</tr>
<tr>
<td>-u</td>
<td>--username arg</td>
<td>Remote JMX agent username.</td>
</tr>
</tbody>
</table>

**Command List**

The available commands are:

<table>
<thead>
<tr>
<th>Command List</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cfhistograms</code></td>
</tr>
<tr>
<td><code>cfstats</code></td>
</tr>
<tr>
<td><code>cleanup</code></td>
</tr>
<tr>
<td><code>clearsnapshot</code></td>
</tr>
<tr>
<td><code>compact</code></td>
</tr>
<tr>
<td><code>compactionstats</code></td>
</tr>
<tr>
<td><code>decomission</code></td>
</tr>
<tr>
<td><code>describering</code></td>
</tr>
<tr>
<td><code>disablegossip</code></td>
</tr>
<tr>
<td><code>disablegossip</code></td>
</tr>
<tr>
<td><code>disablegossip</code></td>
</tr>
</tbody>
</table>
**Command Details**

Details for each command are listed below:

**cfhistograms keyspace cf_name**
Displays statistics on the read/write latency for a column family. These statistics, which include row size, column count, and bucket offsets, can be useful for monitoring activity in a column family.

**cfstats**
Displays statistics for every keyspace and column family.

**cleanup [keyspace][cf_name]**
Triggers the immediate cleanup of keys no longer belonging to this node. This has roughly the same effect on a node that a major compaction does in terms of a temporary increase in disk space usage and an increase in disk I/O. Optionally takes a list of column family names.

**clearsnapshot [keyspaces] -t [snapshotName]**
Deletes snapshots for the specified keyspaces. You can remove all snapshots or remove the snapshots with the given name.

**compact [keyspace][cf_name]**
For column families that use the *SizeTieredCompactionStrategy*, initiates an immediate major compaction of all column families in *keyspace*. For each column family in *keyspace*, this compacts all existing SSTables into a single SSTable. This can cause considerable disk I/O and can temporarily cause up to twice as much disk space to be used. Optionally takes a list of column family names.

**compactionstats**
Displays compaction statistics.

**decommission**
Tells a live node to decommission itself (streaming its data to the next node on the ring). Use *netstats* to monitor the progress.

See also:
- http://wiki.apache.org/cassandra/NodeProbe#Decommission

**describering [keyspace]**
Shows the token ranges for a given keyspace.

**disablegossip**
Disable Gossip. Effectively marks the node dead.

**disablethrift**
Disables the Thrift server.

**drain**
Flushes all memtables for a node and causes the node to stop accepting write operations. Read operations will continue to work. You typically use this command before upgrading a node to a new version of Cassandra.
enablegossip
Re-enables Gossip.

enablethrift
Re-enables the Thrift server.

flush [keyspace] [cf_name]
Flushes all memtables for a keyspace to disk, allowing the commit log to be cleared. Optionally takes a list of column
family names.

getcompactionthreshold keyspace cf_name
Gets the current compaction threshold settings for a column family. See:
   http://wiki.apache.org/cassandra/MemtableSSTable

getendpoints keyspace cf key
Displays the end points that owns the key. The key is only accepted in HEX format.

gossipinfo
Shows the gossip information for the cluster.

info
Outputs node information including the token, load info (on disk storage), generation number (times started), uptime in
seconds, and heap memory usage.

invalidatekeycache [keyspace] [cfnames]
Invalidates, or deletes, the key cache. Optionally takes a keyspace or list of column family names. Leave a blank space
between each column family name.

invalidaterowcache [keyspace] [cfnames]
Invalidates, or deletes, the row cache. Optionally takes a keyspace or list of column family names. Leave a blank space
between each column family name.

join
Causes the node to join the ring. This assumes that the node was initially not started in the ring, that is, started with
-Djoin_ring=false. Note that the joining node should be properly configured with the desired options for seed list,
initial token, and auto-bootstrapping.

move new_token
Moves a node to a new token. This essentially combines decommission and bootstrap. See:
   http://wiki.apache.org/cassandra/Operations#Moving_nodes

netstats [host]
Displays network information such as the status of data streaming operations (bootstrap, repair, move, and
decommission) as well as the number of active, pending, and completed commands and responses.

refresh keyspace cf_name]
Loads newly placed SSTables on to the system without restart.

removetoken status | force | token
Shows status of a current token removal, forces the 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 \(-\text{pr}\) 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 \(-\text{pr}\), all replica ranges that the node is responsible for are repaired.

