Apache Cassandra™ 1.2 Documentation

May 02, 2013

Contents

What's new in Apache Cassandra 1.2.2 and 1.2.3

The latest minor release, Cassandra 1.2.3, includes:

- CQL3-based *implementations of IAuthenticator* and *IAuthorizer* are now available for use with these security features, which were introduced a little earlier:
    - *Internal authentication* based on Cassandra-controlled login accounts and passwords.
    - *Object permission management* using internal authorization to grant or revoke permissions for accessing Cassandra data through the familiar relational database GRANT/REVOKE paradigm.
    - *Client-to-node encryption* that protects data in flight from client machines to a database cluster was also release in Cassandra 1.2.
- This release includes CQL 3, which supports *blob constants*.
- Conversion functions convert *native types to blobs* and perform *timeuuid conversions*.

Key Features

Cassandra 1.2 introduced a number of major improvements:

- *Virtual nodes*: Prior to this release, Cassandra assigned one token per node, and each node owned exactly one contiguous range within the cluster. Virtual nodes change this paradigm from one token and range per node to many tokens per node. This allows each node to own a large number of small ranges distributed throughout the ring. Virtual nodes provide a number of advantages:
    - You no longer have to calculate and assign tokens to each node.
    - Rebalancing a cluster is no longer necessary when adding or removing nodes. When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster. If a node fails, the load is spread evenly across other nodes in the cluster.
    - Rebuilding a dead node is faster because it involves every other node in the cluster and because data is sent to the replacement node incrementally instead of waiting until the end of the validation phase.
    - Improves the use of heterogeneous machines in a cluster. You can assign a proportional number of virtual nodes to smaller and larger machines.
- *Murmur3Partitioner*: This new default partitioner provides faster hashing and improved performance.
- *Faster startup times*: The release provides faster startup/bootup times for each node in a cluster, with internal tests performed at DataStax showing up to 80% less time needed to start primary indexes. The startup reductions were realized through more efficient sampling and loading of indexes into memory caches. The index load time is improved dramatically by eliminating the need to scan the primary index.
- *Improved handling of disk failures*: In previous versions, a single unavailable disk had the potential to make the whole node unresponsive (while still technically alive and part of the cluster). In this scenario, memtables cannot flush and the node eventually runs out of memory. Additionally, if the disk contained the commitlog, data could no longer be appended to the commitlog. Thus, the recommended configuration was to deploy Cassandra on top of RAID 10, but this resulted in using 50% more disk space. Starting with version 1.2, instead of erroring out indefinitely, Cassandra properly reacts to a disk failure, either by stopping the affected node or by blacklisting the failed drive, depending on your availability and consistency requirements. This improvement allows you to deploy Cassandra nodes with large disk arrays without the overhead of RAID 10.
- *Multiple independent leveled compactions in parallel*: Increases the performance of leveled compaction. Cassandra's leveled compaction strategy creates data files of a fixed, relatively small size that are grouped into levels. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Parallel level compaction allows concurrent compactions to be performed between and within different levels, which allows better utilization of all available I/O. For detailed information, see Leveled Compaction in Apache Cassandra.

- *Configurable and more frequent tombstone removal*: Tombstones are removed more often in Cassandra 1.2 and are easier to manage. Cassandra now tracks and removes tombstones automatically. Configuring tombstone removal instead of manually performing compaction can save users time, effort, and disk space.

Concurrent schema changes

While Cassandra 1.1 introduced the ability to modify schema objects in a concurrent fashion across a cluster, it did not include support for programmatically creating and dropping tables (either permanent or temporary) in a concurrent manner. Version 1.2 supplies this functionality. This means multiple users can add/drop tables at the same time in the same cluster allowing programmatic table creation, including temporary tables.

CQL improvements

CQL 3, which was previewed in Beta form in Cassandra 1.1, has been released in Cassandra 1.2.

Note

DataStax Enterprise 3.0.x supports CQL 3 in Beta form. Users need to refer to Cassandra 1.1 documentation for CQL information.

CQL 3 is now the mode for cqlsh. CQL 3 supports schema that map Cassandra storage engine cells to a more powerful and natural row-column representation than earlier CQL versions and the Thrift API. CQL3 transposes data partitions (sometimes called "wide rows") into familiar row-based resultsets, dramatically simplifying data modeling. New features in Cassandra 1.2 include:

- *Collections*: Collections provide easier methods for inserting and manipulating data that consists of multiple items that you want to store in a single column; for example, multiple email addresses for a single employee. There are three different types of collections: set, list, and map. Common tasks that required creating a multiple columns or a separate table can now be accomplished intuitively using a single collection.
- *CQL native/binary protocol*: Although Cassandra continues to support the Thrift RPC indefinitely, the CQL binary protocol is a flexible and higher-performance alternative.
- *Query profiling/request tracing*: This enhancement to cqlsh includes performance diagnostic utilities aimed at helping you understand, diagnose, and troubleshoot CQL statements that are sent to a Cassandra cluster. You can interrogate individual CQL statements in an ad-hoc manner, or perform a system-wide collection of all queries/commands that are sent to a cluster using cqlsh tracing of read/write requests. For collecting all statements that are sent to a database to isolate and tune most resource intensive statements, the nodetool utility, *probabilistic tracing*, has been added.
- *System information*: You can easily retrieve details about your cluster configuration and database objects by querying tables in the system keyspace using CQL.
- *Atomic batches*: Prior versions of Cassandra allowed for batch operations for grouping related updates into a single statement. If some of the replicas for the batch failed mid-operation, the coordinator would hint those rows automatically. However, if the coordinator itself failed in mid operation, you could end up with partially applied batches. In version 1.2 of Cassandra, batch operations are guaranteed by default to be atomic, and are handled differently than in earlier versions of the database.
- *Flat file loader/export utility*: A new cqlsh utility facilitates importing and exporting flat file data to/from Cassandra tables. Although it was initially introduced in Cassandra 1.1.3, the new load utility wasn't formally announced with that version, so an explanation of it is warranted in this document. The utility mirrors the COPY command from the PostgreSQL RDBMS. A variety of file formats are supported including comma-separated value (CSV), tab-delimited, and more, with CSV being the default.

Other enhancements and changes

A number of additional CQL 3 enhancements to Cassandra have been made. See the *list of other features*.

Other CQL 3 Enhancements

Cassandra 1.2 introduced many enhancements in addition to the *key CQL 3 improvements*:

- New statement for altering the replication strategy: *ALTER KEYSPACE*
- *Querying ordered data* using additional operators such as >=.
- *Compound keys and clustering*
- *Composite partition keys*
- *Safeguard against running risky queries*
- *Retrieve more information about cluster topology*
- *Increased query efficiency by using the clustering order*
- *Easy access to column timestamps*
- *Improved secondary index updates*
- A table can now consist of *only a primary key definition*
- *New inet data type*
- *New syntax for setting the replication strategy* using a map collection
- Capability to query *legacy tables*
- Capability to *rename a CQL 3 column*
- *Consistent case-sensitivity rules* for keyspace, table, and column names
- *Configurable tombstone removal* by setting a CQL table property

Other Cassandra 1.2 changes related to CQL 3

Some of these changes affect CQL queries directly and others indirectly.

- *Support for legacy tables*
- *Option used to start cqlsh in CQL 2 mode*
- *CLI command* to query CQL 3 table
- New syntax for setting *compression* and *compaction*
- *New compaction sub-properties* for configuing the size-tiered bucketing process
- Change to level, now set in the driver instead of CQL or globally using *cqlsh*
- Capability to *drop a column from a table* temporarily removed
- The WITH CONSISTENCY LEVEL clause has been removed from CQL commands. Programmatically, you now set the consistency level in the driver. On the command line, a new cqlsh *CONSISTENCY command* has been added.

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 its configuration.

Architecture in brief

Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure. Its architecture is based in the understanding that system and hardware failure can and do occur. Cassandra addresses the problem of failures by employing a peer-to-peer distributed system where all nodes are the same and data is distributed among all nodes in the cluster. Each node exchanges information across the cluster every second. A commit log on each node captures write activity to ensure data durability. Data is also written to an in-memory structure, called a memtable, and then written to a data file called an SStable on disk once the memory structure is full. All writes are automatically partitioned and replicated throughout the cluster.

Cassandra is a row-oriented database. Cassandra's architecture allows any authorized user to connect to any node in any data center and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL. From the CQL perspective the database consists of tables. Typically, a cluster has one keyspace per application. Developers can access CQL through cqlsh as well as via drivers for application languages.

Client read or write requests can go to any node in the cluster. When a client connects to a node with a request, that node serves as the *coordinator* for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured. For more information, see *About client requests*.

The key components for configuring Cassandra are:

- *About internode communications (gossip)*: A peer-to-peer communication protocol to discover and share location and state information about the other nodes in a Cassandra cluster.
- *Partitioner*: A partitioner determines how to distribute the data across the nodes in the cluster. Choosing a partitioner determines which node to place the first copy of data on.
- *Replica placement strategy*: Cassandra stores copies (replicas) of data on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines which nodes to place replicas on. The first replica of data is simply the first copy; it is not unique in any sense.
- *Snitch*: A snitch defines the topology information that the replication strategy uses to place replicas and route requests efficiently.

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. You must configure these cluster contact points for each node in the *cassandra.yaml* configuration file.

Configuring gossip settings

The gossip-related settings are:

Property	Description
cluster_name	Name of the cluster that this node is joining. Should be the same for every node in the cluster.
listen_address	The IP address or hostname that other Cassandra nodes use to connect to this node. Should be changed from `localhost` to the public address for the host.
seed_provider	A `-seeds` list is comma-delimited list of hosts (IP addresses) that gossip uses to learn the topology of the ring. Every node should have the same list of seeds. In multiple data-center clusters, the seed list should include a node from each data center.
storage_port	The intra-node communication port (default is 7000). Must be the same for every node in the cluster.
initial_token	Determines the range of data the node is responsible for in version 1.1 and earlier.
num_tokens	Determines the ranges of data the node is responsible for in version 1.2 and later.

Purging gossip state on a node

Gossip information is persisted locally by each node to use immediately on node 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`` in packed installs or <install_location>/conf in binary installs.

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

To prevent partitions in gossip communications, all nodes in a cluster must 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 remembers 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.

About failure detection and recovery

Failure detection is a method for locally determining, from gossip state and history, if another node in the system is up or down. Cassandra uses this information 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 state from other nodes both directly (nodes gossiping directly to it) and indirectly (nodes communicated about secondhand, thirdhand, and so on). Rather than have a fixed threshold for marking failing nodes, Cassandra uses an accrual detection mechanism to calculate a per-node threshold that takes into account network performance, workload, or other conditions. During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. In Cassandra, configuring the *phi_convict_threshold* property adjusts the sensitivity of the failure detector. Use default value for most situations, but increase it to `12` for Amazon EC2 (due to the frequently experienced network congestion).

Node failures can result from various causes such as hardware failures and network outages. 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 periodically try

to 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* (3 hours 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 distribution and replication

In Cassandra, data distribution and replication go together. This is because Cassandra is designed as a peer-to-peer system that makes copies of the data and distributes the copies among a group of nodes. Data is organized by table and identified by a row key called a primary key. The primary key determines which node the data is stored on. Copies of rows are called replicas. When data is first written, it is also referred to as a replica.

When your create a cluster, you must specify the following:

- *Virtual nodes*: Assigns data ownership to physical machines.
- *Partitioner*: Partitions the data across the cluster.
- *Replication strategy*: Determines the replicas for each row of data.
- *Snitch*: Defines the topology information that the replication strategy uses to place replicas.

Consistent hashing

This section provides more detail about how the consistent hashing mechanism distributes data across a cluster in Cassandra. Consistent hashing partitions data based on the primary key. For example, if you have the following data:

jim	age: 36	car: camaro	gender: M
carol	age: 37	car: bmw	gender: F
johnny	age: 12	gender: M	
suzy	age: 10	gender: F	

Cassandra assigns a hash value to each primary key:

Primary key	Murmur3 hash value
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318

Each node in the cluster is responsible for a range of data based on the hash value:

Node	Murmur3 start range	Murmur3 end range
A	-9223372036854775808	-4611686018427387903
B	-4611686018427387904	-1
C	0	4611686018427387903
D	4611686018427387904	9223372036854775807

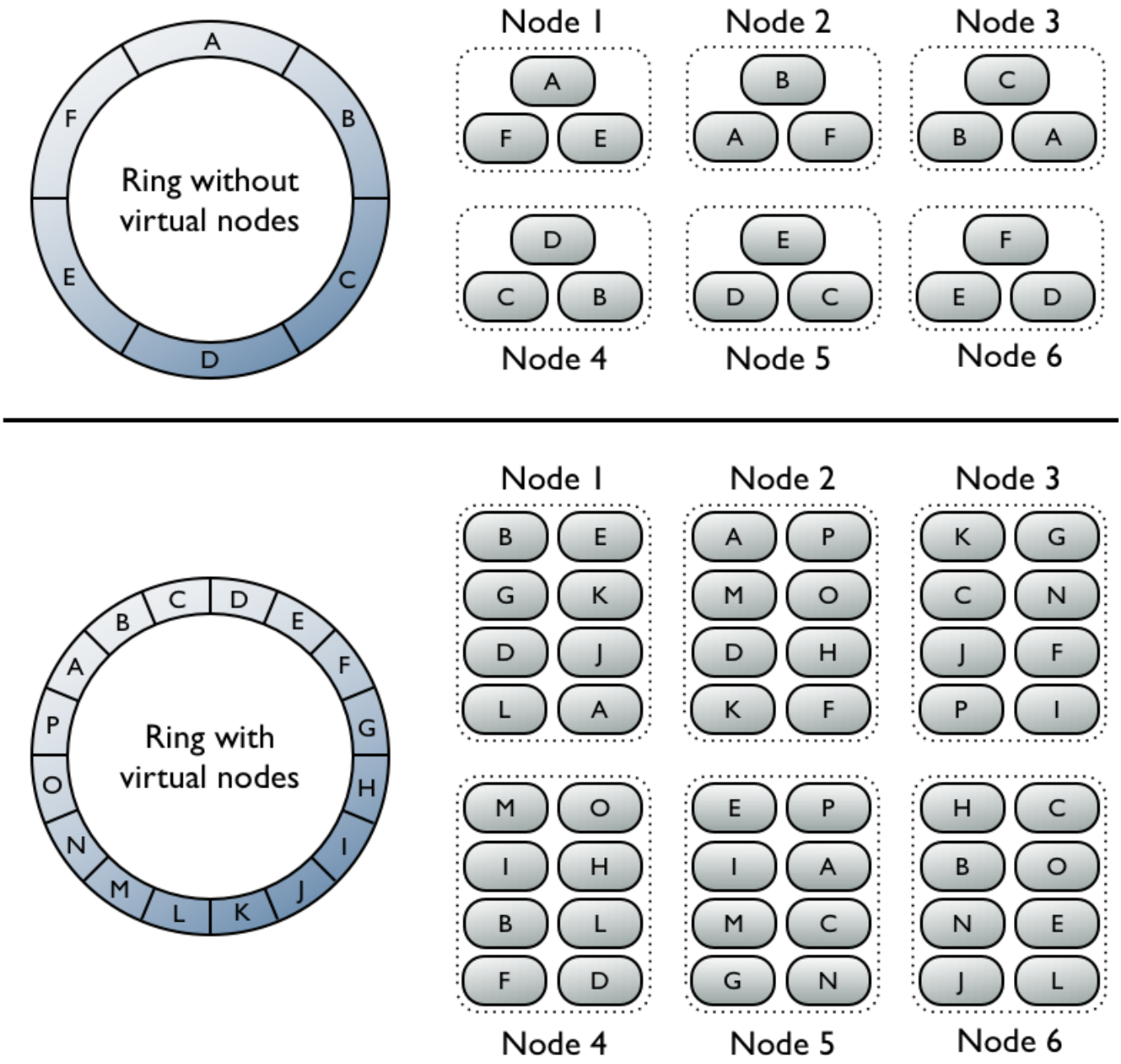
Cassandra places the data on each node according to the value of the primary key and the range that the node is responsible for. For example, in a four node cluster, the data in this example is distributed as follows:

Node	Start range	End range	Primary key	Hash value
A	-9223372036854775808	-4611686018427387903	johnny	-6723372854036780875
B	-4611686018427387904	-1	jim	-2245462676723223822
C	0	4611686018427387903	suzy	1168604627387940318
D	4611686018427387904	9223372036854775807	carol	7723358927203680754

How data is distributed across a cluster (using virtual nodes)

Prior to version 1.2, you had to calculate and assign a single token to each node in a cluster. Each token determined the node's position in the ring and its portion of data according to its hash value. Although the design of consistent hashing used prior to version 1.2 (compared to other distribution designs), allowed moving a single node's worth of data when adding or removing nodes from the cluster, it still required substantial effort to do so.

Starting in version 1.2, Cassandra changes this paradigm from one token and range per node to many tokens per node. The new paradigm is called virtual nodes. Virtual nodes allow each node to own a large number of small ranges distributed throughout the cluster. Virtual nodes also use consistent hashing to distribute data but using them doesn't require token generation and assignment.



The top portion of the graphic shows a cluster without virtual nodes. In this paradigm, each node is assigned a single token that represents a location in the ring. Each node stores data determined by mapping the row key to a token value within a range from the previous node to its assigned value. Each node also contains copies of each row from other nodes in the cluster. For example, range E replicates to nodes 5, 6, and 1. Notice that a node owns exactly one contiguous range in the ring space.

The bottom portion of the graphic shows a ring with virtual nodes. Within a cluster, virtual nodes are randomly selected and non-contiguous. The placement of a row is determined by the hash of the row key within many smaller ranges belonging to each node.

Using virtual nodes

Virtual nodes simplify many tasks in Cassandra:

- You no longer have to calculate and assign tokens to each node.

- Rebalancing a cluster is no longer necessary when adding or removing nodes. When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster. If a node fails, the load is spread evenly across other nodes in the cluster.
- Rebuilding a dead node is faster because it involves every other node in the cluster and because data is sent to the replacement node incrementally instead of waiting until the end of the validation phase.
- Improves the use of heterogeneous machines in a cluster. You can assign a proportional number of virtual nodes to smaller and larger machines.

For more information, see the article Virtual nodes in Cassandra 1.2.

To set up virtual nodes:

Set the number of tokens on each node in your cluster with the *num_tokens* parameter in the *cassandra.yaml* file. The recommended value is 256. Do not set the *initial_token* parameter.

Generally when all nodes have equal hardware capability, they should have the same number of virtual nodes. If the hardware capabilities vary among the nodes in your cluster, assign a proportional number of virtual nodes to the larger machines. For example, you could designate your older machines to use 128 virtual nodes and your new machines (that are twice as powerful) with 256 virtual nodes.

About data replication

Cassandra stores replicas on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines the nodes where replicas are placed.

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 later. 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.

Two replication strategies are available:

- **SimpleStrategy**: Use for a single *data center* only. If you ever intend more than one data center, use the NetworkTopologyStrategy.
- **NetworkTopologyStrategy**: Highly recommended for most deployments because it is much easier to expand to multiple data centers when required by future expansion.

SimpleStrategy

Use only for a single data center. 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 topology (rack or data center location).

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.

NetworkTopologyStrategy places replicas in the same data center by walking the ring clockwise until reaching the first node in another rack. NetworkTopologyStrategy attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) often fail at the same time due to power, cooling, or network issues.

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 data-center 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 either the failure of a one node per replication group at a strong consistency level of LOCAL_QUORUM or 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.

Choosing keyspace replication options

To set the replication strategy for a keyspace, see *CREATE KEYSPACE*.

When you use NetworkToplogyStrategy, during creation of the keyspace *strategy_options*, you use the data center names defined for the *snitch* used by the cluster. To place replicas in the correct location, Cassandra requires a keyspace definition that uses the snitch-aware data center names. For example, if the cluster uses the *PropertyFileSnitch*, create the keyspace using the user-defined data center and rack names in the cassandra-topologies.properties file. If the cluster uses the *EC2Snitch*, create the keyspace using EC2 data center and rack names.

Partitioners

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

Both the Murmur3Partitioner and RandomPartitioner use tokens to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace. This is true even if the tables use different row keys, such as usernames or timestamps. Moreover, the read and write requests to the cluster are also evenly distributed and load balancing is simplified because each part of the hash range receives an equal number of rows on average. For more detailed information, see *Consistent hashing*.

Cassandra offers the following partitioners:

- *Murmur3Partitioner* (default): Uniformly distributes data across the cluster based on MurmurHash hash values.
- *RandomPartitioner*: Uniformly distributes data across the cluster based on MD5 hash values.
- *ByteOrderedPartitioner*: Keeps an ordered distribution of data lexically by key bytes

The Murmur3Partitioner is the default partitioning strategy for new Cassandra clusters and the right choice for new clusters in almost all cases.

About computing tokens

If you are using virtual nodes, you do not need to calculate the tokens.

If you are not using virtual nodes, you must calculate the tokens to assign to the *initial_token* parameter in the cassandra.yaml file. See *Generating tokens* and use the method for the type of partitioner you are using.

About the Murmur3Partitioner

The Murmur3Partitioner (org.apache.cassandra.dht.Murmur3Partitioner) provides faster *hashing* and improved performance than the previous default partitioner (RandomPartitioner).

Note

You can only use Murmur3Partitioner for new clusters; you cannot change the partitioner in existing clusters. If you are switching to the 1.2 *cassandra.yaml*, be sure to change the partitioner setting to match the previous partitioner.

The Murmur3Partitioner uses the MurmurHash function. This hashing function creates a 64-bit hash value of the row key. The possible range of hash values is from -2^{63} to $+2^{63}$.

When using the Murmur3Partitioner, you can page through all rows using the *token function* in a CQL 3 query.

About the RandomPartitioner

Although no longer the default partitioner, you can use the RandomPartitioner (org.apache.cassandra.dht.RandomPartitioner) in version 1.2, even when using virtual nodes. However, if you don't use virtual nodes, you must calculate the tokens, as described in *Generating tokens*.

The RandomPartition distributes data evenly across the nodes using an MD5 hash value of the row key. The possible range of hash values is from 0 to 2^{127} -1.

When using the RandomPartitioner, you can page through all rows using the *token function* in a CQL 3 query.

About the ByteOrderedPartitioner

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

Using the ordered partitioner allows ordered scans by primary key. 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 using randomly partitioned row keys because the keys are stored in the order of their MD5 hash (not sequentially).

Although having the capability to do range scans on rows sounds like a desirable feature of ordered partitioners, there are ways to achieve the same functionality using *table indexes*.

Using an ordered partitioner is not recommended for the following reasons:

- **Difficult load balancing.** More administrative overhead is required 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.
- **Sequential writes can cause hot spots.** If your application tends to write or update a sequential block of rows at a time, then the writes are not be distributed across the cluster; they all go to one node. This is frequently a problem for applications dealing with timestamped data.
- **Uneven load balancing for multiple tables.** If your application has multiple tables, chances are that those tables have different row keys and different distributions of data. An ordered partitioner that is balanced for one table may cause hot spots and uneven distribution for another table in the same cluster.

Types of snitches

A snitch has two functions:

- It determines which data centers and racks are written to and read from and informs Cassandra about the network topology so that requests are routed efficiently.
- It allows Cassandra to distribute replicas by grouping machines into data centers and racks. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).

Note

If you change the snitch after data is inserted into the cluster, you must run a full repair, since the snitch affects where replicas are placed.

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).

Using a SimpleSnitch, the only keyspace *strategy option* you specify is a replication factor.

RackInferringSnitch

The RackInferringSnitch determines the location of nodes by rack and data center, which are assumed to correspond to the 3rd and 2nd octet of the node's IP address, respectively. Use this snitch as an example of writing a custom Snitch class.

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 cassandra-topology.properties file. Use this snitch when your node IPs are not uniform or if you have complex replication grouping requirements as shown in *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 you define in the `cassandra-topology.properties` file correlates to the name of your data centers in your keyspace *strategy_options*. Every node in the cluster should be described in the `cassandra-topology.properties` file, and this file should be exactly the same on every node in the cluster.

The location of the `cassandra-topology.properties` file depends on the type of installation; see *Cassandra Configuration Files Locations* or DataStax Enterprise Configuration Files Locations.

Configuring the PropertyFileSnitch

If 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, the `cassandra-topology.properties` file might look like this:

```
# Data Center One

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

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

# Data Center Two
```

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

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

# Analytics Replication Group

172.106.12.120=DC3:RAC1
172.106.12.121=DC3:RAC1
172.106.12.122=DC3:RAC1

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

GossipingPropertyFileSnitch

The GossipingPropertyFileSnitch defines a local node's data center and rack; it uses gossip for propagating this information to other nodes. The conf/cassandra-rackdc.properties file defines the default data center and rack used by this snitch:

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

The location of the `conf` directory depends on the type of installation; see *Cassandra Configuration Files Locations* or DataStax Enterprise Configuration Files Locations

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

EC2Snitch

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

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

EC2MultiRegionSnitch

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

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

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

```
seeds: 50.34.16.33, 60.247.70.52
```

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

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

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

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

About Dynamic Snitching

By default, all snitches also use a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes. The dynamic snitch is enabled by default and is recommended for use in most deployments. For information on how this works, see Dynamic snitching in Cassandra: past, present, and future.

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

About client requests

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.

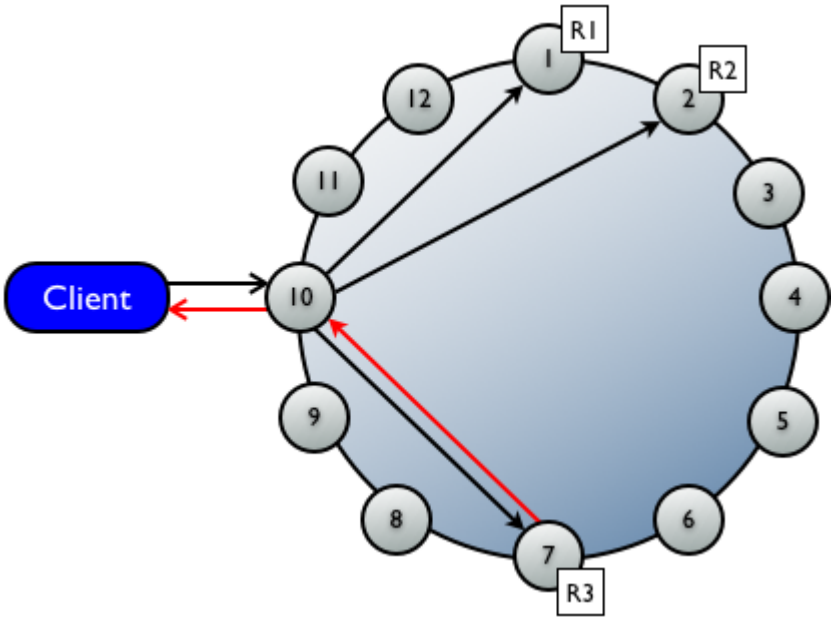
The job of the coordinator is to act as a proxy between the client application and the nodes (or replicas) that own the data being requested. The coordinator determines which nodes in the ring should get the request based on the cluster configured *partitioner* and *replica placement strategy*.

About write requests

The coordinator sends a write request 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 acknowledgment in order for the write to be considered successful. Success means that the data was written to the commit log and the memtable as described in *About writes*.

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, Cassandra will make the row consistent later using one of its *built-in repair mechanisms*: hinted handoff, read repair, or anti-entropy node repair.

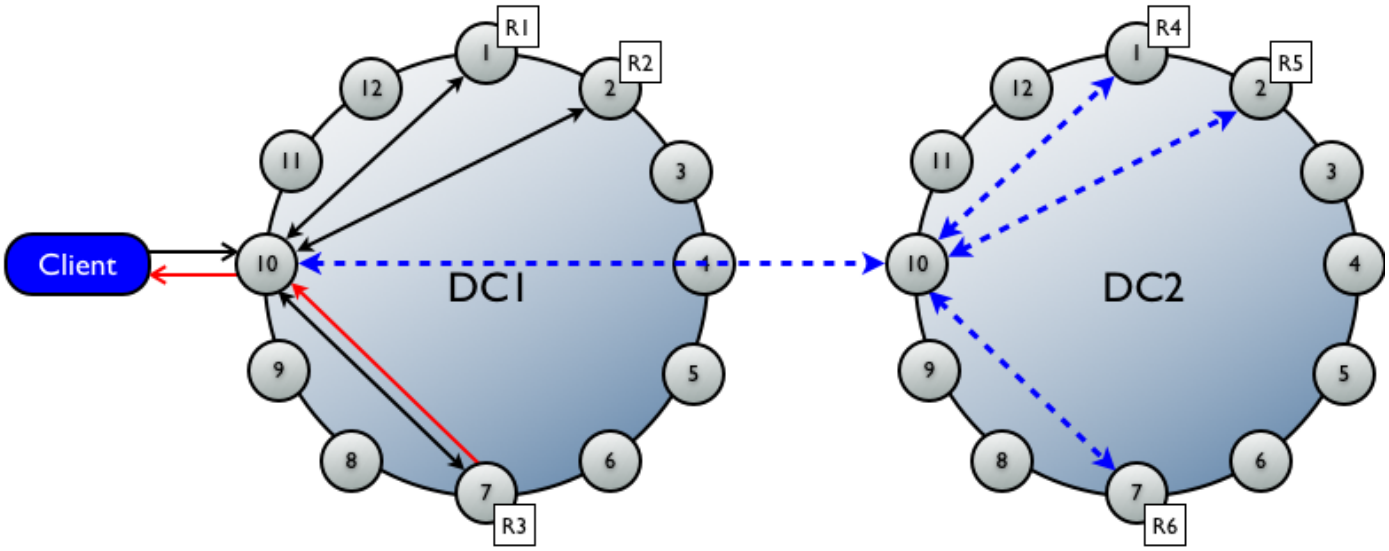
That node forwards the write to all replicas of that row. It will respond back to the client once it receives a write acknowledgment from the number of nodes specified by the consistency level. when a node writes and responds, that means it has written to the commit log and puts the mutation into a memtable.



About multiple data center write requests

In multiple 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

There are two types of read requests that a coordinator can send to a replica:

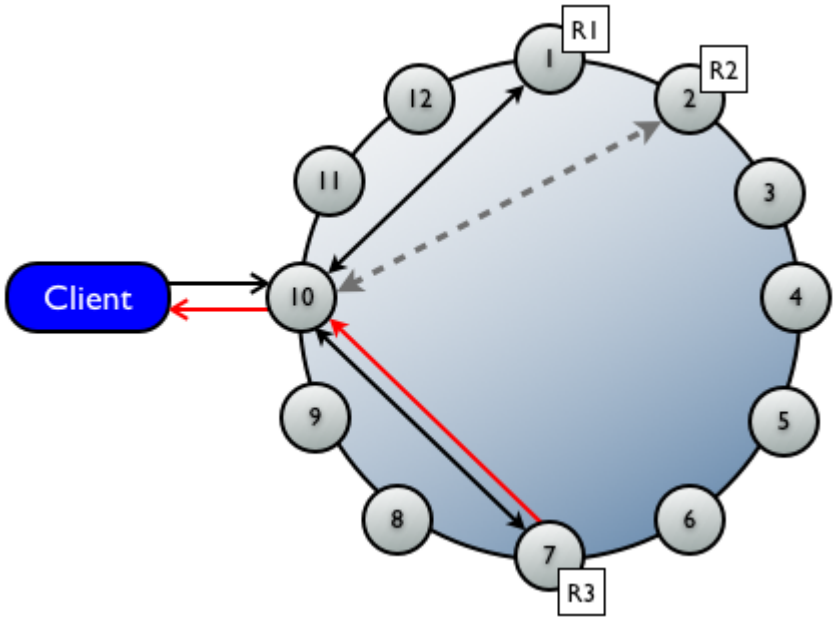
- A direct read request
- A background *read repair* request

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

To ensure that all replicas have the most recent version of frequently-read data, the coordinator also contacts and compares the data from all the remaining replicas that own the row in the background. If the replicas are inconsistent, the coordinator issues writes to the out-of-date replicas to update the row to the most recent values. This process is known as *read repair*. Read repair can be configured per table (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.



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. After reading this section, it recommended that you read *Anti-patterns in Cassandra*.

Selecting hardware for enterprise implementations

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

Memory

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

- For dedicated hardware, the optimal price-performance sweet spot is 16GB to 64GB; the minimum is 8GB.
- For a virtual environments, the optimal range may be 8GB to 16GB; the minimum is 4GB.
- For testing light workloads, Cassandra can run on a virtual machine as small as 256MB.
- For setting Java heap space, see *Tuning Java resources*.

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

Disk space depends a lot on usage, so it's important to understand the mechanism. Cassandra writes data to disk when appending data to the *commit log* for durability and when flushing *memtables* to *SSTable* data files for persistent storage. SSTables are periodically compacted. Compaction improves performance by merging and rewriting data and discarding old data. However, depending on the type of *compaction_strategy* and size of the compactions, compaction can substantially increase disk utilization and data directory volume. For this reason, you should leave an adequate amount of free disk space available on a node: 50% (worst case) for SizeTieredCompactionStrategy and large compactions, and 10% for LeveledCompactionStrategy. The following links provide information about compaction:

- *Configuring compaction and compression*
- The Apache Cassandra storage engine
- Leveled Compaction in Apache Cassandra
- When to Use Leveled Compaction

For information on calculating disk size, see *Calculating usable disk capacity* and *Choosing node configuration options*.

Recommendations:

- **Capacity and I/O**: When choosing disks, consider both capacity (how much data you plan to store) and I/O (the write/read throughput rate). Some workloads are best served by using less expensive SATA disks and scaling disk capacity and I/O by adding more nodes (with more RAM).
- **Solid-state drives**: SSDs are the recommended choice for Cassandra. Cassandra's sequential, streaming write patterns minimize the undesirable effects of write amplification associated with SSDs. This means that Cassandra deployments can take advantage of inexpensive consumer-grade SSDs. Enterprise level SSDs are not necessary because Cassandra's SSD access wears out consumer-grade SSDs in the same time frame as more expensive enterprise SSDs.
- **Number of disks - SATA**: 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 - SATA**: The disk 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 on data disks**: It is generally not necessary to use RAID for the following reasons:
  - Data is replicated across the cluster based on the replication factor you've chosen.
  - Starting in version 1.2, Cassandra includes takes care of disk management with the JBOD (Just a bunch of disks) support feature. Because Cassandra properly reacts to a disk failure, based on your availability/consistency requirements, either by stopping the affected node or by blacklisting the failed drive, this allows you to deploy Cassandra nodes with large disk arrays without the overhead of RAID 10.
- **RAID on the commit log disk**: Generally RAID is not needed for the commit log disk. Replication adequately prevents data loss. If you need the extra redundancy, use RAID 1.
- **Extended file systems**: DataStax recommends deploying Cassandra on XFS. 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 using large disks, 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

Prior to version 1.2, the recommended size of disk space per node was 300 to 500GB. Improvement to Cassandra 1.2, such as JBOD support, virtual nodes, off-heap Bloom filters, and parallel leveled compaction (SSD nodes only), allow you to use few machines with multiple terabytes of disk space.

Network

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

- Recommended 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 efficiently routes requests to replicas that are geographically closest to the coordinator node and chooses a replica in the same rack if possible; it always chooses 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. See *Configuring firewall port access*.

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

Planning an Amazon EC2 cluster

DataStax provides an Amazon Machine Image (AMI) to allow you to quickly deploy a multi-node Cassandra cluster on Amazon EC2. The DataStax AMI initializes all nodes in one availability zone using the *SimpleSnitch*.

If you want an EC2 cluster that spans multiple regions and availability zones, do not use the DataStax AMI. Instead, install Cassandra on your EC2 instances as described in *Installing Cassandra Debian packages*, and then configure the cluster as a *multiple data center cluster*.

Use the following guidelines when setting up your cluster:

- For production Cassandra clusters on EC2, use Large or Extra Large instances with local storage.

  Amazon Web Service recently reduced the number of default ephemeral disks attached to the image from four to two. Performance will be slower for new nodes unless you manually attach the additional two disks; see Amazon EC2 Instance Store.
- RAID 0 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 more data redundancy, consider deploying your Cassandra cluster across multiple availability zones or using EBS volumes to store your Cassandra backup files.
- Cassandra JBOD support allows you to use standard disks, but you may get better throughput with RAID0. RAID0 splits every block to be on another device so that writes are written in parallel fashion instead of written serially on disk.
- EBS volumes are not recommended for Cassandra data volumes for the following reasons:
  - EBS volumes contend directly for network throughput with standard packets. This means that EBS throughput is likely to fail if you saturate a network link.
  - EBS volumes have unreliable performance. I/O performance can be exceptionally slow, causing the system to backload reads and writes until the entire cluster becomes unresponsive.
  - Adding capacity by increasing the number of EBS volumes per host does not scale. You can easily surpass the ability of the system to keep effective buffer caches and concurrently serve requests for all of the data it is responsible for managing.

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

Calculating usable disk capacity

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

Start with the raw capacity of the physical disks:

```
raw_capacity = disk_size * number_of_data_disks
```

Account for file system formatting overhead (roughly 10 percent):

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

During normal operations, Cassandra routinely requires disk capacity for compaction and repair operations. For optimal performance and cluster health, DataStax recommends not filling your disks to capacity, but running at 50% to 80% capacity depending on the *compaction_strategy* and size of the compactions. With this in mind, calculate the usable disk space as follows:

```
formatted_disk_space * (0.5 to 0.8) = 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 is about two 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 tables. The following calculations 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 table 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 total). So the total size of a regular column is:

```
regular_total_column_size = column_name_size + column_value_size + 15

counter-expiring_total_column_size = column_name_size + column_value_size + 23
```

- **Row Overhead**: Every row in Cassandra incurs 23 bytes of overhead.
- **Primary Key Index**: Every table also maintains a primary index of its row keys. Sizing of the primary row key index can be estimated as follows (in bytes):

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

- **Replication Overhead**: The replication factor 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 data is stored in the cluster). If replication factor is greater than 1, then your total data storage requirement will include replication overhead.

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

Choosing node configuration options

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

- **Storage settings**: By default, a node is configured to store the data it manages in the /var/lib/cassandra directory. In a production cluster deployment, you change the *commitlog_directory* to a different disk drive from the *data_file_directories*.
- **Gossip settings**: The *gossip-related settings* control a node's participation in a cluster and how the node is known to the cluster.
- **Purging gossip state on a node**: Gossip information is also persisted locally by each node to use immediately when a node restarts. You may want to *purge gossip history* on node restart for various reasons, such as when the node's IP addresses has changed.
- **Partitioner settings**: You must set the *partitioner type* and assign the node a *num_tokens* value for each node. If not using virtual nodes, use the *initial_token* setting instead.
- **Snitch settings**: You need to configure a *snitch* when you create a cluster. The snitch is responsible for knowing the location of nodes within your network topology and distributing replicas by grouping machines into data centers and racks.
- **Keyspace replication settings**: When you create a keyspace, you must define the *replica placement strategy* and the number of replicas you want.

Anti-patterns in Cassandra

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

Network attached storage

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

- Router latency.

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

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

Shared network file systems

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

Excessive heap space size

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

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

For information on heap sizing, see *Tuning Java resources*.

Cassandra's rack feature

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

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

To use racks correctly:

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

Multiple-gets

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

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

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

Super columns

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

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

Using the Byte Ordered Partitioner

The Byte Ordered Partitioner (BOP) is not recommended.

Use *virtual nodes* and use either the *Murmur3Partitioner* (default) or *RandomPartitioner*. Virtual nodes allow each node to own a large number of small ranges distributed throughout the cluster. Using virtual nodes saves you the effort of generating tokens and assigning tokens to your nodes. If not using virtual nodes, these partitioners are recommended because all writes occur on the hash of the key and are therefore spread out throughout the ring amongst tokens range. These partitioners ensure that your cluster evenly distributes data by placing the key at the correct token using the key's hash value. Even if data becomes stale and needs to be deleted, this ensures that data removal also takes place while evenly distributing data around the cluster.

Reading before writing

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

Load balancers

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

Insufficient testing

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

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

Lack of familiarity with Linux

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

1 Not maxed out.

- **Parallel SSH and Cluster SSH**: The pssh and cssh tools allow SSH access to multiple nodes. This is useful for inspections and cluster wide changes.
- **Passwordless SSH**: SSH authentication is carried out by using public and private keys. This allows SSH connections to easily hop from node to node without password access. In cases where more security is required, you can implement a password Jump Box and/or VPN.
- **Useful common command-line tools include:**
    - **top**: Provides an ongoing look at processor activity in real time.
    - **System performance tools**: Tools such as iostat, mpstat, iftop, sar, lsof, netstat, htop, vmstat, and similar can collect and report a variety of metrics about the operation of the system.
    - **vmstat**: Reports information about processes, memory, paging, block I/O, traps, and CPU activity.
    - **iftop**: Shows a list of network connections. Connections are ordered by bandwidth usage, with the pair of hosts responsible for the most traffic at the top of list. This tool makes it easier to identify the hosts causing network congestion.

Running without the recommended settings

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

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

More anti-patterns

For more about anti-patterns, visit the Matt Dennis slideshare.

Installing a Cassandra Cluster

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

Note

For information on installing for evaluation or installing on Windows, see the Quick Start Documentation.

Installing Cassandra RHEL or CentOS packages

DataStax provides `yum` repositories for CentOS and RedHat Enterprise. To install on SUSE, use the *Cassandra binary tarball distribution*.

For a complete list of supported platforms, see DataStax Community – Supported Platforms.

Note

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

Prerequisites

Before installing Cassandra make sure the following prerequisites are met:

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

Steps to install Cassandra

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

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

   ```
   java -version
   ```

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

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

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

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

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

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

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

5. Install the package using `yum`.

```
$ sudo yum install dsc12
```

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

Next steps

- *Initializing a multiple node cluster*
- *Install locations*

Installing Cassandra Debian packages

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

Note

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

Prerequisites

Before installing Cassandra make sure the following prerequisites are met:

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

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
   ```

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

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

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

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

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

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

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

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

5. Install the package.

   ```
   $ sudo apt-get update
   $ sudo apt-get install dsc12
   ```

   This installs the DataStax Community distribution of Cassandra and the OpsCenter Community Edition. By default, the Debian packages start the Cassandra service automatically.

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

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

Next steps

- *Initializing a multiple node cluster*
- *Install locations*

Installing the Cassandra binary tarball distribution

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

Note

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

Prerequisites

Before installing Cassandra make sure the following prerequisites are met:

- The latest version of Oracle Java SE Runtime Environment (JRE) **6** is installed. Java 7 is not recommended.

- Java Native Access (JNA) is required for production installations. See *Installing JNA*.
- Also see *Recommended settings for production installations*.

> ***Note***
> If you are using Ubuntu 10.04 LTS, you need to update to JNA 3.4, as described in *Install JNA on Ubuntu 10.04*.

Steps to install Cassandra

1. Download the Cassandra DataStax Community tarball:

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

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

   ```
   java -version
   ```

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

3. Unpack the distribution:

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

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

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

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

Next steps

- *Initializing a multiple node cluster*
- *Install locations*

Recommended settings for production installations

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

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

```
*  soft  nproc  10240
```

For more information, see *Insufficient user resource limits errors*.

Disable swap

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

```
sudo swapoff --all
```

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

Synchronize clocks

The clocks on all nodes should be synchronized. You can use NTP (Network Time Protocol) or other methods.

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

Optimum blockdev --setra settings for RAID

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

Check to ensure setra is not set to 65536:

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

To set setra:

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

Java Virtual Machine

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

Java Native Access

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

Installing the JRE and JNA

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

Installing Oracle JRE

Select one of the following:

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

Note

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

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

Installing the JRE on RHEL or CentOS Systems

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

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

   ```
   java -version
   ```

2. If necessary, go to Oracle Java SE Downloads, accept the license agreement, and download the `Linux x64-RPM` installer.
3. Go to the directory where you downloaded the JRE package, and change the permissions so the file is executable:

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

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

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

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

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

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

   ```
   $ java -version

     java version "1.6.0_43"
     Java(TM) SE Runtime Environment (build 1.6.0_43-b05)
     Java HotSpot(TM) 64-Bit Server VM (build 20.13-b02, mixed mode)
   ```

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

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

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

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

Installing the JRE on Debian or Ubuntu Systems

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

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

```
java -version
```

2. If necessary, download the appropriate version of the JRE, such as `jre-6u43-linux-i586.bin`, for your system and place it in `/usr/java/latest`.
3. Make the file executable:

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

4. Go to the new folder:

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

5. Execute the file:

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

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

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

Note

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

7. Set the new JRE as the default:

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

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

```
$ java -version

  java version "1.6.0_43"
  Java(TM) SE Runtime Environment (build 1.6.0_43-b05)
  Java HotSpot(TM) 64-Bit Server VM (build 20.13-b02, mixed mode)
```

Installing the JRE on SUSE Systems

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

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

   ```
   java -version
   ```

2. If necessary, go to Oracle Java SE Downloads, accept the license agreement, and download the `Linux x64-RPM` installer.
3. Go to the directory where you downloaded the JRE package, and change the permissions so the file is executable:

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

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

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

   The RPM installs the JRE into `/usr/java/`.
5. Make sure your system is now using the correct JRE. For example:

   ```
   $ java -version

     java version "1.6.0_43"
     Java(TM) SE Runtime Environment (build 1.6.0_43-b05)
     Java HotSpot(TM) 64-Bit Server VM (build 20.13-b02, mixed mode)
   ```

Installing JNA

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

Debian or Ubuntu Systems

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

Ubuntu 10.04 LTS

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

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

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

RHEL or CentOS Systems

Install with the following command:

```
# yum install jna
```

SUSE Systems

Install with the following commands:

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

Tarball Installations

Install with the following commands:

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

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

Installing a Cassandra cluster on Amazon EC2

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

The DataStax Cassandra AMI does the following:

- Launches the newest stable release of DataStax Community Edition.
- Installs Cassandra on an Ubuntu 12.04 LTS (Precise Pangolin) image (Ubuntu Cloud 20121218 release).
- Uses RAID0 ephemeral disks for data storage and commit logs.
- Uses the private interface for intra-cluster communication.
- Configures a Cassandra cluster using the RandomPartitioner.
- Configures the Cassandra replication strategy using the EC2Snitch.
- Sets the seed node cluster-wide.
- Starts Cassandra on all the nodes.
- Installs DataStax OpsCenter on the first node in the cluster (by default).

If you want an EC2 cluster that spans multiple regions and availability zones, do not use the DataStax AMI. Instead, install Cassandra on your EC2 instances as described in *Installing Cassandra Debian packages*, and then configure the cluster as a *multiple data center cluster*.

Production considerations

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 more data redundancy, consider deploying your Cassandra cluster across multiple availability zones or using EBS volumes to store your Cassandra backup files.

Creating an EC2 security group for DataStax Community Edition

An EC2 Security Group acts as a firewall that allows you to choose which protocols and ports are open in your cluster. You can specify the protocols and ports either by a range of IP addresses or by security group. The default EC2 security

group opens all ports and protocols only to computers that are members of the default group. This means you must define a security group for your Cassandra cluster. Be aware that specifying a Source IP of 0.0.0.0/0 allows every IP address access by the specified protocol and port range.

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

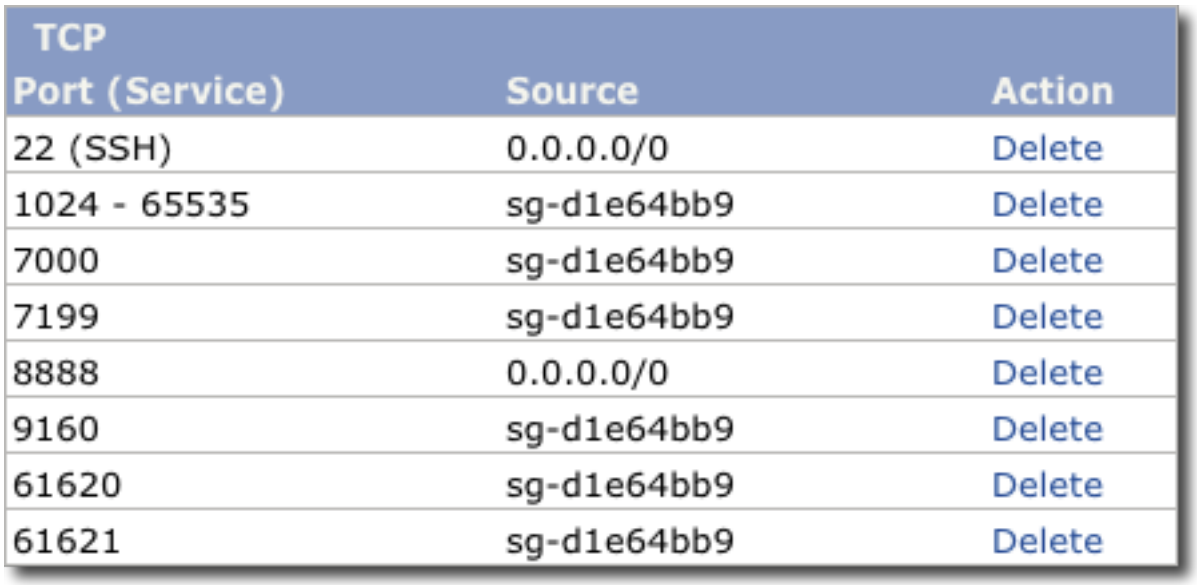
3. Click the **Inbound** tab and add rules for the ports listed in the table below:
   - **Create a new rule**: Custom TCP rule.
   - **Port range**: See table.
   - **Source**: See table. To create rules that open a port to other nodes in the same security group, use the **Group ID** listed in the Group Details tab.

Port	Description
Public Facing Ports	
22	SSH port.
8888	OpsCenter website port.
Cassandra Inter-node Ports	
1024+	JMX reconnection/loopback ports. See description for port 7199.
7000	Cassandra inter-node cluster communication.
7199	Cassandra JMX monitoring port. After the initial handshake, the JMX protocol requires that the client reconnects on a randomly chosen port (1024+).
9160	Cassandra client port (Thrift).
OpsCenter ports	
61620	OpsCenter monitoring port. The opscenterd daemon listens on this port for TCP traffic coming from the agent.
61621	OpsCenter agent port. The agents listen on this port for SSL traffic initiated by OpsCenter.

Note

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

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



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

Warning

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

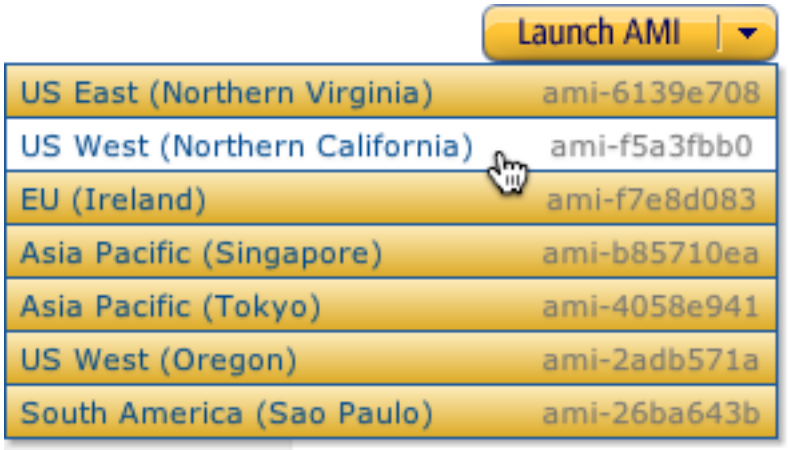
Launching the DataStax Community AMI

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

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

   https://aws.amazon.com/amis/datastax-auto-clustering-ami-2-2

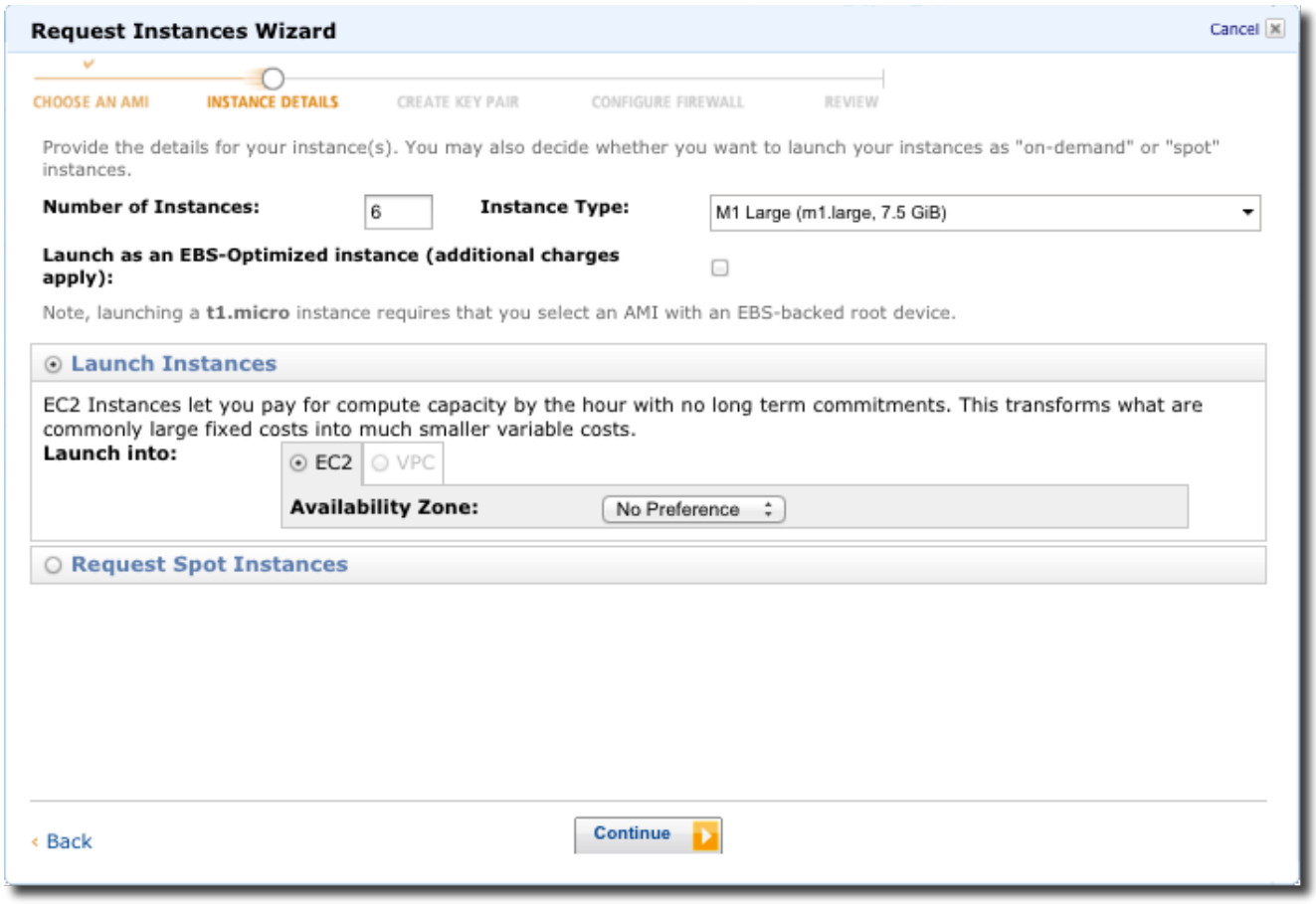
2. Click **Launch AMI**, then select the region where you want to launch the AMI.



3. On the **Request Instances Wizard** page, verify the settings and then click **Continue**.
4. On the **Instance Details** page, enter the total number of nodes that you want in your cluster, select the **Instance Type**, and then click **Continue**.

   Use the following guidelines when selecting the type of instance:

   - Extra large for production.
   - Large for development and light production.
   - Small and Medium not supported.



Note

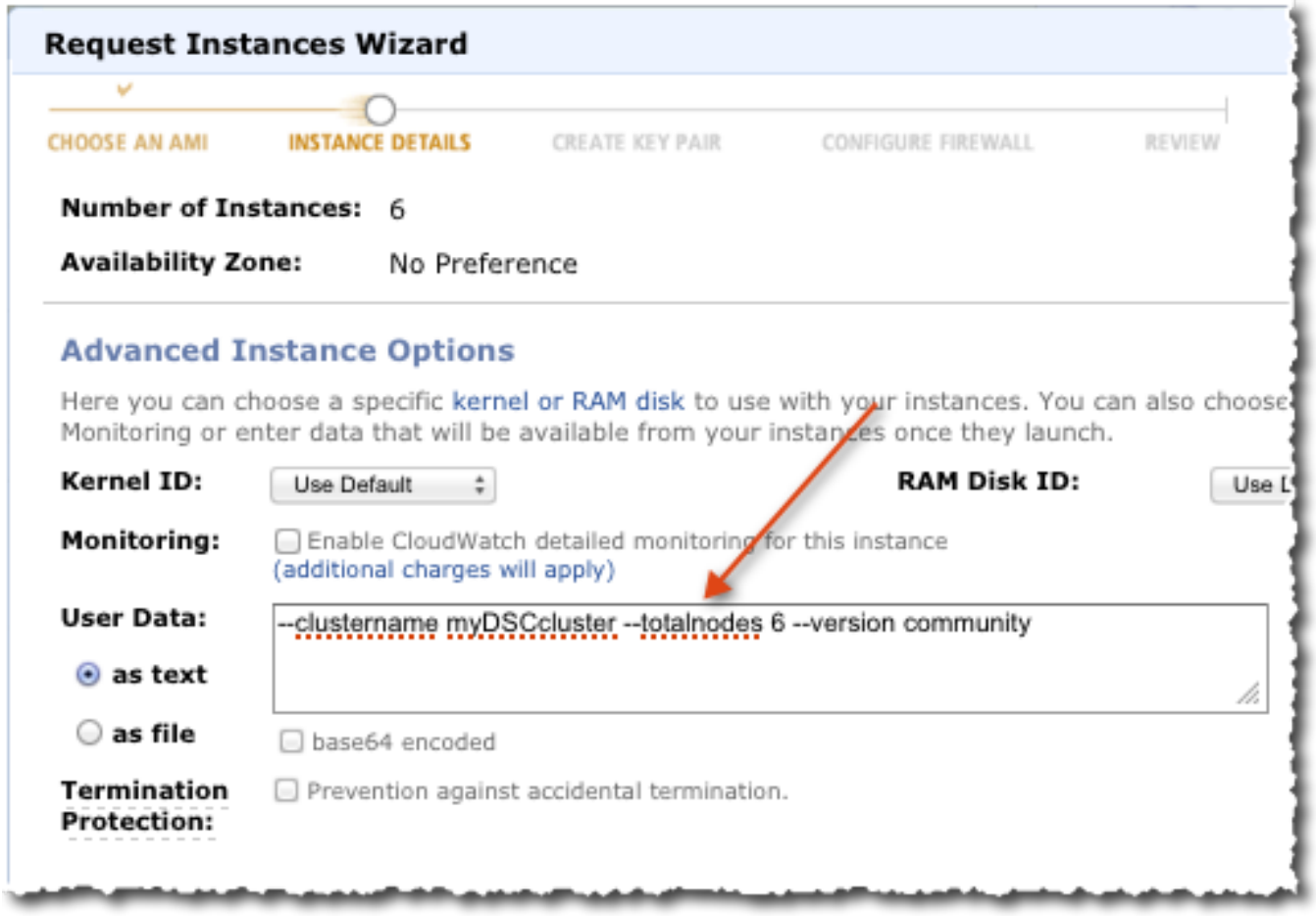
EBS volumes are not recommended. In Cassandra data volumes, EBS throughput may fail in a saturated network link, I/O may be exceptionally slow, and adding capacity by increasing the number of EBS volumes per host does not scale. For more information and graphs related to ephemeral versus EBS performance, see the blog article at http://blog.scalyr.com/2012/10/16/a-systematic-look-at-ec2-io/.

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

   For new clusters the available options are:

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

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



6. On the **Storage Device Configuration** page, you can add ephemeral drives if needed.

Note

Amazon Web Service recently reduced the number of default ephemeral disks attached to the image from four to two. Performance will be slower for new nodes unless you manually attach the additional two disks; see Amazon EC2 Instance Store.

7. On the **Tags** page, give a name to your DataStax Community instance, such as `cassandra-node`, and then click **Continue**.
8. On the **Create Key Pair** page, create a new key pair or select an existing key pair, and then click **Continue**. Save this key (`.pem` file) to your local machine; you will need it to log in to your DataStax Community instance.
9. On the **Configure Firewall** page, select the security group that you created earlier and click **Continue**.
10. On the **Review** page, review your cluster configuration and then click **Launch**.
11. Close the **Launch Install Wizard** and go to the **My Instances** page to see the status of your Cassandra instance. Once a node has a status of **running**, you can connect to it.

Connecting to your DataStax Community EC2 instance

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

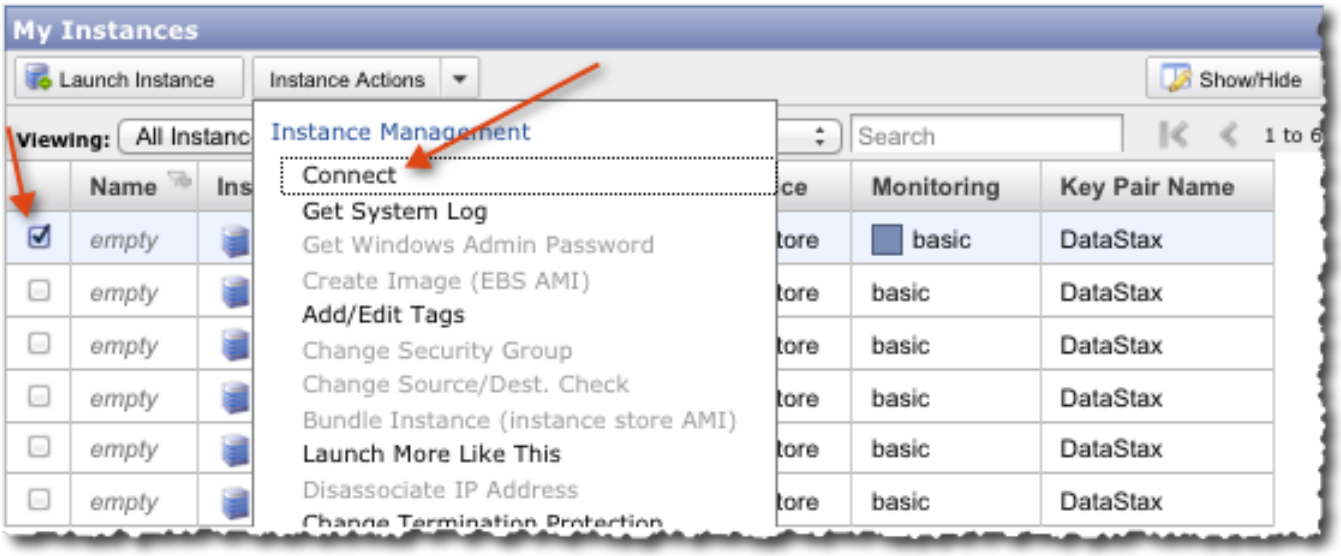
Connect as user `ubuntu` rather than as `root`.

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

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

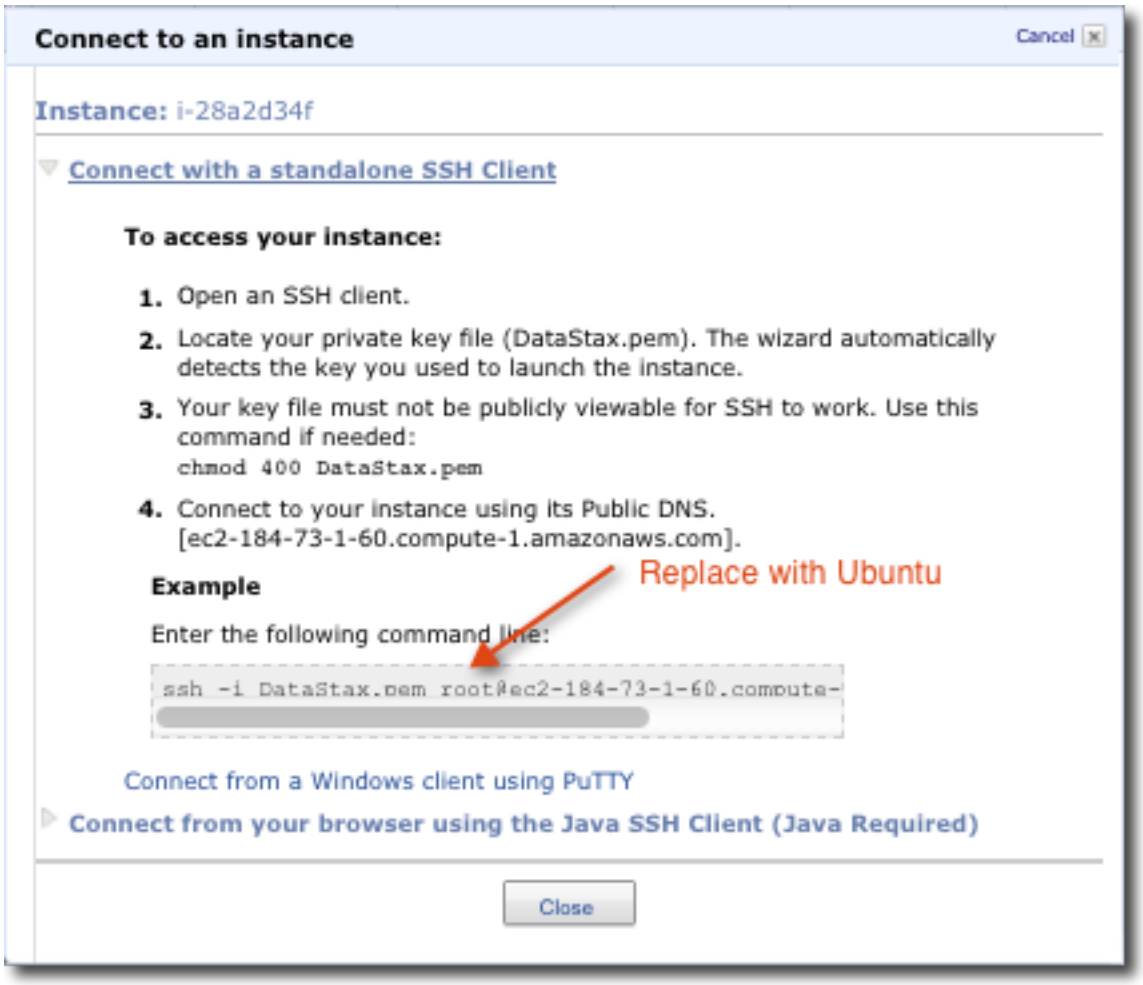
1. From the **My Instances** page in your AWS EC2 Dashboard, select the node that you want to connect to.

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



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

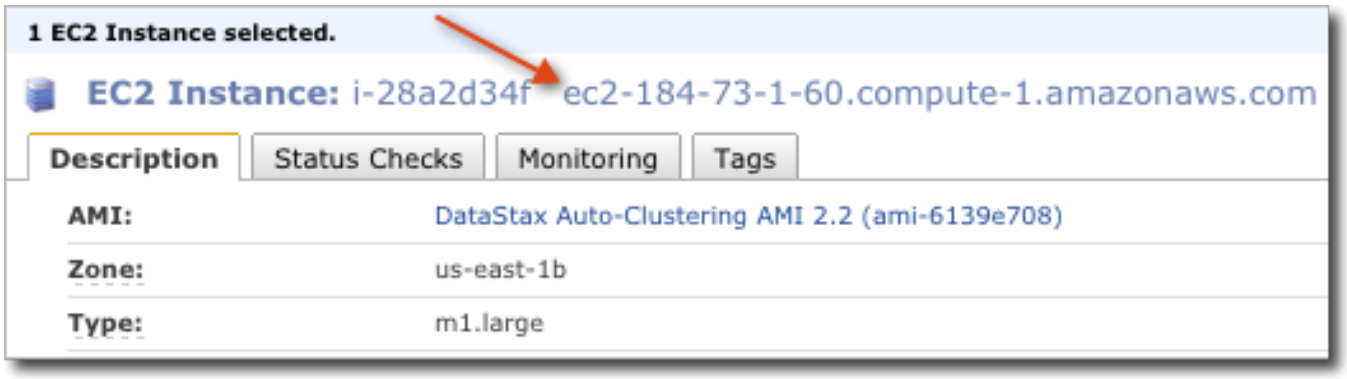
3. In the **Connect Help - Secure Shell (SSH)** page, copy the command line and change the connection user from **root** to **ubuntu**, then paste it into your SSH client.



4. The AMI image configures your cluster and starts the Cassandra services. After you have logged into a node, run the `nodetool status` command to make sure your cluster is running. For more information, see the *nodetool utility*.

```
ubuntu@ip-10-169-22-55:~$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address           Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160    53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48      93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
UN  10.196.14.239     37.51 KB   256     8.2%   d9bc122b-be89-46b2-ce5d-f5489575c1fa  rack1
```

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



6. After the OpsCenter loads, you must install the OpsCenter agents to see the cluster performance data.

   a. Click the **Fix** link located near the top of the Dashboard in the left navigation pane to install the agents.

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

Next steps

Expanding a Cassandra AMI cluster

This section contains instructions for expanding a cluster created with version 1.2 of the DataStax Community Edition AMI (Amazon Machine Image).

Note

For adding nodes to clusters created prior to Cassandra 1.2, follow the instructions in the 1.1 topic Expanding a Cassandra AMI cluster.

Adding nodes to a Cassandra AMI cluster

Virtual nodes greatly simplify adding nodes to an existing cluster. For a detailed explanation about how this works, see Virtual nodes in Cassandra 1.2.

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

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

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

   ```
   sudo service cassandra stop
   ```

3. Clear the data in each node:

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

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

   b. Remove the data directories:

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

   ### Note

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

4. Set the following properties in the *cassandra.yaml* configurtion file. For example:

   ```
   cluster_name: 'NameOfExistingCluster'
   ...
   num_tokens: 256
   ...
   seed_provider:
       - class_name: org.apache.cassandra.locator.SimpleSeedProvider
         parameters:
             - seeds: "110.82.155.0,110.82.155.3"
   ```

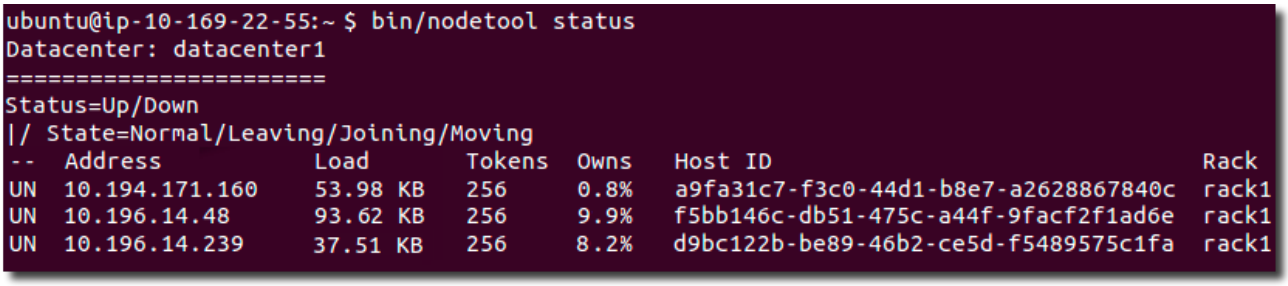
   Do not set the `initial_token`.

5. Start each node in **two** minute intervals. You can monitor the startup and data streaming process using *nodetool netstats*.

   ```
   $ sudo service cassandra start
   ```

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

   ```
   $ nodetool status
   ```



Upgrading Cassandra

This section describes how to upgrade an earlier version of Cassandra or DataStax Community Edition to DataStax Community 1.2.3. This section contains the following topics:

- *Best Practices for upgrading Cassandra*
- *Pre-requisite steps*
- *To upgrade a binary tarball installation*

Best Practices for upgrading Cassandra

The following steps are recommended before upgrading Cassandra:

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

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

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

Pre-requisite steps

1. Date strings (and timestamps) are no longer accepted as valid timeuuid values. This change requires modifying queries that use these values. *New methods have been added* for working with timeuuid values.
2. Cassandra 1.2.3 is not network-compatible with versions older than 1.0. If you want to perform a *rolling restart*, first upgrade the cluster to 1.0.x or 1.1.x, and then to 1.2.3, as described in the Cassandra 1.1 documentation.

   Data files from Cassandra 0.6 and later are compatible with Cassandra 1.2 and later. If it's practical to shut down cluster instead of performing a rolling restart, you can skip upgrading to an interim release and upgrade from Cassandra 0.6 or later to 1.2.3.

3. Do not upgrade if nodes in the cluster are down. The hints schema changed from 1.1 to 1.2.3. Cassandra automatically snapshots and then truncates the hints column family as part of starting up 1.2.3 for the first time. Additionally, upgraded nodes will not store new hints destined for older (pre-1.2) nodes. Use the *nodetool removenode* command, which was called nodetool removetoken in earlier releases, to removed dead nodes.

To upgrade a binary tarball installation

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

   #### Note
   Do not use the default *partitioner* setting because it has changed in this release to the Murmur3Partitioner. The Murmur3Partitioner can be used only for new clusters. After data has been added to the cluster, you cannot change the partitioner without reworking tables, which is not practical. Use your old partitioner setting in the new cassandra.yaml file.

7. Follow steps for *completing the upgrade*.

To upgrade a RHEL or CentOS installation

1. On each of your Cassandra nodes, run `sudo yum remove apache-cassandra11`, then run `sudo yum install dsc12`. 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, including the partitioner setting, from the old file into the new one. For details, see the note in *step 6 of the tarball upgrade procedure*. Save the file as cassandra.yaml.
4. Follow steps for *completing the upgrade*.

To upgrade a Debian or Ubuntu installation

1. Save the cassandra.yaml file from the old installation to a safe place.
2. On each of your Cassandra nodes, run `sudo apt-get install dsc12`.
3. Open the old and new cassandra.yaml files and diff them.
4. Merge the diffs by hand, including the partitioner setting, from the old file into the new one. For details, see the note in *step 6 of the tarball upgrade procedure*. Save the file as cassandra.yaml.
5. Follow steps for *completing the upgrade*.

Completing the upgrade

To complete the upgrade, perform the following steps:

1. Account for new parameters in cassandra.yaml.
2. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.

3. Run *nodetool drain* before shutting down the existing Cassandra service. This prevents overcounts of counter data, and will also speed up restart post-upgrade.
4. Stop the old Cassandra process, then start the new binary process.
5. Monitor the log files for any issues.
6. After upgrading and restarting all Cassandra processes, restart client applications.

Other Cassandra 1.2.3 changes that affect upgrades

Noteworthy changes are:

- Tables using LeveledCompactionStrategy do not create a row-level bloom filter by default. In versions of Cassandra before 1.2.2 the default value differs the current value; manually set the false positive rate to 1.0 (to disable) or 0.01 (to enable, if you make many requests for rows that do not exist).
- The default version of CQL (and cqlsh) is CQL 3.0. CQL 2 is still available but you will have to use the thrift set_cql_version method that is already supported in 1.1 to use CQL 2.
- In CQL 3, the DROP behavior has been removed temporarily from ALTER TABLE because it was not correctly implemented.
- In earlier releases, CQL3 property map keys used in ALTER and CREATE statements were case-insensitive. For example, CLASS or class and REPLICATION_FACTOR or replication_factor were permitted. The case-sensitivity of the property map keys was inconsistent with the treatment of other string literals and incompatible with formatting of NetworkTopologyStrategy property maps, which have case-sensitive data center names. In this release property map keys, such as class and replication_factor are case-sensitive. Use lowercase property map keys as shown in this example:

```
CREATE KEYSPACE test WITH replication =
  { 'class' : 'SimpleStrategy', 'replication_factor' : '1' };
```

Security

Cassandra 1.2.2 and later includes a number of features for securing data.

Client-to-node encryption

Client-to-node encryption protects data in flight from client machines to a database cluster. It establishes a secure channel between the client and the coordinator node. For information about generating SSL certificates, see *Preparing server certificates*.

SSL settings for Cassandra client-to-node encryption

To enable client-to-node SSL, you must set the client encryption options in the *cassandra.yaml* file. On each node under client_encryption_options:

- Enable encryption.
- Set the appropriate paths to your .keystore and .truststore files.
- Provide the required passwords. The passwords must match the passwords used when generating the keystore and truststore.
- To enable client certificate authentication, set require_client_auth to true. (Available starting with Cassandra 1.2.3.)

```
client_encryption_options:
   enabled: true
   keystore: conf/.keystore ## The path to your .keystore file
   keystore_password: <keystore password> ## The password you used when generating the keystore.
   truststore: conf/.truststore
   truststore_password: <truststore password>
   require_client_auth: <true or false>
```

Using cqlsh with SSL encryption

To run cqlsh, you must create a .cqlshrc file in your home or client program directory. This means you don't have to override the SSL_CERTFILE environmental variables every time.

Note

You cannot use cqlsh when client certificate authentication is enabled (*require_client_auth=true*).

Sample files are available in the following directories:

- Packaged installs: /etc/cassandra/conf
- Binary installs: <install_location>/conf

For example:

```
[authentication]
username = fred
password = !!bang!!$

[connection]
hostname = 127.0.0.1
port = 9160
factory = cqlshlib.ssl.ssl_transport_factory
```

```
[ssl]
certfile = ~/keys/cassandra.cert
validate = true ## Optional, true by default.

[certfiles] ## Optional section, overrides the default certfile in the [ssl] section.
192.168.1.3 = ~/keys/cassandra01.cert
192.168.1.4 = ~/keys/cassandra02.cert
```

When validate is enabled, the host in the certificate is compared to the host of the machine that it is connected to. The SSL certificate must be provided either in the configuration file or as an environment variable. The environment variables (SSL_CERTFILE and SSL_VALIDATE) override any options set in this file.

Node-to-node encryption

Node-to-node encryption protects data transferred between nodes in a cluster using SSL (Secure Sockets Layer). For information about generating SSL certificates, see *Preparing server certificates*.

SSL settings for Cassandra node-to-node encryption

To enable node-to-node SSL, you must set the server encryption options in the cassandra.yaml file.

On each node, under server_encryption_options:

- Enable the internode_encryption options (described below).
- Set the appropriate paths to your .keystore and .truststore files.
- Provide the required passwords. The passwords must match the passwords used when generating the keystore and truststore.
- To enable client certificate authentication, set require_client_auth to true.

The available inter-node options are:

- **all**
- **none**
- **dc**: Cassandra encrypts the traffic between the data centers.
- **rack**: Cassandra encrypts the traffic between the racks.

```
server_encryption_options:
   internode_encryption: <internode_option>
   keystore: resources/dse/conf/.keystore
   keystore_password: <keystore password>
   truststore: resources/dse/conf/.truststore
   truststore_password: <truststore password>
   require_client_auth: <true or false>
```

Preparing server certificates

This topic provides information about generating SSL certificates for *client-to-node encryption* or *node-to-node encryption*. If you generate the certificates for one type of encryption, you do not need to generate them again for the other: the same certificates are used for both.

All nodes must have all the relevant SSL certificates on all nodes. A keystore contains private keys. The truststore contains SSL certificates for each node and doesn't require signing by a trusted and recognized public certification authority.

To prepare server certificates:

1. Generate the private and public key pair for the nodes of the cluster.

   A prompt for the new keystore and key password appears.
2. Leave key password the same as the keystore password.
3. Repeat steps 1 and 2 on each node using a different alias for each one.

   ```
   keytool -genkey -alias <cassandra_node0> -keystore .keystore
   ```
4. Export the public part of the certificate to a separate file and copy these certificates to all other nodes.

   ```
   keytool -export -alias cassandra -file cassandranode0.cer -keystore .keystore
   ```
5. Add the certificate of each node to the truststore of each node, so nodes can verify the identity of other nodes.

   A prompt for setting a password for the newly created truststore appears.

   ```
   keytool -import -v -trustcacerts -alias <cassandra_node0> -file <cassandra_node0>.cer -keystore .truststore
   keytool -import -v -trustcacerts -alias <cassandra_node1> -file <cassandra_node1>.cer -keystore .truststore
   . . .
   ```
6. Distribute the .keystore and .truststore files to all Cassandra nodes.
7. Make sure .keystore is readable only to the Cassandra daemon and not by any user of the system.

Adding new trusted users

When client certificate authentication is enabled (*require_client_auth=true*), generate the certificate as described above. Then import the user's certificate into every node's truststore using keytool:

```
keytool -import -v -trustcacerts -alias <username> -file <certificate file> -keystore .truststore
```

Configuring and using internal authentication

Like *object permission management* using internal authorization, internal authentication is based on Cassandra-controlled login accounts and passwords. Internal authentication works for the following clients when you provide a user name and password to start up the client:

- Astyanax
- cassandra-cli
- cqlsh
- Hector
- pycassa

Internal authentication stores usernames and bcrypt-hashed passwords in the system_auth.credentials column family.

Configuring and using authentication

PasswordAuthenticator is an IAuthorizer implementation, available in Cassandra 1.2.2 and later, that you can use to configure Cassandra for internal authentication out-of-the-box. You make a few changes to the cassandra.yaml as described in this procedure. Then, to use authentication, you start up the client using the default *superuser* name and password (cassandra/cassandra).

The syntax for starting up the client is:

```
<client startup string> -u cassandra -p cassandra
```

Change the superuser password:

1. Create another superuser (not named cassandra).
2. Log in as that new superuser.
3. Change the user password, cassandra, to something long and incomprehensible, and then forget about it. It won't be used again.
4. Take away the superuser status of the user named cassandra.

To configure and use internal authentication

1. Change the cassandra.yaml authenticator setting to PasswordAuthenticator:

   ```
   authenticator: org.apache.cassandra.auth.PasswordAuthenticator
   ```

2. *Configure the replication factor for the system_auth keyspace*.
3. Restart Cassandra.

   A default superuser name and password (cassandra) that you use to start the supported client is stored in Cassandra. For example, to start cqlsh:

   ```
   ./cqlsh -u cassandra -p cassandra
   ```

   You can now set up user accounts and authorize users to access the database objects by using CQL to grant them permissions on those objects.

CQL 3 supports the following authentication statements:

- *ALTER USER*
- *CREATE USER*
- *DROP USER*
- *LIST USERS*

Managing object permissions using internal authorization

You use familiar relational database GRANT/REVOKE paradigm to grant or revoke permissions to access Cassandra data. A *superuser* grants initial permissions, and subsequently a user may or may not be given the permission to grant/revoke permissions.

Accessing system resources

Read access to these resources is implicitly given to every authenticated user because the tables are used by most Cassandra tools:

- system.schema_keyspace
- system.schema_columns
- system.schema_columnfamilies
- system.local
- system.peers

Configuration

CassandraAuthorizer is one of many possible IAuthorizer implementations, and the one that stores permissions in the system_auth.permissions column family to support all authorization-related CQL 3 statements. Configuration consists mainly of changing the authorizer option in the cassandra.yaml to use the CassandraAuthorizer.

To configure internal authorization for managing object (resource) permissions:

1. In the cassandra.yaml, comment out the default AllowAllAuthorizer and add the CassandraAuthorizer as shown here:

   ```
   #authorizer: org.apache.cassandra.auth.AllowAllAuthorizer
   authorizer: org.apache.cassandra.auth.CassandraAuthorizer
   ```

   You can use any authenticator except AllowAll.
2. *Configure the system_auth keyspace replication factor*.
3. Fetching permissions can be an expensive operation. If necessary, adjust the *validity period for permissions caching* in the cassandra.yaml. You can disable permission caching by setting this option to 0.
4. Restart Cassandra after changing the cassandra.yaml file.

CQL 3 supports the following authorization statements:

- *GRANT*
- *LIST PERMISSIONS*
- *REVOKE*

Configuring system_auth keyspace replication

If you use a non-default authenticator and/or authorizer, such as PasswordAuthenticator, alter the replication factor for the system_auth keyspace. In a multi-node cluster, using the default replication factor of 1 for the system_auth keyspace precludes logging into any node when the node that stores the user data is down. For all system_auth-related queries, Cassandra uses the *QUORUM consistency level*.