Optionally takes a list of column family names.

\textbf{ring}

Displays node status and information about the ring as determined by the node being queried. This can give you an idea of the load balance and if any nodes are down. If your cluster is not properly configured, different nodes may show a different ring; this is a good way to check that every node views the ring the same way.

\textbf{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 \texttt{upgradesstables}. While \texttt{scrub} rebuilds SSTables, it also discards data that it deems broken and creates a snapshot, which you have to remove manually.

\textbf{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).

\textbf{setcompactionthreshold keyspace cf_name min_threshold max_threshold}

The \texttt{min\_threshold} parameter controls how many SSTables of a similar size must be present before a minor compaction is scheduled. The \texttt{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

\textbf{setcompactionthroughput value\_in\_mb}

Set the maximum throughput for compaction in the system in megabytes per second. Set to 0 to disable throttling.

\textbf{setstreamthroughput value\_in\_mb}

Set the maximum streaming throughput in the system in megabytes per second. Set to 0 to disable throttling (available from Cassandra 1.0.7 onwards).

\textbf{snapshot [keyspace] -t [snapshot-name]}

Takes an online snapshot of Cassandra’s data. Before taking the snapshot, the node is flushed. The results are stored in Cassandra’s data directory under the \texttt{snapshots} directory of each keyspace. See \textit{Install Locations}. For more details, see:

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

\textbf{statusthrift}

Status of the thrift server.

\textbf{tpstats}

Displays the number of active, pending, and completed tasks for each of the thread pools that Cassandra uses for stages of operations. A high number of pending tasks for any pool can indicate performance problems. See:

http://wiki.apache.org/cassandra/Operations#Monitoring

\textbf{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).

\textbf{version}

Displays the 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access.properties=&lt;filename&gt;</td>
<td>The file location of the access.properties file.</td>
</tr>
<tr>
<td>cassandra-pidfile=&lt;filename&gt;</td>
<td>Log the Cassandra server process ID in the named file. Useful for stopping Cassandra by killing its PID.</td>
</tr>
<tr>
<td>cassandra.config=&lt;directory&gt;</td>
<td>The directory location of the Cassandra configuration files.</td>
</tr>
<tr>
<td>cassandra.initial_token=&lt;token&gt;</td>
<td>Sets the initial partitioner token for a node the first time the node is started.</td>
</tr>
<tr>
<td>cassandra.join_ring=&lt;true</td>
<td>false&gt;</td>
</tr>
<tr>
<td>cassandra.load_ring_state=&lt;true</td>
<td>false&gt;</td>
</tr>
<tr>
<td>cassandra.renew_counter_id=&lt;true</td>
<td>false&gt;</td>
</tr>
<tr>
<td>cassandra.replace_token=&lt;token&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 --cassandra.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 --cassandra.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 `stressed`. 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 OrderPreservingPartitioner)
- indexed range slicing (works with RandomPartitioner on indexed ColumnFamilies).

You can use these modes with or without the `stressed` 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 <SUPERCOLUMNNS>, --supercolumns <SUPERCOLUMNNS>
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 RandomPartitioner. You must set the OrderPreservingPartitioner in your storage configuration (note that you will need to wipe all existing data when switching partitioners.)
-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.


**sstable2json**
This converts the on-disk SSTable representation of a column family into a JSON formatted document.

**Usage**

```
bin/sstable2json [-f OUT_FILE] SSTABLE
   [-k KEY [-k KEY [...]]] [-x KEY [-x KEY [...]]] [-e]
```

SSTABLE should be a full path to a `column-family-name-Data.db` file in Cassandra's data directory. For example, `/var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db`.

- `-k` allows you to include a specific set of keys. The `KEY` must be in HEX format. Limited to 500 keys.
- `-x` allows you to exclude a specific set of keys. Limited to 500 keys.
- `-e` causes keys to only be enumerated

**Output Format**

The output of `sstable2json` for standard column families is:

```
{
  ROW_KEY:
  {
    [ 
    [COLUMN_NAME, COLUMN_VALUE, COLUMN_TIMESTAMP, IS_MARKED_FOR_DELETE],
    [COLUMN_NAME, ... ],
    ...
    ],
  },
  ROW_KEY:
  {
    ...
  },
  ...
}
```

The output for super column families is:

```
{
  ROW_KEY:
  {
    SUPERCOLUMN_NAME:
    {
      deletedAt: DELETION_TIME,
      subcolumns:
    [ 
      [COLUMN_NAME, COLUMN_VALUE, COLUMN_TIMESTAMP, IS_MARKED_FOR_DELETE],
    ...
    ],
  }
  }
```
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 KEYSSPACE -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.

**Install Locations**

**Locations of the Configuration Files**

The configuration files, such as `cassandra.yaml`, are located in the following directories:

- Cassandra packaged installs: `/etc/cassandra/conf`
- Cassandra binary installs: `<install_location>/conf`

For DataStax Enterprise installs, see Configuration Files Locations.
**Packaged Installs Directories**

The packaged releases install into the following directories.

- `/var/lib/cassandra` (data directories)
- `/var/log/cassandra` (log directory)
- `/var/run/cassandra` (runtime files)
- `/usr/share/cassandra` (environment settings)
- `/usr/share/cassandra/lib` (JAR files)
- `/usr/bin` (binary files)
- `/usr/sbin`
- `/etc/cassandra` (configuration files)
- `/etc/init.d` (service startup script)
- `/etc/security/limits.d` (cassandra user limits)
- `/etc/default`

**Binary Tarball Install Directories**

The following directories are installed in the installation home directory.

- `bin` (utilities and start scripts)
- `conf` (configuration files and environment settings)
- `interface` (Thrift and Avro client APIs)
- `javadoc` (Cassandra Java API documentation)
- `lib` (JAR and license files)

**Configuring Firewall Port Access**

If you have a firewall running on the nodes in your Cassandra cluster, you must open up the following ports to allow communication between the nodes, including certain Cassandra ports. If this isn't done, when you start Cassandra (or Hadoop in DataStax Enterprise) on a node, the node will act as a standalone database server rather than joining the database cluster.

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Public Facing Ports</strong></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>SSH (default)</td>
</tr>
<tr>
<td><strong>OpsCenter Specific</strong></td>
<td></td>
</tr>
<tr>
<td>8888</td>
<td>OpsCenter website port</td>
</tr>
<tr>
<td><strong>Intranode Ports</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Cassandra Specific</strong></td>
<td></td>
</tr>
<tr>
<td>1024+</td>
<td>JMX reconnection/loopback ports</td>
</tr>
<tr>
<td>7000</td>
<td>Cassandra intra-node port</td>
</tr>
<tr>
<td>7199</td>
<td>Cassandra JMX monitoring port</td>
</tr>
<tr>
<td>9160</td>
<td>Cassandra client port</td>
</tr>
<tr>
<td><strong>OpsCenter Specific</strong></td>
<td></td>
</tr>
</tbody>
</table>
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
```

Start Cassandra in the background (default):