Setting the system_auth keyspace replication factor

To change the replication factor of the system_auth keyspace:

1. Change the replication factor using CQL:

   Example for SimpleStrategy

   ```
   ALTER KEYSPACE system_auth WITH REPLICATION =
     {'class' : 'SimpleStrategy',  'replication_factor' : 3 };
   ```

   Example for NetworkTopologyStrategy

   ```
   ALTER KEYSPACE system_auth WITH REPLICATION =
     {'class' : 'NetworkTopologyStrategy',
     'dc1' : 3, 'dc2' : 3};
   ```
2. If you change the system_auth keyspace on an existing cluster:
   a. Make sure every node uses the same settings.
   b. On each affected node, run *nodetool repair* to repair *only* the system_auth keyspace. Wait until repair completes on a node before moving to the next node.

About the system_auth keyspace

Cassandra uses the system_auth keyspace for storing security authentication and authorization information:

- **Cassandra:** the internal user list (in system_auth.users column family).
- **PasswordAuthenticator:** the users' hashed passwords (in system_auth.credentials column family)

- **CassandraAuthorizer:** the users' permissions (in system_auth.permissions column family)

Configuring firewall port access

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

Port	Description
Public Facing Ports	
22	SSH port.
8888	OpsCenter website port.
Cassandra Inter-node Ports	
1024+	JMX reconnection/loopback ports. See description for port 7199.
7000	Cassandra inter-node cluster communication.
7199	Cassandra JMX monitoring port. After the initial handshake, the JMX protocol requires that the client reconnects on a randomly chosen port (1024+).
9160	Cassandra client port (Thrift).
OpsCenter ports	
61620	OpsCenter monitoring port. The opscenterd daemon listens on this port for TCP traffic coming from the agent.
61621	OpsCenter agent port. The agents listen on this port for SSL traffic initiated by OpsCenter.

Initializing a Cassandra cluster

Initializing a Cassandra cluster involves configuring each node so that it is prepared to join the cluster. After configuring the nodes, start each one sequentially beginning with the seed node(s). For information about choosing the right configuration options for your environment, see *Planning a Cassandra cluster deployment*.

Initializing a multiple node cluster

You can use initialize a Cassandra cluster with one or more 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.

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.

Prerequisites

Each node must be correctly configured before starting the cluster. You must determine or perform the following before starting the cluster:

- Install Cassandra on each node.
- Choose a name for the cluster.
- Get the IP address of each node.
- Determine which nodes will be seed nodes. (Cassandra nodes use the seed node list for finding each other and learning the topology of the ring.)
- Determine the *snitch*.
- If the nodes are behind a firewall, open the required ports for internal/external communication. See *Configuring firewall port access*.
- If using multiple data centers, determine a naming convention for each data center and rack, for example: DC1, DC2 or 100, 200 and RAC1, RAC2 or R101, R102.
- Other possible configuration settings are described in *Choosing node configuration options* and *Node and cluster configuration (cassandra.yaml)*.

The following examples demonstrate initializing Cassandra:

- *Configuration example for single data center*
- *Configuration example for multiple data centers*

Configuration example for single data center

This example describes installing a six node cluster spanning two racks in a single data center. Each node is configured to use the RackInferringSnitch (multiple rack aware) and 256 virtual nodes (recommended).

It is recommended to have more than one seed node per data center.

To initialize the cluster:

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.

1. Suppose you install Cassandra on these nodes with one node per rack serving 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

   It is a best practice to have at more than one seed node per data center.
2. 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*.
3. If the Cassandra is running, stop the node and clear the data.
   - For packaged installs, run the following commands:

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

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

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

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

     `$ sudo rm -rf /var/lib/cassandra/*` (clears the data from the **default** directories)
4. Modify the following property settings in the `cassandra.yaml` file for each node:
   - `num_tokens: -`
   - `-seeds: <internal IP_address of each seed node>`
   - `listen_address: <localhost IP address>`
   - `endpoint_snitch` <name of snitch> - See *endpoint_snitch*.

   node0

```
cluster_name: 'MyDemoCluster'
num_tokens: 256
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 these nodes are the same as **node0** except for the listen_address.

5. 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)
6. To check that the ring is up and running, run the *nodetool status* command.

```
paul@ubuntu:~/cassandra-1.2.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
DN  10.196.14.239    ?          256     8.2%   null                                  rack1
```

Configuration example for multiple data centers

This example describes installing a six node cluster spanning two data centers. Each node is configured to use the PropertyFileSnitch (uses a user-defined description of the network details) and 256 virtual nodes (recommended).

It is recommended to have more than one seed node per data center.

To configure a cluster with multiple data centers:

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

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

1. Suppose you install Cassandra on these nodes:
   - node0 10.168.66.41 (seed1)
   - node1 10.176.43.66
   - node2 10.168.247.41
   - node3 10.176.170.59 (seed2)
   - node4 10.169.61.170
   - node5 10.169.30.138
2. 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*.

3. If the Cassandra is running, stop the node and clear the data.
   - For packaged installs, run the following commands:

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

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

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

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

     `$ sudo rm -rf /var/lib/cassandra/*` (clears the data from the **default** directories)
4. Modify the following property settings in the `cassandra.yaml` file for each node:
   - `num_tokens: -`
   - `-seeds: <internal IP_address of each seed node>`
   - `listen_address: <localhost IP address>`
   - `endpoint_snitch` <name of snitch> - See *endpoint_snitch*.

   node0:

   ```
   cluster_name: 'MyDemoCluster'
   num_tokens: 256
   seed_provider:
      - class_name: org.apache.cassandra.locator.SimpleSeedProvider
         parameters:
           - seeds: "10.168.66.41,10.176.170.59"
   listen_address: 10.168.66.41
   endpoint_snitch: PropertyFileSnitch
   ```

   Note

   Include at least one node from *each* data center.

   node1 to node5

   The properties for these nodes are the same as **node0** except for the listen_address.
5. In the `cassandra-topology.properties` file, assign the data center and rack names you determined in the Prerequisites 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
   ```
6. 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
   ```

7. 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)
8. To check that the ring is up and running, run the *nodetool status* command.

```
paul@ubuntu:~/cassandra-1.2.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
DN  10.196.14.239    ?          256     8.2%   null                                  rack1
```

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

You do not need to generate tokens when using *virtual nodes* in Cassandra 1.2 and later clusters.

If you are not using virtual nodes, you still need to calculate tokens for your cluster. The following topics in the Cassandra 1.1 documentation provides conceptual information about tokens:

- Data Distribution in the Ring
- Replication Strategy

Calculating tokens for single or multiple data centers

When calculating tokens for single data center deployments, you calculate tokens by dividing the hash range by the number of nodes in the cluster. For multiple data center deployments, calculate the tokens for each data center so that the hash range is evenly divided for the nodes in each data center. For more explanation, see be sure to read the conceptual information mentioned above.

The method used for calculating tokens depends on the type of partitioner.

Calculating tokens for the RandomPartitioner

To calculate tokens when using the *RandomPartitioner* in Cassandra 1.2 clusters, use the Cassandra 1.1 Token Generating Tool.

Calculating tokens for the Murmur3Partitioner

Use this method for generating tokens when you are **not** using virtual nodes and using the *Murmur3Partitioner* (default). This partitioner uses a maximum possible range of hash values from -2^{63} to $+2^{63}-1$. To calculate tokens for this partitioner:

```
python -c 'print [str(((2**64 / number_of_tokens) * i) - 2**63) for i in range(number_of_tokens)]'
```

For example, to generate tokens for 6 nodes:

```
python -c 'print [str(((2**64 / 6) * i) - 2**63) for i in range(6)]'
```

The command displays the token for each node:

```
['-9223372036854775808', '-6148914691236517206', '-3074457345618258604', '-2',
 '3074457345618258600', '6148914691236517202']
```

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.

Anatomy of a table

CQL 3 tables, rows, and columns can be viewed much the same way as SQL, which is different from the internal implementation in Cassandra. In SQL 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. In Cassandra, you also define tables and metadata about the columns, but the actual columns that make up a row are determined by the client application.

CQL 3 improves upon CQL 2 in a number of ways:

- Clients no longer need to manually decode the CompositeType packing when processing values from a table like playlists in the *music service example*. Non-Java clients, such as Python cqlsh, no longer need to reverse-engineer the serialization format, and Java clients need not perform cumbersome unpacking.
- The row-oriented parts of SQL are akin to the CQL 3 abstraction of the Cassandra database, which does away with the CQL 2 syntax using FIRST to limit the number of columns, distinct from LIMIT for the number of rows.
- CQL 3 addresses a number of other CQL 3 problems with handling wide rows, such as indexing their data and performing row-oriented functions, such as count. CQL 3 is easier to use and intuitive for SQL users.

The *COMPACT STORAGE* directive provides backward compatibility for tables you create in CQL 3 and can also be used to direct Cassandra to use a more space-efficient storage format for the table at the cost of making all primary key columns non-updateable.

Example -- a music service

This example contrasts two approaches to applying Cassandra's storage model:

- Using legacy sparse objects. Objects and fields correspond to rows and columns, respectively, but we do not know what the fields are in advance.
- Using CQL 3 to transpose data partitions (wide rows) into familiar row-based resultsets, dramatically simplifying data modeling.

This example of a social music service requires a songs table having a title, album, and artist column, plus a column called data for the actual audio file itself. The table uses a UUID as a primary key.

```
CREATE TABLE songs (
  id uuid PRIMARY KEY,
  title text,
  album text,
  artist text,
  data blob
);
```

In a relational database, you would create a playlists table with a foreign key to the songs, but in Cassandra, you denormalize the data. Cassandra packs each song in a playlist into one column, so fetching a playlist is just a single primary key lookup. The column value holds the song id. To represent the playlist data in CQL3, you can create a table like this:

```
CREATE TABLE playlists (
  id uuid,
  song_id uuid,
  title text,
  album text,
  artist text,
  PRIMARY KEY (id, song_id)
);
```

The combination of the id and song_id in the playlists table uniquely identifies a row in the playlists table. You can have more than one row with the same id as long as the rows contain different song_ids.

Using UUIDs as surrogate keys instead of sequential integers is a Cassandra best practice; using natural keys is also a good option.

Note

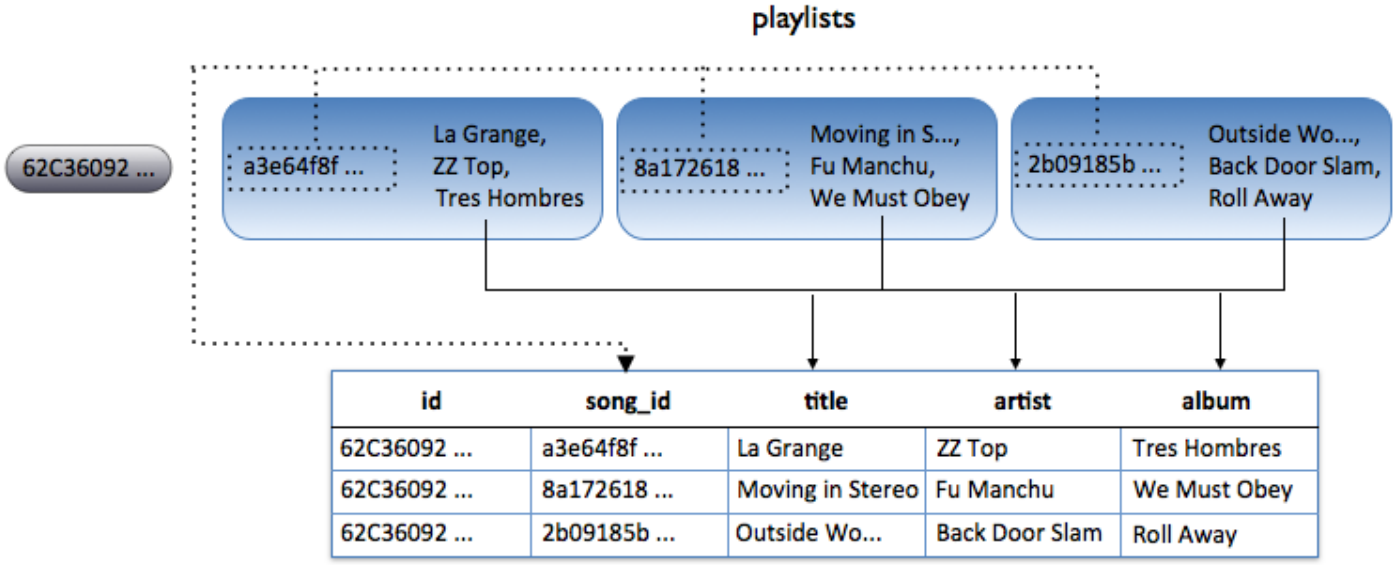
UUIDs are handy for sequencing the data or automatically incrementing synchronization across multiple machines.

After *inserting the example data* into playlists, the output of selecting all the data looks like this:

```
SELECT * FROM playlists;

 id          | song_id     | album        | artist         | title
-------------+-------------+--------------+----------------+---------------------
 62c36092... | 2b09185b... |    Roll Away | Back Door Slam | Outside Woman Blues
 62c36092... | 8a172618... | We Must Obey |      Fu Manchu |    Moving in Stereo
 62c36092... | a3e64f8f... | Tres Hombres |         ZZ Top |           La Grange
```

The CQL 3 data maps to the storage engine's representation of the data as follows:



The solid lines show how the cell values get unpacked into three CQL 3 columns. The dotted lines show how the clustering key portion of the compound primary key becomes the song_id column. Presenting a storage engine row as a partition of two or more object rows is a more natural row-column representation than earlier CQL versions and the Thrift API.

The next example illustrates how you can create a query that uses the artist as a filter. First, add a little more data to the playlist table to make things interesting for the collections examples later:

```
INSERT INTO playlists (id, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204,
  7db1a490-5878-11e2-bcfd-0800200c9a66,
  'Ojo Rojo', 'Fu Manchu', 'No One Rides for Free');
```

With the schema as given so far, a query that includes the artist filter would require a sequential scan across the entire playlists dataset. Cassandra will reject such a query. If you first create a secondary index on artist, Cassandra can efficiently pull out the records in question.

```
CREATE INDEX ON playlists(artist);
```

Now, you can query the playlists for songs by Fu Manchu, for example:

```
SELECT * FROM playlists WHERE artist = 'Fu Manchu';
```

The output looks something like this:

```
 id          | song_id     | album                  | artist    | title
-------------+-------------+------------------------+-----------+------------------
62c36092... | 7db1a490... | No One Rides for Free | Fu Manchu |         Ojo Rojo
62c36092... | 8a172618... |          We Must Obey | Fu Manchu | Moving in Stereo
```

Compound keys and clustering

A compound primary key gives you clustering guarantees per partition as well as to the normal SQL semantics of the compound primary key.

About the partition key and clustering columns

Cassandra uses the first column name in the primary key definition as the *partition key*. For example, in the playlists table, id is the partition key. The remaining column, or columns that are not partition keys in the primary key definition are the clustering columns. In the case of the playlists table, the song_id is the clustering column. The data for each partition is *clustered* by the remaining column or columns of the primary key definition. On a physical node, when rows for a partition key are stored in order based on the clustering columns, retrieval of rows is very efficient. For example, because the id in the playlists table is the partition key, all the songs for a playlist are clustered in the order of the remaining song_id column.

Insertion, update, and deletion operations on rows sharing the same partition key for a table are performed *atomically* and in *isolation*.

Using the CQL 3 model, you can query a single sequential set of data on disk to get the songs for a playlist.

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
  ORDER BY song_id DESC LIMIT 50;
```

The output looks something like this:

```
 id          | song_id     | album                  | artist         | title
-------------+-------------+------------------------+----------------+--------------------
 62c36092... | a3e64f8f... |          Tres Hombres |         ZZ Top |           La Grange
 62c36092... | 8a172618... |          We Must Obey |      Fu Manchu |    Moving in Stereo
 62c36092... | 2b09185b... |             Roll Away | Back Door Slam | Outside Woman Blues
 62c36092... | 7db1a490... | No One Rides for Free |      Fu Manchu |            Ojo Rojo
```

Collection columns

CQL 3 introduces these collection types:

- *collections set*

- *list*
- *map*

Handling some tasks in earlier releases of Cassandra was not as elegant as in a relational database. For example, in a relational database, to allow users to have multiple email addresses, you create an email_addresses table having a many-to-one (joined) relationship to a users table. In earlier releases of Cassandra, you denormalized the data, and stored it in multiple columns: email1, email2, and so on. This early Cassandra approach involved doing a read before adding a new email address to know which column name to use, but otherwise, involved no performance hit because adding new columns is virtually free in Cassandra.

CQL 3 includes the capability to handle the classic multiple email addresses use case, and other use cases, by defining columns as collections. Using the set collection type to solve the multiple email addresses problem is convenient and intuitive using CQL 3.

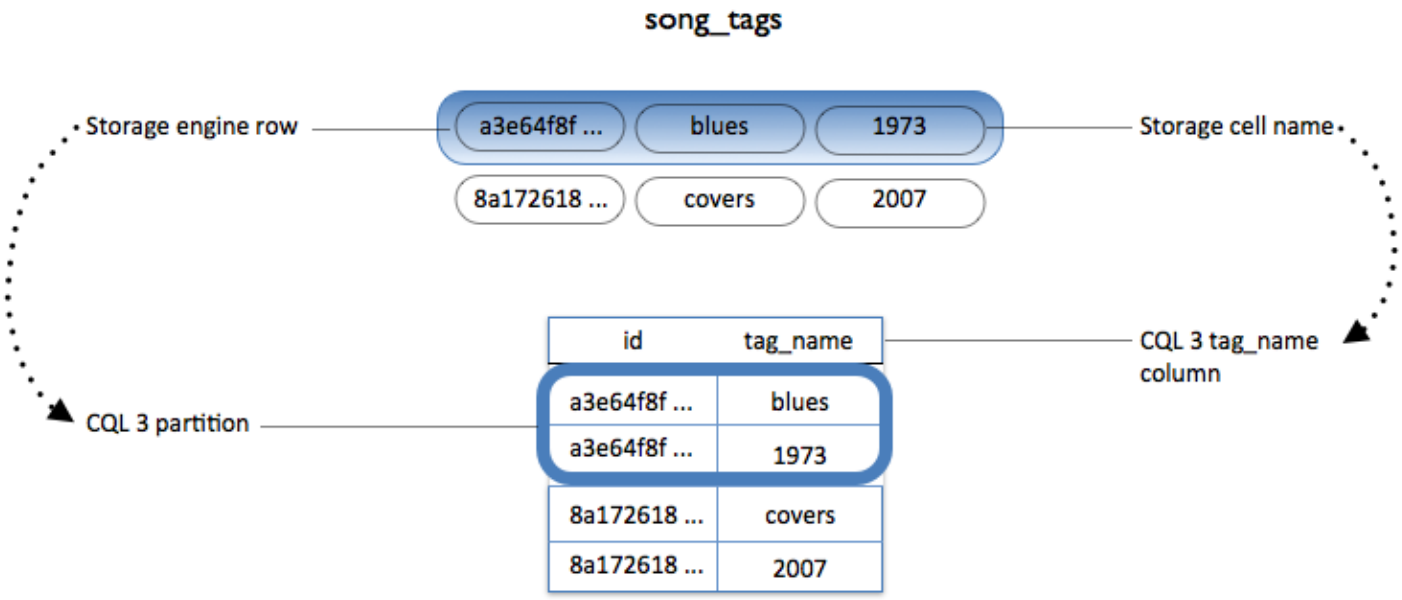
Another use of a collection type can be demonstrated using the music service example.

Adding a collection to a table

The music service example includes the capability to tag the songs. In CQL 2, in addition to the songs table presented earlier, you would need to add a second table to handle the song tags. The song_tags table, for example, looks like this:

```
CREATE TABLE song_tags (
  id uuid,
  tag_name text,
  PRIMARY KEY (id, tag_name)
);
```

By creating the song_tags table, you give the storage engine cell name, which is used as a map key, its own CQL3 column.



From a relational standpoint, you can think of storage engine rows as partitions, within which (object) rows are clustered. After tagging songs, the table would look like this:

```
SELECT * FROM song_tags;

 id                                   | tag_name
--------------------------------------+----------
 a3e64f8f-bd44-4f28-b8d9-6938726e34d4 |     1973
 a3e64f8f-bd44-4f28-b8d9-6938726e34d4 |    blues
 8a172618-b121-4136-bb10-f665cfc469eb |     2007
 8a172618-b121-4136-bb10-f665cfc469eb |   covers
```

To tag songs, in CQL 3 you don't need the song_tags table. Drop the song_tags table and represent the sparse tag collection directly in the songs table, as a set data type. A collection set can be declared using the CREATE TABLE or ALTER TABLE statements. Because the songs table already exists from the earlier example, just alter that table to add a tags set:

```
DROP table song_tags;
ALTER TABLE songs ADD tags set<text>;
```

Updating a collection

Update the songs table to insert the tags data:

```
UPDATE songs SET tags = tags + {'2007'}
  WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;
UPDATE songs SET tags = tags + {'covers'}
  WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;
UPDATE songs SET tags = tags + {'1973'}
  WHERE id = a3e64f8f-bd44-4f28-b8d9-6938726e34d4;
UPDATE songs SET tags = tags + {'blues'}
  WHERE id = a3e64f8f-bd44-4f28-b8d9-6938726e34d4;
UPDATE songs SET tags = tags + {'rock'}
  WHERE id = 7db1a490-5878-11e2-bcfd-0800200c9a66;
```

A music reviews list and a schedule (map collection) of live appearances can be added to the table:

```
ALTER TABLE songs ADD reviews list<text>;
ALTER TABLE songs ADD venue map<timestamp, text>;
```

Each element of a map, list, or map is internally stored as one Cassandra column. To update a set, use the UPDATE command and the addition (+) operator to add an element or the subtraction (-) operator to remove an element. For example, to update a set:

```
UPDATE songs
  SET tags = tags + {'rock'}
  WHERE id = 7db1a490-5878-11e2-bcfd-0800200c9a66;
```

To update a list, a similar syntax using square brackets instead of curly brackets is used.

```
UPDATE songs
  SET reviews = reviews + [ 'hot dance music' ]
  WHERE id = 7db1a490-5878-11e2-bcfd-0800200c9a66;
```

To update a map, use INSERT to specify the data in a map collection.

```
INSERT INTO songs (id, venue)
  VALUES (7db1a490-5878-11e2-bcfd-0800200c9a66,
  { '2013-9-22 12:01'  : 'The Fillmore',
  '2013-10-1 18:00' : 'The Apple Barrel'});
```

Inserting data into the map replaces the entire map.

Querying a collection

To query a collection, include the name of the collection column in the select expression. For example, selecting the tags set returns the set of tags, sorted alphabetically in this case because the tags set is of the text data type:

```
SELECT id, tags FROM songs;

 id                                   | tags
--------------------------------------+----------------
 7db1a490-5878-11e2-bcfd-0800200c9a66 |         {rock}
 a3e64f8f-bd44-4f28-b8d9-6938726e34d4 |  {blues, 1973}
 8a172618-b121-4136-bb10-f665cfc469eb | {2007, covers}

SELECT id, venue FROM songs;

 id                                   | venue
--------------------------------------+----------------------------------------------------------------------
 7db1a490... | {2013-10-01 18:00:00-0700: The Apple Barrel, 2013-09-22 12:01:00-0700: The Fillmore}
 a3e64f8f... |                                                                                          null
 8a172618... |                                                                                          null
```

The collection types are described in more detail in *Using collections: set, list, and map*.

Expiring columns

Data in a column can 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 data in the column. TTL columns are marked as having the data deleted (with a tombstone) after the requested amount of time has expired. After columns are marked with a tombstone, they are automatically removed during the normal compaction (defined by the *gc_grace_seconds*) and repair processes.

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

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

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

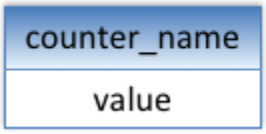
An expiring column has an additional overhead of 8 bytes in memory and on disk (to record the TTL and expiration time) compared to standard columns.

Counter columns

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

Counter column tables must use Counter data type. Counters may only be stored in dedicated tables.

After a counter is defined, the client application then updates the counter 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 a source such as network time protocol (NTP).

Unlike normal columns, a write to a counter requires a read in the background to ensure that distributed counter values remain consistent across replicas. Typically, you use a consistency level of ONE with counters because during a write operation, the implicit read does not impact write latency.

Working with pre-CQL 3 applications

Internally, CQL 3 does not change the row and column mapping from the Thift API mapping. CQL 3 and Thrift use the same storage engine. CQL 3 supports the same query-driven, denormalized data modeling principles as Thrift. Existing applications do not have to be upgraded to CQL 3. The CQL 3 abstraction layer makes CQL 3 easier to use for new applications. For an in-depth comparison of Thrift and CQL 3, see A Thrift to CQL 3 Upgrade Guide and CQL 3 for Cassandra experts.

Creating legacy tables

You can create Thrift/CLI-compatible tables in CQL 3 using the COMPACT STORAGE directive. The *compact storage directive* used with the CREATE TABLE command provides backward compatibility with older Cassandra applications; new applications should generally avoid it.

Compact storage stores an entire row in a single column on disk instead of storing each non-primary key column in a column that corresponds to one column on disk. Using compact storage prevents you from adding new columns that are not part of the PRIMARY KEY.

Querying a legacy table

You can use the CLI GET command to query tables created with or without the COMPACT STORAGE directive in CQL 3 or created with the Thrift API, CLI, or early version of CQL.

Using CQL 3, you can query a legacy table. A legacy table managed in CQL 3 includes an implicit WITH COMPACT STORAGE directive.

Using the CLI GET command

Continuing with the *music service example*, select all the columns in the playlists table that was created in CQL 3. This output appears:

```
[default@music] GET playlists [62c36092-82a1-3a00-93d1-46196ee77204];
  => (column=7db1a490-5878-11e2-bcfd-0800200c9a66:,
    value=, timestamp=1357602286168000)
  => (column=7db1a490-5878-11e2-bcfd-0800200c9a66:album,
    value=4e6f204f6e652052696465732066f722046726565, timestamp=1357602286168000)
.
.
.
  => (column=a3e64f8f-bd44-4f28-b8d9-6938726e34d4:title,
    value=4c612047726616e6765, timestamp=1357599350478000)
Returned 16 results.
```

The output of cell values is unreadable because GET returns the values in byte format.

Using a CQL 3 query

A CQL 3 query resembles a SQL query to a greater extent than a CQL 2 query. CQL 3 no longer uses the REVERSE keyword, for example.

In the example *GET command* and in CLI output in general, there is a timestamp value that doesn't appear in the CQL 3 output.

This timestamp represents the date/time that a write occurred to a columns. In CQL 3, you use WRITETIME in the select expression to get this timestamp. For example, to get the date/times that a write occurred to the body column:

```
SELECT WRITETIME (title)
  FROM songs
  WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;

 writetime(title)
------------------
 1353890782373000
```

The output in microseconds shows the write time of the data in the title column of the songs table.

When you use CQL 3 to query legacy tables with no column names defined for data within a partition, CQL 3 generates the names (column1 and value1) for the data. Using the CQL *RENAME clause*, you can change the default name to a more meaningful name.

```
ALTER TABLE users RENAME key to user_id;
```

CQL 3 supports dynamic tables created in the Thrift API, CLI, and earlier CQL versions. For example, a dynamic table is represented and queried like this in CQL 3:

```
CREATE TABLE clicks (
  userid uuid,
  url text,
  timestamp date
  PRIMARY KEY (userid, url)
) WITH COMPACT STORAGE;

SELECT url, timestamp
  FROM clicks
  WHERE userid = 148e9150-1dd2-11b2-0000-242d50cf1fff;

SELECT timestamp
  FROM clicks
  WHERE userid = 148e9150-1dd2-11b2-0000-242d50cf1fff
  AND url = 'http://google.com';
```

In these queries, only equality conditions are valid.

Super columns not supported

CQL does not support super columns. For new projects, use compound keys and collections instead. For old projects, the Thrift supercolumn API will remain supported.

Starting CQLsh using the CQL 2 mode

CQL 2 supports dynamic columns, but not compound primary keys. To start cqlsh 2.0.0 using the legacy CQL 2 mode from the Cassandra bin directory on Linux, for example:

```
./cqlsh -cql2
```

Or on Windows, from the Cassandra bin directory in Command Prompt:

```
python cqlsh -cql2
```

About indexes

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 table is the index of its row keys. Each node maintains this index for the data it manages.

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

With randomly partitioned row keys (the default in Cassandra), row keys are partitioned by their MD5 hash and cannot be scanned in order like traditional b-tree indexes. The CQL 3 *token function* introduced in Cassandra 1.1.1 now makes it possible to page through non-ordered partitioner results.

About secondary indexes

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

```
USE myschema;

CREATE TABLE users (
  userID uuid,
  fname text,
  lname text,
  state text,
  zip int,
  PRIMARY KEY (userID, zip)
);

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

Secondary index names, in this case users_state, must be unique within the keyspace. After creating a secondary index for the state column and inserting values into the table, greater efficiency is achieved when you query Cassandra directly for users who live in a given state:

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

Using multiple secondary indexes

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

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

CREATE INDEX users_lname
```

```
  ON users (lname);

SELECT * FROM users
  WHERE fname = 'bob' AND lname = 'smith'
  ALLOW FILTERING;
```

When there are multiple conditions in a WHERE clause, Cassandra selects the least-frequent occurrence of a condition for processing first for efficiency. In this example, Cassandra queries on the last name first if there are fewer Smiths than Bobs in the database or on the first name first if there are fewer Bobs than Smiths. When you attempt a potentially expensive query, such as searching a range of rows, Cassandra requires the ALLOW FILTERING directive.

When to use secondary indexes

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

When not to use secondary indexes

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

Building and using secondary indexes

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

Maintaining secondary indexes

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

Planning the data model

Different design considerations are involved in planning a Cassandra data model and a relational database data model. Ultimately, the data model you design depends largely on the data you want to capture and how you plan to access it. However, there are a few important design considerations that apply universally to Cassandra data model planning.

Start with queries

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

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

model design.

Denormalize to optimize

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

Cassandra does not have foreign key relationships like a relational database does, which means you cannot join multiple tables to satisfy a given query request. Cassandra performs best when the data needed to satisfy a given query is located in the *same* table. Try to plan your data model so that one or more rows in a single table 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 table, 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 table. Unlike the primary key in a relational database, Cassandra does not enforce uniqueness. 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 table. 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 tables, the data may contain values that are guaranteed to be unique and are not typically updated after a row is created. For example, the user name in a users table. 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 uniqueness, it could potentially overwrite column data.

For more information about data modeling, see *Anatomy of a table*.

Querying Cassandra

You use cqlsh for querying the Cassandra database from the command line. All of the commands included in CQL are available on the cqlsh command line. Several cqlsh commands, however, are not included in CQL. The *command table of contents* lists these commands by type.You can run cqlsh-only commands from only the command line.

This document describes CQL 3.0.0, cqlsh, and the *Command Line Interface (CLI)* for querying Cassandra.

Getting started using CQL

Cassandra Query Language (CQL) is a SQL (Structured Query Language)-like language for querying Cassandra. Although CQL has many similarities to SQL, there are some fundamental differences. CQL doesn't support joins, which make no sense in Cassandra. Cassandra is a distributed store, so joins are too expensive to perform on distributed data over many machines. Joins require expensive random reads, which need to be merged across the network, dramatically increasing the overhead involved. Instead of joins, Cassandra promotes collocating data accessed together through denormalization and the ordering provided by the Cassandra storage engine. Using this kind of model, the Cassandra DBMS can handle up to two billion columns per row.

CQL 3 is the default and primary interface into the Cassandra DBMS. CQL 3 provides a new API to Cassandra that is simpler than the Thrift API for new applications. The Thrift API, the Cassandra Command Line Interface (CLI), and legacy versions of CQL expose the internal storage structure of Cassandra. CQL 3 adds an abstraction layer that hides implementation details and provides native syntaxes for CQL 3 collections and other common encodings. For more information about backward compatibility and working with database objects created outside of CQL 3, see *Working with pre-CQL 3 applications*.

Note

DataStax Enterprise 3.0.x supports CQL 3 in Beta form. Users need to refer to Cassandra 1.1 documentation for CQL information.

Activating CQL 3

You activate the CQL mode in one of these ways:

- Start *cqlsh*, a Python-based command-line client.
- Use the set_sql_version Thrift method.
- Specify the desired CQL mode in the connect() call to the Python driver:

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

CQL 3 supports *compound keys and clustering*. Also, super columns are not supported by either CQL version; column_type and subcomparator arguments are not valid.

Running CQL

Developers can access CQL commands in a variety of ways. Drivers are available for Python, PHP, Ruby, Node.js, and JDBC-based client programs. For the purposes of administrators, cqlsh is the most direct way to run simple CQL commands. Using cqlsh, you can run CQL commands from the command line. The location of cqlsh is <install_location>/bin for tarball installations, or /usr/bin for packaged installations.

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

Starting cqlsh using the CQL 3 mode

If you use security features, *provide a user name and password* to start cqlsh.

Linux

To start cqlsh using the default CQL 3 mode from the Cassandra bin directory on Linux:

```
./cqlsh
```

The default user name and password are cassandra/cassandra.

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

```
./cqlsh 1.2.3.4 9160
```

Windows

To start cqlsh using the default CQL 3 mode from the Cassandra bin directory on Windows, in Command Prompt:

```
python cqlsh
```

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

```
python cqlsh 1.2.3.4 9160
```

Linux and Windows

To exit cqlsh:

```
cqlsh> exit
```

CQL syntax is SQL-like. To query the Cassandra database, follow steps in *Querying Cassandra*. CQL commands are described in detail in the *CQL Reference*.

Using tab completion to enter commands

You can use tab completion to see hints about cqlsh commands. Some platforms, such as Mac OSX, do not ship with tab completion installed. You can use easy_install to install tab completion capabilities on Mac OSX:

```
easy_install readline
```

Creating and updating a keyspace

The first CQL command you need to know is CREATE KEYSPACE. A keyspace is the CQL counterpart to the SQL database, but different in that the Cassandra keyspace is a namespace that defines how data is replicated on nodes. Typically, a cluster has one keyspace per application. Replication is controlled on a per-keyspace basis, so data that has different replication requirements typically resides in different keyspaces. Keyspaces are not designed to be used as a significant map layer within the data model. Keyspaces are designed to control data replication for a set of tables.

To create a keyspace, use the CREATE KEYSPACE command.

```
cqlsh> CREATE KEYSPACE demodb
         WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor': 3};
```

Using SimpleStrategy is fine for evaluating Cassandra. For production use or for use with mixed workloads, use NetworkTopologyStrategy. For more information about keyspace options, see *About data replication*.

Using a keyspace

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

```
cqlsh> USE demodb;
```

Next, create a table and populate it with data.

Changing the replication factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster. This example shows how to change the replication factor of a keyspace.

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

   ```
   ALTER KEYSPACE "Excalibur" WITH REPLICATION =
     { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
   ```

   Or if using NetworkTopologyStrategy:

   ```
   ALTER KEYSPACE system_auth WITH REPLICATION =
     {'class' : 'NetworkTopologyStrategy', 'dc1' : 3, 'dc2' : 2};
   ```

2. On each affected node, run *nodetool repair*. Wait until repair completes on a node before moving to the next node.

Creating a table

Continuing with the previous example, create a users table in the newly created keyspace:

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

The users table has a single primary key.

Using compound primary keys

Use a compound primary key when you want to create columns that you can query to return sorted results. To create a table having a compound primary key, use two or more columns as the primary key:

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

The compound primary key is made up of the empID and deptID columns in this example. The empID acts as a partition key for distributing data in the table among the various nodes that comprise the cluster. The remaining component of the primary key, the deptID, acts as a clustering mechanism and ensures that the data is stored in ascending order on disk (much like a clustered index in Microsoft SQL Server). For more information about compound and clustering columns, see the *Compound keys and clustering* section.

Inserting data into the table

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

The following example shows how to use cqlsh to insert employee information for Jane Smith.

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

Querying system tables

The system keyspace includes a number of tables that contain details about your Cassandra database objects and cluster configuration. Cassandra populates these tables and others in the system keyspace:

Table name	Column names
schema_keyspaces	keyspace_name, durable_writes, strategy_class, strategy_options
local	"key", bootstrapped, cluster_name, cql_version, data_center, gossip_generation, partitioner, rack, release_version, ring_id, schema_version, thrift_version, tokens set, truncated at map (1
peers	peer, data_center, rack, release_version, ring_id, rpc_address, schema_version, tokens set (2
schema_columns	keyspace_name, columnfamily_name, column_name, component_index, index_name, index_options, index_type, validator (3
schema_columnfamilies	See (4

(1 Information a node has about itself and a superset of *gossip*.

(2 Each node records what other nodes tell it about themselves over the *gossip*.

(3 Used internally with compound primary keys.

(4 You can inspect schema_columnfamilies to get detailed information about specific column families.

Keyspace Information

An alternative to the Thrift API describe_keyspaces function is querying the system tables directly in CQL 3. For example, you can query the defined keyspaces:

```
SELECT * from system.schema_keyspaces;
```

The cqlsh output includes information about defined keyspaces. For example:

```
 keyspace | durable_writes | name    | strategy_class | strategy_options
----------+----------------+---------+----------------+--------------------------
  history |           True | history | SimpleStrategy | {"replication_factor":"1"}
  ks_info |           True | ks_info | SimpleStrategy | {"replication_factor":"1"}
```

You can also retrieve information about tables by querying system.schema_columnfamilies and about column metadata by querying system.schema_columns.

Cluster information

You can query system tables to get cluster topology information. You can get the IP address of peer nodes, data center and rack names, token values, and other information.

For example, after setting up a 3-node cluster using ccm on the Mac OSX, query the peers and local tables.

```
USE system;
select * from peers;
```

Output from querying the peers table looks something like this:

```
   peer    | data_center | rack  | release_version | ring_id         | rpc_address | schema_version   | tokens
-----------+-------------+-------+-----------------+-----------------+-------------+------------------+-------. . .
 127.0.0.3 | datacenter1 | rack1 | 1.2.0-beta2     | 53d171bc-ff. . .| 127.0.0.3   | 59adb24e-f3 . . .|  {3074. . .
 127.0.0.2 | datacenter1 | rack1 | 1.2.0-beta2     | 3d19cd8f-c9. . .| 127.0.0.2   | 59adb24e-f3 . . .| {-3074. . .}
```

For more information about system keyspaces, see The data dictionary article.

Retrieving columns

To retrieve results, use the SELECT statement.

```
SELECT * FROM users WHERE first_name = 'jane' and last_name='smith';
```

Retrieving and sorting results

Similar to a SQL query, use the WHERE clause and then the ORDER BY clause to retrieve and sort results:

```
cqlsh:demodb> SELECT * FROM emp WHERE empID IN (130,104) ORDER BY deptID DESC;

 empid | deptid | first_name | last_name
-------+--------+------------+-----------
   104 |     15 |       jane |     smith
   130 |      5 |     sughit |     singh

cqlsh:demodb> SELECT * FROM emp where empID IN (130,104) ORDER BY deptID ASC;

 empid | deptid | first_name | last_name

-------+--------+------------+-----------
   130 |      5 |     sughit |     singh
   104 |     15 |       jane |     smith
```

See the *music service example* for more information about using compound primary keys.

Using the keyspace qualifier

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

```
INSERT INTO Music.songs (id, title, artist, album) VALUES (
a3e64f8f-bd44-4f28-b8d9-6938726e34d4, 'La Grange', 'ZZ Top', 'Tres Hombres');
```

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

- ALTER TABLE
- CREATE TABLE
- DELETE
- INSERT
- SELECT
- TRUNCATE
- UPDATE

For more information, see the *CQL Reference*.

Determining time-to-live and write time

You can use these functions in a SELECT statement to determine:

- *TTL*: How much longer an expiring column has to live
- WRITETIME: The date/time that a write to a column occurred

Determining how long a column has to live

This procedure creates a table, inserts data into two columns, and calls the TTL function to retrieve the date/time of the writes to the columns.

1. Create a users table named clicks in the excelsior keyspace.

```
CREATE TABLE excelsior.clicks (
  userid uuid,
  url text,
  date timestamp,  //unrelated to WRITETIME discussed in the next section
  name text,
  PRIMARY KEY (userid, url)
);
```

2. Insert data into the table, including a date in yyyy-mm-dd format, and set that data to expire in a day (86400 seconds).

```
INSERT INTO excelsior.clicks (
  userid, url, date, name)
  VALUES (
    3715e600-2eb0-11e2-81c1-0800200c9a66,
    'http://apache.org',
    '2013-10-09', 'Mary')
    USING TTL 86400;
```

   To set time-to-live on a column, insert the new data, the primary key, and selected columns followed by the USING TTL clause.

3. Wait for a while and then issue a SELECT statement to determine how much longer the data entered in step 2 has to live.

```
SELECT TTL (name) from excelsior.clicks
  WHERE url = 'http://apache.org';
```

   Output is, for example, 85908 seconds:

```
 ttl(name)
-----------
 85908
```

Determining the date/time that a write occurred

Using the WRITETIME function in a SELECT statement returns the date/time in microseconds that the column was written to the database. This procedure continues the example from the previous procedure and calls the WRITETIME function to retrieve the date/time of the writes to the columns.

1. Insert more data into the table.

```
INSERT INTO excelsior.clicks (
  userid, url, date, name)
  VALUES (
    cfd66ccc-d857-4e90-b1e5-df98a3d40cd6,
    'http://google.com',
    '2013-10-11', 'Bob'
);
```

2. Retrieve the date/time that the value Mary was written to the name column. Use the WRITELINE function in a SELECT statement, followed by the name of a column in parentheses:

```
SELECT WRITETIME (name) FROM excelsior.clicks
  WHERE url = 'http://apache.org';


 writetime(name)
------------------
 1353010594789000
```

   The writetime output in microseconds converts to November 15, 2012 at 12:16:34 GMT-8.

3. Retrieve the date/time that the timestamp value 2013-10-09 was written to the date column.

```
SELECT WRITETIME (date) FROM excelsior.clicks
  WHERE url = 'http://google.org';


 writetime(date)
------------------
 1353010564029000
```

   The writetime output in microseconds converts to November 15, 2012 at 12:16:04 GMT-8.

Altering a table to add columns

The ALTER TABLE command adds new columns to a table. For example, to add a coupon_code column with the varchar data type to the users table:

```
cqlsh:demodb> ALTER TABLE users ADD coupon_code varchar;
```

This creates the column metadata and adds the column to the table schema, but does not update any existing rows.

Altering column metadata

Using ALTER TABLE, you can change the data type of a column after it is defined or added to a table. For example, to change the coupon_code column to store coupon codes as integers instead of text, change the data type as follows:

```
cqlsh:demodb> ALTER TABLE users ALTER coupon_code TYPE int;
```

Only newly inserted values, not existing coupon codes are validated against the new type.

Removing data

To remove data, you can set column values for automatic removal using the *TTL* (time-to-expire) table attribute. You can also drop a table or keyspace, and delete keyspace column metadata.

Expiring columns

Both the INSERT and UPDATE commands support setting a time for data in a column to expire. The expiration time (TTL) is set using CQL. The following example first shows an INSERT statement that sets a password column in the users table to expire in 86400 seconds, or one day. If you wanted to extend the expiration period to five days, use the UPDATE command as shown in the second example:

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

cqlsh:demodb> UPDATE users USING TTL 432000 SET 'password' = 'ch@ngem4a'
                WHERE user_name = 'cbrown';
```

Dropping tables and keyspaces

Using cqlsh commands, you can drop a keyspace or table. This example shows the commands that drops the users table and then, the demodb keyspace:

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

Deleting columns and rows

CQL provides the DELETE command to delete a column or row. This example deletes user jsmith's session token column, and then deletes jsmith's entire row.

```
cqlsh:demodb> DELETE session_token FROM users where pk = 'jsmith';
cqlsh:demodb> DELETE FROM users where pk = 'jsmith';
```

Deleted values are removed completely by the first compaction following deletion.

Using collections: set, list, and map

This release of Cassandra includes collection types, *introduced earlier* in this document, that provide an improved way of handling tasks, such as building multiple email address capability into tables.

You can expire each element of a collection by setting an individual time-to-live (TTL) property, as shown in *Setting Expiration*.

Using the set type

A set stores a group of elements that are returned in sorted order when queried. A column of type set consists of unordered unique values. Using the set data type, you can solve the multiple email problem in an intuitive way that does not require a read before adding a new email address. For example, you can define an emails set in the users table to accommodate multiple email address:

```
CREATE TABLE users (
  user_id text PRIMARY KEY,
  first_name text,
  last_name text,
  emails set<text>
);
```

Insertion

To insert data into the set, enclose values in curly brackets. Set values must be unique. For example:

```
INSERT INTO users (user_id, first_name, last_name, emails)
  VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com', 'baggins@gmail.com'});
```

Addition

To add an element to a set, use the UPDATE command and the addition (+) operator:

```
UPDATE users
  SET emails = emails + {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

Deletion

To remove an element from a set, use the subtraction (-) operator.

```
UPDATE users
  SET emails = emails - {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

Retrieval

When you query a table containing a collection, Cassandra retrieves the collection in its entirety; consequently, keep collections small enough to be manageable, or construct a data model to replace collections that can accommodate large amounts of data.

To return the set of email belonging to frodo, for example:

```
SELECT user_id, emails FROM users WHERE user_id = 'frodo';
```

Cassandra returns results in an order based on the type of the elements in the collection. For example, a set of text elements is returned in alphabetical order.

```
 user_id | emails
---------+-----------------------------------------------------------------
 frodo   | {"baggins@caramail.com","f@baggins.com","fb@friendsofmordor.org"}
```

If you want elements of the collection returned in insertion order, use a *list*.

An empty set

To remove all elements from a set, you can use the UPDATE or DELETE statement:

```
UPDATE users SET emails = {} WHERE user_id = 'frodo';

DELETE emails FROM users WHERE user_id = 'frodo';
```

A set, *list*, or *map* needs to have at least one element; otherwise, Cassandra cannot distinguish the set from a null value.

```
SELECT user_id, emails FROM users WHERE user_id = 'frodo';

 user_id | emails
---------+------------------------------------------------
 frodo   | null
```

Using the list type

When the order of elements matters, which may not be the natural order dictated by the type of the elements, use a list. Also, use a list when you need to store same value multiple times. List values are returned according to their index value in the list, whereas set values are returned in alphabetical order, assuming the values are text.

For example, for each user in a users table, you want to add a list of their preferred places, then query the database for the top x places for a user.

Insertion

To add a list declaration to a table, add a column top_places of the list type to the users table:

```
ALTER TABLE users ADD top_places list<text>;
```

Next, use the UPDATE command to insert values into the list.

```
UPDATE users
  SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';
```

Addition

To prepend an element to the list, enclose it in square brackets, and use the addition (+) operator:

```
UPDATE users
  SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';
```

To append an element to the list, switch the order of the new element data and the list name in the UPDATE command:

```
UPDATE users
  SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';
```

These update operations are implemented internally without any read-before-write. Appending and prepending a new element to the list writes only the new element.

To add an element at a particular position, use the list index position in square brackets:

```
UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';
```

When you add an element at a particular position, Cassandra reads the entire list, and then writes only the updated element. Consequently, adding an element at a particular position results in greater latency than appending or prefixing an element to a list.

Deletion

To remove an element from a list, use the DELETE command and the list index position in square brackets:

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';
```

To remove all elements having a particular value, use the UPDATE command, the subtraction operator (-), and the list value in square brackets:

```
UPDATE users
  SET top_places = top_places - ['riddermark'] WHERE user_id = 'frodo';
```

The former, indexed method of removing elements from a list requires a read internally. Using the UPDATE command as shown here is recommended over emulating the operation client-side by reading the whole list, finding the indexes that contain the value to remove, and then removing those indexes. This emulation would not be thread-safe. If another thread/client prefixes elements to the list between the read and the write, the wrong elements are removed. Using the UPDATE command as shown here does not suffer from that problem.

Retrieval

A query returns a list of top places.

```
SELECT user_id, top_places FROM users WHERE user_id = 'frodo';
```

Using the map type

As its name implies, a map maps one thing to another. A map is a typed pair of unique keys and values. One use case for the map type is storing timestamp-related information in user profiles.

Insertion

To add a simple todo list to every user profile in an existing users table, use the CREATE TABLE or ALTER statement, specifying the map type and enclosing the data types for the name-value pair in angle brackets. For example, enclose the timestamp and reminder text in angle brackets:

```
ALTER TABLE users ADD todo map<timestamp, reminder_text>;
```

Modification and replacement

To set or replace map data, you can use the INSERT or UPDATE command. Enclose the timestamp and text values in a map collection: curly brackets, separated by a colon.

```
UPDATE users
  SET todo =
  { '2012-9-24' : 'enter mordor',
  '2012-10-2 12:00' : 'throw ring into mount doom' }
  WHERE user_id = 'frodo';
```

You can also update or set a specific element using the UPDATE command. Enclose the timestamp of the element in square brackets and use the equals operator to map the value to that timestamp:

```
UPDATE users SET todo['2012-10-2 12:00'] = 'throw my precious into mount doom'
WHERE user_id = 'frodo';
```

To use INSERT to set or replace map data, specify data in a map collection.

```
INSERT INTO users (todo)
  VALUES ( { '2013-9-22 12:01'  : 'birthday wishes to Bilbo',
  '2013-10-1 18:00' : 'Check into Inn of Prancing Pony' });
```

Inserting this data into the map replaces the entire map.

Deletion

To delete an element from the map, use the DELETE command and enclose the timestamp of the element in square brackets:

```
DELETE todo['2012-9-24'] FROM users WHERE user_id = 'frodo';
```

Retrieval

Like the output of a query on a set, the order of the output of a map is based on the type of the map. To retrieve the todo map:

```
SELECT user_id, todo FROM users WHERE user_id = 'frodo';
```

Setting Expiration

Each element of the map is internally stored as one Cassandra column. Each element can have an individual TTL for instance. If you want elements of the todo list to expire the day of their timestamp, you can compute the correct TTL and set it as follows:

```
UPDATE users USING TTL <computed_ttl>
  SET todo['2012-10-1'] = 'find water' WHERE user_id = 'frodo';
```

Indexing a column

You can use cqlsh can to create *a secondary index* (indexes on column values). This example creates an index on the state and birth_year columns in the users table.

```
cqlsh:demodb> CREATE INDEX state_key ON users (state);
cqlsh:demodb> CREATE INDEX birth_year_key ON users (birth_year);
```

Because you created the secondary index on the two columns, the column values can be queried directly:

```
cqlsh:demodb> SELECT * FROM users
                WHERE gender = 'f' AND
                state = 'TX' AND
                birth_year > 1968
                ALLOW FILTERING;
```

Paging through unordered partitioner results

When using the RandomPartitioner or Murmur3Partitioner, Cassandra rows are ordered by the hash of their value and hence the order of rows is not meaningful. Despite that fact, given the following definition:

```
CREATE TABLE test (
  k int PRIMARY KEY,
  v1 int,
  v2 int
);
```

and assuming RandomPartitioner or Murmur3Partitioner, CQL 2 allows queries like:

```
SELECT * FROM test WHERE k > 42;
```

The semantics of such a query is to query all rows for which the hash of the key was bigger than the hash of 42. The query would return results where $k \leq 42$, which is unintuitive.

CQL 3 forbids such a query unless the partitioner in use is ordered. Even when using the random partitioner or the murmur3 partitioner, it can sometimes be useful to page through all rows. For this purpose, CQL 3 includes the token function:

```
SELECT * FROM test WHERE token(k) > token(42);
```

The ByteOrdered partitioner arranges tokens the same way as key values, but the RandomPartitioner and Murmur3Partitioner distribute tokens in a completely unordered manner. The token function makes it possible to page through unordered partitioner results. Using the token function actually queries results directly using tokens. Underneath, the token function makes token-based comparisons and does not convert keys to tokens (not k > 42).

Getting started using the Cassandra CLI

The legacy Cassandra CLI client utility can be used to do limited Thrift data definition (DDL) and data manipulation (DML) within a Cassandra cluster. CQL 3 is the recommended API for Cassandra. You can *access CQL 3 tables* using CLI. The CLI utility 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. Cassandra connects to the cluster named in the *cassandra.yaml* file. Test Cluster is the default cluster name. 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;
```

A command must be terminated by a semicolon (;). Using the return key without a semicolon at the end of the line echoes 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. This example creates a keyspace called demo, with a replication factor of 1 and using the SimpleStrategy replica placement strategy.

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.

Accessing CQL 3 tables

In Cassandra 1.2 and later, you can use the CLI GET command to query tables created with or without the COMPACT STORAGE directive in CQL 3. The CLI SET command can also be used with CQL 3 tables. For more examples of querying CQL 3 tables, see *Querying a legacy table*.

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 CLI and Thrift, 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*. Cassandra validates that data type of the keys of rows. Columns are sorted, and stored in sorted order on disk, so you have to specify a comparator for columns. You can define the validator and comparator when you create your table schema (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 table schema (or if not specified by the client request).

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

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

About validators

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

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

You cannot know the column names of dynamic tables ahead of time, so specify a *default_validation_class* instead of defining the per-column data types.

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

About the comparator

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

Typically, static table names will be strings, and the sort order of columns is not important in that case. For dynamic tables, however, sort order is important. For example, in a table 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.

Creating a table

First, connect to the keyspace where you want to define the table with the USE command.

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

In this example, we create a users table in the demo keyspace. This table defines a few columns: full_name, email, state, gender, and birth_year. This is considered a static table because the column names are specified 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 values set 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. To create a table using the CLI, you use the *column family keyword*.

```
[default@unknown] USE demo;

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

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

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

Creating a counter table

A counter table 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 table, you must create a table specifically to hold counters.

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

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

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

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

To increment the counter:

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

Inserting rows and columns

The following examples illustrate using the SET command to insert columns for a particular row key into the users table. 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 table 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*.

Reading rows and columns

Use the GET command within Cassandra CLI to retrieve a particular row from a table. 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 table:

```
[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 table, 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 table 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 table or add it later using the UPDATE COLUMN FAMILY command.

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

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

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

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

Deleting rows and columns

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

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

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

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

Or to delete an entire row:

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

Dropping tables and keyspaces

With Cassandra CLI commands you can drop tables 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 table and then drop the demo keyspace altogether:

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

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

CQL 3 Reference

This document covers the CQL 3.0.1 reference for querying Cassandra. For information about CQL 2.0.0, see the CQL reference for Cassandra 1.0.

CQL binary protocol

Although Cassandra continues to support the Thrift RPC indefinitely, the CQL binary protocol summarized in a recent blog post and fully described by the CQL Binary Protocol specification is a flexible alternative.

The CQL binary protocol is a frame-based transport designed for CQL 3. There are a number of benefits that the new 1.2 native CQL transport provides:

- Thrift is a synchronous transport meaning only one request can be active at a time for a connection. By contrast, the new native CQL transport allows each connection to handle more than one active request at the same time. This translates into client libraries only needing to maintain a relatively low number of open connections to a Cassandra node in order to maximize performance, and helps scale large clusters.
- Thrift is an RPC mechanism, which means you cannot have a Cassandra server push information to a client. However the new native CQL protocol allows clients to register for certain types of event notifications from a server. As of 1.2, currently supported events include cluster topology changes, such as a node joins the cluster, is removed, is moved; status changes, such as a node is detected up/down; schema changes, such as a table has been modified. These new capabilities allow clients to stay up to date with the state of the Cassandra cluster without having to poll the cluster regularly.
- The new native protocol allows for messages to be compressed if desired.

To use the new binary protocol, change the start_native_transport option to true in the cassandra.yaml file (you can also turn start_rpc to false if you're not going to use the thrift interface). An open source DataStax Java Driver and .NET Driver supports the CQL binary protocol.

CQL lexical structure

CQL input consists of statements. Like SQL, statements change data, look up data, store data, or change the way data is stored. Statements end in a semicolon (;).

For example, the following is valid CQL syntax:

```
SELECT * FROM MyTable;

UPDATE MyTable
  SET SomeColumn = 'SomeValue'
  WHERE columnName = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

This is a sequence of two CQL statements. This example shows one statement per line, although a statement can usefully be split across lines as well.

Uppercase/lowercase sensitivity

Keyspace, column, and table names created using CQL 3 are case-insensitive unless enclosed in double quotation marks. If you enter names for these objects using any uppercase letters, Cassandra stores the names in lowercase. You can force the case by using double quotation marks. For example:

```
CREATE TABLE test (
  Foo int PRIMARY KEY,
  "Bar" int
);
```

The following table shows partial queries that work and do not work to return results from the test table:

Queries that Work	Queries that Don't Work
SELECT foo FROM ...	SELECT "Foo" FROM ...
SELECT Foo FROM ...	SELECT "BAR" FROM ...
SELECT FOO FROM ...	SELECT bar FROM ...
SELECT "foo" FROM ...	
SELECT "Bar" FROM ...	

SELECT "foo" FROM ... works because internally, Cassandra stores foo in lowercase. The double-quotation mark character can be used as an escape character for the double quotation mark.

Case sensitivity rules in earlier versions of CQL apply when handling legacy tables.

CQL keywords are case-insensitive. For example, the keywords SELECT and select are equivalent. This document shows keywords in uppercase.

Escaping unreadable characters

Column names that contain characters that CQL cannot parse need to be enclosed in double quotation marks in CQL 3. In CQL 2, single quotation marks were used.

Valid literals

Valid literal consist of these kinds of values:

- blob: hexidecimal.
- boolean: true or false, case-insensitive, and in CQL 3, enclosure in single quotation marks is not required prior to release 1.2.2. In 1.2.2 and later, using quotation marks is not allowed.
- A numeric constant can consist of integers 0-9 and a minus sign prefix. A numeric constant can also be float. A float can be a series of one or more decimal digits, followed by a period, ., and one or more decimal digits. There is no optional + sign. The forms .42 and 42 are unacceptable. You can use leading or trailing zeros before and after decimal points. For example, 0.42 and 42.0. A float constant, expressed in E notation, consists of the characters in this regular expression:

  ```
  '-'?[0-9]+('.'[0-9]*)?([eE][+-]?[0-9+])?
  ```

  The next section presents an example.
- identifier: A letter followed by any sequence of letters, digits, or the underscore. Names of tables, columns, and other objects are identifiers. Enclose them in double quotation marks.
- integer: An optional minus sign, -, followed by one or more digits.
- 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, use '' to make dog plural: dog''s.
- 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
- timeuuid: Uses the time in 100 nanosecond intervals since 00:00:00.00 UTC (60 bits), a clock sequence number for prevention of duplicates (14 bits), plus the IEEE 801 MAC address (48 bits) to generate a unique identifier. For example: d2177dd0-eaa2-11de-a572-001b779c76e3
- whitespace: Separates terms and used inside string literals, but otherwise CQL ignores whitespace.

Using exponential notation in CQL 3.0.1

Cassandra 1.2.1 and later supports exponential notation. This example shows exponential notation in the output from a cqlsh 3.0.1 command:

```
CREATE TABLE test(
  id varchar PRIMARY KEY,
  value_double double,
  value_float float
);

INSERT INTO test (id, value_float, value_double)
  VALUES ('test1', -2.6034345E+38, -2.6034345E+38);

SELECT * FROM test;

 id    | value_double | value_float
-------+--------------+-------------
 test1 |  -2.6034e+38 | -2.6034e+38
```

CQL Keywords

Here is a list of keywords and whether or not the words are reserved. A reserved keyword cannot be used as an identifier unless you enclose the word in double quotation marks. Non-reserved keywords have a specific meaning in certain context but can be used as an identifier outside this context.

Word	Reserved
ADD	yes
ALL	no
ALLOW	yes
ALTER	yes
AND	yes
APPLY	yes
ASC	yes
ASCII	no
AUTHORIZE	yes
BATCH	yes
BEGIN	yes
BIGINT	no
BLOB	no
BOOLEAN	no
BY	yes
CLUSTERING	no
COLUMNFAMILY	yes
COMPACT	no
COUNT	no
COUNTER	no
CREATE	yes

DECIMAL	no
DELETE	yes
DESC	yes
DOUBLE	no
DROP	yes
FILTERING	no
FLOAT	no
FROM	yes
GRAMT	yes
IN	yes
INDEX	yes
INET	yes
INSERT	yes
INT	no
INTO	yes
KEY	no
KEYSPACE	yes
KEYSPACES	yes
LIMIT	yes
LIST	no
MAP	no
MODIFY	yes
NORECURSIVE	yes
NOSUPERUSER	no
OF	yes
ON	yes
ORDER	yes
PASSWORD	yes
PERMISSION	no
PERMISSIONS	no
PRIMARY	yes
RENAME	yes
REVOKE	yes
SCHEMA	yes
SELECT	yes
SET	yes
STORAGE	no

SUPERUSER	no
TABLE	yes
TEXT	no
TIMESTAMP	no
TIMEUUID	no
TO	yes
TOKEN	yes
TRUNCATE	yes
TTL	no
TYPE	no
UNLOGGED	yes
UPDATE	yes
USE	yes
USER	no
USERS	no
USING	yes
UUID	no
VALUES	no
VARCHAR	no
VARINT	no
WHERE	yes
WITH	yes
WRITETIME	no

CQL data types

CQL comes with the following built-in data types for columns. One exception is *counter type*, which is allowed only as a column value (not allowed for row key values).

CQL Type	Constants	Description
ascii	strings	US-ASCII character string
bigint	integers	64-bit signed long
blob	blobs	Arbitrary bytes (no validation), expressed as hexadecimal
boolean	booleans	true or false
counter	integers	Distributed counter value (64-bit long)
decimal	integers, floats	Variable-precision decimal
double	integers	64-bit IEEE-754 floating point
float	integers, floats	32-bit IEEE-754 floating point
inet	strings	IP address string in IPv4 or IPv6 format [1]

int	integers	32-bit signed integer
list	n/a	A collection of one or more ordered elements
map	n/a	A collection of one or more timestamp, value pairs
set	n/a	A collection of one or more elements
text	strings	UTF-8 encoded string
timestamp	integers, strings	Date plus time, encoded as 8 bytes since epoch
uuid	uuids	Type 1 or type 4 UUID in standard UUID format
timeuuid	uuids	Type 1 UUID only (CQL 3)
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer

In addition to the CQL types listed in this table, you can use a string containing the name of a JAVA 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.

Enclose ASCII text, timestamp, and inet values in single quotation marks. Enclose names of a keyspace, table, or column in double quotation marks.

Blob

Cassandra 1.2.3 still supports blobs as string constants for input (to allow smoother transition to blob constant). Blobs as strings are now deprecated and will not be supported in the near future. If you were using strings as blobs, update your client code to switch to blob constants.

A blob constant is an hexadecimal number defined by 0[xX](hex)+ where hex is an hexadecimal character, e.g. [0-9a-fA-F]. For example, 0xcafe.

Blob conversion functions

A number of functions convert the native types into binary data (blob). For every <native-type> nonblob type supported by CQL3, the typeAsBlob function takes a argument of type type and returns it as a blob. Conversely, the blobAsType function takes a 64-bit blob argument and converts it to a bigint value. For example, bigintAsBlob(3) is 0x0000000000000003 and blobAsBigint(0x0000000000000003) is 3.

The map, set, and list collection types

A collection column is declared using the collection type, followed by another type, such as int or text, in angle brackets. For example, you can *create a table* having a list of textual elements, a list of integers, or a list of some other element types.

```
list<text>
list<int>
```

Collection types cannot currently be nested. For example, you cannot define a list within a list:

```
list<list<int>>     \\not allowed
```

Currently, you cannot create a secondary index on a column of type map, set, or list.

2 Used by python-cql driver and binary protocols.

UUID types for column names

The UUID (universally unique id) comparator type is used to avoid collisions in column names. Alternatively, you can use the *timeuuid*.

Timeuuid type

A value of the timeuuid type is a Type 1 UUID. A type 1 UUID includes the time of its generation and are sorted by timestamp, making them ideal for use in applications requiring conflict-free timestamps. For example, you can use this type to identify a column (such as a blog entry) by its timestamp and allow multiple clients to write to the same row key simultaneously. Collisions that would potentially overwrite data that was not intended to be overwritten cannot occur.

A valid timeuuid conforms to the timeuuid format shown in *valid expressions*.

Timeuuid functions

You can use these functions with the timeuuid type:

- dateOf()

  Used in a SELECT clause, this function extracts the timestamp of a timeuuid column in a resultset. This function returns the extracted timestamp as a date. Use unixTimestampOf() to get a raw timestamp.

- now()

  Generates a new unique timeuuid when the statement is executed. This method is useful for inserting values. The value returned by now() is guaranteed to be unique.

- minTimeuuid() and maxTimeuuid()

  Returns a UUID-like result given a conditional time component as an argument. For example:

  ```
  SELECT * FROM myTable
     WHERE t > maxTimeuuid('2013-01-01 00:05+0000')
     AND t < minTimeuuid('2013-02-02 10:00+0000')
  ```

  This example selects all rows where the timeuuid column, t, is strictly later than 2013-01-01 00:05+0000 but strictly earlier than 2013-02-02 10:00+0000. The t >= maxTimeuuid('2013-01-01 00:05+0000') does not select a timeuuid generated exactly at 2013-01-01 00:05+0000 and is essentially equivalent to t > maxTimeuuid('2013-01-01 00:05+0000').

  The values returned by minTimeuuid and maxTimeuuid functions are not true UUIDs in that the values do not conform to the Time-Based UUID generation process specified by the RFC 4122.

  ### Warning

  The values returned by these methods are not unique. Use these methods for querying only. Inserting the result of these methods in the database is not recommended.

- unixTimestampOf()

  Used in a SELECT clause, this functions extracts the timestamp of a timeuuid column in a resultset. Returns the value as a raw, 64-bit integer timestamp.

Timestamp type

Values for the timestamp and timeuuid types 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 and timeuuid types can be entered as integers for CQL input.

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
```

where Z is the RFC-822 4-digit time zone, expressing the time zone's difference from UTC. For example, for the date and time of Jan 2, 2003, at 04:05:00 AM, GMT:

```
2011-02-03 04:05+0000
2011-02-03 04:05:00+0000
2011-02-03T04:05+0000
2011-02-03T04:05:00+0000
```

The +0000 is the RFC 822 4-digit time zone specification for GMT. US Pacific Standard Time is -0800. The time zone may be omitted. For example:

```
2011-02-03 04:05
2011-02-03 04:05:00
2011-02-03T04:05
2011-02-03T04:05:00
```

If no time zone is specified, the time zone of the Cassandra coordinator node handing the write request is used. For accuracy, DataStax recommends specifying the time zone rather than relying on the time zone configured on the Cassandra nodes.

If you only want to capture date values, the time of day can also be omitted. For example:

```
2011-02-03
2011-02-03+0000
```

In this case, the time of day defaults to 00:00:00 in the specified or default time zone.

Timestamp output appears in the following format by default:

```
yyyy-mm-dd HH:mm:ssZ
```

You can change the format by setting the time_format property in the [ui] section of the .cqlshrc file.

counter type

To use counter types, see the DataStax blog about counters. Do not assign this type to a column that serves as the primary key. Also, do not use the counter type in a table that contains anything other than counter types (and primary key). To generate sequential numbers for surrogate keys, use the timeuuid type instead of the counter type.

CQL storage parameters

Certain CQL commands allow a WITH clause for setting certain properties on a keyspace or table. Note that CQL does not currently offer support for using *all of the Cassandra table attributes* as CQL storage parameters, just a subset.

CQL keyspace storage parameters

CQL supports setting the following keyspace properties.

- *strategy_class* The name of the replication strategy: SimpleStrategy or NetworkTopologyStrategy.
- *strategy_options* Replication strategy options.

CQL 3 table storage properties

CQL supports Cassandra table attributes through the CQL properties listed in the following table. The table includes the default value of the property. In CQL statements, such as CREATE TABLE, you format properties in either the name-value pair or collection map format.

The name-value pair property syntax is:

```
name = value AND name = value
```

The collection map format is:

```
{ name : value, name, value : name, value ... }
```

See *CREATE TABLE* for examples.

CQL Properties Name	Format
bloom_filter_fp_chance	name: value
caching	name: value
comment	name: value
compaction	map
compression	map
dclocal_read_repair_chance	name: value
gc_grace_seconds	name: value
read_repair_chance	name: value
replicate_on_write	name: value

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.

CQL 3 subproperties of compaction

Using CQL, you can configure compaction for a table by constructing a map of the compaction property and the following subproperties:

CQL Compaction Subproperties Name	Supported Strategy
bucket_high	SizeTieredCompactionStrategy
bucket_low	SizeTieredCompactionStrategy
max_threshold	SizeTieredCompactionStrategy
min_threshold	SizeTieredCompactionStrategy
min_sstable_size	SizeTieredCompactionStrategy
sstable_size_in_mb	LeveledCompactionStrategy

tombstone_compaction_interval	all
tombstone_threshold	all

CQL 3 subproperties of compression

Using CQL, you can configure compression for a table by constructing a map of the compression property and the following subproperties:

- *sstable_compression*
- *chunk_length_kb*
- *crc_check_chance*

To query the Cassandra database, *use CQL commands* described in the next section.

CQL Command Table of Contents

CQL Commands	cqlsh Commands
ALTER KEYSPACE	*ASSUME*
ALTER TABLE	*CAPTURE*
ALTER USER	*CONSISTENCY*
BATCH	*COPY*
CREATE INDEX	*DESCRIBE*
CREATE KEYSPACE	*EXIT*
CREATE TABLE	*SHOW*
CREATE USER	*SOURCE*
DELETE	*TRACING*
DROP INDEX	
DROP KEYSPACE	
DROP TABLE	
DROP USER	
GRANT	
INSERT	
LIST PERMISSIONS	
LIST USERS	
REVOKE	
SELECT	
TRUNCATE	
UPDATE	
USE	

ALTER KEYSPACE

Change property values of a keyspace.

Synopsis

```
ALTER KEYSPACE | SCHEMA keyspace_name
  WITH REPLICATION = map
  | WITH DURABLE_WRITES = true | false
  | WITH REPLICATION = map
  AND DURABLE_WRITES = true | false
```

map is a map collection:

```
{ property : value, property, value : property, value ... }
```

Synopsis legend

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- « means a non-literal, open parenthesis used to indicate scope
- » means a non-literal, close parenthesis used to indicate scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

ALTER KEYSPACE changes the map that defines the replica placement strategy and/or the durable_writes value. You can also use the alias ALTER SCHEMA. Use these properties and values to construct the map. To set the replica placement strategy, construct a map of properties and values, as shown in *the table of map properties* on the CREATE KEYSPACE reference page.

You cannot change the name of the keyspace.

Example

Continuing with the example in *CREATE KEYSPACE*, change the definition of the Excalibur keyspace to use the SimpleStrategy and a replication factor of 3.

```
ALTER KEYSPACE "Excalibur" WITH REPLICATION =
  { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

ALTER TABLE

Modifies the column metadata of a table.

Synopsis

```
ALTER TABLE keyspace_name.table_name
ALTER column_name TYPE cql_type
| ADD column_name cql_type
| DROP column_name
| RENAME column_name TO column_name
| WITH property AND property ...
```

cql_type is a type, other than a collection or *counter type* type, listed in *CQL data types*. Exceptions: ADD supports a collection type and also, if the table is a counter, a counter type.

property is a *CQL table storage property* and value, such as caching = 'all'

Synopsis legend

Description

ALTER TABLE manipulates the table metadata. You can change the data storage type of columns, add new columns, drop existing columns, and change table properties. No results are returned.

You can also use the alias ALTER COLUMNFAMILY.

See *CQL data types* for the available data types and *CQL 3 table storage properties* for column properties and their default values.

First, specify the name of the table to be changed after the ALTER TABLE keywords, followed by the type of change: ALTER, ADD, DROP, RENAME, or WITH. Next, provide the rest of the needed information, as explained in the following sections.

You can qualify table names by keyspace. For example, to alter the addamsFamily table in the monsters keyspace:

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

Examples:

- *Changing the type of a column*
- *Adding a column*
- *Dropping a column*, not available in this release.
- *Modifying table options*
- *Renaming a column*
- *Modifying the compression or compaction setting*

Changing the type of a column

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

```
ALTER TABLE addamsFamily ALTER plot_number TYPE uuid;
```

The column must already exist in current rows. The bytes stored in values for that column remain unchanged, and if existing data cannot be deserialed according to the new type, your CQL driver or interface might report errors.

These changes to a column type are not allowed:

- Changing the type of a *clustering column*.
- Changing columns on which a *secondary index* is defined.

Adding a column

To add a column, other than a column of a collection type, to a table, use ALTER TABLE and the ADD keyword in the following way:

```
ALTER TABLE addamsFamily ADD gravesite varchar;
```

To add a column of the *collection type*:

```
ALTER TABLE users ADD top_places list<text>;
```

The column may or may not already exist in current rows. No validation of existing data occurs.

These additions to a table are not allowed:

- Adding a column having the same name as an existing column.
- Adding columns to tables defined with COMPACT STORAGE.

Dropping a column

This feature is not ready in Cassandra 1.2 but will be available in a subsequent version.

To drop a column from the table, use ALTER TABLE and the DROP keyword in the following way:

```
ALTER TABLE addamsFamily DROP gender;
```

Dropping a column removes the column from current rows.

Renaming a column

The main purpose of the RENAME clause is to change the names of CQL 3-generated row key and column names that are missing from a *legacy tables*.

Modifying table options

To change the table storage options established during creation of the table, use ALTER TABLE and the WITH keyword. To change multiple properties, use *AND* as shown in this example:

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

See *CQL 3 table storage properties* for the table options you can define.

You cannot modify table options of a table having compact storage.

Modifying the compression or compaction setting

Changing any compaction or compression option erases all previous compaction or compression settings.

```
ALTER TABLE addamsFamily
  WITH compression =
  { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 };

ALTER TABLE users
  WITH compaction =
  { 'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 };
```

ALTER USER

Alter existing user options.

Synopsis

```
ALTER USER user_name
  WITH PASSWORD 'password' NOSUPERUSER | SUPERUSER
```

Synopsis legend

Description

Superusers can change a user's password or superuser status. To prevent disabling all superusers, superusers cannot change their own superuser status. Ordinary users can change only their own password.

Enclose the user name in single quotation marks if it contains non-alphanumeric characters. Enclose the password in single quotation marks.

Example

```
ALTER USER moss WITH PASSWORD 'bestReceiver';
```

BATCH

Writes multiple DML statements and sets a client-supplied timestamp for all columns written by the statements in the batch.

Synopsis

```
BEGIN BATCH
  | BEGIN UNLOGGED
  | BEGIN COUNTER
  USING TIMESTAMP timestamp;
  dml_statement
  dml_statement
  ...
APPLY BATCH;
```

dml_statement is:

- *INSERT*
- *UPDATE*
- *DELETE*

Synopsis legend

Description

A BATCH statement combines multiple data modification language (DML) statements (INSERT, UPDATE, DELETE) into a single logical operation. Batching multiple statements saves network exchanges between the client/server and server coordinator/replicas.

In Cassandra 1.2 and later, batches are atomic by default. In the context of a Cassandra batch operation, atomic means that if any of the batch succeeds, all of it will. To achieve atomicity, Cassandra first writes the serialized batch to the batchlog system table that consumes the serialized batch as blob data. When the rows in the batch have been successfully written and persisted (or hinted) the batchlog data is removed. There is a performance penalty for atomicity. If you do not want to incur this penalty, prevent Cassandra from writing to the batchlog system by using the UNLOGGED option: BEGIN UNLOGGED BATCH

Although an atomic batch guarantees that if any part of the batch succeeds, all of it will, no other transactional enforcement is done at the batch level. For example, there is no batch isolation. Other clients are able to read the first updated rows from the batch, while other rows are in progress. However, transactional row updates within a single row are isolated: a partial row update cannot be read.

Using a timestamp

BATCH supports setting a client-supplied timestamp, an integer, in the USING clause that is used by all batched operations. If not specified, the current time of the insertion (in microseconds) is used. Individual DML statements inside a BATCH cannot specify a timestamp.

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.

Batching counter updates

Use BEGIN COUNTER BATCH in a batch statement for batched counter updates. Unlike other writes in Cassandra, counter updates are not *idempotent*, so replaying them automatically from the new system table is not safe. Counter batches are thus strictly for improved performance when updating multiple counters in the same partition.

Example

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

CREATE INDEX

Define a new, secondary index on a single column of a table.

Synopsis

```
CREATE INDEX index_name
  ON keyspace_name.table_name ( column_name )
```

Synopsis legend

Description

CREATE INDEX creates a new, automatic secondary index on the given table 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. The column and its data type must be specified when the table is created, or added afterward by altering the table.

If data already exists for the column, Cassandra indexes the data during the execution of this statement. After the index is created, Cassandra indexes new data for the column automatically when new data is inserted.

In this release, Cassandra supports creating an index on a table having a *compound primary key*. You cannot create a secondary index on the primary key itself. Cassandra does not support secondary indexes on collections.

Examples

Define a table and then create a secondary index on two of its named columns:

```
 CREATE TABLE myschema.users (
   userID uuid,
   fname text,
   lname text,
   email text,
   address text,
   zip int,
   state text,
   PRIMARY KEY (userID)
 );

CREATE INDEX user_state
   ON myschema.users (state);

CREATE INDEX ON myschema.users (zip);
```

Define a table having a compound primary key and create a secondary index on it.

```
USE myschema;

CREATE TABLE timeline (
  user_id varchar,
  email_id uuid,
  author varchar,
  body varchar,
  PRIMARY KEY (user_id, email_id)
);

CREATE INDEX ON timeline (author);
```

CREATE KEYSPACE

Define a new keyspace and its replica placement strategy.

Synopsis

```
CREATE KEYSPACE | SCHEMA keyspace_name WITH REPLICATION = map
  AND DURABLE_WRITES = true | false
```

map is a map collection:

```
{ property : value, property, value : property, value ... }
```

Synopsis legend

Description

CREATE KEYSPACE creates a top-level namespace and sets the keyspace name, replica placement *strategy class*, *replication options*, and *durable_writes* options for the keyspace. Keyspace names are 32 or fewer alpha-numeric characters and underscores, the first of which is an alpha character. Keyspace names are case-insensitive. To make a name case-sensitive, enclose it in double quotation marks.

To set the replica placement strategy, construct a map of properties and values.

Table of map properties and values

Property	Value	Value Description
'class'	'SimpleStrategy' or 'NetworkTopologyStrategy'	Required. The name of the *replica placement strategy* class for the new keyspace.
'replication_factor'	An integer	Required if class is SimpleStrategy; otherwise, not used. The number of replicas of data on multiple nodes.
'<datacenter name>'	An integer	Required if class is NetworkTopologyStrategy; otherwise, not used. The number of replicas of data on each node in the data center.
'<datacenter name>'	An integer	Optional if class is NetworkTopologyStrategy. The number of replicas of data on each node in the data center.
. . .	. . .	More optional replication factors for additional named data centers.

You can also use the alias CREATE SCHEMA.

Example of Setting the SimpleStrategy class

Construct the CREATE KEYSPACE statement by first declaring the name of the keyspace, followed by the WITH REPLICATION keywords and the equals symbol. Next, to create a keyspace that is not optimized for multiple data centers, use SimpleStrategy for the class value in the map. Set replication_factor properties, separated by a colon and enclosed in curly brackets. For example:

```
CREATE KEYSPACE Excelsior
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

Using SimpleStrategy is fine for evaluating Cassandra. For production use or for use with mixed workloads, use NetworkTopologyStrategy.

Example of Setting the NetworkToplogyStrategy class

Before creating a keyspace that is optimized for multiple data centers, configure the cluster that will use the keyspace. Configure the cluster to use a network-aware snitch, such as the *PropertyFileSnitch*. Create a keyspace using NetworkToplogyStrategy for the class value in the map. Set one or more key-value pairs consisting of the data center name and number of replicas per data center, separated by a colon and enclosed in curly brackets. For example:

```
CREATE KEYSPACE "Excalibur"
  WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc1' : 3, 'dc2' : 2};
```

This example sets one replica for a data center named dc1 and two replicas for a data center named dc2. The data center name you use depends on the cluster-configured *snitch* you are using. There is a correlation between the data center name defined in the map 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.

For more information about choosing a replication strategy for a cluster, see *Choosing keyspace replication options*.

Example of setting the durable_writes property

You can set the durable_writes option after the map specification of the CREATE KEYSPACE command. When set to false, data written to the keyspace bypasses the commit log. Be careful using this option because you risk losing data. Do not set this attribute on a keyspace using the SimpleStrategy.

```
CREATE KEYSPACE Risky
  WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
  'dc1' : 1 } AND durable_writes = false;
```

Checking created keyspaces

Check that the keyspaces were created:

```
select * from system.schema_keyspaces;
```

```
keyspace_name  | durable_writes | strategy_class                                          | strategy_options
---------------+----------------+---------------------------------------------------------+---------------------------
     excelsior |           True |          org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"3"}
     Excalibur |           True | org.apache.cassandra.locator.NetworkTopologyStrategy |      {"dc2":"2","dc1":"3"}
         risky |          False | org.apache.cassandra.locator.NetworkTopologyStrategy |                 {"dc1":"1"}
        system |           True |           org.apache.cassandra.locator.LocalStrategy |                          {}
 system_traces |           True |          org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}
```

Cassandra converted the excelsior keyspace to lowercase because quotation marks were not used to create the keyspace and retained the initial capital letter for the Excalibur because quotation marks were used.

CREATE TABLE

Define a new table.

Synopsis

```
CREATE TABLE keyspace_name.table_name
( column_definition, column_definition, ...)
WITH property AND property ...
```

column_definition is:

```
column_name cql_type
| column_name cql_type PRIMARY KEY
| PRIMARY KEY ( partition_key )
| column_name collection_type
```

cql_type is a type, other than a collection or *counter type* type, listed in *CQL data types*. Exceptions: ADD supports a collection type and also, if the table is a counter, a counter type.

partition_key is:

```
column_name
| ( column_name1, column_name2, column_name3 ...)
| ((column_name1■, column_name2■), column_name3■,
  column_name4■ . . .)
```

column_name1 is the partition key. column_name2, column_name3 ... are clustering keys. column_name1■, column_name2■ are partitioning keys. column_name3■, column_name4■... are clustering keys.

collection_type is:

```
LIST <cql_type>
| SET <cql_type>
| MAP <cql_type, cql_type>
```

property is a one of the *CQL table storage options* or a directive. A directive is either:

- COMPACT STORAGE
- CLUSTERING ORDER followed by the *clustering order specification*.

Synopsis legend

Description

CREATE TABLE creates a new *table* under the current keyspace. You can also use the alias CREATE COLUMNFAMILY. Valid table names are strings of alphanumeric characters and underscores, which begin with a letter. If you add the keyspace name followed by a period to the name of the table, Cassandra creates the table in the specified keyspace, but does not change the current keyspace; otherwise, if you do not use a keyspace name, Cassandra creates the table within the current keyspace.

Examples:

- *Defining a primary key column*
- *Using compound primary keys*
- *Defining columns*
- *Setting table options*
- *Using compact storage*
- *Using clustering order*
- *Using collections: set, list, and map*

Defining a primary key column

The only schema information that must be defined for a table is the primary key (or row key) and its associated data type. Unlike earlier versions, CQL 3 does not require a column in the table that is not part of the primary key. A primary key can have any number (1 or more) of component columns.

If the primary key consists of only one column, you can use the keywords, PRIMARY KEY, after the column definition:

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

Alternatively, you can declare the primary key consisting of only one column in the same way as you declare a compound primary key. Do not use a counter column for a key.

Using compound primary keys

Compound primary keys consist of more than one column. Use the keywords, PRIMARY KEY, followed by the comma-separated list of column names enclosed in parentheses.

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

Using composite partition keys

The first column declared in the PRIMARY KEY definition is the partition key, as discussed in *Compound keys and clustering*. Cassandra stores rows that share the partition key on the same physical node. You can declare a composite partition key formed of multiple columns by using an extra set of parentheses to define which columns form the compound partition key.

```
CREATE TABLE Cats (
  block_id uuid,
  breed text,
  color text,
  short_hair boolean,
  PRIMARY KEY ((block_id, breed), color, short_hair)
);
```

The composite partition key consists of block_id and breed. The clustering columns, color and short_hair, determine the clustering order of rows that facilitates retrieval.

Defining columns

You assign columns a type during table creation. Column types, other than collection-type columns, are specified as a parenthesized, comma-separated list of column name and type pairs. See *CQL data types* for the available types.

This example shows how to create a table that includes collection-type columns: map, set, and list.