```
$ bin/cassandra -f
```

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:

- Schema Disagreement Exception Occurs
- 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
Troubleshooting Guide

- 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

**Schema Disagreement Exception Occurs**

When setting up a Cassandra cluster, you get an error message about a schema disagreement when multiple schema updates are performed simultaneously, resulting in some nodes in your cluster having a different schema than others.

**To correct this problem:**

Using the Cassandra CLI, enter the `DESCRIBE CLUSTER` command. The output contains the IDs of the schema versions and their corresponding nodes. Remove the problematic schema and migration sstables in your system keyspace as described on the Cassandra wiki.

**To prevent this problem:**

Perform schema changes one at a time, at a steady pace, and from the same node.

**Reads are getting slower while writes are still fast**

Check the SSTable counts in `cfstats`. If the count is continually growing, the cluster’s IO capacity is not enough to handle the write load it is receiving. Reads have slowed down because the data is fragmented across many SSTables and compaction is continually running trying to reduce them. Adding more IO capacity, either via more machines in the cluster, or faster drives such as SSDs, will be necessary to solve this.

If the SSTable count is relatively low (32 or less) then the amount of file cache available per machine compared to the amount of data per machine needs to be considered, as well as the application's read pattern. The amount of file cache can be formulated as \((\text{TotalMemory} - \text{JVMHeapSize})\) and if the amount of data is greater and the read pattern is approximately random, an equal ratio of reads to the cache: data ratio will need to seek the disk. With spinning media, this is a slow operation. You may be able to mitigate many of the seeks by using a key cache of 100%, and a small amount of row cache (10000-20000) if you have some 'hot' rows and they are not extremely large.

**Nodes seem to freeze after some period of time**

Check your system.log for messages from the GCInspector. If the GCInspector is indicating that either the ParNew or ConcurrentMarkSweep collectors took longer than 15 seconds, there is a very high probability that some portion of the JVM is being swapped out by the OS. One way this might happen is if the mmap DiskAccessMode is used without JNA support. The address space will be exhausted by mmap, and the OS will decide to swap out some portion of the JVM that isn’t in use, but eventually the JVM will try to GC this space. Adding the JNA libraries will solve this (they cannot be shipped with Cassandra due to carrying a GPL license, but are freely available) or the DiskAccessMode can be switched to mmap_index_only, which as the name implies will only mmap the indices, using much less address space. DataStax recommends that Cassandra nodes disable swap entirely, 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.
Troubleshooting Guide

- The memtable sizes are too large for the amount of heap allocated to the JVM
  - You can expect \( N + 2 \) memtables resident in memory, where \( N \) is the number of column families. Adding another 1GB on top of that for Cassandra itself is a good estimate of total heap usage.
If none of these seem to apply to your situation, try loading the heap dump in MAT and see which class is consuming the bulk of the heap for clues.

**Nodetool or JMX connections failing on remote nodes**

If you can run nodetool commands locally but not on other nodes in the ring, you may have a common JMX connection problem that is resolved by adding an entry like the following in `<install_location>/conf/cassandra-env.sh` on each node:

```
JVM_OPTS="$JVM_OPTS -Djava.rmi.server.hostname=<public name>"
```

If you still cannot run nodetool commands remotely after making this configuration change, do a full evaluation of your firewall and network security. The nodetool utility communicates through JMX on port 7199.

**View of ring differs between some nodes**

This is an indication that the ring is in a bad state. This can happen when there are token conflicts (for instance, when bootstrapping two nodes simultaneously with automatic token selection.) Unfortunately, the only way to resolve this is to do a full cluster restart; a rolling restart is insufficient since gossip from nodes with the bad state will repopulate it on newly booted nodes.

**Java reports an error saying there are too many open files**

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.

**Insufficient User Resource Limits Errors**

Insufficient resource limits may result in a number of errors in Cassandra, DataStax Enterprise, and OpsCenter, including the following:

**Cassandra Errors**

**Insufficient as (address space) or memlock setting:**

```
ERROR [SSTableBatchOpen:1] 2012-07-25 15:46:02,913 AbstractCassandraDaemon.java (line 139)
Fatal exception in thread Thread[SSTableBatchOpen:1,5,main]
java.io.IOException: java.io.IOException: Map failed  at ...
```

**Insufficient memlock settings:**

```
This can result in part of the JVM being swapped out, especially with mmapped I/O enabled.
```
Troubleshooting Guide

Increase RLIMIT_MEMLOCK or run Cassandra as root.

**Insufficient nofiles setting:**

WARN 05:13:43,644 Transport error occurred during acceptance of message.
org.apache.thrift.transport.TTransportException: java.net.SocketException: Too many open files ...

**Insufficient nofiles setting:**

java.lang.OutOfMemoryError: unable to create new native thread

**OpsCenter Errors**

**Insufficient nofiles setting:**

2012-08-13 11:22:51-0400 [] INFO: Could not accept new connection (EMFILE)

**Recommended Settings**

You can view the current limits using the `ulimit -a` command. Although limits can also be temporarily set using this command, DataStax recommends permanently changing the settings by adding the following entries to your `/etc/security/limits.conf` file:

* soft nofile 32768
* hard nofile 32768
root soft nofile 32768
root hard nofile 32768
* soft memlock unlimited
* hard memlock unlimited
root soft memlock unlimited
root hard memlock unlimited
* soft as unlimited
* hard as unlimited
root soft as unlimited
root hard as unlimited

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

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

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

**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.