```
CREATE TABLE users (
  userid text PRIMARY KEY,
  first_name text,
  last_name text,
  emails set<text>,
  top_scores list<int>,
  todo map<timestamp, text>
);
```

For information about using collections, see *Using collections: set, list, and map*.

Setting table options

Using the optional WITH clause and keyword arguments, you can control the configuration of a new table. You can use the WITH clause to specify the attributes of tables listed in *CQL 3 table storage properties*. For example:

```
CREATE TABLE MonkeyTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
)
```

```
WITH comment='Important biological records'
AND read_repair_chance = 1.0;

CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH compression =
    { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 }
  AND compaction =
    { 'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 };
```

You can specify *using compact storage* or *clustering order* using the WITH clause.

Using compact storage

When you create a table using compound primary keys, for every piece of data stored, the column name needs to be stored along with it. Instead of each non-primary key column being stored such that each column corresponds to one column on disk, an entire row is stored in a single column on disk. If you need to conserve disk space, use the WITH COMPACT STORAGE directive that stores data essentially the same as it was stored under CQL 2.

```
CREATE TABLE sblocks (
  block_id uuid,
  subblock_id uuid,
  data blob,
  PRIMARY KEY (block_id, subblock_id)
)
WITH COMPACT STORAGE;
```

Using the compact storage directive prevents you from adding more than one column that is not part of the PRIMARY KEY.

At this time, updates to data in a table created with compact storage are not allowed. The table with compact storage that uses a compound primary key must define at least one clustering column.

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

Using clustering order

You can order query results to make use of the on-disk sorting of columns. You can order results in ascending or descending order. The ascending order will be more efficient than descending. If you need results in descending order, you can specify a clustering order to store columns on disk in the reverse order of the default. Descending queries will then be faster than ascending ones.

The following example shows a table definition that changes the clustering order to descending by insertion time.

```
create table timeseries (
  event_type text,
  insertion_time timestamp,
  event blob,
  PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

CREATE USER

Create a new user.

Synopsis

```
CREATE USER user_name
  WITH PASSWORD 'password' NOSUPERUSER | SUPERUSER
```

Synopsis legend

Description

CREATE USER defines a new database user account. By default users accounts do not have superuser status. Only a superuser can issue CREATE USER requests.

User accounts are required for logging in under *internal authentication* and *authorization*.

Enclose the user name in single quotation marks if it contains non-alphanumeric characters. You cannot recreate an existing user. To change the superuser status or password, use *ALTER USER*.

Creating internal user accounts

You need to use the WITH PASSWORD clause when creating a user account for internal authentication. Enclose the password in single quotation marks.

Example

```
CREATE USER spillman WITH PASSWORD 'Niner27';
CREATE USER akers WITH PASSWORD 'Niner2' SUPERUSER;
CREATE USER boone WITH PASSWORD 'Niner75' NOSUPERUSER;
```

If internal authentication has not been set up, you do not need the WITH PASSWORD clause:

```
CREATE USER test NOSUPERUSER;
```

DELETE

Removes entire rows or one or more columns from one or more rows.

Synopsis

```
DELETE column_name, ... | collection_colname [ term ]  FROM keyspace_name.table_name
  USING TIMESTAMP integer
  WHERE row_specification
```

term is:

```
[ list_index_position | [ list_value ]
```

row_specification is:

```
primary_key_name = key_value
primary_key_name IN ( key_value, key_value, ...)
```

Synopsis legend

Description

A DELETE statement removes one or more columns from one or more rows in a table, or it removes the entire row if no columns are specified. Cassandra applies seletions within the same partition key *atomically* and in *isolation*.

Specifying Columns

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

```
DELETE col1, col2, col3 FROM Planeteers WHERE userID = 'Captain';
```

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

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

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 the table

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

Identifying a column for deletion using a TIMESTAMP

You can identify the column for deletion using a timestamp.

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

The TIMESTAMP input is an integer representing microseconds.

Specifying Rows

The WHERE clause specifies which row or rows to delete from the table.

```
DELETE col1 FROM SomeTable WHERE userID = 'some_key_value';
```

This form provides a list of key names using the IN notation and a parenthetical list of comma-delimited key names.

```
DELETE col1 FROM SomeTable WHERE userID IN (key1, key2);
```

Example

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

Using a collection set, list or map

To delete an element from the map, use the DELETE command and enclose the timestamp of the element in square brackets:

```
DELETE todo ['2012-9-24'] FROM users WHERE user_id = 'frodo';
```

To remove an element from a list, use the DELETE command and the list index position in square brackets:

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';
```

To remove all elements from a set, you can use the DELETE statement:

```
DELETE emails FROM users WHERE user_id = 'frodo';
```

DROP TABLE

Removes the named table.

Synopsis

```
DROP TABLE keyspace_name.table_name
```

Synopsis legend

Description

A DROP TABLE statement results in the immediate, irreversible removal of a table, including all data contained in the table. You can also use the alias DROP COLUMNFAMILY.

Example

```
DROP TABLE worldSeriesAttendees;
```

DROP INDEX

Drops the named secondary index.

Synopsis

```
DROP INDEX name
```

Synopsis legend

Description

A DROP INDEX statement removes an existing secondary index. If the index was not given a name during creation, the index name is `<table_name>_<column_name>_idx`.

Example

```
DROP INDEX user_state;

DROP INDEX users_zip_idx;
```

DROP KEYSPACE

Removes the keyspace.

Synopsis

```
DROP KEYSPACE | SCHEMA name
```

Synopsis legend

Description

A DROP KEYSPACE statement results in the immediate, irreversible removal of a keyspace, including all tables and data contained in the keyspace. You can also use the alias DROP SCHEMA.

Example

```
DROP KEYSPACE MyTwitterClone;
```

DROP USER

Synopsis

```
DROP USER user_name
```

Synopsis legend

Description

DROP USER removes an existing user. You have to be logged in as a superuser to issue a DROP USER statement. A user cannot drop themselves.

Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

GRANT

Provides users access to database objects.

Synopsis

```
GRANT permission_name PERMISSION
| GRANT ALL PERMISSIONS
    ON resource TO user
```

permission_name is one of these:

- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resource is one of these:

- ALL KEYSPACES

- KEYSPACE keyspace_name
- TABLE *keyspace_name*.table_name

Synopsis legend

Description

Permissions to access all keyspaces, a named keyspace, or a table can be granted to a user. Enclose the user name in single quotation marks if it contains non-alphanumeric characters.

This table lists the permissions needed to use CQL statements:

Permission	CQL Statements
ALTER	ALTER KEYSPACE, ALTER TABLE, CREATE INDEX, DROP INDEX
AUTHORIZE	GRANT, REVOKE
CREATE	CREATE KEYSPACE, CREATE TABLE
DROP	DROP KEYSPACE, DROP TABLE
MODIFY	INSERT, DELETE, UPDATE, TRUNCATE
SELECT	SELECT

To be able to perform SELECT queries on a table, you have to have SELECT permission on the table, on its parent keyspace, or on ALL KEYSPACES. To be able to CREATE TABLE you need CREATE permission on its parent keyspace or ALL KEYSPACES. You need to be a superuser or to have AUTHORIZE permission on a resource (or one of its parents in the hierarchy) plus the permission in question to be able to GRANT or REVOKE that permission to or from a user. GRANT, REVOKE and LIST permissions check for the existence of the table and keyspace before execution. GRANT and REVOKE check that the user exists.

Examples

Give 'spillman' permission to perform SELECT queries on all tables in all keyspaces:

```
GRANT SELECT ON ALL KEYSPACES TO spillman;
```

Give 'akers' permission to perform INSERT, UPDATE, DELETE and TRUNCATE queries on all tables in the 'field' keyspace:

```
GRANT MODIFY ON KEYSPACE field TO akers;
```

Give 'boone' permission to perform ALTER KEYSPACE queries on the 'forty9ers' keyspace, and also ALTER TABLE, CREATE INDEX and DROP INDEX queries on all tables in 'forty9ers' keyspace:

```
GRANT ALTER ON KEYSPACE forty9ers TO boone;
```

Give 'boone' permission to run all types of queries on ravens.plays table:

```
GRANT ALL PERMISSIONS ON ravens.plays TO boone;
```

To grant access to a keyspace to just one user, assuming nobody else has ALL KEYSPACES access, you use this statement:

```
GRANT ALL ON KEYSPACE keyspace_name TO user_name
```

INSERT

Adds or updates one or more columns in the identified row of a table.

Synopsis

```
INSERT INTO keyspace_name.table_name
  ( identifier, identifier...)
  VALUES ( value, value ... )
  USING option AND option
```

identifier is a column or a collection name.

Value is one of:

- a column value
- a set:

  ```
  { item1, item2, . . . }
  ```

- a list:

  ```
  [ name, value ]
  ```

- a map:

  ```
  { name : value, name : value, . . . }
  ```

option is one of:

- TIMESTAMP string
- TTL seconds

Synopsis legend

Description

An INSERT writes one or more columns to a record in a Cassandra table *atomically* and in *isolation*. No results are returned. You do not have to define all columns, except those that make up the key. Missing columns occupy no space on disk.

If the column exists, it is updated. You can qualify table names by keyspace. INSERT does not support counters, but UPDATE does.

Example

```
INSERT INTO playlists (id, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204,
  a3e64f8f-bd44-4f28-b8d9-6938726e34d4, 'La Grange', 'ZZ Top', 'Tres Hombres');

INSERT INTO playlists (id, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204,
  8a172618-b121-4136-bb10-f665cfc469eb, 'Moving in Stereo', 'Fu Manchu', 'We Must Obey');

INSERT INTO playlists (id, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204,
  2b09185b-fb5a-4734-9b56-49077de9edbf, 'Outside Woman Blues', 'Back Door Slam', 'Roll Away');
```

Specifying TIMESTAMP and TTL

You can specify one or more of these options after the USING keyword:

- Time-to-live (*TTL*) in seconds

- Timestamp

```
INSERT INTO Hollywood.NerdMovies (user_uuid, fan)
  VALUES (cfd66ccc-d857-4e90-b1e5-df98a3d40cd6, 'johndoe')
  USING TTL 86400;
```

TTL input is in seconds. TTL column values are automatically marked as deleted (with a *tombstone*) after the requested amount of time has expired. TTL marks the inserted values, not the column itself, for expiration. Any subsequent update of the column resets the TTL to the TTL specified in the update. By default, values never expire.

The TIMESTAMP input is in one of the following formats:

- A string must be in *ISO 8601 format*.
- An integer representing microseconds.

If not specified, the time (in microseconds) that the write occurred to the column is used.

Using a collection set or map

To insert data into the set, enclose values in curly brackets. Set values must be unique. For example:

```
INSERT INTO users (user_id, first_name, last_name, emails)
  VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com', 'baggins@gmail.com'});
```

Insert a map named todo to insert a reminder, 'die' on October 2 for user frodo.

```
INSERT INTO users (user_id, todo )
  VALUES('frodo', {'2012-10-2 12:10' : 'die' } );
```

LIST PERMISSIONS

Lists permissions granted to a user.

Synopsis

```
LIST permission_name PERMISSION
| LIST ALL PERMISSIONS
    ON resource OF user_name
    NORECURSIVE
```

permission_name is one of these:

- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resource is one of these:

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name.*table_name

Synopsis legend

Description

Permissions checks are recursive. If you omit the NORECURSIVE specifier, permission on the requests resource and its parents in the hierarchy are shown.

- Omitting the resource name (ALL KEYSPACES, keyspace, or table), lists permissions on all tables and all keyspaces.
- Omitting the user name lists permissions of all users. You need to be a superuser to list permissions of all users. If you are not, you must add `of <myusername>`.
- Omitting the NORECURSIVE specifier, lists permissions on the resource and its parent resources.
- Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

After creating users in *Creating internal user accounts* and granting the permissions in the *GRANT* examples, you can list permissions that users have on resources and their parents.

Example

Assuming you completed the examples in *Examples*, list all permissions given to akers:

```
LIST ALL PERMISSIONS OF akers;
```

Output

```
 username | resource           | permission
----------+--------------------+------------
    akers | <keyspace field>   |     MODIFY
```

List permissions given to all the users:

```
LIST ALL PERMISSIONS;
```

Output

```
 username | resource             | permission
----------+----------------------+------------
    akers |     <keyspace field> |     MODIFY
    boone | <keyspace forty9ers> |      ALTER
    boone | <table ravens.plays> |     CREATE
    boone | <table ravens.plays> |      ALTER
    boone | <table ravens.plays> |       DROP
    boone | <table ravens.plays> |     SELECT
    boone | <table ravens.plays> |     MODIFY
    boone | <table ravens.plays> |  AUTHORIZE
 spillman |      <all keyspaces> |     SELECT
```

List all permissions on the plays table:

```
LIST ALL PERMISSIONS ON ravens.plays;

 username | resource             | permission
----------+----------------------+------------
    boone | <table ravens.plays> |     CREATE
    boone | <table ravens.plays> |      ALTER
    boone | <table ravens.plays> |       DROP
    boone | <table ravens.plays> |     SELECT
    boone | <table ravens.plays> |     MODIFY
    boone | <table ravens.plays> |  AUTHORIZE
```

```
  spillman |      <all keyspaces> |     SELECT
```

List all permissions on the ravens.plays table and its parents:

```
 LIST ALL PERMISSIONS ON ravens.plays NORECURSIVE;

  username | resource             | permission
----------+----------------------+------------
     boone | <table ravens.plays> |     CREATE
     boone | <table ravens.plays> |      ALTER
     boone | <table ravens.plays> |       DROP
     boone | <table ravens.plays> |     SELECT
     boone | <table ravens.plays> |     MODIFY
     boone | <table ravens.plays> |  AUTHORIZE
```

LIST USERS

Lists existing users and their superuser status.

Synopsis

```
    LIST USERS
```

Synopsis legend

Description

Assuming you use internal authentication, created the users in *Creating internal user accounts*, and have not yet changed the default user, the following example shows the output of LIST USERS.

Example

```
 LIST USERS;
```

Output

```
  name      | super
-----------+-------
  cassandra |  True
      boone | False
      akers |  True
   spillman | False
```

REVOKE

Synopsis

```
    REVOKE permission_name PERMISSION
    | REVOKE ALL PERMISSIONS
       ON resource FROM user_name
```

permission_name is one of these:

- ALTER

- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resource is one of these:

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name*.table_name

Synopsis legend

Description

Permissions to access all keyspaces, a named keyspace, or a table can be revoked from a user. Enclose the user name in single quotation marks if it contains non-alphanumeric characters.

This table lists the permissions needed to use CQL statements:

Permission	CQL Statements
ALTER	ALTER KEYSPACE, ALTER TABLE, CREATE INDEX, DROP INDEX
AUTHORIZE	GRANT, REVOKE
CREATE	CREATE KEYSPACE, CREATE TABLE
DROP	DROP KEYSPACE, DROP TABLE
MODIFY	INSERT, DELETE, UPDATE, TRUNCATE
SELECT	SELECT

Example

```
REVOKE SELECT ON ravens.plays FROM boone;
```

The user boone can no longer perform SELECT queries on the ravens.plays table. Exceptions: Because of inheritance, the user can perform SELECT queries on revens.plays if one of these conditions is met:

- The user is a superuser
- The user has SELECT on ALL KEYSPACES permissions
- The user has SELECT on the ravens keyspace

SELECT

Retrieves data, including Solr data, from a Cassandra table.

Synopsis

```
SELECT select_expression
  FROM keyspace_name.table_name
  WHERE clause AND clause ...
  ORDER BY compound_key_2 ASC | DESC
  LIMIT n
  ALLOW FILTERING
```

select expression is:

```
selection_list
| COUNT ( * | 1 )
```

selection_list is:

```
selector , selector, ...| *
```

selector is:

```
WRITETIME (col_name)
| TTL «(col_name) | * »| function (selector , selector, ...)
```

function is a *timeuuid function*, a *token function*, or a *blob conversion function*.

WHERE clause syntax is:

```
relation AND relation
primary_key_name = «| < | > | <= | >= »key_value
| primary_key_name> IN ( key_value,... )
| TOKEN (partitioner_key)  «| < | > | <= | >= »  «term | TOKEN ( term ) »
```

Synopsis legend

Description

A SELECT statement reads one or more records from a Cassandra table. The input to the SELECT statement is the select expression. The output of the select statement depends on the select expression:

Select Expression	Output
Column or list of columns	Rows having a key value and collection of columns
COUNT aggregate function	One row with a column that has the value of the number of rows in the resultset
WRITETIME function	The date/time that a write to a column occurred
TTL function	The remaining time-to-live for a column

Specifying columns

The SELECT expression determines which columns, if any, appear in the result. Using the asterisk specifies selection of all columns:

```
SELECT * from People;
```

Select two columns, Name and Occupation, from three rows having employee ids (primary key) 199, 200, or 207:

```
SELECT Name, Occupation FROM People WHERE empID IN (199, 200, 207);
```

A simple form is a comma-separated list of column names. The list can consist of a range of column names.

Counting returned rows

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

Count the number of rows in the users table:

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

Specifying rows returned using LIMIT

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

```
SELECT COUNT(*) FROM big_table;
SELECT COUNT(*) FROM big_table LIMIT 50000;
SELECT COUNT(*) FROM big_table LIMIT 200000;
```

The output of these statements is: 10000, 50000, and 105291

Specifying the table using FROM

The FROM clause specifies the table to query. Optionally, specify a keyspace for the table followed by a period, (.), then the table name. If a keyspace is not specified, the current keyspace will be used.

Count the number of rows in the Migrations table in the system keyspace:

```
SELECT COUNT(*) FROM system.Migrations;
```

Filtering data using WHERE

The WHERE clause specifies which rows to query. The WHERE clause is composed of conditions on the columns that are part of the primary key or are indexed in a secondary index. Use of the primary key in the WHERE clause tells Cassandra to race to the specific node that has the data. Using the equals conditional operators (= or IN) is unrestricted. The term on the left of the operator must be the name of the column, and the term on the right must be the column value to filter on. There are restrictions on other conditional operators.

Supported conditional operators

Cassandra supports these conditional operators: =, >, >=, <, or <=, but not all in certain situations.

- A filter based on a non-equals condition on a *partition key* is supported only if the partitioner is an ordered one.
- WHERE clauses can include a greater-than and less-than comparisons, but for a given partition key, the conditions on the *clustering column* are restricted to the filters that allow Cassandra to select a contiguous ordering of rows.

For example:

```
CREATE TABLE ruling_stewards (
  steward_name text,
  king text,
  reign_start int,
  event text,
  PRIMARY KEY (steward_name, king, reign_start)
);
```

This query constructs a filter that selects data about stewards whose reign started by 2450 and ended before 2500. If king were not a component of the primary key, you would need to create a secondary index on king to use this query:

```
Select * FROM ruling_stewards
  WHERE king = 'Brego'
  AND reign_start >= 2450
  AND reign_start < 2500 ALLOW FILTERING;
```

The output is:

```
 steward_name | king  | reign_start | event
--------------+-------+-------------+--------------------
      Boromir | Brego |        2477 |   Attacks continue
       Cirion | Brego |        2489 | Defeat of Balchoth
```

To allow Cassandra to select a contiguous ordering of rows, you need to include the king component of the primary key in the filter using an equality condition. The ALLOW FILTERING clause, described in the next section, is also required.

ALLOW FILTERING clause

When you attempt a potentially expensive query, such as searching a range of rows, this prompt appears:

```
Bad Request: Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance. If you want
to execute this query despite the performance unpredictability,
use ALLOW FILTERING.
```

To run the query, use the ALLOW FILTERING clause. Imposing a limit using the LIMIT *n* clause is recommended to reduce memory used. For example:

```
Select * FROM ruling_stewards
  WHERE king = 'none'
  AND reign_start >= 1500
  AND reign_start < 3000 LIMIT 10 ALLOW FILTERING;
```

Critically, LIMIT doesn't protect you from the worst liabilities. For instance, what if there are no entries with no king? Then you have to scan the entire list no matter what LIMIT is.

ALLOW FILTERING will probably become less strict as we collect more statistics on our data. For example, if we knew that 90% of entries have no king we would know that finding 10 such entries should be relatively inexpensive.

Paging through unordered results

The *TOKEN function* can be used with a condition operator on the *partition key* column to query. The query selects rows based on the token of their partition key rather than on their value. The token of a key depends on the partitioner in use. The RandomPartitioner and Murmur3Partitioner do not yield a meaningful order.

For example, assuming you have this table defined, the following query shows how to use the TOKEN function:

```
CREATE TABLE periods (
  period_name text,
  event_name text,
  event_date timestamp,
  weak_race text,
  strong_race text,
  PRIMARY KEY (period_name, event_name, event_date)
);

SELECT * FROM periods
  WHERE TOKEN(period_name) > TOKEN('Third Age')
  AND TOKEN(period_name) < TOKEN('Fourth Age');
```

Querying compound primary keys and sorting results

ORDER BY clauses can select a single column only. That column has to be the second column in a compound PRIMARY KEY. This also applies to tables with more than two column components in the primary key. Ordering can be done in ascending or descending order, default ascending, and specified with the ASC or DESC keywords.

For example, set up the playlists table described in the *Compound keys and clustering* section, *insert the example data*, and use this query to get information about a particular playlist, ordered by the id of the song. As of Cassandra 1.2, you do not need to include the ORDER BY column in the select expression.

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
  ORDER BY song_id DESC LIMIT 50;
```

Output

```
 id          | song_id     | album                  | artist          | title
-------------+-------------+------------------------+-----------------+---------------------
62c36092... | a3e64f8f... |          Tres Hombres |          ZZ Top |           La Grange
62c36092... | 8a172618... |          We Must Obey |       Fu Manchu |    Moving in Stereo
62c36092... | 2b09185b... |             Roll Away | Back Door Slam | Outside Woman Blues
62c36092... | 7db1a490... | No One Rides for Free |       Fu Manchu |            Ojo Rojo
```

Or, create an index on playlist artists, and use this query to get titles of Fu Manchu songs on the playlist:

```
CREATE INDEX ON playlists(artist)

SELECT title FROM playlists WHERE artist = 'Fu Manchu';
```

Output

```
 title
------------------
          Ojo Rojo
  Moving in Stereo
```

Querying a collection set, list, or map

When you query a table containing a collection, Cassandra retrieves the collection in its entirety.

To return the set of email belonging to frodo, for example:

```
SELECT user_id, emails
  FROM users
  WHERE user_id = 'frodo';
```

Retrieving the date/time a write occurred

Using *WRITETIME* followed by the name of a column in parentheses returns date/time in microseconds that the column was written to the database.

Retrieve the date/time that a write occurred to the first_name column of the user whose last name is Jones:

```
SELECT WRITETIME (first_name) FROM users
  WHERE last_name = 'Jones';

 writetime(first_name)
-----------------------
 1353010594789000
```

The writetime output in microseconds converts to November 15, 2012 at 12:16:34 GMT-8

TRUNCATE

Removes all data from a table.

Synopsis

```
TRUNCATE keyspace_name.table_name
```

Synopsis legend

Description

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

Example

```
TRUNCATE user_activity;
```

UPDATE

Updates one or more columns in the identified row of a table.

Synopsis

```
UPDATE keyspace_name.table_name
USING TTL seconds
SET assignment , assignment, ...
WHERE row_specification
```

assignment is one of:

```
column_name = value

set_or_list_item «+ | - »«set | list »
map_name + map

collection_column_name [ term ] = value

counter_column_name = counter_column_name «+ | - «integer
```

set is:

```
{ item1, item2, . . . }
```

list is:

```
[ name, value ]
```

map is:

```
{ name : value, name : value, . . . }
```

term is:

```
    [ list_index_position | [ list_value ]
```

row_specification is:

```
    primary key name = key_value
    primary key name IN (key_value ,...)
```

Synopsis legend

Description

An UPDATE writes one or more column values to existing columns in a Cassandra table. No results are returned. A statement begins with the UPDATE keyword followed by a Cassandra table name. To update multiple columns, separate the name-value pairs using commas.

The SET clause specifies the column name-value pairs to update. Separate multiple name-value pairs using commas. If the named column exists, its value is updated. If the column does not exist, use ALTER TABLE to create the new column.

To update a counter column value in a counter table, specify a value to increment or decrement value the current value of the counter column.

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

In an UPDATE statement, you can specify these options:

- *TTL* seconds
- Timestamp

TTL input is in seconds. TTL column values are automatically marked as deleted (with a *tombstone*) after the requested amount of time has expired. TTL marks the inserted values, not the column itself, for expiration. Any subsequent update of the column resets the TTL to the TTL specified in the update. By default, values never expire.

The TIMESTAMP input is in one of the following formats:

- A string in *ISO 8601 format*
- An integer representing microseconds

If not specified, the time (in microseconds) that the write occurred to the column is used.

Each update statement requires a precise set of row keys to be specified using a WHERE clause. You need to specify all keys in a table having compound and clustering columns. For example, update the value of a column in a table having a compound primary key, userid and url:

```
UPDATE excelsior.clicks USING TTL 432000
  SET user_name = 'bob'
  WHERE userid=cfd66ccc-d857-4e90-b1e5-df98a3d40cd6 AND
    url='http://google.com';

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

Examples

Update a column in several rows at once:

```
UPDATE users
  SET state = 'TX'
  WHERE user_uuid
```

```
  IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
    06a8913c-c0d6-477c-937d-6c1b69a95d43,
    bc108776-7cb5-477f-917d-869c12dfffa8);
```

Update several columns in a single row:

```
UPDATE users
  SET name = 'John Smith',
  email = 'jsmith@cassie.com'
  WHERE user_uuid = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

Update the value of a counter column:

```
UPDATE page_views
  USING TIMESTAMP 1355384955054
  SET index.html = 'index.html' + 1
  WHERE url_key = 'www.datastax.com';
```

Using a collection set

To add an element to a set, use the UPDATE command and the addition (+) operator:

```
UPDATE users
  SET emails = emails + {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

To remove an element from a set, use the subtraction (-) operator.

```
UPDATE users
  SET emails = emails - {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

To remove all elements from a set, you can use the UPDATE statement:

```
UPDATE users SET emails = {} WHERE user_id = 'frodo';
```

Using a collection map

To set or replace map data, you can use the UPDATE command. Enclose the timestamp and text values in map collection syntax: strings in curly brackets, separated by a colon.

```
UPDATE users
  SET todo =
  { '2012-9-24' : 'enter mordor',
  '2012-10-2 12:00' : 'throw ring into mount doom' }
  WHERE user_id = 'frodo';
```

You can also update or set a specific element using the UPDATE command. For example, update a map named todo to insert a reminder, 'die' on October 2 for user frodo.

```
UPDATE users SET todo['2012-10-2 12:10'] = 'die'
  WHERE user_id = 'frodo';
```

You can set the a *TTL* for each map element:

```
UPDATE users USING TTL <ttl value>
 SET todo['2012-10-1'] = 'find water' WHERE user_id = 'frodo';
```

Using a collection list

To insert values into the list.

```
UPDATE users
  SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';
```

To prepend an element to the list, enclose it in square brackets, and use the addition (+) operator:

```
UPDATE users
  SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';
```

To append an element to the list, switch the order of the new element data and the list name in the UPDATE command:

```
UPDATE users
  SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';
```

To add an element at a particular position, use the list index position in square brackets:

```
UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';
```

To remove all elements having a particular value, use the UPDATE command, the subtraction operator (-), and the list value in square brackets:

```
UPDATE users
  SET top_places = top_places - ['riddermark'] WHERE user_id = 'frodo';
```

USE

Connects the current client session to a keyspace.

Synopsis

```
USE keyspace_name | keyspace_name
```

Description

A USE statement identifies the keyspace that contains the tables to query for the current client session. All subsequent operations on tables 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.

To use a case-sensitive keyspace, enclose the keyspace name in double quotation marks.

Example

```
USE PortfolioDemo;
```

Continuing with the example in *Checking created keyspaces*:

```
USE "Excalibur";
```

ASSUME

Treats a column name or value as a specified type, even if that type information is not specified in the table's metadata.

Synopsis

```
ASSUME keyspace_name.table_name
storage_type_definition , storage_type_definition ..., ...
```

storage_type_definition is:

```
(column_name) VALUES ARE datatype
| NAMES ARE datatype
| VALUES ARE datatype
```

Synopsis legend

Description

ASSUME treats all values in the given column in the given table as being of the specified type when the storage_type_definition is:

```
(column_name) VALUES ARE datatype
```

This overrides any other information about the type of a value.

ASSUME treats all column names in the given table as being of the given type when the storage_type_definition is:

```
NAMES ARE <type>
```

ASSUME treats all column values in the given table as being of the given type unless overridden by a column-specific ASSUME or column-specific metadata in the table's definition.

Assigning multiple data type overrides

Assign multiple overrides at once for the same table by separating storage_type_definitions with commas:

```
ASSUME ks.table NAMES ARE uuid,
  VALUES ARE int, (col)
  VALUES ARE ascii
```

Examples

```
ASSUME users NAMES ARE text, VALUES are text;
```

```
ASSUME users(user_id) VALUES are uuid;
```

CAPTURE

Captures command output and appends it to a file.

Synopsis

```
CAPTURE '<file>' | OFF
```

Synopsis legend

Description

To start capturing the output of a query, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to $HOME:

Example

```
CAPTURE '~/mydir/myfile.txt';
```

Output is not shown on the console while it is captured. Only query result output is captured. Errors and output from cqlsh-only commands still appear.

To stop capturing output and return to normal display of output, use CAPTURE OFF.

To determine the current capture state, use CAPTURE with no arguments.

CONSISTENCY

Shows the current *consistency level*, or given a level, sets it.

Synopsis

```
CONSISTENCY level
```

Synopsis legend

Description

Providing an argument to the CONSISTENCY command overrides the default consistency level of ONE, setting the consistency level for future requests. Valid values are: ANY, ONE, TWO, THREE, QUORUM, ALL, LOCAL_QUORUM and EACH_QUORUM. See *About data consistency* for more information about these settings.

Providing no argument shows the current consistency level.

Example

```
CONSISTENCY
```

If you haven't changed the default, the output of the CONSISTENCY command with no arguments is: `Current consistency level is ONE.`

COPY

Imports and exports CSV (comma-separated values) data to and from Cassandra 1.1.3 and higher.

Synopsis

```
COPY table_name ( column, ...)
FROM ( 'file_name' | STDIN )
WITH option = 'value' AND ...

COPY table_name ( column , ... )
TO ( 'file_name' | STDOUT )
WITH option = 'value' AND ...
```

Synopsis legend

Description

Using the COPY options in a WITH clause, you can change the CSV format. This table describes these options:

COPY Options	Default Value	Use To
DELIMITER	comma (,)	Set the character that separates fields in the file.
QUOTE	quotation mark(")	Set the character that encloses field values.[1]
ESCAPE	backslash (\)	Set the character that escapes literal uses of the QUOTE character.
HEADER	false	Set true to indicate that first row of the file is a header.
ENCODING	UTF8	Set the COPY TO command to output unicode strings.
NULL	an empty string	Set the COPY TO command to represent the absence of a value.

[1] Applies only to fields having newline characters.

The ENCODING and NULL options cannot be used in the COPY FROM command.

This table shows that, by default, Cassandra expects the CSV data to consist of fields separated by commas (,), records separated by line separators (a newline, \r\n), and field values enclosed in double-quotation marks (""). Also, to avoid ambiguity, escape a literal double-quotation mark using a backslash inside a string enclosed in double-quotation marks ("\""). By default, Cassandra does not expect the CSV file to have a header record on the first line that consists of the column names. COPY TO includes the header in the output if HEADER=true. COPY FROM ignores the first line if HEADER=true.

COPY FROM a CSV file

By default, when you use the COPY FROM command, Cassandra expects every row in the CSV input to contain the same number of columns. The number of columns in the CSV input is the same as the number of columns in the Cassandra table metadata. Cassandra assigns fields in the respective order. To apply your input data to a particular set of columns, specify the column names in parentheses after the table name.

COPY FROM is intended for importing small datasets (a few million rows or less) into Cassandra. For importing larger datasets, use *Cassandra bulk loader* or the *sstable2json/json2sstable2 utility* `.

COPY TO a CSV file

For example, assume you have the following table in CQL:

```
cqlsh> SELECT * FROM test.airplanes;

 name          | mach | manufacturer | year
---------------+------+--------------+------
 P38-Lightning |  0.7 |     Lockheed | 1937
```

After inserting data into the table, you can copy the data to a CSV file in another order by specifying the column names in parentheses after the table name:

```
COPY airplanes
(name, mach, year, manufacturer)
 TO 'temp.csv'
```

Specifying the source or destination files

Specify the source file of the CSV input or the destination file of the CSV output by a file path. Alternatively, you can use the STDIN or STDOUT keywords to import from standard input and export to standard output. When using stdin, signal

the end of the CSV data with a backslash and period (“\.“) on a separate line. If the data is being imported into a table that already contains data, COPY FROM does not truncate the table beforehand.

You can copy only a partial set of columns. Specify the entire set or a subset of column names in parentheses after the table name in the order you want to import or export them. By default, when you use the COPY TO command, Cassandra copies data to the CSV file in the order defined in the Cassandra table metadata. In version 1.1.6 and later, you can also omit listing the column names when you want to import or export all the columns in the order they appear in the source table or CSV file.

Examples

Copy a table to a CSV file

1. Using CQL 3, create a table named airplanes and copy it to a CSV file.

```
CREATE KEYSPACE test
  WITH REPLICATION = {'class' : 'SimpleStrategy'
  'replication_factor' : 3 };

USE test;

CREATE TABLE airplanes (
  name text PRIMARY KEY,
  manufacturer ascii,
  year int,
  mach float
);

INSERT INTO airplanes
  (name, manufacturer, year, mach)
  VALUES ('P38-Lightning', 'Lockheed', 1937, '.7');

COPY airplanes (name, manufacturer, year, mach) TO 'temp.csv';
1 rows exported in 0.004 seconds.
```

2. Clear the data from the airplanes table and import the data from the temp.csv file

```
TRUNCATE airplanes;

COPY airplanes (name, manufacturer, year, mach) FROM 'temp.csv';
1 rows imported in 0.087 seconds.
```

Copy data from standard input to a table

1. Enter data directly during an interactive cqlsh session, using the COPY command defaults.

```
COPY airplanes (name, manufacturer, year, mach) FROM STDIN;
```

2. At the `[copy]` prompt, enter the following data:

```
"F-14D Super Tomcat", Grumman,"1987", "2.34"
"MiG-23 Flogger", Russian-made, "1964", "2.35"
"Su-27 Flanker", U.S.S.R.,"1981", "2.35"
\.
3 rows imported in 55.204 seconds.
```

3. Query the airplanes table to see data imported from STDIN:

```
SELECT * FROM airplanes;
```

Output

```
 name               | manufacturer | year | mach
--------------------+------+--------------+-------------
 F-14D Super Tomcat |      Grumman | 1987 | 2.35
      P38-Lightning |     Lockheed | 1937 | 0.7
      Su-27 Flanker |     U.S.S.R. | 1981 | 2.35
     MiG-23 Flogger | Russian-made | 1967 | 2.35
```

DESCRIBE

Provides information about the connected Cassandra cluster, or about the data objects stored in the cluster.

Synopsis

```
DESCRIBE CLUSTER | SCHEMA
| KEYSPACES
| KEYSPACE keyspace_name
| TABLES
| TABLE table_name
```

Synopsis legend

Description

The DESCRIBE or DESC command outputs information about the connected Cassandra cluster, or about the data stored on it. To *query the system tables* directly, use SELECT.

The keyspace and table name arguments are case-sensitive and need to match the the upper or lowercase names stored internally. Use the DESCRIBE commands to list objects by their internal names.

DESCRIBE functions in the following ways:

- DESCRIBE CLUSTER

  Output is the information about the connected Cassandra cluster, such as the cluster name, and the partitioner and snitch in use. When you are connected to a non-system keyspace, this command also shows endpoint-range ownership information for the Cassandra ring.

- DESCRIBE SCHEMA

  Output is a list of CQL commands that could be used to recreate the entire schema. Works as though DESCRIBE KEYSPACE <k> was invoked for each keyspace k.

- DESCRIBE KEYSPACES

  Output is a list of all keyspace names.

- DESCRIBE KEYSPACE *keyspace_name*

  Output is a list of CQL commands that could be used to recreate the given keyspace, and the tables in it. In some cases, as the CQL interface matures, there will be some metadata about a keyspace that is not representable with CQL. That metadata will not be shown.

  The <keyspacename> argument can be omitted when using a non-system keyspace; in that case, the current keyspace is described.

- DESCRIBE TABLES

  Output is a list of the names of all tables in the current keyspace, or in all keyspaces if there is no current keyspace.

- DESCRIBE TABLE *table_name*

  Output is a list of CQL commands that could be used to recreate the given table. In some cases, there might be table metadata that is not representable and it is not be shown.

Examples

```
DESCRIBE CLUSTER;

DESCRIBE KEYSPACES;

DESCRIBE KEYSPACE PortfolioDemo;

DESCRIBE TABLES;

DESCRIBE TABLE Stocks;
```

EXIT

Terminates cqlsh.

Synopsis

```
EXIT | QUIT
```

SHOW

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

Synopsis

```
SHOW VERSION
| HOST
| ASSUMPTIONS
```

Description

A SHOW command displays this information about the current cqlsh client session:

- The version and build number of the connected Cassandra instance, as well as the CQL mode for cqlsh and the Thrift protocol used by the connected Cassandra instance.
- The host information of the Cassandra node that the cqlsh session is currently connected to.
- The data type assumptions for the current cqlsh session as specified by the ASSUME command.

Examples

```
SHOW VERSION;

SHOW HOST;

SHOW ASSUMPTIONS;
```

SOURCE

Executes a file containing CQL statements.

Synopsis

```
SOURCE 'file'
```

Description

To execute the contents of a file, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to $HOME:

Example

```
SOURCE '~/mydir/myfile.txt';
```

The output for each statement, if there is any, appears in turn, including any error messages. Errors do not abort execution of the file.

Alternatively, use the `--file` option to execute a file while starting CQL.

TRACING

Enables or disables request tracing.

Synopsis

```
TRACING ON | OFF
```

Synopsis legend

Description

To turn tracing read/write requests on or off, use the TRACING command. After turning on tracing, database activity creates output that can help you understand Cassandra internal operations and troubleshoot performance problems.

For 24 hours, Cassandra saves the tracing information in the tables, which are in the system_traces keyspace:

```
CREATE TABLE sessions (
  session_id uuid PRIMARY KEY,
  coordinator inet,
  duration int,
  parameters map<text, text>,
  request text,
  started_at timestamp
```

```
);

CREATE TABLE events (
 session_id uuid,
 event_id timeuuid,
 activity text,
 source inet,
 source_elapsed int,
 thread text,
 PRIMARY KEY (session_id, event_id)
);
```

To keep tracing information, copy the data in sessions and event tables to another location.

Tracing a write request

This example shows tracing activity on a 3-node cluster created by ccm on Mac OSX. Using a keyspace that has a replication factor of 3 and an employee table similar to the one in *Creating a table*, the tracing shows:

- The coordinator identifies the target nodes for replication of the row.
- Writes the row to the commitlog and memtable.
- Confirms completion of the request.

```
TRACING ON;

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

Cassandra provides a description of each step it takes to satisfy the request, the names of nodes that are affected, the time for each step, and the total time for the request.

```
 Tracing session: 740b9c10-3506-11e2-0000-fe8ebeead9ff

 activity                                | timestamp    | source    | source_elapsed
-----------------------------------------+--------------+-----------+----------------
                  execute_cql3_query | 16:41:00,754 | 127.0.0.1 |              0
                   Parsing statement | 16:41:00,754 | 127.0.0.1 |             48
                  Peparing statement | 16:41:00,755 | 127.0.0.1 |            658
   Determining replicas for mutation | 16:41:00,755 | 127.0.0.1 |            979
    Message received from /127.0.0.1 | 16:41:00,756 | 127.0.0.3 |             37
      Acquiring switchLock read lock | 16:41:00,756 | 127.0.0.1 |           1848
       Sending message to /127.0.0.3 | 16:41:00,756 | 127.0.0.1 |           1853
              Appending to commitlog | 16:41:00,756 | 127.0.0.1 |           1891
       Sending message to /127.0.0.2 | 16:41:00,756 | 127.0.0.1 |           1911
              Adding to emp memtable | 16:41:00,756 | 127.0.0.1 |           1997
      Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.3 |            395
    Message received from /127.0.0.1 | 16:41:00,757 | 127.0.0.2 |             42
              Appending to commitlog | 16:41:00,757 | 127.0.0.3 |            432
      Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.2 |            168
              Adding to emp memtable | 16:41:00,757 | 127.0.0.3 |            522
              Appending to commitlog | 16:41:00,757 | 127.0.0.2 |            211
              Adding to emp memtable | 16:41:00,757 | 127.0.0.2 |            359
    Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |           1282
    Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |           1024
       Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |           1469
       Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |           1179
```

```
   Message received from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |          10966
   Message received from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |          10966
Processing response from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |          11063
Processing response from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |          11066
                 Request complete | 16:41:00,765 | 127.0.0.1 |          11139
```

Tracing a sequential scan

Due to the log structured design of Cassandra, a single row is spread across multiple SSTables. Reading one row involves reading pieces from multiple SSTables, as shown by this trace of a request to read the employee table, which was pre-loaded with 10 rows of data.

```
SELECT * FROM emp;
```

Output is:

```
 empid | deptid | first_name | last_name
-------+--------+------------+-----------
   110 |     16 |      naoko |     murai
   105 |     15 |       john |     smith
   111 |     15 |       jane |     thath
   113 |     15 |       lisa |     amato
   112 |     20 |       mike |     burns
   107 |     15 |     sukhit |       ran
   108 |     16 |        tom |     brown
   109 |     18 |        ann |     green
   104 |     15 |       jane |     smith
   106 |     15 |        bob |     jones
```

The tracing output of this read request looks something like this (a few rows have been truncated to fit on this page):

```
 Tracing session: bf5163e0-350f-11e2-0000-fe8ebeead9ff

 activity                                                | timestamp    | source    | source_elapsed
--------------------------------------------------------+--------------+-----------+----------------
                              execute_cql3_query | 17:47:32,511 | 127.0.0.1 |              0
                               Parsing statement | 17:47:32,511 | 127.0.0.1 |             47
                             Preparing statement | 17:47:32,511 | 127.0.0.1 |            249
                Determining replicas to query | 17:47:32,511 | 127.0.0.1 |            383
                Sending message to /127.0.0.2 | 17:47:32,512 | 127.0.0.1 |            883
              Message received from /127.0.0.1 | 17:47:32,512 | 127.0.0.2 |             33
  Executing seq scan across 0 sstables for . . . | 17:47:32,513 | 127.0.0.2 |            670
             Read 1 live cells and 0 tombstoned | 17:47:32,513 | 127.0.0.2 |            964
             Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2 |           1268
             Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2 |           1502
             Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2 |           1673
                   Scanned 4 rows and matched 4 | 17:47:32,514 | 127.0.0.2 |           1721
              Enqueuing response to /127.0.0.1 | 17:47:32,514 | 127.0.0.2 |           1742
                 Sending message to /127.0.0.1 | 17:47:32,514 | 127.0.0.2 |           1852
              Message received from /127.0.0.2 | 17:47:32,515 | 127.0.0.1 |           3776
            Processing response from /127.0.0.2 | 17:47:32,515 | 127.0.0.1 |           3900
                 Sending message to /127.0.0.2 | 17:47:32,665 | 127.0.0.1 |         153535
              Message received from /127.0.0.1 | 17:47:32,665 | 127.0.0.2 |             44
  Executing seq scan across 0 sstables for . . . | 17:47:32,666 | 127.0.0.2 |           1068
             Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2 |           1454
             Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2 |           1640
                   Scanned 2 rows and matched 2 | 17:47:32,667 | 127.0.0.2 |           1694
              Enqueuing response to /127.0.0.1 | 17:47:32,667 | 127.0.0.2 |           1722
```

SOURCE

```
                  Sending message to /127.0.0.1 | 17:47:32,667 | 127.0.0.2 |           1825
                Message received from /127.0.0.2 | 17:47:32,668 | 127.0.0.1 |         156454
             Processing response from /127.0.0.2 | 17:47:32,668 | 127.0.0.1 |         156610
   Executing seq scan across 0 sstables for . . . | 17:47:32,669 | 127.0.0.1 |         157387
              Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         157729
              Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         157904
              Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         158054
              Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         158217
                    Scanned 4 rows and matched 4 | 17:47:32,669 | 127.0.0.1 |         158270
                                Request complete | 17:47:32,670 | 127.0.0.1 |         159525
```

The sequential scan across the cluster shows:

- The first scan found 4 rows on node 2.
- The second scan found 2 more rows found on node 2.
- The third scan found the 4 rows on node 1.

For examples of tracing indexed queries and diagnosing performance problems using tracing, see Request tracing in Cassandra 1.2.

Managing and accessing data

This section includes a description of the journey that data written to and read from the database takes, the hinted handoff feature, areas of ACID conformance and non-conformance, and data consistency. Also covered are the client utilities and application programming interfaces (APIs) for developing applications for data storage and retrieval.

About writes

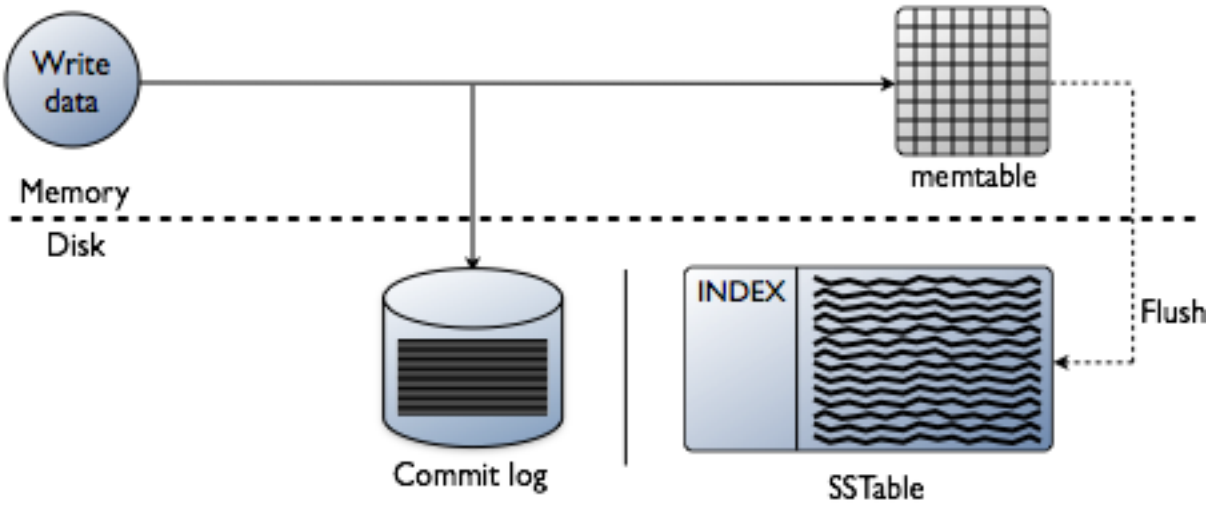
Cassandra delivers high availability for writing through its data replication strategy. Cassandra duplicates data on multiple peer nodes to ensure reliability and fault tolerance. Relational databases, on the other hand, typically structure tables to keep data duplication at a minimum. The relational database server has to do additional work to ensure data integrity across the tables. In Cassandra, maintaining integrity between related tables is not an issue. Cassandra tables are not related. Usually, Cassandra performs better on writes than relational databases.

About the write path

When a write occurs, Cassandra stores the data in a structure in memory, the memtable, and also appends writes to the commit log on disk, providing *configurable durability*.

The commit log receives every write made to a Cassandra node, and these durable writes survive permanently even after hardware failure.

The more a table is used, the larger its memtable needs to be. Cassandra can dynamically allocate the right amount of memory for the memtable or you can manage the amount of memory being utilized yourself. When memtable contents exceed a *configurable threshold*, the memtable data, which includes secondary indexes, is put in a queue to be flushed to disk. You can configure the length of the queue by changing memtable_flush_queue_size in the cassandra.yaml. If the data to be flushed exceeds the queue size, Cassandra blocks writes. The memtable data is flushed to *SSTables* on disk using sequential I/O. Data in the commit log is purged after its corresponding data in the memtable is flushed to the SSTable.



Memtables and SSTables are maintained per table. SSTables are immutable, not written to again after the memtable is flushed. Consequently, a row is typically stored across multiple SSTable files.

For each SSTable, Cassandra creates these in-memory structures:

- Primary index - A list of row keys and the start position of rows in the data file.
- Index summary - A subset of the primary index. By default 1 row key out of every 128 is sampled.

How Cassandra stores data

In the memtable, data is organized in sorted order by row key.

For efficiency, Cassandra does not repeat the names of the columns in memory or in the SSTable. For example, the following writes occur:

```
write (k1, c1:v1)
write (k2, c1:v1 C2:v2)
write (k1, c1:v4 c3:v3 c2:v2)
```

In the memtable, Cassandra stores this data after receiving the writes:

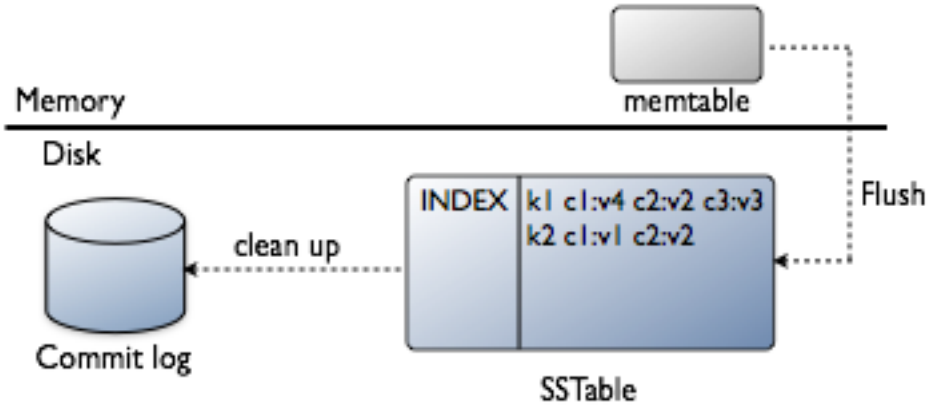
```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

In the commit log on disk, Cassandra stores this data after receiving the writes:

```
k1, c1:v1
k2, c1:v1 C2:v2
k1, c1:v4 c3:v3 c2:v2
```

In the SSTable on disk, Cassandra stores this data after flushing the memtable:

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```



About secondary indexes updates

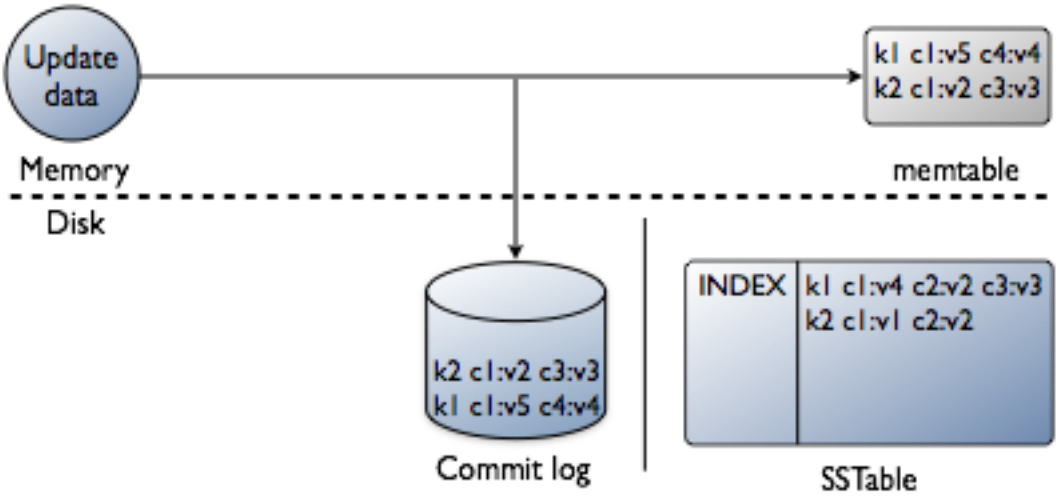
To update secondary indexes Cassandra appends data to the commit log, updates the memtable, and updates the secondary indexes. Writing to a table having a secondary index involves more work than writing to a table without a secondary index, but the update process has been improved in Cassandra 1.2. The need for a synchronization lock to prevent concurrency issues for heavy insert loads has been removed.

When a column is updated, the secondary index is updated. If the old column value was still in the memtable, which typically occurs when updating a small set of rows repeatedly, Cassandra removes the index entry; otherwise, the old entry remains to be purged by compaction. If a read sees a stale index entry before compaction purges it, the reader thread invalidates it.

As with relational databases, keeping indexes up to date is not free, so unnecessary indexes should be avoided.

About inserts and updates

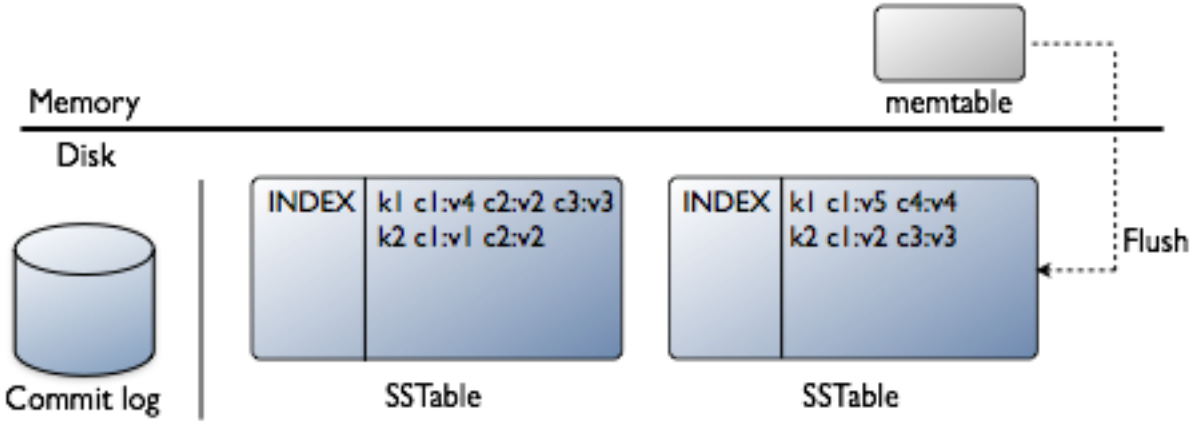
As updates come in, Cassandra does not overwrite the rows in place, but instead groups updates in the *memtable*.



Any number of columns may be inserted at the same time. When inserting or updating columns in a table, 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 table. However, unlike inserting a primary key, inserting a duplicate row key does not result in a primary key constraint violation.

The write path of an update

Inserting a duplicate row key is treated as an *upsert*. Eventually, the updates are streamed to disk using sequential I/O and stored in a new SSTable.



Columns are overwritten only 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), for example.

About deletes

Cassandra deletes data in a different way from a traditional, relational database. A relational database might spend time scanning through data looking for expired data and throwing it away or an administrator might have to partition expired data by month, for example, to clear it out faster. In Cassandra, you do not have to manually remove expired data. Two facts about deleted Cassandra data to keep in mind are:

- Cassandra does not immediately remove deleted data from disk.
- A deleted column can reappear if you do not run node repair routinely.

After an SSTable is written, it is immutable (the file is not updated by further DML operations). Consequently, a deleted column is not removed immediately. Instead 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 table). When the

grace period expires, the *compaction process* permanently deletes the column.

Marking a deleted column with a tombstone signals Cassandra to retry sending a delete request to a replica that was down at the time of delete. If the replica comes back up within the grace period of time, it eventually receives the delete request. However, if a node is down longer than the grace period, 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).

Managing data on disk

Cassandra uses a storage structure similar to a Log-Structured Merge Tree, unlike a typical relational database that uses a B-Tree. The storage engine writes sequentially to disk in append mode and stores data continguously. Operations are parallel within cross nodes and within an individual machine. Because Cassandra does not use a B-tree, concurrency control is unnecessary. Nothing needs to be updated when writing.

Cassandra accommodates modern solid-state disks (SSDs) extremely well. Inexpensive, consumer SSDs are fine for use with Cassandra because Cassandra minimizes wear and tear on an SSD. The disk I/O performed by Cassandra is minimal.

Throughput and latency

Throughput and latency are key factors affecting Cassandra performance in managing data on disk.

- Throughput is operations per second.
- Latency is the round trip time to complete a request.

When database operations are serial, throughput and latency are interchangeable. Cassandra operations are performed in parallel, so throughput and latency are independent. Unlike most databases, Cassandra achieves excellent throughput and latency.

Writes are very efficient in Cassandra and very inefficient in storage engines that scatter random writes around while making in-place updates. When you're doing many random writes of small amounts of data, Cassandra reads in the SSD sector. No random seeking occurs as it does in relational databases. Cassandra's log-structured design obviates the need for disk seeks. As database updates are received, Cassandra does not overwrite rows in place. In-place updates would require doing random I/O. Cassandra updates the bytes and rewrites the entire sector back out instead of modifying the data on disk. Eliminating on-disk data modification and erase-block cycles prolongs the life of the SSD and saves time: one or two milliseconds.

Cassandra does not lock the fast write request path that would negatively affect throughput. Because there is no modification of data on disk, locking for concurrency control of data on disk is unnecessary. THe operational design integrates nicely with the operating system page cache. Because Cassandra does not modify the data, dirty pages that would have to be flushed are not even generated.

Using SSDs instead of rotational disks is necessary for achieving low latency. Cassandra runs the same code on every node and has no master node and no single point of failure, which also helps achieve high throughput.

Separate table directories

Cassandra 1.1 and later releases provide fine-grained control of table storage on disk, writing tables to disk using separate table directories within each keyspace directory. Data files are stored using this directory and file naming format:

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

The new file name format includes the keyspace name to distinguish which keyspace and table the file contains when streaming or bulk loading data. Cassandra creates a subdirectory for each table, which allows you to symlink a table to a chosen physical drive or data volume. This provides the capability to move very active tables to faster media, such as SSD’s for better performance, and also divvy up tables across all attached storage devices for better I/O balance at the storage layer.

About hinted handoff writes

Hinted handoff is an optional feature of Cassandra. Hinted handoff has two uses:

- To reduce the time to restore a failed node to consistency after the failed node returns to the cluster
- To ensure 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 Cassandra knows a replica is down at the time the write occurs, a corresponding live replica stores a hint. The hint consists of:

- The location of the replica that is down
- The row key that requires a replay
- The actual data being written

If all replicas storing the affected row key are down, a write can succeed if write operation uses a *write consistency* level of ANY. Under this case, the hint and written data are stored on the coordinator node, but are not available to reads until the hint is written 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 three hours after a replica fails because if the replica is down longer than that, it is likely permanently dead. In this case, you should run a *repair* to re-replicate the data before the failure occurred. You can configure this time using the *max_hint_window_in_ms* property in the `cassandra.yaml` file.

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

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

Note

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

About reads

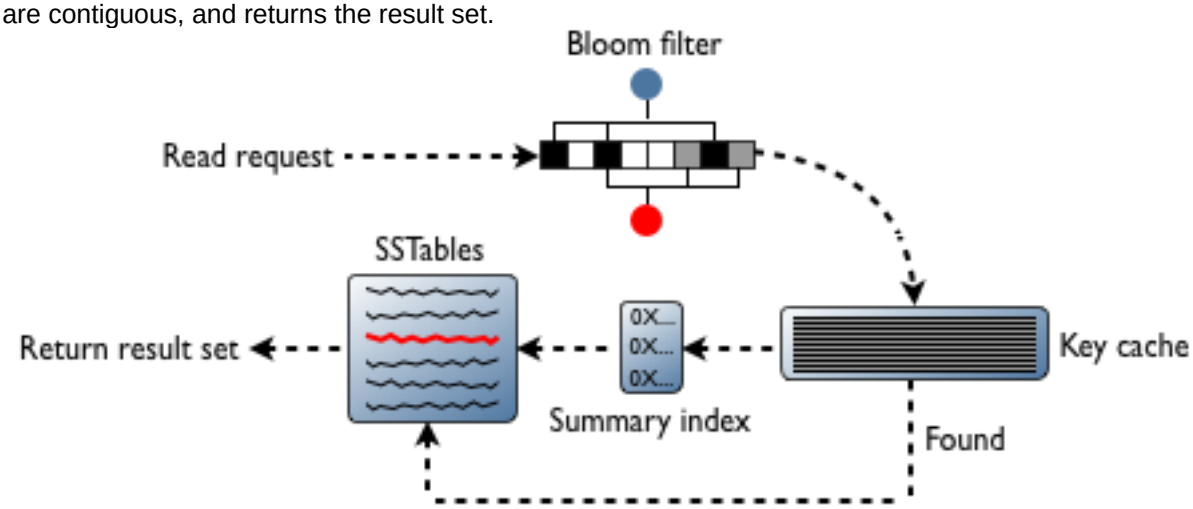
Cassandra performs random reads from SSD in parallel with extremely low latency, unlike most databases. Rotational disks are not recommended.

Cassandra reads, as well as writes, data by primary key, eliminating complex queries required by a relational database. First, Cassandra checks the *Bloom filter*. Each SSTable has a Bloom filter associated with it that checks if any data for the requested row exists in the SSTable before doing any disk I/O.

Next, Cassandra checks the global *key cache*. If the requested data is not in the key cache, Cassandra performs a binary search of the *index summary* to find a row. By default, 1 row key out of every 128 is sampled from the primary index to create the index summary. You configure sample frequency by changing the index_interval property in the

cassandra.yaml file. You can probably increase the index_interval to 512 without seeing degradation.

Finally, Cassandra performs a single seek and a sequential read of columns (a range read) in the SSTable if the columns are contiguous, and returns the result set.



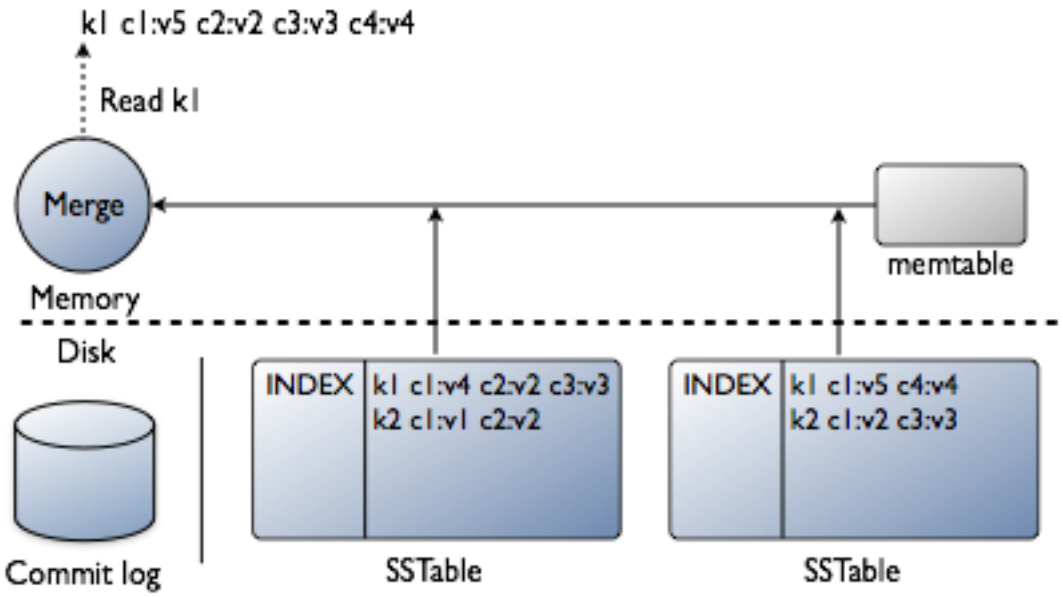
Disk reads take place on a block level. One disk read of the index block corresponds to the closest sampled entry. Cassandra reads a row, plus some selection of columns or a range of columns. This process, in conjunction with fast lookup of data through *primary and secondary indexes* makes Cassandra is very performant on reads when compared to other storage systems, even for read-heavy workloads. Faster startup/bootup times for each node in a cluster are realized through the efficient sampling and loading of SSTable indexes into memory caches. The SSTable index load time is improved dramatically by eliminating the need to go through the whole primary index.

Reading a clustered row

Using a CQL 3 schema, Cassandra's storage engine uses compound columns to store clustered rows. All the logical rows with the same partition key get stored as a single, physical row. Within a partition, all rows are not equally expensive to query. The very beginning of the partition -- the first rows, clustered by your key definition -- is slightly less expensive to query because there is no need to consult the partition-level index. For more information about clustered rows, see *Compound keys and clustering*.

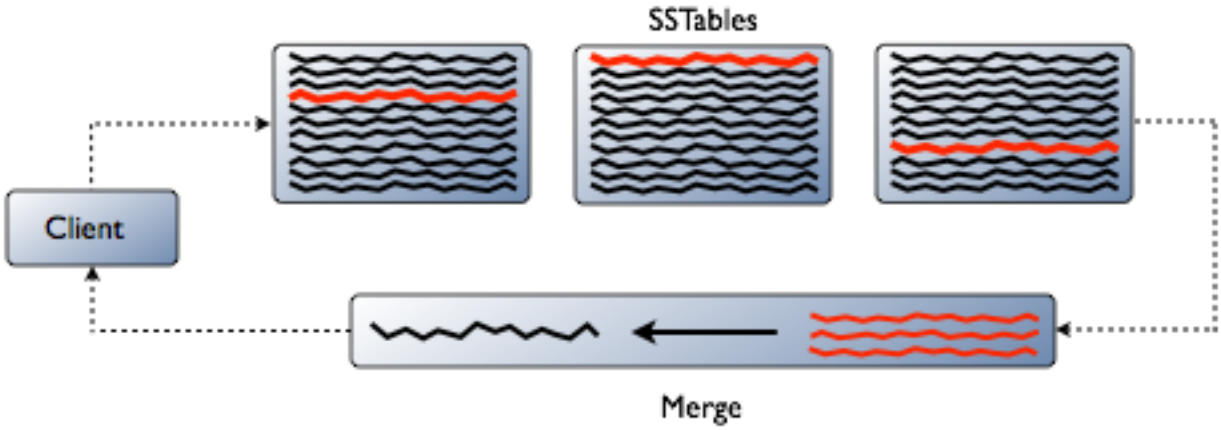
About the read path

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. This diagram depicts the read path of a read request, continuing the example in *The write path of an update*:



For example, you have a row of user data and need to update the user email address. Cassandra doesn't rewrite the entire row into a new data file, but just puts new email address in the new data file. The user name and password are still in the old data file.

The red lines in the SSTables in this diagram are fragments of a row that Cassandra needs to combine to give the user the requested results. Cassandra caches the merged value, not the raw row fragments. That saves some CPU and disk I/O.



The row cache is a write-through cache, so if you have a cached row and you update that row, it will be updated in the cache and you still won't have to merge that again.

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

How write patterns affect reads

The type of *compaction strategy* Cassandra performs on your data is configurable and can significantly affect read preformance. Using the SizeTieredCompactionStrategy tends to cause data fragmentation when rows are frequently updated. The LeveledCompactionStrategy (LCS) was designed to prevent fragmentation under this condition. For more information about LCS, see the article Leveled Compaction in Apache Cassandra.

How the row cache affects reads

Typical of 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*. For rows that are accessed frequently, Cassandra has a built-in key cache and an optional row cache.

How compaction and compression affect reads

To prevent read speed from deteriorating, *compaction* runs in the background without random I/O. *Compression* maximizes the storage capacity of nodes and reduces disk I/O, particularly for read-dominated workloads.

When I/O activity starts to increase in Cassandra due to increased read load, typically the remedy is to add more nodes to the cluster. Cassandra avoids decompressing data in the middle of reading a data file, making its compression application-transparent.

About transactions and concurrency control

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

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

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

Atomicity in Cassandra

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

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

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

Tunable consistency in Cassandra

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

Isolation in Cassandra

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

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

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

Durability in Cassandra

Writes in Cassandra are durable. All writes to a replica node are recorded both in memory and in a commit log on disk 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. In addition to the local durability (data immediately written to disk), the replication of data on other nodes strengthens durability.

About data consistency

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

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

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

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

Tunable consistency for client requests

Consistency levels in Cassandra can be set at to manage response time versus data accuracy. You can set consistency on a cluster, data center, or individual I/O operation basis. Very strong or eventual data consistency among all participating nodes can be set globally and also controlled on a per-operation basis (for example insert or update) using Cassandra's drivers and client libraries.

About write consistency

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

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

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

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

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

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

About read consistency

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

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

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

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

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

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

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

Note

LOCAL_QUORUM and EACH_QUORUM are designed for use in multiple data center clusters using a rack-aware replica placement strategy (such as `NetworkTopologyStrategy`) and a properly configured snitch. These consistency levels will fail when using `SimpleStrategy`.

Choosing client consistency levels

Choosing a consistency level 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 always reflects the most recent write by using the following formula:

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

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

Consistency levels for multiple data center clusters

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.

Setting client consistency levels

You can use a new cqlsh command, *CONSISTENCY*, to set the consistency level for the keyspace. The WITH CONSISTENCY clause has been removed from CQL 3 commands in the release version of CQL 3. Programmatically, set the consistency level at the driver level. For example, call execute_cql3_query with the required binary query, the compression settings, and consistency level.

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*
- *Hinted Handoff*
- *Anti-Entropy Node Repair*

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 the 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.

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

In Cassandra 1.1, CQL became the primary interface into the DBMS. The CQL mode was promoted to CQL 3, although CQL 2 remained the default because CQL 3 is not backward compatible. The most significant enhancement of CQL 3 was support for *compound and clustering columns*.

In Cassandra 1.2, CQL 3 became the default interface into the DBMS.

The Python driver includes a command-line interface, cql.sh for using *cqlsh*.

Other high-level clients

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

A list of other available clients are 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.

Python: Pycassa client API

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

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.

Configuration

Like any modern server-based software, Cassandra has a number of configuration options to tune the system toward 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 require some modifications of the default configuration. For information about JVM configuration, see *Tuning Java resources*.

Node and cluster configuration (cassandra.yaml)

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

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

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

Note

** Some default values are set at the class level and may be missing or commented out in the cassandra.yaml file. Additionally, values in commented out options may not match the default value: they are the recommended value when changing from the default.

Option	Option
authenticator	multithreaded_compaction
authorizer	native_transport_max_threads
auto_bootstrap	native_transport_port
auto_snapshot	num_tokens
broadcast_address	partitioner
client_encryption_options	permissions_validity_in_ms
cluster_name	phi_convict_threshold
column_index_size_in_kb	range_request_timeout_in_ms
commitlog_directory	read_request_timeout_in_ms
commitlog_segment_size_in_mb	reduce_cache_capacity_to
commitlog_sync	reduce_cache_sizes_at
commitlog_total_space_in_mb	request_scheduler_id
compaction_preheat_key_cache	request_scheduler_options
compaction_throughput_mb_per_sec	request_scheduler
concurrent_compactors	request_timeout_in_ms
concurrent_reads	request_timeout_in_ms
concurrent_writes	row_cache_keys_to_save
cross_node_timeout	row_cache_provider
data_file_directories	row_cache_save_period

disk_failure_policy	row_cache_size_in_mb
dynamic_snitch_badness_threshold	rpc_address
dynamic_snitch_reset_interval_in_ms	rpc_keepalive
dynamic_snitch_update_interval_in_ms	rpc_max_threads
endpoint_snitch	rpc_min_threads
flush_largest_memtables_at	rpc_port
hinted_handoff_enabled	rpc_recv_buff_size_in_bytes
hinted_handoff_throttle_in_kb	rpc_send_buff_size_in_bytes
in_memory_compaction_limit_in_mb	rpc_server_type
incremental_backups	saved_caches_directory
index_interval	seed_provider
initial_token	server_encryption_options
inter_dc_tcp_nodelay	snapshot_before_compaction
internode_compression	ssl_storage_port
internode_recv_buff_size_in_bytes	start_native_transport
internode_send_buff_size_in_bytes	start_rpc
key_cache_keys_to_save	storage_port
key_cache_save_period	stream_throughput_outbound_megabits_per_sec
key_cache_size_in_mb	streaming_socket_timeout_in_ms
listen_address	thrift_framed_transport_size_in_mb
max_hint_window_in_ms	thrift_max_message_length_in_mb
max_hints_delivery_threads	trickle_fsync
memtable_flush_queue_size	truncate_request_timeout_in_ms
memtable_total_space_in_mb	write_request_timeout_in_ms

Node and cluster initialization properties

The following properties are used to initialize a new cluster or when introducing a new node to an established cluster. They control how a node is configured within a cluster, including inter-node communication, data partitioning, and replica placement. It is recommended that you carefully evaluate your requirements and make any changes before starting a node for the first time.

auto_bootstrap

(Default: true) This setting has been removed from default configuration. It makes new (non-seed) nodes automatically migrate the right data to themselves. It is referenced here because `auto_bootstrap: true` is explicitly added to the cassandra.yaml file in an *AMI* installation. Setting this property to false is not recommended and is necessary only in rare instances.

broadcast_address

(Default: listen_address**) If your Cassandra cluster is deployed across multiple Amazon EC2 regions and you use the *EC2MultiRegionSnitch*, set the broadcast_address to public IP address of the node and the listen_address to the private IP.

cluster_name

(Default: Test Cluster) The name of the cluster; used to prevent machines in one logical cluster from joining another. All nodes participating in a cluster must have the same value.

commitlog_directory

(Default: /var/lib/cassandra/commitlog) The directory where the commit log is stored. For optimal write performance, it is recommended the commit log be on a separate disk partition (ideally, a separate physical device) from the data file directories.

data_file_directories

(Default: /var/lib/cassandra/data) The directory location where table data (*SSTables*) is stored.

disk_failure_policy

(Default: stop) Sets how Cassandra responds to disk failure.

- **stop**: Shuts down gossip and Thrift, leaving the node effectively dead, but it can still be inspected using JMX.
- **best_effort**: Cassandra does its best in the event of disk errors. If it cannot write to a disk, the disk is blacklisted for writes and the node continues writing elsewhere. If Cassandra cannot read from the disk, those SSTables are marked unreadable, and the node continues serving data from readable SSTables. This means you will see obsolete data at consistency level of ONE.
- **ignore**: Use for upgrading. Cassandra acts as in versions prior to 1.2. Ignores fatal errors and lets the requests fail; all file system errors are logged but otherwise ignored. It is recommended using stop or best_effort.

endpoint_snitch

(Default: org.apache.cassandra.locator.SimpleSnitch) Sets which snitch Cassandra uses for locating nodes and routing requests. It must be set to a class that implements IEndpointSnitch. For descriptions of the snitches, see *Types of snitches*.

initial_token

(Default: n/a) Used in versions prior to 1.2. If you haven't specified *num_tokens* or have set it to the default value of 1, you should always specify this parameter when setting up a production cluster for the first time and when adding capacity. For more information, see this parameter in the 1.1 Node and Cluster Configuration topic.

listen_address

(Default: localhost) The IP address or hostname that other Cassandra nodes use to connect to this node. If left unset, the hostname must resolve to the IP address of this node using /etc/hostname, /etc/hosts, or DNS. Do not specify 0.0.0.0.

num_tokens

(Default: 1**) Defines the number of tokens randomly assigned to this node on the ring. The more tokens, relative to other nodes, the larger the proportion of data that the node stores. Generally all nodes should have the same number of tokens assuming they have equal hardware capability. Specifying the *initial_token* overrides this setting. The recommended value is 256.

If left unspecified, Cassandra uses the default value of 1 token (for legacy compatibility) and uses the initial_token. If you already have a cluster with one token per node, and wish to migrate to multiple tokens per node, see http://wiki.apache.org/cassandra/Operations.

partitioner

(Default: org.apache.cassandra.dht.Murmur3Partitioner) Distributes rows (by key) across nodes in the cluster. Any IPartitioner may be used, including your own as long as it is on the classpath. Cassandra provides the following partitioners:

- *org.apache.cassandra.dht.Murmur3Partitioner*
- *org.apache.cassandra.dht.RandomPartitioner*
- *org.apache.cassandra.dht.ByteOrderedPartitioner*
- org.apache.cassandra.dht.OrderPreservingPartitioner (deprecated)
- org.apache.cassandra.dht.CollatingOrderPreservingPartitioner (deprecated)

rpc_address

(Default: localhost) The listen address for client connections (Thrift remote procedure calls). Valid values are:

- **0.0.0.0**: Listens on all configured interfaces.
- **IP address**
- **hostname**
- **unset**: Resolves the address using the hostname configuration of the node.

If left unset, the hostname must resolve to the IP address of this node using /etc/hostname, /etc/hosts, or DNS.

rpc_port

(Default: 9160) The port for the Thrift RPC service, which is used for client connections.

start_rpc

(Default: true) Starts the Thrift RPC server.

saved_caches_directory

(Default: /var/lib/cassandra/saved_caches) The directory location where table key and row caches are stored.

seed_provider

(Default: org.apache.cassandra.locator.SimpleSeedProvider) A list of comma-delimited hosts (IP addresses) to use as contact points when a node joins a cluster. Cassandra also uses this list to learn the topology of the ring. When running multiple nodes, you must change the - seeds list from the default value (127.0.0.1). In multiple data-center clusters, the - seeds list should include at least one node from each data center (replication group).

start_native_transport

(Default: false) Enable or disable the native transport server. Currently, only the Thrift server is started by default because the native transport is considered beta. Note that the address on which the native transport is bound is the same as the rpc_address. However, the port is different from the rpc_port and specified in native_transport_port.

native_transport_port

(Default: 9042) Port on which the CQL native transport listens for clients.

native_transport_max_threads

(Default: unlimited**) The maximum number of thread handling requests. The meaning is the same as rpc_max_threads.

storage_port

(Default: 7000) The port for inter-node communication.

Global row and key caches properties

When creating or modifying tables, you enable or disable the key or row caches for that table by setting the *caching parameter*. Other row and key cache tuning and configuration options are set at the global (node) level. Cassandra uses these settings to automatically distribute memory for each table on the node based on the overall workload and specific table usage. You can also configure the save periods for these caches globally. For more information, see *Configuring caches*.

key_cache_keys_to_save

(Default: disabled - all keys are saved**) Number of keys from the key cache to save.

key_cache_save_period

(Default: 14400 - 4 hours) Duration in seconds that keys are saved in cache. Caches are saved to *saved_caches_directory*. Saved caches greatly improve cold-start speeds and has relatively little effect on I/O.

key_cache_size_in_mb

(Default: empty, which automatically sets it to the smaller of 5% of the available heap, or 100MB) A global cache setting for tables. It is the maximum size of the key cache in memory. To disable set to 0.

row_cache_keys_to_save

(Default: disabled - all keys are saved**) Number of keys from the row cache to save.

row_cache_size_in_mb

(Default: 0 - disabled) A global cache setting for tables.

row_cache_save_period

(Default: 0 - disabled) Duration in seconds that rows are saved in cache. Caches are saved to *saved_caches_directory*.

row_cache_provider

(Default: SerializingCacheProvider) Specifies what kind of implementation to use for the row cache.

- **SerializingCacheProvider**: Serializes the contents of the row and stores it in native memory, that is, off the JVM Heap. Serialized rows take significantly less memory than live rows in the JVM, so you can cache more rows in a given memory footprint. Storing the cache off-heap means you can use smaller heap sizes, which reduces the impact of garbage collection pauses. It is valid to specify the fully-qualified class name to a class that implements org.apache.cassandra.cache.IRowCacheProvider.
- **ConcurrentLinkedHashCacheProvider**: Rows are cached using the JVM heap, providing the same row cache behavior as Cassandra versions prior to 0.8.

The SerializingCacheProvider is 5 to 10 times more memory-efficient than ConcurrentLinkedHashCacheProvider for applications that are not blob-intensive. However, SerializingCacheProvider may perform worse in update-heavy workload situations because it invalidates cached rows on update instead of updating them in place as

ConcurrentLinkedHashCacheProvider does.

Performance tuning properties

The following properties tune performance and system resource utilization, such as memory, disk I/O, and CPU, for reads and writes.

column_index_size_in_kb

(Default: 64) Add column indexes to a row when the data reaches this size. This value defines how much row data must be deserialized to read the column. Increase this setting if your column values are large or if you have a very large number of columns. If consistently reading only a few columns from each row or doing many partial-row reads, keep it small. All index data is read for each access, so take that into consideration when setting the index size.

commitlog_segment_size_in_mb

(Default: 32 for 32-bit JVMs, 1024 for 64-bit JVMs) Sets the size of the individual commitlog file segments. A commitlog segment may be archived, deleted, or recycled after all its data has been flushed to SSTables. This amount of data can potentially include commitlog segments from every table in the system. The default size is usually suitable for most commitlog archiving, but if you want a finer granularity, 8 or 16 MB is reasonable. See *Commit log archive configuration*.

commitlog_sync

(Default: periodic) The method that Cassandra uses to acknowledge writes in milliseconds:

- **periodic**: Used with commitlog_sync_period_in_ms (default: 10000 - 10 seconds) to control how often the commit log is synchronized to disk. Periodic syncs are acknowledged immediately.
- **batch**: Used with commitlog_sync_batch_window_in_ms (default: disabled**) to control how long Cassandra waits for other writes before performing a sync. When using this method, writes are not acknowledged until fsynced to disk.

commitlog_total_space_in_mb

(Default: 32 for 32-bit JVMs, 1024 for 64-bit JVMs**) Total space used for commitlogs. If the used space goes above this value, Cassandra rounds up to the next nearest segment multiple and flushes memtables to disk for the oldest commitlog segments, removing those log segments. This reduces the amount of data to replay on startup, and prevents infrequently-updated tables from indefinitely keeping commitlog segments. A small total commitlog space tends to cause more flush activity on less-active tables.

compaction_preheat_key_cache

(Default: true) When set to true, cached row keys are tracked during compaction, and re-cached to their new positions in the compacted SSTable. If you have extremely large key caches for tables, set the value to false; see *Global row and key caches properties*.

compaction_throughput_mb_per_sec

(Default: 16) Throttles compaction to the given total throughput across the entire system. The faster you insert data, the faster you need to compact in order to keep the SSTable count down. The recommended Value is 16 to 32 times the rate of write throughput (in MBs/second). Setting the value to 0 disables compaction throttling.

concurrent_compactors

(Default: 1 per CPU core**) Sets the number of concurrent compaction processes allowed to run simultaneously on a node, not including validation compactions for anti-entropy repair. Simultaneous compactions help preserve read performance in a mixed read-write workload by mitigating the tendency of small SSTables to accumulate during a single

long-running compaction. If compactions run too slowly or too fast, change compaction_throughput_mb_per_sec first.

concurrent_reads

(Default: 32) For workloads with more data than can fit in memory, the bottleneck is reads fetching data from disk. Setting to (16 * number_of_drives) allows operations to queue low enough in the stack so that the OS and drives can reorder them.

concurrent_writes

(Default: 32) Writes in Cassandra are rarely I/O bound, so the ideal number of concurrent writes depends on the number of CPU cores in your system. The recommended value is (8 * number_of_cpu_cores).

cross_node_timeout

(Default: false) Enable or disable operation timeout information exchange between nodes (to accurately measure request timeouts). If disabled Cassandra assumes the request was forwarded to the replica instantly by the coordinator.

Warning

Before enabling this property make sure NTP (network time protocol) is installed and the times are synchronized between the nodes.

flush_largest_memtables_at

(Default: 0.75) When Java heap usage (after a full concurrent mark sweep (CMS) garbage collection) exceeds the set value, Cassandra flushes the largest memtables to disk to free memory. This parameter is an emergency measure to prevent sudden out-of-memory (OOM) errors. Do **not** use it as a tuning mechanism. It is most effective under light to moderate loads or read-heavy workloads; it will fail under massive write loads. A value of 0.75 flushes memtables when Java heap usage is above 75% total heap size. Set to 1.0 to disable. Other emergency measures are reduce_cache_capacity_to and reduce_cache_sizes_at.

in_memory_compaction_limit_in_mb

(Default: 64) Size limit for rows being compacted in memory. Larger rows spill to disk and use a slower two-pass compaction process. When this occurs, a message is logged specifying the row key. The recommended value is 5 to 10 percent of the available Java heap size.

index_interval

(Default: 128) Controls the sampling of entries from the primary row index. The interval corresponds to the number of index entries that are skipped between taking each sample. By default Cassandra samples one row key out of every 128. The larger the interval, the smaller and less effective the sampling. The larger the sampling, the more effective the index, but with increased memory usage. Generally, the best trade off between memory usage and performance is a value between 128 and 512 in combination with a large table key cache. However, if you have small rows (many to an OS page), you may want to increase the sample size, which often lowers memory usage without an impact on performance. For large rows, decreasing the sample size may improve read performance.

memtable_flush_queue_size

(Default: 4) The number of full memtables to allow pending flush (memtables waiting for a write thread). At a minimum, set to the maximum number of secondary indexes created on a single table.

memtable_flush_writers

(Default: 1 per data directory**) Sets the number of memtable flush writer threads. These threads are blocked by disk I/O, and each one holds a memtable in memory while blocked. If you have a large Java heap size and many *data directories*, you can increase the value for better flush performance.

memtable_total_space_in_mb

(Default: 1/3 of the heap**) Specifies the total memory used for all *memtables* on a node. This replaces the per-table storage settings memtable_operations_in_millions and memtable_throughput_in_mb.

multithreaded_compaction

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

populate_io_cache_on_flush

(Default: false**) Populates the page cache on memtable flush and compaction. Enable this setting **only** when the whole node's data fits in memory.

reduce_cache_capacity_to

(Default: 0.6) Sets the size percentage to which maximum cache capacity is reduced when Java heap usage reaches the threshold defined by reduce_cache_sizes_at. Together with flush_largest_memtables_at, these properties constitute an emergency measure for preventing sudden out-of-memory (OOM) errors.

reduce_cache_sizes_at

(Default: 0.85) When Java heap usage (after a full concurrent mark sweep (CMS) garbage collection) exceeds this percentage, Cassandra reduces the cache capacity to the fraction of the current size as specified by reduce_cache_capacity_to. To disable, set the value to 1.0.

stream_throughput_outbound_megabits_per_sec

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

trickle_fsync

(Default: false) When doing sequential writing, enabling this option tells fsync to force the operating system to flush the dirty buffers at a set interval (trickle_fsync_interval_in_kb [default: 10240]). Enable this parameter to avoid sudden dirty buffer flushing from impacting read latencies. Recommended to use on SSDs, but not on HDDs.

Binary protocol timeout properties

The following timeout properties are used by the binary protocol.

read_request_timeout_in_ms

(Default: 10000) The time in milliseconds that the coordinator waits for read operations to complete.

range_request_timeout_in_ms

(Default: 10000) The time in milliseconds that the coordinator waits for sequential or index scans to complete.

request_timeout_in_ms

(Default: 10000) The default timeout for other, miscellaneous operations.

truncate_request_timeout_in_ms

(Default: 60000) The time in milliseconds that the coordinator waits for truncates to complete. The long default value allows for flushing of all tables, which ensures that anything in the commitlog is removed that could cause truncated data to reappear. If auto_snapshot is disabled, you can reduce this time.

write_request_timeout_in_ms

(Default: 10000) The time in milliseconds that the coordinator waits for write operations to complete.

request_timeout_in_ms

(Default: 10000) The default timeout for other, miscellaneous operations.

Remote procedure call tuning (RPC) properties

The following properties are used to configure and tune RPCs (client connections).

request_scheduler

(Default: org.apache.cassandra.scheduler.NoScheduler) Defines a scheduler to handle incoming client requests according to a defined policy. This scheduler is useful for throttling client requests in single clusters containing multiple keyspaces. Valid values are:

- **org.apache.cassandra.scheduler.NoScheduler**: No scheduling takes place and does not have any options.
- **org.apache.cassandra.scheduler.RoundRobinScheduler**: See request_scheduler_options properties.
- A Java class that implements the RequestScheduler interface.

request_scheduler_id

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

request_scheduler_options

(Default: disabled) Contains a list of properties that define configuration options for request_scheduler:

- **throttle_limit**: (Default: 80) The number of active requests per client. Requests beyond this limit are queued up until running requests complete. Recommended value is ((concurrent_reads + concurrent_writes) * 2).
- **default_weight**: (Default: 1**) How many requests are handled during each turn of the RoundRobin.
- **weights**: (Default: 1 or default_weight) How many requests are handled during each turn of the RoundRobin, based on the request_scheduler_id. Takes a list of keyspaces: weights.

rpc_keepalive

(Default: true) Enable or disable keepalive on client connections.

rpc_max_threads

(Default: unlimited**) Regardless of your choice of RPC server (rpc_server_type), the number of maximum requests in the RPC thread pool dictates how many concurrent requests are possible. However, if you are using the parameter sync in the rpc_server_type, it also dictates the number of clients that can be connected. For a large number of client

connections, this could 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 blocks additional connections until a client disconnects.

rpc_min_threads

(Default: 16**) Sets the minimum thread pool size for remote procedure calls.

rpc_recv_buff_size_in_bytes

(Default: N/A**) Sets the receiving socket buffer size for remote procedure calls.

rpc_send_buff_size_in_bytes

(Default: N/A**) Sets the sending socket buffer size in bytes for remote procedure calls.

streaming_socket_timeout_in_ms

(Default: 0 - never timeout streams**) Enable or disable socket timeout for streaming operations. When a timeout occurs during streaming, streaming is retried from the start of the current file. Avoid setting this value too low, as it can result in a significant amount of data re-streaming.

rpc_server_type

(Default: sync) Cassandra provides three options for the RPC server. On Windows, sync is about 30% slower than hsha. On Linux, sync and hsha performance is about the same, but hsha uses less memory.

- **sync**: (default) One connection per thread in the RPC pool. For a very large number of clients, memory is the limiting factor. On a 64 bit JVM, 128KB is the minimum stack size per thread. Connection pooling is strongly recommended.
- **hsha**: Half synchronous, half asynchronous. The RPC thread pool is used to manage requests, but the threads are multiplexed across the different clients. All Thrift clients are handled asynchronously using a small number of threads that does not vary with the number of clients (and thus scales well to many clients). The RPC requests are synchronous (one thread per active request).
- **Your own RPC server**: You must provide a fully-qualified class name of an o.a.c.t.TServerFactory that can create a server instance.

thrift_framed_transport_size_in_mb

(Default: 15) Frame size (maximum field length) for Thrift. The frame is the row or part of the row the application is inserting.

thrift_max_message_length_in_mb

(Default: 16) The maximum length of a Thrift message in megabytes, including all fields and internal Thrift overhead (1 byte of overhead for each frame).

Message length is usually used in conjunction with batches. A frame length greater than or equal to 24 accommodates a batch with four inserts, each of which is 24 bytes. The required message length is greater than or equal to 24+24+24+24+4 (number of frames).

Fault detection properties

dynamic_snitch_badness_threshold

(Default: 0.0) Sets the performance threshold for dynamically routing requests away from a poorly performing node. A value of 0.2 means Cassandra continues to prefer the static snitch values until the node response time is 20% worse than the best performing node. Until the threshold is reached, incoming client requests are statically routed to the closest replica (as determined by the snitch). Having requests consistently routed to a given replica can help keep a working set of data hot when *read repair* is less than 1.

dynamic_snitch_reset_interval_in_ms

(Default: 600000) Time interval in milliseconds to reset all node scores, which allows a bad node to recover.

dynamic_snitch_update_interval_in_ms

(Default: 100) The time interval in milliseconds for calculating read latency.

hinted_handoff_enabled

(Default: true) Enable or disable hinted handoff. A hint indicates that the write needs to be replayed to an unavailable node. Where Cassandra writes the hint depends on the version:

- Prior to 1.0: Writes to a live replica node.
- 1.0 and later: Writes to the coordinator node.

hinted_handoff_throttle_in_kb

(Default: 1024) Rate per *delivery thread* that hints are sent to the node in kilobytes per second.

max_hint_window_in_ms

(Default: 10800000 - 3 hours) Defines how long in milliseconds to generate and save hints for an unresponsive node. After this interval, new hints are no longer generated until the node is back up and responsive. If the node goes down again, a new interval begins. This setting can prevent a sudden demand for resources when a node is brought back online and the rest of the cluster attempts to replay a large volume of hinted writes.

max_hints_delivery_threads

(Default: 2) Number of threads with which to deliver hints. For multiple data center deployments, consider increasing this number because cross data-center handoff is generally slower.

phi_convict_threshold

(Default: 8**) Adjusts the sensitivity of the failure detector on an exponential scale. Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures will cause a node failure. In unstable network environments (such as EC2 at times), raising the value to 10 or 12 helps prevent false failures. Values higher than 12 and lower than 5 are not recommended.

Automatic Backup Properties

auto_snapshot

(Default: true) Enable or disable whether a snapshot is taken of the data before keyspace truncation or dropping of tables. To prevent data loss, using the default setting is **strongly** advised. If you set to false, you will lose data on truncation or drop.

incremental_backups

(Default: false) Backs up data updated since the last snapshot was taken. When enabled, Cassandra creates a hard link to each SSTable flushed or streamed locally in a backups/ subdirectory of the keyspace data. Removing these links is the operator's responsibility.

snapshot_before_compaction

(Default: false) Enable or disable taking a snapshot before each compaction. This option is useful to back up data when there is a data format change. Be careful using this option because Cassandra does not clean up older snapshots automatically.

Security properties

authenticator

(Default: org.apache.cassandra.auth.AllowAllAuthenticator) The authentication backend. It implements IAuthenticator, which is used to identify users.

authorizer

(Default: org.apache.cassandra.auth.AllowAllAuthorizer) The authorization backend. It implements IAuthenticator, which limits access and provides permissions.

permissions_validity_in_ms

(Default: `2000`) How long permissions in cache remain valid. Depending on the authorizer, fetching permissions can be resource intensive. This setting is automatically disabled when *AllowAllAuthorizer* is set.

server_encryption_options

Enable or disable inter-node encryption. You must also generate keys and provide the appropriate key and trust store locations and passwords. No custom encryption options are currently enabled.

The available options are:

- **internode_encryption**: (Default: none) Enable or disable encryption of inter-node communication using the TLS_RSA_WITH_AES_128_CBC_SHA cipher suite for authentication, key exchange, and encryption of data transfers. The available inter-node options are:
  - **all**: Encrypt all inter-node communications.
  - **none**: No encryption.
  - **dc**: Encrypt the traffic between the data centers (server only).
  - **rack**: Encrypt the traffic between the racks(server only).
- **keystore**: (Default: conf/.keystore) The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), which is the Java version of the Secure Sockets Layer (SSL), and Transport Layer Security (TLS) protocols. The keystore contains the private key used to encrypt outgoing messages.
- **keystore_password**: (Default: cassandra) Password for the keystore.
- **truststore**: (Default: conf/.truststore) Location of the truststore containing the trusted certificate for authenticating remote servers.
- **truststore_password**: (Default: cassandra) Password for the truststore.

The passwords used in these options must match the passwords used when generating the keystore and truststore. For instructions on generating these files, see: Creating a Keystore to Use with JSSE.

The advanced settings are:

- **protocol**: (Default: TLS)
- **algorithm**: (Default: SunX509)
- **store_type**: (Default: JKS)
- **cipher_suites**: (Default: TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA)
- **require_client_auth**: (Default: false) Enables or disables certificate authentication.

client_encryption_options

Enable or disable client-to-node encryption. You must also generate keys and provide the appropriate key and trust store locations and passwords. No custom encryption options are currently enabled.

- enabled: (Default: false) To enable, set to true.
- **keystore**: (Default: conf/.keystore) The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), which is the Java version of the Secure Sockets Layer (SSL), and Transport Layer Security (TLS) protocols. The keystore contains the private key used to encrypt outgoing messages.
- **keystore_password**: (Default: cassandra) Password for the keystore. This must match the password used when generating the keystore and truststore.
- **require_client_auth**: (Default: false) Enables or disables certificate authentication. (Available starting with Cassandra 1.2.3.)
- **truststore**: (Default: conf/.truststore) Set if require_client_auth is true.
- **truststore_password**: <truststore_password> Set if require_client_auth is true.

The advanced settings are:

- **protocol**: (Default: TLS)
- **algorithm**: (Default: SunX509)
- **store_type**: (Default: JKS)
- **cipher_suites**: (Default: TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA)

internode_send_buff_size_in_bytes

(Default: N/A**) Sets the sending socket buffer size in bytes for inter-node calls.

internode_recv_buff_size_in_bytes

(Default: N/A**) Sets the receiving socket buffer size in bytes for inter-node calls.

internode_compression

(Default: all) Controls whether traffic between nodes is compressed. The valid values are:

- **all**: All traffic is compressed.
- **dc**: Traffic between data centers is compressed.
- **none**: No compression.

inter_dc_tcp_nodelay

(Default: false) Enable or disable tcp_nodelay for inter-data center communication. When disabled larger, but fewer, network packets are sent. This reduces overhead from the TCP protocol itself. However, if cross data-center responses are blocked, it will increase latency.

ssl_storage_port

(Default: 7001) The SSL port for encrypted communication. Unused unless enabled in encryption_options.

Keyspace and table storage configuration

Cassandra stores storage configuration attributes in the `system` keyspace. You set storage configuration attributes on a per-keyspace or per-column family basis programmatically or using a client application, such as CQL. The attribute names documented in this section are the names as they are stored in the `system` keyspace within Cassandra. A few attributes have slightly different names in CQL.

Keyspace attributes

A keyspace must have a user-defined name, a replica placement strategy, and options that specify the number of copies per data center or node.

Attribute	Default Value
name	N/A (A user-defined value is required)
placement_strategy	SimpleStrategy
strategy_options	N/A (container attribute)
durable_writes	true

name

Required. The name for the keyspace.

placement_strategy

Required. Called strategy_class in CQL. Determines how Cassandra distributes replicas for a keyspace among nodes in the ring.

Values are:

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

NetworkTopologyStrategy requires a *properly configured snitch* to be able to determine rack and data center locations of a node. For more information about replication placement strategy, see *About data replication*.

strategy_options

Specifies configuration options for the chosen replication strategy class. The replication factor option is the total number of replicas across the cluster. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means there are 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.

When the 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 set a placement strategy and options using CQL, see *CREATE KEYSPACE*. For more information about configuring the replication placement strategy for a cluster and data centers, see *Choosing keyspace replication options*.

durable_writes

(Default: true) When set to false, data written to the keyspace bypasses the commit log. Be careful using this option because you risk losing data. Do not set this attribute on a keyspace that uses the SimpleStrategy. Change the durable_writes attribute using CQL.

Table attributes

The following attributes can be declared per table.

Option	Default Value
bloom_filter_fp_chance	0.01 or 0.1 depending on compaction strategy
bucket_high	1.5
bucket_low	0.5
caching	keys_only
column_metadata	n/a (container attribute)
column_type	Standard
comment	n/a
compaction_strategy	SizeTieredCompactionStrategy
compaction_strategy_options	n/a (container attribute)
comparator	BytesType
compare_subcolumns_with [1]	BytesType
compression_options	sstable_compression='SnappyCompressor'
default_validation_class	n/a
dclocal_read_repair_chance	0.0
gc_grace_seconds	864000 (10 days)
key_validation_class	n/a
max_compaction_threshold [2]	32
max_threshold [3]	32
min_compaction_threshold [2]	4
min_threshold [3]	4
memtable_flush_after_mins [1]	n/a
memtable_operations_in_millions [1]	n/a
memtable_throughput_in_mb [1]	n/a
min_sstable_size	50MB
name	n/a
read_repair_chance	0.1 or 1 (See description below.)
replicate_on_write	true
sstable_size_in_mb	5MB
tombstone_compaction_interval	1 day
tombstone_threshold	0.2

bloom_filter_fp_chance

(Default: 0.01 for SizeTieredCompactionStrategy, 0.1 for LeveledCompactionStrategy) Desired false-positive probability for SSTable Bloom filters. When data is requested, the Bloom filter checks if the requested row exists before doing any disk I/O. Valid values are 0 to 1.0. A setting of 0 means that the unmodified (effectively the largest possible) Bloom filter is enabled. Setting the Bloom Filter at 1.0 disables it. The higher the setting, the less memory Cassandra uses. The maximum recommended setting is 0.1, as anything above this value yields diminishing returns. For detailed information, see *Tuning Bloom filters*.

bucket_high

(Default: 1.5) Size-tiered compaction considers SSTables to be within the same bucket if the SSTable size diverges by 50% or less from the default bucket_low and default bucket_high values: [average_size * bucket_low, average_size * bucket_high].

bucket_low

(Default: 0.5) See bucket_high for a description.

caching

(Default: keys_only) Optimizes the use of cache memory without manual tuning. Set caching to one of the following values:

- all
- keys_only
- rows_only
- none

Cassandra weights the cached data by size and access frequency. In Cassandra 1.1 and later, use this parameter to specify a key or row cache instead of a table cache, as in earlier versions.

3	Ignored in Cassandra 1.2, but can still be declared for backward compatibility.
4	Used by Thrift and CQL 2; ignored in CQL 3.
5	The CQL 3 attribute name for the max_compaction_threshold and min_compaction_threshold Cassandra storage options.

chunk_length_kb

(Default: 64KB) On disk SSTables are compressed by block (to allow random reads). This subproperty of compression defines the size (in KB) of the block. Values larger than the default value might improve the compression rate, but increases the minimum size of data to be read from disk when a read occurs. The default value (64) is a good middle-ground for compressing tables. Adjust compression size to account for read/write access patterns (how much data is typically requested at once) and the average size of rows in the table.

column_metadata

(Default: N/A - container attribute) Column metadata defines these attributes of a column:

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

Setting a value for the name option is required. The validation_class is set to the default_validation_class of the table if you do not set the validation_class option explicitly. The value of index_type must be set to create a secondary index for a column. The value of index_name is not valid unless index_type is also set.

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

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

column_type

(Default: Standard) The standard type of table contains regular columns.

comment

(Default: N/A) A human readable comment describing the table.

compaction_strategy

(Default: SizeTieredCompactionStrategy) Sets the compaction strategy for the table. The available strategies are:

- **SizeTieredCompactionStrategy**: The default compaction strategy and the only compaction strategy available in releases earlier than Cassandra 1.0. This strategy triggers a minor compaction whenever there are a number of similar sized SSTables on disk (as configured by min_threshold). Using 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 table 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. For more information, see the articles When to Use Leveled Compaction and Leveled Compaction in Apache Cassandra.

compaction_strategy_options

(Default: N/A - container attribute) Sets attributes related to the chosen compaction_strategy. Attributes are:

- bucket_high
- bucket_low
- max_threshold
- min_threshold
- min_sstable_size
- sstable_size_in_mb
- tombstone_compaction_interval
- tombstone_threshold

CQL examples show how to set and update compaction properties.

comparator

(Default: BytesType) Defines the data types used to validate and sort column names. There are several built-in *column comparators* available. The comparator cannot be changed after you create a table.

compare_subcolumns_with

(Default: BytesType) Required when the *column_type attribute* is set to Super. Same as comparator but for the sub-columns of a super column. Ignored by Cassandra 1.2, but can be declared for backward compatibility.

compression_options

(Default: N/A - container attribute) Sets the compression algorithm and subproperties for the table. Choices are:

sstable_compression chunk_length_kb crc_check_chance

Using CQL presents examples of setting and updating compression properties.

crc_check_chance

(Default 1.0) When compression is enabled, each compressed block includes a checksum of that block for the purpose of detecting disk bitrot and avoiding the propagation of corruption to other replica. This option defines the probability with which those checksums are checked during read. By default they are always checked. Set to 0 to disable checksum checking and to 0.5, for instance, to check them on every other read.

default_validation_class

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

dclocal_read_repair_chance

(Default: 0.0) Specifies the probability of read repairs being invoked over all replicas in the current data center. Contrast *read_repair_chance*.

gc_grace_seconds

(Default: 864000 [10 days]) Specifies the time to wait before garbage collecting tombstones (deletion markers). The default value allows a great deal of time for consistency to be achieved prior to deletion. In many deployments this interval can be reduced, and in a single-node cluster it can be safely set to zero. When using CLI, use gc_grace instead

of gc_grace_seconds.

key_validation_class

(Default: N/A) Defines the data type used to validate row key values. There are several built-in *key validators* available, however CounterColumnType (distributed counters) cannot be used as a row key validator.

max_compaction_threshold

(Default: 32) Used by Thrift and CQL2. Ignored in CQL3; replaced by max_threshold. Sets the maximum number of SSTables processed by one minor compaction.

max_threshold

(Default: 32) Maximum number of SSTables processed by one minor compaction when using sizeTieredCompactionStrategy.

min_compaction_threshold

(Default: 4) Used by Thrift and CQL2. Ignored in CQL3; replaced by min_threshold. Sets the minimum number of SSTables to trigger a minor compaction when compaction_strategy=sizeTieredCompactionStrategy.

min_threshold

(Default: 4) Sets the minimum number of SSTables to start a minor compaction when using sizeTieredCompactionStrategy. Raising this value causes minor compactions to start less frequently and be more I/O-intensive.

memtable_flush_after_mins

Deprecated as of Cassandra 1.0. Can still be declared (for backward compatibility) but is ignored. Use *commitlog_total_space_in_mb*.

memtable_operations_in_millions

Deprecated as of Cassandra 1.0. Can still be declared (for backward compatibility) but is ignored. Use *commitlog_total_space_in_mb*.

memtable_throughput_in_mb

Deprecated as of Cassandra 1.0. Can still be declared (for backward compatibility) but is ignored. Use *commitlog_total_space_in_mb*.

min_sstable_size

(Default: 50MB) The size-tiered compaction strategy groups SSTables for compaction into buckets. The bucketing process groups SSTables that differ in size by less than 50%. This results in a bucketing process that is too fine grained for small SSTables. If your SSTables are small, use min_sstable_size to defines a size threshold (in bytes) below which all SSTables belong to one unique bucket.

name

(Default: N/A) Required. The user-defined name of the table.

read_repair_chance

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

replicate_on_write

(Default: true) Applies only to counter tables. 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 tables, this should always be set to true.

sstable_size_in_mb

(Default: 5MB) The target size for sstables that use the leveled compaction strategy. Although SSTable sizes should be less or equal to sstable_size_in_mb, it is possible to have a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. The data is not split into two SSTables.

sstable_compression

(Default: SnappyCompressor) The compression algorithm to use. Valid values are SnappyCompressor (Snappy compression library) and DeflateCompressor (Java zip implementation). Use an empty string ('') to disable compression. 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. Specify the full class name as a "string constant".

tombstone_compaction_interval

(Default: 1 day) The mininum time to wait after an SSTable creation time before considering the SSTable for tombstone compaction. Tombstone compaction is the compaction triggered if the SSTable has more garbage-collectable tombstones than tombstone_threshold.

tombstone_threshold

(Default: 0.2) A ratio of garbage-collectable tombstones to all contained columns, which if exceeded by the SSTable triggers compaction (with no other sstables) for the purpose of purging the tombstones.

Configuring the heap dump directory

Cassandra starts Java with the option -XX:-HeapDumpOnOutOfMemoryError. Using this option triggers a heap dump in the event of an out-of-memory condition. The heap dump file consists of references to objects that cause the heap to overflow. Analyzing the heap dump file can help troubleshoot memory problems. By default, Cassandra puts the file a subdirectory of the working, root directory when running as a service. If Cassandra does not have write permission to the root directory, the heap dump fails. If the root directory is too small to accommodate the heap dump, the server crashes.

For a heap dump to succeed and to prevent crashes, configure a heap dump directory that meets these requirements:

- Accessible to Cassandra for writing
- Large enough to accommodate a heap dump

Base the size of the directory on the value of the Java -mx option.

To configure the heap dump directory:

1. Open the cassandra-env.sh file for editing. This file is located in:
   - Packaged installs

     /etc/dse/cassandra
   - Binary installs

     <install_location>/resources/cassandra/conf
2. Scroll down to the comment about the heap dump path:

   ```
   # set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
   ```

3. On the line after the comment, set the CASSANDRA_HEAPDUMP_DIR to the path you want to use:

   ```
   # set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
   CASSANDRA_HEAPDUMP_DIR=<path>
   ```

4. Save the cassandra-env.sh file and restart DataStax Enterprise.

Authentication and authorization configuration

Note

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

Using authentication and authorization requires configuration changes in cassandra.yaml and two additional files:

- One file for assigning users and their permissions to keyspaces and tables
- Another file for assigning passwords to those users.

These files are named access.properties and passwd.properties, respectively, and are located in the `examples` directory of the Apache Cassandra project repository. To test simple authentication, you can move these files to the `conf` directory.

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

To set up simple authentication and authorization

1. Edit *cassandra.yaml*, setting org.apache.cassandra.auth.SimpleAuthenticator as the authenticator value. The default value of AllowAllAuthenticator is equivalent to no authentication.
2. Edit `access.properties`, adding entries for users and their permissions to read and write to specified keyspaces and tables. 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* for details and examples.
4. After making the required configuration changes, 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[.TABLE].PERMISSION=USERS

- KEYSPACE is the keyspace name.
- TABLE is the table 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 tables, 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 table in Keyspace1, you would create the following entry to allow the specified users read-write access:

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

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

```
<modify-keyspaces>=jsmith
```

passwd.properties

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

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

Logging Configuration

Cassandra provides logging functionality using Simple Logging Facade for Java (SLF4J) with a log4j backend. Additionally, the `output.log` captures the stdout of the Cassandra process, which is configurable using the standard Linux logrotate facility. You can also change logging levels via JMX using the *JConsole* tool.

Changing the Rotation and Size of Cassandra Logs

You can control the rotation and size of both the `system.log` and `output.log`. Cassandra's `system.log` logging configuration is controlled by the `log4j-server.properties` file in the following directories:

- **Packaged installs:** `/etc/dse/cassandra`
- **Binary installs:** `<install_location>/resources/cassandra/conf`

system.log

The maximum log file size and number of backup copies are controlled by the following lines:

```
log4j.appender.R.maxFileSize=20MB
log4j.appender.R.maxBackupIndex=50
```

The default configuration rolls the log file once the size exceeds 20MB and maintains up to 50 backups. When the maxFileSize is reached, the current log file is renamed to `system.log.1` and a new `system.log` is started. Any previous backups are renumbered from `system.log.n` to `system.log.n+1`, which means the higher the number, the older the file. When the maximum number of backups is reached, the oldest file is deleted.

If an issue occurred but has already been rotated out of the current `system.log`, check to see if it is captured in an older backup. If you want to keep more history, increase the maxFileSize, maxBackupIndex, or both. However, make sure you have enough space to store the additional logs.

By default, logging output is placed the `/var/log/cassandra/system.log`. You can change the location of the output by editing the `log4j.appender.R.File` path. Be sure that the directory exists and is writable by the process running Cassandra.

output.log

The `output.log` stores the stdout of the Cassandra process; it is not controllable from log4j. However, you can rotate it using the standard Linux logrotate facility. To configure logrotate to work with cassandra, create a file called `/etc/logrotate.d/cassandra` with the following contents:

```
/var/log/cassandra/output.log {
  size 10M
  rotate 9
  missingok
  copytruncate
  compress
}
```

The copytruncate directive is critical because it allows the log to be rotated without any support from Cassandra for closing and reopening the file. For more information, refer to the logrotate man page.

Changing Logging Levels

If you need more diagnostic information about the runtime behavior of a specific Cassandra node than what is provided by Cassandra's JMX MBeans and the *nodetool* utility, you can increase the logging levels on specific portions of the system using log4j. The logging levels from most to least verbose are:

```
TRACE
DEBUG
INFO
WARN
ERROR
FATAL
```

Note

Be aware that increasing logging levels can generate a lot of logging output on even a moderately trafficked cluster.

Logging Levels

The default logging level is determined by the following line in the `log4j-server.properties` file:

```
log4j.rootLogger=INFO,stdout,R
```

To exert more fine-grained control over your logging, you can specify the logging level for specific categories. The categories usually (but not always) correspond to the package and class name of the code doing the logging.

For example, the following setting logs DEBUG messages from all classes in the org.apache.cassandra.db package:

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

In this example, DEBUG messages are logged specifically from the StorageProxy class in the org.apache.cassandra.service package:

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

Finding the Category of a Log Message

To determine which category a particular message in the log belongs to, change the following line:

```
log4j.appender.R.layout.ConversionPattern=%5p [%t] %d{ISO8601} %F (line %L) %m%n
```

1. Add %c at the beginning of the conversion pattern:

   ```
   log4j.appender.R.layout.ConversionPattern=%c %5p [%t] %d{ISO8601} %F (line %L) %m%n
   ```

   Each log message is now prefixed with the category.
2. After Cassandra runs for a while, use the following command to determine which categories are logging the most messages:

   ```
   cat system.log.* | egrep 'TRACE|DEBUG|INFO|WARN|ERROR|FATAL' | awk '{ print $1 }' | sort | uniq -c | sort -n
   ```
3. If you find that a particular class logs too many messages, use the following format to set a less verbose logging level for that class by adding a line for that class:

   ```
   loggerog4j.logger.package.class=WARN
   ```

   For example a busy Solr node can log numerous INFO messages from the SolrCore, LogUpdateProcessorFactory, and SolrIndexSearcher classes. To suppress these messages, add the following lines:

   ```
   log4j.logger.org.apache.solr.core.SolrCore=WARN
   log4j.logger.org.apache.solr.update.processor.LogUpdateProcessorFactory=WARN
   log4j.logger.org.apache.solr.search.SolrIndexSearcher=WARN
   ```
4. After determining which category a particular message belongs to you may want to revert the messages back to the default format. Do this by removing %c from the `ConversionPattern`.

Commit log archive configuration

Cassandra provides commitlog archiving and point-in-time recovery. You configure this feature in the commitlog_archiving.properties configuration file, which is located in the following directories:

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

Commands

The commands archive_command and restore_command expect only a single command with arguments. STDOUT and STDIN or multiple commands cannot be executed. To workaround, you can script multiple commands and add a pointer to this file. To disable a command, leave it blank.

Archive a segment

Archive a particular commitlog segment.

Command	archive_command=	
Parameters	<path>	Fully qualified path of the segment to archive.
	<name>	Name of the commit log.
Example	archive_command=/bin/ln <path> /backup/<name>	

Restore an Archived Commitlog

Make an archived commitlog live again.

Command	restore_command=	
Parameters	<from>	Fully qualified path of the an archived commitlog segment from the <restore_directories>.
	<to>	Name of live commit log directory.
Example	restore_command=cp -f <from> <to>	

Restore Directory Location

Set the directory for the recovery files are placed.

Command	restore_directories=
Format	restore_directories=<restore_directory location>

Restore Mutations

Restore mutations created up to and including the specified timestamp.

Command	restore_point_in_time=
Format	<timestamp>
Example	restore_point_in_time=2012-08-16 20:43:12

Restore stops when the first client-supplied timestamp is greater than the restore point timestamp. Because the order in which Cassandra receives mutations does not strictly follow the timestamp order, this can leave some mutations unrecovered.

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:

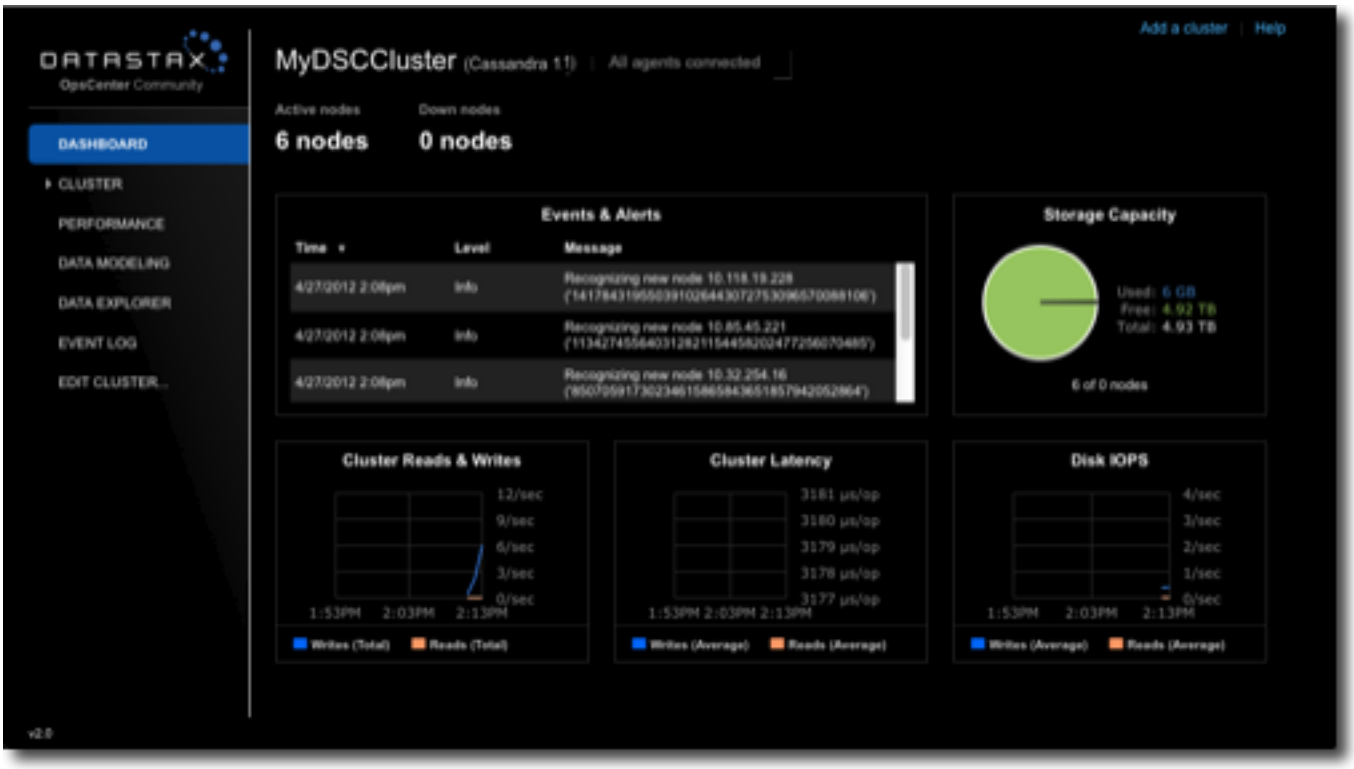
- JConsole
- The Cassandra *The nodetool utility* utility
- DataStax OpsCenter management console.

Using the same tools, you can perform certain administrative commands and operations such as flushing caches or doing a *node repair*.

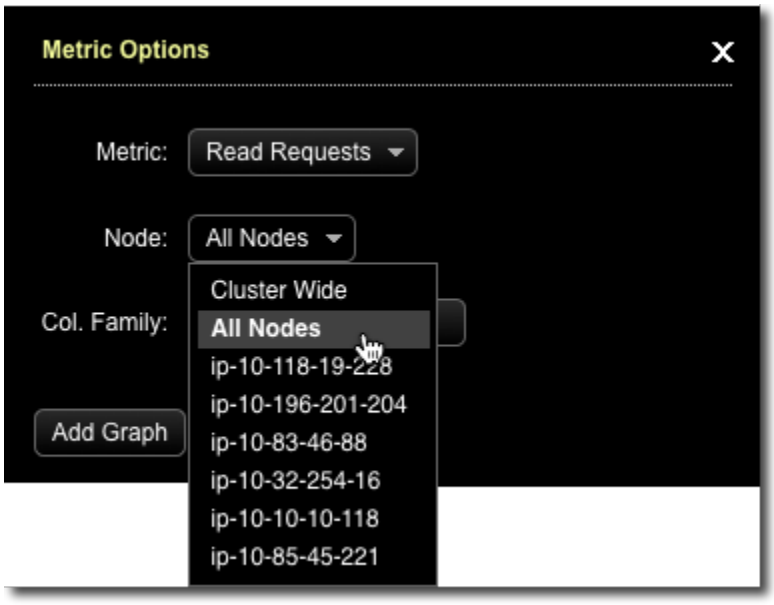
Monitoring using DataStax OpsCenter

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. You can register for a free version for development or non-production use.

OpsCenter provides a graphical representation of performance trends in a summary view that is hard to obtain with other monitoring tools. The GUI provides views for different time periods as well as the capability to drill down on single data points. Both real-time and historical performance data for a Cassandra or DataStax Enterprise cluster are available in OpsCenter. OpsCenter metrics are captured and stored within Cassandra.



Within OpsCenter you can customrize the performance metrics viewed to meet your monitoring needs. Administrators can also perform routine node administration tasks from OpsCenter. Metrics within OpsCenter are divided into three general categories: table metrics, cluster metrics, and OS metrics. For many of the available metrics, you can view aggregated cluster-wide information or view information on a per-node basis.



Monitoring using the nodetool utility

The *nodetool utility* is a command-line interface for monitoring Cassandra and performing routine database operations. Included in the Cassandra distribution, nodetool 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 *status* command. For example:

```
paul@ubuntu:~/cassandra-1.2.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address           Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160    53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48      93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
DN  10.196.14.239     ?          256     8.2%   null                                  rack1
```

The nodetool utility provides commands for viewing detailed *metrics for tables*, server metrics, and compaction statistics. Commands include decommissioning a node, running repair, and moving partitioning tokens.

Monitoring using JConsole

JConsole is a JMX-compliant tool for monitoring Java applications such as Cassandra. It is included with Sun JDK 5.0 and higher. JConsole consumes the JMX metrics and operations exposed by Cassandra and displays them in a well-organized GUI. For each node monitored, JConsole provides these six separate tab views:

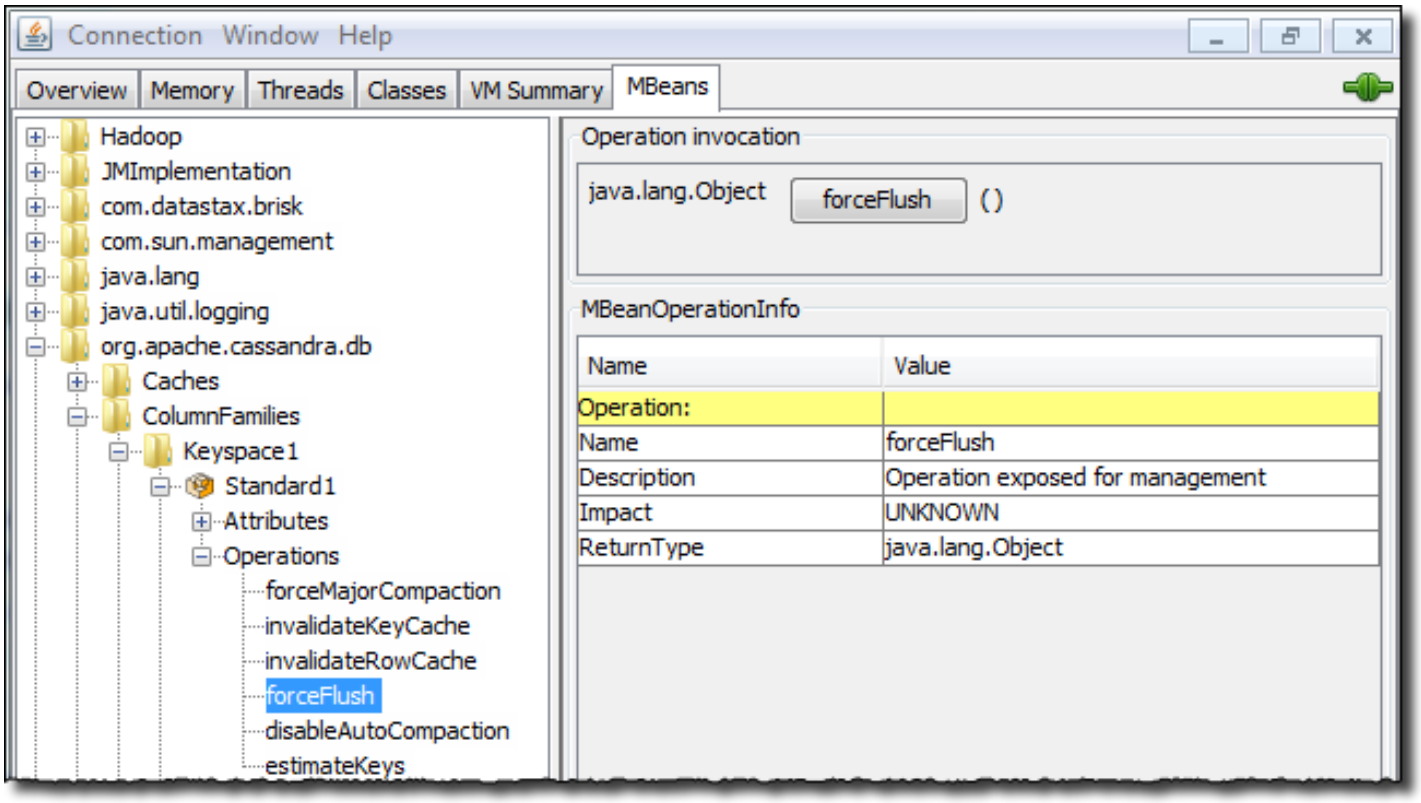
- **Overview** - Displays overview information about the Java VM and monitored values.
- **Memory** - Displays information about memory use.Threads - Displays information about thread use.
- **Classes** - Displays information about class loading.
- **VM Summary** - Displays information about the Java Virtual Machine (VM).
- **Mbeans** - Displays information about MBeans.

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

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

- org.apache.cassandra.db - Includes caching, table 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 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 table 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.

The JConsole CompactionManagerMBean exposes *compaction metrics* that can indicate when you need to add capacity to your cluster.

Compaction metrics

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

Attribute	Description
CompletedTasks	Number of completed compactions since the last start of this Cassandra instance
PendingTasks	Number of estimated tasks remaining to perform
ColumnFamilyInProgress	The table currently being compacted. This attribute is null if no compactions are in progress.
BytesTotalInProgress	Total number of data bytes (index and filter are not included) being compacted. This attribute is null if no compactions are in progress.
BytesCompacted	The progress of the current compaction. This attribute is null if no compactions are in progress.

Thread pool statistics

Cassandra maintains distinct thread pools for different stages of execution. Each of the thread pools provide statistics on the number of tasks that are active, pending, and completed. Trends on these pools for increases in the pending tasks column indicate when to add additional capacity. After a baseline is established, configure alarms for any increases above normal in the pending tasks column. Use *nodetool tpstats* on the command line to view the thread pool details shown in the following table.

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

Read/Write latency metrics

Cassandra tracks latency (averages and totals) of read, write, and slicing operations at the server level through StorageProxyMBean.

Table Statistics

For individual tables, 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:

Attribute	Description
MemtableDataSize	The total size consumed by this table's data (not including metadata).
MemtableColumnsCount	Returns the total number of columns present in the *memtable* (across all keys).
MemtableSwitchCount	How many times the memtable has been flushed out.
RecentReadLatencyMicros	The average read latency since the last call to this bean.
RecentWriterLatencyMicros	The average write latency since the last call to this bean.
LiveSSTableCount	The number of live SSTables for this table.

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

You can set a threshold and monitor LiveSSTableCount to ensure that the number of *SSTables* for a given table does not become too great.

Tuning Bloom filters

Cassandra uses Bloom filters to determine whether an SSTable has data for a particular row. Bloom filters are unused for range scans, but are used for index scans. Bloom filters are probabilistic sets that allow you to trade memory for accuracy. This means that higher Bloom filter attribute settings (*bloom_filter_fp_chance*) use less memory, but will result in more disk I/O if the SSTables are highly fragmented. Bloom filter settings range from 0 to 1.0 (disabled). The default value of bloom_filter_fp_chance depends on the *compaction_strategy*. The LeveledCompactionStrategy uses a higher default value (0.1) because it generally defragments more effectively than the SizeTieredCompactionStrategy, which has a default of 0.01. Memory savings are nonlinear; going from 0.01 to 0.1 saves about one third of the memory.

The settings you choose depend the type of workload. For example, to run an analytics application that heavily scans a particular table, you would want to inhibit the Bloom filter on the table by setting it high.

To view the observed Bloom filters false positive rate and the number of SSTables consulted per read use *cfstats* in the nodetool utility.

Starting in version 1.2, Bloom filters are stored off-heap so you don't need include it when determining the `-Xmx` settings (the maximum memory size that the heap can reach for the JVM).

To change the *Bloom filter attribute* on a column family, use CQL. For example:

```
ALTER TABLE addamsFamily WITH bloom_filter_fp_chance = 0.1;
```

After updating the value of bloom_filter_fp_chance on a table, Bloom filters need to be regenerated in one of these ways:

- *Initiate compaction*
- *Upgrade SSTables*

You do not have to restart Cassandra after regenerating SSTables.

Enabling and configuring data caches

Cassandra has offered built-in key and the row caches for a long time. As of Cassandra 1.1, cache tuning was completely revamped to make caches easier to use effectively. Caching is integrated with the database. One advantage of this integration is that Cassandra distributes cache data around the cluster for you. When a node goes down, the client can read from another cached replica of the data. The integrated architecture also facilitates troubleshooting because there is no separate caching tier, and cached data matches what's in the database exactly. The integrated cache solves the cold start problem by virtue of saving your cache to disk periodically and being able to read contents back in when it restarts—you never have to start with a cold cache.

About the key cache

The key cache is a cache of the *primary key index* for a Cassandra table. Using the key cache instead of relying on the OS page cache saves CPU time and memory. However, enabling just the key cache results in disk (or OS page cache) activity to actually read the requested data rows.

About the row cache

The row cache is similar to a traditional cache like memcached: when a row is accessed, the entire row is pulled into memory (merging from multiple SSTables if necessary) and cached so that further reads against that row can be satisfied without hitting disk at all.

Typically, you enable either the key or row cache for a table. The main exception is for archive tables that are infrequently read. You should disable caching entirely for archive tables.

Configuring caches

In the *cassandra.yaml* file, set these main caching options:

- *key_cache_size_in_mb*: The capacity in megabytes of all key caches on the node.
- *row_cache_size_in_mb*: The capacity in megabytes of all row caches on the node.

Unlike earlier Cassandra versions, cache sizes do not need to be specified per table. Just set *caching* to all, keys_only, rows_only, or none. For example, in CQL create a table to set caching to all:

```
CREATE TABLE users (
  userid text PRIMARY KEY,
  first_name text,
  last_name text,
)
with caching = 'all';
```

Other caching options set in the cassandra.yaml are:

- *key_cache_save_period*: How often to save the key caches to disk.
- *row_cache_save_period*: How often to save the row caches to disk.
- *row_cache_provider*: The implementation used for row caches on the node.

You can set the row_cache_provider to one of these options:

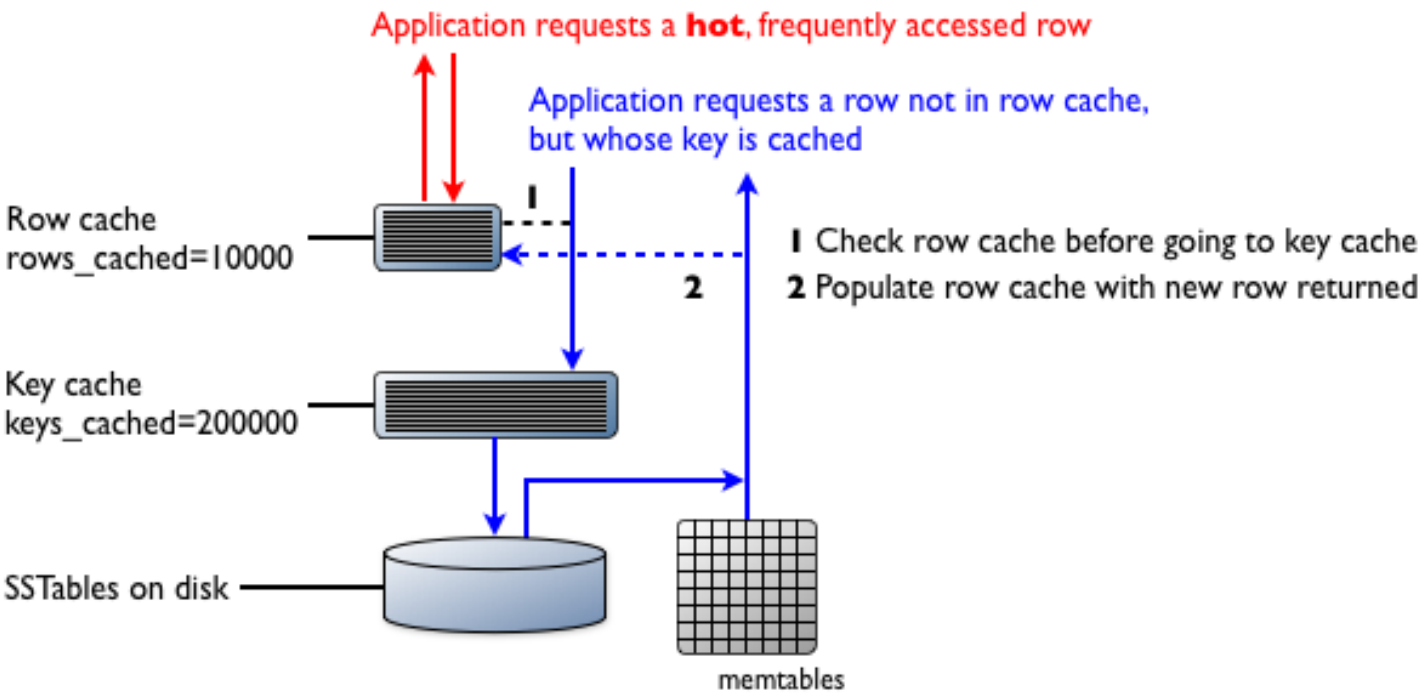
- SerializingCacheProvider
- ConcurrentLinkedHashCacheProvider

SerializingCacheProvider is the default and more memory-efficient option, between five and ten times more efficient for applications that are not blob-intensive. Using ConcurrentLinkedHashCacheProvider makes sense for use with update-heavy workloads because it updates data in place. SerializingCacheProvider, on the other hand, invalidates cached rows on update, and therefore, might not perform as well with update-heavy workloads.

Enable the key and row caches at the table level using *CQL to set the caching option*. Set the caching parameter to enable or disable caching on the keys or rows, or both of a table.

How caching works

When both row and key caches are configured, the row cache returns results whenever possible. In the event of a row cache miss, the key cache might still provide a hit that makes the disk seek much more efficient. This diagram depicts two read operations on a table with both caches already populated.



One read operation hits the row cache, returning the requested row without a disk seek. The other read operation requests a row that is not present in the row cache but is present in the key cache. After accessing the row in the *SSTable*, the system returns the data and populates the row cache with this read operation.

Tips for efficient cache use

Some tips for efficient cache use are:

- Store lower-demand data or data with extremely long rows in a table with minimal or no caching.
- Deploy a large number of Cassandra nodes under a relatively light load per node.
- Logically separate heavily-read data into discrete tables.

Cassandra's *memtables* have overhead for index structures on top of the actual data they store. If the size of the values stored in the heavily-read columns is small compared to the number of columns and rows themselves, this overhead can be substantial. Rows having this type of data do not lend themselves to efficient row caching.

Monitoring and adjusting cache performance

Make changes to cache options in small, incremental adjustments, then monitor the effects of each change using DataStax OpsCenter http://www.datastax.com/products/opscenter. In the event of high memory consumption, consider tuning data caches.

Configuring memtable throughput

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

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

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

Configuring compaction and compression

Consolidating SSTables and configuring compression of tables in the Cassandra database can improve performance. This section discusses these configuration topics and how to test your configuration changes to compaction and compression:

- *Configuring compaction*
- *Configuring compression*
- *Testing compaction and compression*

Compacting SSTables

In the background, Cassandra periodically merges SSTables together into larger SSTables using a process called *compaction*. Compaction merges row fragments, removes expired *tombstones*, and rebuilds primary and secondary indexes. Because the SSTables are sorted by row key, this merge is efficient (no random disk I/O). After 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.

Cassandra 1.2 tracks the times that tombstones can be dropped for TTL-configured and deleted columns and performs compaction when columns exceed a *CQL-configurable threshold*. Also, as of 1.2, you can better manage tombstone removal and avoid manually performing user-defined compaction to recover disk space. The *CQL-configurable* sets the minimum time to wait after an SSTable creation time before considering the SSTable for tombstone compaction.

The capability to perform multiple, independent leveled compactions in parallel promotes full I/O utilization when using SSD hardware, which is not bottlenecked by I/O. Cassandra's leveled compaction strategy creates SSTables of a fixed, relatively small size that are grouped into levels. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Cassandra executes compactions in parallel between different levels, and performs multiple compactions per level. To configure this feature, set the *multithreaded_compaction* setting to true in the cassandra.yaml configuration file and set the *compaction_strategy* as described in *Configuring compaction* below.

Compaction impacts *reads* in two ways. During compaction temporary increases in disk I/O and disk utilization 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 to complete a read request.

Configuring compaction

In addition to consolidating SSTables, the compaction process merges keys, combines columns, discards tombstones, and creates a new index in the merged SSTable.

There are two different *compaction strategies* that you can configure on a table:

- Size-tiered compaction
- Leveled compaction

To set compaction, construct a property map using CQL. Set compaction properties using a map collection:

```
name = { 'name' : value, 'name', value : 'name', value ... }
```

In this string, italics indicates optional.

To create or update a table to set the compaction strategy, use the ALTER or CREATE TABLE statements. For example:

```
ALTER TABLE users WITH
  compaction =
  { 'class' : 'LeveledCompactionStrategy', 'sstable_size_in_mb' : 10 }
```

For the list of options and more information, see *CQL 3 table storage properties*.

Configuring size-tiered compaction

Control the frequency and scope of a minor compaction of a table that uses the default size-tiered compaction strategy by setting the *CQL 3 min_threshold attribute*. The size-tiered compaction strategy triggers a minor compaction when a number SSTables on disk are of the size configured by min_threshold. Configure this value per table using CQL. For example:

```
ALTER TABLE users
  WITH compaction =
  {'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 }
```

By default, a minor compaction can begin any time Cassandra creates four SSTables on disk for a table. A minor compaction *must* begin before the total number of SSTables reaches 32.

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

Cassandra provides a startup option for *testing compaction strategies* without affecting the production workload.

For information about compaction metrics, see *Compaction metrics*.

Configuring compression

Compression maximizes the storage capacity of Cassandra nodes by reducing the volume of data on disk and disk I/O, particularly for read-dominated workloads. Cassandra quickly finds the location of rows in the SSTable index and decompresses the relevant row chunks.

Write performance is not negatively impacted by compression in Cassandra as it is in traditional databases. In traditional relational databases, writes require overwrites to existing data files on disk. The database has to locate the relevant pages on disk, decompress them, overwrite the relevant data, and finally recompress. In a relational database, compression is an expensive operation in terms of CPU cycles and disk I/O. Because Cassandra SSTable data files are immutable (they are not written to again after they have been flushed to disk), there is no recompression cycle necessary in order to process writes. SSTables are compressed only once when they are written to disk. Writes on compressed tables can show up to a 10 percent performance improvement.

How to enable and disable Compression

Compression is enabled by default in Cassandra 1.1. To disable compression, use CQL to set the compression parameters to an empty string:

```
CREATE TABLE DogTypes (
              block_id uuid,
              species text,
              alias text,
              population varint,
              PRIMARY KEY (block_id)
            )
            WITH compression = { 'sstable_compression' : 'DeflateCompressor' };
```

To enable or change compression on an existing table, use ALTER TABLE and set the *compression algorithm* sstable_compression to SnappyCompressor or DeflateCompressor.

How to change and tune compression

Change or tune data compression on a per-table basis using CQL to alter a table and set the *compression* attributes:

```
ALTER TABLE users
  WITH compression = { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 }
```

When to use compression

Compression is best suited for tables that have many rows and each row has the same columns, or at least as many columns, as other rows. For example, a table containing user data such as username, email, and state, is a good candidate for compression. The greater the similarity of the data across rows, the greater the compression ratio and gain in read performance.

A table that has rows of different sets of columns is not well-suited for compression. Dynamic tables do not yield good compression ratios.

Don't confuse table compression with *compact storage* of columns, which is used for backward compatibility of old applications with CQL 3.

Depending on the data characteristics of the table, compressing its data can result in:

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

After configuring compression on an existing table, subsequently created SSTables are compressed. Existing SSTables on disk are not compressed immediately. Cassandra compresses existing SSTables when the normal Cassandra compaction process occurs. Force existing SSTables to be rewritten and compressed by using *nodetool upgradesstables* (Cassandra 1.0.4 or later) or *nodetool scrub*.

Testing compaction and compression

Write survey mode is a Cassandra startup option for testing new compaction and compression strategies. In write survey mode, you can test out new compaction and compression strategies on that node and benchmark the write performance differences, without affecting the production cluster.

Write survey mode adds a node to a database cluster. The node accepts all write traffic as if it were part of the normal Cassandra cluster, but the node does not officially join the ring.

To enable write survey mode, start a Cassandra node using the option shown in this example:

```
bin/cassandra – Dcassandra.write_survey=true
```

Also use write survey mode to try out a new Cassandra version. The nodes you add in write survey mode to a cluster must be of the same major release version as other nodes in the cluster. The write survey mode relies on the streaming subsystem that transfers data between nodes in bulk and differs from one major release to another.

If you want to see how read performance is affected by modifications, stop the node, bring it up as a standalone machine, and then benchmark read operations on the node.

Tuning Java resources

Consider tuning Java resources in the event of a performance degradation or high memory consumption.

Java and system environment configuration

There are two files that control environment settings for Cassandra:

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

Heap sizing options

If you decide to change the Java heap sizing, both MAX_HEAP_SIZE and HEAP_NEWSIZE should should be set together in `conf/cassandra-env.sh`.

- 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.

- 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.

Tuning the Java heap

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

System Memory	Heap Size
Less than 2GB	1/2 of system memory
2GB to 4GB	1GB
Greater than 4GB	1/4 system memory, but not more than 8GB

General guidelines

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

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

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.

Guidelines for DSE Search/Solr users

Some Solr users have reported that increasing the stack size improves performance under Tomcat. To increase the stack size, uncomment and modify the default -Xss128k setting in the cassandra-env.sh file. Also, decreasing the memtable space to make room for Solr caches might improve performance. Modify the memtable space using the memtable_total_space_in_mb property in the cassandra.yaml file.

Guidelines for Analytics/Hadoop users

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

About the off-heap row cache

Cassandra can store cached rows in native memory, outside the Java heap. This results in both a smaller per-row memory footprint and reduced JVM heap requirements, which helps keep the heap size in the sweet spot for JVM garbage collection performance.

Using the off-heap row cache requires the JNA library to be installed; otherwise, Cassandra falls back on the on-heap cache provider.

Tuning Java garbage collection

Cassandra's GCInspector class logs information about garbage collection whenever a garbage collection takes longer than 200ms. Garbage collections that occur frequently and take a moderate length of time to complete (such as ConcurrentMarkSweep taking a few seconds), indicate that there is a lot of garbage collection pressure on the JVM. Remedies include adding nodes, lowering cache sizes, or adjusting the JVM options regarding garbage collection.

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, *The nodetool utility* and DataStax OpsCenter are examples of JMX-compliant management tools.

By default, you can modify the following properties in the conf/cassandra-env.sh file to configure JMX to listen on port 7199 without authentication.

- 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.

Repairing nodes

This section discusses running routine node repair.

Running routine node repair

The *nodetool repair* command repairs inconsistencies across all of the replicas for a given range of data. Run repair in these situations:

- During normal operation as part of regular, scheduled cluster maintenance unless Cassandra applications perform no deletes.
- During node recovery, for example, when bringing a node back into the cluster after a failure
- On nodes containing data that is not read frequently
- To update data on a node that has been down

Frequency of node repair

The guidelines for running node repair include:

- The hard requirement for repair frequency is the value of *gc_grace_seconds*. 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.

  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* with minimal impact to disk space. This allows wider intervals for scheduling repair operations with the *nodetool utility*.

Replacing or adding a node or data center

This section discusses the following tasks:

- *Adding nodes an existing cluster*
- *Adding a data center to a cluster*
- *Replacing a dead node*

Adding nodes an existing cluster

Virtual nodes greatly simplify adding nodes to an existing cluster:

- Calculating tokens and assigning them to each node is no longer required.
- Rebalancing a cluster is no longer necessary because a node joining the cluster assumes responsibility for an even portion of the data.

For a detailed explanation about how this works, see Virtual nodes in Cassandra 1.2.

Note

If you do not use virtual nodes, follow the instructions in the 1.1 topic Adding Capacity to an Existing Cluster.

To add nodes to a cluster

1. Install Cassandra on the new nodes, but do not start Cassandra. (If you used a packaged install, Cassandra starts automatically and you must *stop the node* and *clear the data*.)
2. Set the following properties in the *cassandra.yaml* and *cassandra-topology.properties* configurtion files:
    - *cluster_name*: The name of the cluster the new node is joining.
    - *listen_address*/*broadcast_address*: The IP address or host name that other Cassandra nodes use to connect to the new node.
    - *endpoint_snitch*: The snitch Cassandra uses for locating nodes and routing requests.
    - *num_tokens*: The number of *virtual nodes* to assign to the node. If the hardware capabilities vary among the nodes in your cluster, you can assign a proportional number of virtual nodes to the larger machines.
    - *seed_provider*: The `- seeds` list in this setting determines which nodes the new node should contact to learn about the cluster and establish the gossip process.
    - Change other non-default settings you have made to your existing cluster in the cassandra.yaml file and cassandra-topology.properties files. Use the diff command to find and merge (by head) any differences between existing and new nodes.
3. *Start Cassandra* on each new node. Allow two minutes between node initializations. You can monitor the startup and data streaming process using *nodetool netstats*.
4. After all new nodes are running, run *nodetool cleanup* on each of the previously 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.

1. Ensure that you are using *NetworkTopologyStrategy* for all of your keyspaces.
2. For each new node, edit the *configuration properties* in the *cassandra.yaml* file:
   - Add (or edit) *auto_bootstrap: false*. By default, this setting is true and not listed in the cassandra.yaml file. Setting this parameter to false prevent the new nodes from attempting to get all the data from the other nodes in the data center. When you run *nodetool rebuild* in the last step, each node is properly mapped.
   - Set the necessary properties as described in *step 2* above.
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 auto-detect 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: set strategy options to DC1:2, DC2:2. For more information, see *ALTER KEYSPACE*.
   b. Run *nodetool rebuild* on all nodes in the new data center.

Replacing a dead node

To replace a node that has died for some reason, such as hardware failure, prepare and start the replacement node, integrate it into the cluster, and then remove the dead node.

To prepare the replacement node:

1. Confirm that the node is dead by running *nodetool status* (if using virtual nodes) or *nodetool ring*.

   The nodetool command shows a down status for the dead node:

```
paul@ubuntu:~/cassandra-1.2.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
DN  10.196.14.239    ?          256     8.2%   null                                  rack1
```

2. Add and start the replacement node as described in the *steps above*.
3. Remove the dead node from the cluster using the *removenode* command. Use the force option of this command if necessary to remove the node.

Backing up and restoring data

Cassandra backs up data by taking a snapshot of all on-disk data files (SSTable files) stored in the data directory. You can take a snapshot of all keyspaces, a single keyspace, or a single table while the system is online.

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

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

Note

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

Taking a snapshot

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

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

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

To create a snapshot of a node

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

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

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

Deleting snapshot files

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

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

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

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

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

Enabling incremental backups

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

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

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

Restoring from a Snapshot

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

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

Node restart method

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

Note

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

To restore a node from a snapshot and incremental backups:

1. Shut down the node.
2. Clear all files in /var/lib/cassandra/commitlog.
3. Delete all **.db* files in this directory:

   <data_directory_location>/<keyspace_name>/<table_name>

   DO NOT delete the /snapshots and /backups subdirectories.
4. Locate the most recent snapshot folder in this directory:

   <data_directory_location>/<keyspace_name>/<table_name>/snapshots/<snapshot_name>

   Copy its contents into this directory:

   <data_directory_location>/<keyspace_name>/<table_name> directory.
5. If using incremental backups, copy all contents of this directory:

   <data_directory_location>/<keyspace_name>/<table_name>/backups

   Paste it into this directory:

   <data_directory_location>/<keyspace_name>/<table_name>.
6. Restart the node.

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

References

The nodetool utility

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

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

Standard usage:

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

RMI usage:

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

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

Options

The available options are:

Flag	Option	Description
-h	--host *arg*	Hostname of node or IP address.
-p	--port *arg*	Remote JMX agent port number.
-pr	--partitioner-range	Repair only the first range returned by the partitioner for the node.
-pw	--password *arg*	Remote JMX agent password.
-u	--username *arg*	Remote JMX agent username.
Snapshot Options Only		
-cf	--column-family *arg*	Only take a snapshot of the specified table.
-snapshot	--with-snapshot	Repair one node at a time using snapshots.
-t	--tag *arg*	Optional name to give a snapshot.

Command list

The available commands are:

Command List		
cfhistograms	*getendpoints*	*repair*
cfstats	*getsstables*	*ring*
cleanup	*gossipinfo*	*scrub*
clearsnapshot	*info*	*setcompactionthreshold*
compact	*invalidatekeycache*	*setcompactionthroughput*
compactionstats	*invalidaterowcache*	*setstreamthroughput*
decomission	*join*	*settraceprobability*
describering	*move*	*snapshot*

disablegossip	*netstats*	*status*
disablethrift	*rangekeysample*	*statusthrift*
drain	*rebuild*	*stop*
enablegossip	*rebuild_index*	*tpstats*
enablethrift	*refresh*	*upgradesstables*
flush	*removenode*	*version*
getcompactionthreshold		

Command details

Details for each command are listed below:

cfhistograms *keyspace table*

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

cfstats

Displays statistics for every keyspace and table.

cleanup [*keyspace*][*table*]

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 table 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*][*table*]

For tables that use the *SizeTieredCompactionStrategy*, initiates an immediate major compaction of all tables in *keyspace*. For each table 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 table names.

compactionstats

Displays compaction statistics.

decommission

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

http://wiki.apache.org/cassandra/NodeProbe#Decommission

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

describering [*keyspace*]

Shows the token ranges for a given keyspace.

disablegossip

Disable Gossip. Effectively marks the node dead.

disablethrift

Disables the Thrift server.

drain

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

enablegossip

Re-enables Gossip.

enablethrift

Re-enables the Thrift server.

flush [*keyspace*] [*table*]

Flushes all memtables for a keyspace to disk, allowing the commit log to be cleared. Optionally takes a list of table names.

getcompactionthreshold *keyspace table*

Gets the current compaction threshold settings for a table. See http://wiki.apache.org/cassandra/MemtableSSTable.

getendpoints *keyspace table key*

Displays the end points that owns the key. The `key` is only accepted in HEX format.

getsstables *keyspace table key*

Displays the sstable filenames that own the key.

gossipinfo

Shows the gossip information for the cluster.

info

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

invalidatekeycache [*keyspace*] [*tables*]

Invalidates, or deletes, the key cache. Optionally takes a keyspace or list of table names. Leave a blank space between each table name.

invalidaterowcache [*keyspace*] [*tables*]

Invalidates, or deletes, the row cache. Optionally takes a keyspace or list of table names. Leave a blank space between each table name.

join

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

move *new_token*

Moves a node to a new token. This essentially combines decommission and bootstrap. See:

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

netstats [*host*]

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

rangekeysample

Displays the sampled keys held across all keyspaces.

rebuild [*source_dc_name*]

Rebuilds data by streaming from other nodes (similar to bootstrap). Use this command to bring up a new data center in an existing cluster. See *Adding a data center to a cluster*.

rebuild_index *keyspace table_name.index_name,index_name1*

Fully rebuilds of native secondary index for a given table. Example of *index_names*: Standard3.IdxName,Standard3.IdxName1

refresh *keyspace table*]

Loads newly placed SSTables on to the system without restart.

removenode force | status *Host ID*

Remove node by host ID, or force completion of pending removal. Show status of current node removal. To get host ID, run *status*.

repair *keyspace* [*table*] [-pr]

Begins an anti-entropy node repair operation. If the `-pr` option is specified, only the first range returned by the partitioner for a node is repaired. This allows you to repair each node in the cluster in succession without duplicating work.

Without `-pr`, all replica ranges that the node is responsible for are repaired.

Optionally takes a list of table names.

ring

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

If you are using virtual nodes, use *nodetool status*; it is much less verbose.

scrub [*keyspace*][*table*]

Rebuilds SSTables on a node for the named tables and snapshots data files before rebuilding as a safety measure. If possible use `upgradesstables`. While `scrub` rebuilds SSTables, it also discards data that it deems broken and creates a snapshot, which you have to remove manually.

setcompactionthreshold *keyspace table min_threshold max_threshold*

The *min_threshold* parameter controls how many SSTables of a similar size must be present before a minor compaction is scheduled. The *max_threshold* sets an upper bound on the number of SSTables that may be compacted in a single minor compaction. See also:

http://wiki.apache.org/cassandra/MemtableSSTable

setcompactionthroughput *value_in_mb*

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

setstreamthroughput *value_in_mb*

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

settraceprobability *value*

Probabilistic tracing is useful to determine the cause of intermittent query performance problems by identifying which queries are responsible. This option traces some or all statements sent to a cluster. Tracing a request usually requires at least 10 rows to be inserted.

A probability of 1.0 will trace everything whereas lesser amounts (for example, 0.10) only sample a certain percentage of statements. Care should be taken on large and active systems, as system-wide tracing will have a performance impact. Unless you are under very light load, tracing all requests (probability 1.0) will probably overwhelm your system. Start with a small fraction, for example, 0.001 and increase only if necessary. The trace information is stored in a systems_traces keyspace that holds two tables – sessions and events, which can be easily queried to answer questions, such as what the most time-consuming query has been since a trace was started. Query the parameters map

and thread column in the system_traces.sessions and events tables for probabilistic tracing information.

snapshot -cf [*tableName*] [*keyspace*] -t [*snapshot-name*]

Takes an online snapshot of Cassandra's data. Before taking the snapshot, the node is flushed. The results are stored in Cassandra's data directory under the `snapshots` directory of each keyspace. See *Install locations*. See:

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

status

Display cluster information, such as state, load, host ID, and token.

statusthrift

Status of the thrift server.

stop [*operation type*]

Stops an operation from continuing to run. Options are COMPACTION, VALIDATION, CLEANUP, SCRUB, INDEX_BUILD. For example, this allows you to stop a compaction that has a negative impact on the performance of a node. After the compaction stops, Cassandra continues with the rest in the queue. Eventually, Cassandra restarts the compaction.

tpstats

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

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

upgradesstables [*keyspace*][*table*]

Rebuilds SSTables on a node for the named tables. Use when upgrading your server or changing compression options (available from Cassandra 1.0.4 onwards).

version

Displays the Cassandra release version for the node being queried.

The cassandra utility

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, on Linux, Cassandra uses an include file, `cassandra.in.sh`, to source these environment variables. It will check the following locations for this file:

- Environment setting for CASSANDRA_INCLUDE if set
- <install_location>/bin
- /usr/share/cassandra/cassandra.in.sh

- /usr/local/share/cassandra/cassandra.in.sh
- /opt/cassandra/cassandra.in.sh
- <USER_HOME>/.cassandra.in.sh

Cassandra also uses the Java options set in `$CASSANDRA_CONF/cassandra-env.sh`. If you want to pass additional options to the Java virtual machine, such as maximum and minimum heap size, edit the options in that file rather than setting JVM_OPTS in the environment.

Options

- -f Start the cassandra process in foreground (default is to start as a background process).
- -p <filename> Log the process ID in the named file. Useful for stopping Cassandra by killing its PID.
- -v Print the version and exit.
- -D <parameter>

  Passes in one of the following startup parameters:

Parameter	Description
access.properties=<filename>	The file location of the access.properties file.
cassandra-pidfile=<filename>	Log the Cassandra server process ID in the named file. Useful for stopping Cassandra by killing its PID.
cassandra.config=<directory>	The directory location of the Cassandra configuration files.
cassandra.initial_token=<token>	Sets the initial partitioner token for a node the first time the node is started.
cassandra.join_ring=<true|false>	Set to false to start Cassandra on a node but not have the node join the cluster.
cassandra.load_ring_state= <true|false>	Set to false to clear all gossip state for the node on restart. Use if you have changed node information in cassandra.yaml (such as listen_address).
cassandra.renew_counter_id= <true|false>	Set to true to reset local counter info on a node. Used to recover from data loss to a counter table. First remove all SSTables for counter tabless on the node, then restart the node with -Dcassandra.renew_counter_id=true, then run nodetool repair once the node is up again.
cassandra.replace_token=<token>	To replace a node that has died, restart a new node in its place and use this parameter to pass in the token that the new node is assuming. The new node must not have any data in its data directory and the token passed must already be a token that is part of the ring.
cassandra.write_survey=true	For testing new compaction and compression strategies. It allows you to experiment with different strategies and benchmark write performance differences without affecting the production workload. See *Testing compaction and compression*.

```
cassandra.framed
cassandra.host
cassandra.port=<port>
cassandra.rpc_port=<port>
cassandra.start_rpc=<true|false>
```

```
cassandra.storage_port=<port>
corrupt-sstable-root
legacy-sstable-root
mx4jaddress
mx4jport
passwd.mode
passwd.properties=<file>
```

Examples

Start Cassandra on a node and log its PID to a file:

```
cassandra -p ./cassandra.pid
```

Clear gossip state when starting a node. This is useful if the node has changed its configuration, such as its listen IP address:

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

Start Cassandra on a node in stand-alone mode (do not join the cluster configured in the `cassandra.yaml` file):

```
cassandra -Dcassandra.join_ring=false
```

Cassandra bulk loader

The sstableloader tool provides the ability to bulk load external data into a cluster, load existing SSTables into another cluster with a different number nodes or replication strategy, and restore snapshots.

About sstableloader

The sstableloader tool streams a set of SSTable data files to a live cluster. It does not simply copy the set of SSTables to every node, but transfers the relevant part of the data to each node, conforming to the replication strategy of the cluster. The table into which the data is loaded does not need to be empty.

Because sstableloader uses Cassandra gossip, make sure that the cassandra.yaml configuration file is in the classpath and set to communicate with the cluster. At least one node of the cluster must be configured as seed. If necessary, properly configure the following properties: listen_address, storage_port, rpc_address, and rpc_port.

If you use sstableloader to load external data, you must first generate SSTables. If you use DataStax Enterprise, you can use Sqoop to migrate your data or if you use Cassandra, follow the procedure described in Using the Cassandra Bulk Loader blog.

Before loading the data, you must define the schema of the column families with *CLI*, Thrift, or *CQL*.

To get the best throughput from SSTable loading, you can use multiple instances of sstableloader to stream across multiple machines. No hard limit exists on the number of SSTables that sstablerloader can run at the same time, so you can add additional loaders until you see no further improvement.

If you use sstableloader on the same machine as the Cassandra node, you can't use the same network interface as the Cassandra node. However, you can use the JMX > StorageService > bulkload() call from that node. This method takes the absolute path to the directory where the SSTables are located, and loads them just as sstableloader does. However, because the node is both source and destination for the streaming, it increases the load on that node. This means that you should load data from machines that are not Cassandra nodes when loading into a live cluster.

Using sstableloader

In binary installations, sstableloader is located in the <install_location>/bin directory.

The sstableloader bulk loads the SSTables found in the directory <dir_path> to the configured cluster. The parent directory of <dir_path> is used as the keyspace name. For example to load an SSTable named Standard1-he-1-Data.db into keyspace Keyspace1, the files Keyspace1-Standard1-he-1-Data.db and Keyspace1-Standard1-he-1-Index.db must be in a directory called Keyspace1/Standard1/.

```
bash sstableloader [options] <dir_path>
```

Example:

```
$ ls -1 Keyspace1/Standard1/
Keyspace1-Standard1-he-1-Data.db
Keyspace1-Standard1-he-1-Index
$ <path_to_install>/bin/sstableloader -d localhost Keyspace1/<dir_name>/
```

where <dir_name> is the directory containing the SSTables. Only the -Data and -Index components are required; -Statistics and -Filter are ignored.

The sstableloader has the following options:

Option	Description
-d,--nodes <initial hosts>	Connect to comma separated list of hosts for initial ring information.
--debug	Display stack traces.
-h,--help	Display help.
-i,--ignore <NODES>	Do not stream to this comma separated list of nodes.
--no-progress	Do not display progress.
-p,--port <rpc port>	RPC port (default 9160).
-t,--throttle <throttle>	Throttle speed in Mbits (default unlimited).
-v,--verbose	Verbose output.

The sstable2json / json2sstable utility

The sstable2json utility converts the on-disk SSTable representation of a table into a JSON formatted document. Its counterpart, json2sstable, does exactly the opposite: it converts a JSON representation of a table to a Cassandra usable SSTable format. Converting SSTables this way is useful for testing and debugging.

Note

Starting with version 0.7, json2sstable and sstable2json must be run so that the schema can be loaded from system tables. This means that the cassandra.yaml file must be in the classpath and refer to valid storage directories. For more information, see the Import/Export section of http://wiki.apache.org/cassandra/Operations.

sstable2json

This converts the on-disk SSTable representation of a table into a JSON formatted document.

Usage

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

`SSTABLE` should be a full path to a {table-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 tables is:

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

Row keys, column names and values are written in as the HEX representation of their byte arrays. Line breaks are only in between row keys in the actual output.

json2sstable

This converts a JSON representation of a table (aka column family) to a Cassandra usable SSTable format.

Usage

```
bin/json2sstable -K KEYSPACE -c COLUMN_FAMILY JSON SSTABLE
```

JSON should be a path to the JSON file

SSTABLE should be a full path to a {table-name}-Data.d` 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 table's data, it dumps only the keys.

Usage

```
bin/sstablekeys SSTABLE
```

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

Install locations

Cassandra 1.2 unpacks files into directories listed in this section.

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 File Locations.

Packaged install 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)

Starting and stopping Cassandra

On initial start-up, each node must be started one at a time, starting with your seed nodes.

Starting Cassandra as a stand-alone process

Start the Cassandra Java server process starting with the seed nodes:

`$ cd <install_location>` `$ bin/cassandra` (in the background - default) `$ bin/cassandra -f` (in the foreground)

Starting Cassandra as a service

Packaged installations provide startup scripts in `/etc/init.d` for starting Cassandra as a service. The service runs as the `cassandra` user.

To start the Cassandra service, you must have root or sudo permissions:

```
$ sudo service cassandra start
```

Note

On Enterprise Linux systems, the Cassandra service runs as a java process. On Debian systems, the Cassandra service runs as a jsvc process.

Stopping Cassandra as a stand-alone process

To stop the Cassandra process, find the Cassandra Java process ID (PID), and then kill the process using its PID number:

```
$ ps auwx | grep cassandra
$ sudo kill <pid>
```

Stopping Cassandra as a service

To stop the Cassandra service, you must have root or sudo permissions:

```
$ sudo service cassandra stop
```

Clearing the data as a stand-alone process

For binary installs, after stopping the process, run the following command from the install directory:

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

Clearing the data as a service

For packaged installs, after stopping the service, run the following command:

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

Troubleshooting Guide

This page contains recommended fixes and workarounds for issues commonly encountered with Cassandra:

Reads are getting slower while writes are still fast

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

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

Nodes seem to freeze after some period of time

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

If the GCInspector isn't reporting very long GC times, but is reporting moderate times frequently (ConcurrentMarkSweep taking a few seconds very often) then it is likely that the JVM is experiencing extreme GC pressure and will eventually OOM. See the section below on OOM errors.

Nodes are dying with OOM errors

If nodes are dying with OutOfMemory exceptions, check for these typical causes:

- **Row cache is too large, or is caching large rows.** Row cache is generally a high-end optimization. Try disabling it and see if the OOM problems continue.

- **The memtable sizes are too large for the amount of heap allocated to the JVM.** You can expect N + 2 memtables resident in memory, where N is the number of column families. Adding another 1GB on top of that for Cassandra itself is a good estimate of total heap usage.

If none of these seem to apply to your situation, try loading the heap dump in MAT and see which class is consuming the bulk of the heap for clues.

Nodetool or JMX connections failing on remote nodes

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

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

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

View of ring differs between some nodes

This is an indication that the ring is in a bad state. This can happen when not using virtual nodes and there are token conflicts (for instance, when bootstrapping two nodes simultaneously with automatic token selection.) Unfortunately, the only way to resolve this is to do a full cluster restart; a rolling restart is insufficient since gossip from nodes with the bad state will repopulate it on newly booted nodes.

Java reports an error saying there are too many open files

Java is not allowed to open enough file descriptors. Cassandra generally needs more than the default (1024) amount. To increase the number of file descriptors, change the security limits on your Cassandra nodes as described in the *Recommended settings* section of *Insufficient user resource limits errors*.

Another, much less likely possibility, is a file descriptor leak in Cassandra. Run `lsof -n | grep java` to check that the number of file descriptors opened by Java is reasonable and reports the error if the number is greater than a few thousand.

Insufficient user resource limits errors

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

Cassandra errors

Insufficient as (address space) or memlock setting:

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

Insufficient memlock settings:

```
WARN [main] 2011-06-15 09:58:56,861 CLibrary.java (line 118) Unable to lock JVM memory (ENOMEM).
This can result in part of the JVM being swapped out, especially with mmapped I/O enabled.
Increase RLIMIT_MEMLOCK or run Cassandra as root.
```

Insufficient nofiles setting:

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

Insufficient nofiles setting:

```
ERROR [MutationStage:11] 2012-04-30 09:46:08,102 AbstractCassandraDaemon.java (line 139)
Fatal exception in thread Thread[MutationStage:11,5,main]
java.lang.OutOfMemoryError: unable to create new native thread
```

OpsCenter errors

See the OpsCenter Troubleshooting documentation.

Recommended settings

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

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

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

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

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

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

```
*  soft  nproc  10240
```

Cannot initialize class org.xerial.snappy.Snappy

The following error may occur when Snappy compression/decompression is enabled although its library is available from the classpath:

```
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`

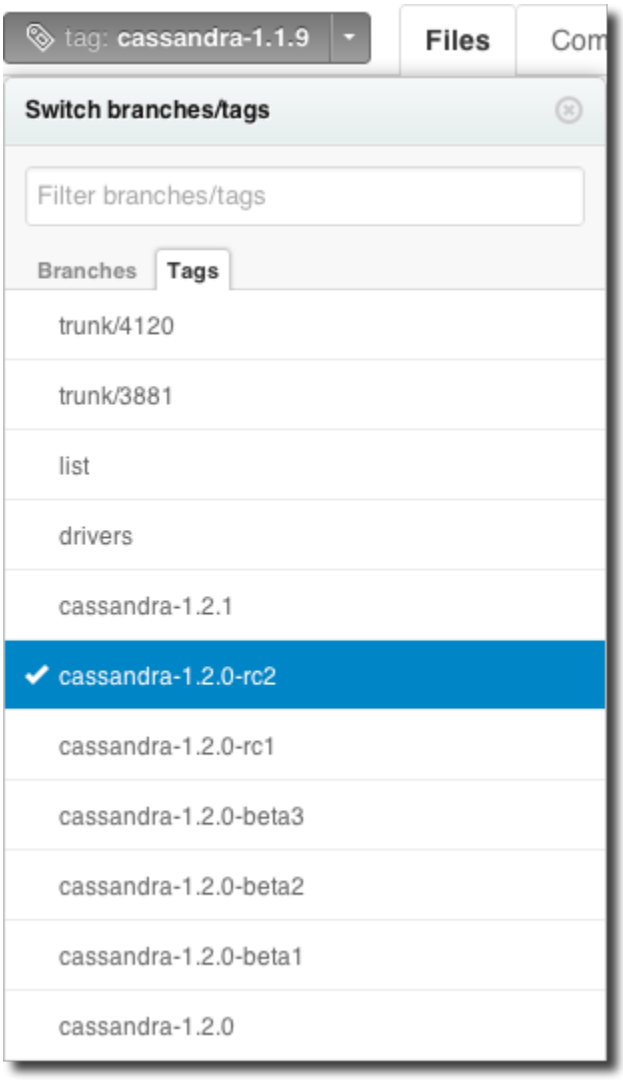
  The default `cassandra-env.sh` looks for the variable and appends to it when starting the JVM.

DataStax Community release notes

Current DataStax Community version: Cassandra 1.2.3.

Fixes and New Features in Cassandra 1.2.3

For a list of fixes and new features, see the CHANGES.txt. You can view all version changes by branch or tag in the **branch** drop-down list:



Glossary of Cassandra Terms

anti-entropy

The synchronization of replica data on nodes to ensure that the data is fresh.

Bloom filter

An off-heap structure associated with each SSTable that checks if any data for the requested row exists in the SSTable before doing any disk I/O.

cluster

Two or more Cassandra instances the exchange messages using the gossip protocol.

clustering

The storage engine process that creates an index and keeps data in order based on the index.

clustering column

Columns other than the *partition key* in a compound primary key definition.

column

The smallest increment of data, which contains a name, a value and a timestamp.

column family

A container for rows, similar to the table in a relational system. Replaced by *table* in CQL 3. The CLI, API classes, and OpsCenter continue to use column family.

commit log

A file to which Cassandra appends changed data for recovery in the event of a hardware failure.

compaction

A process that consists primarily of consolidating *SSTables*, but also discards tombstones and regenerates the index in the SSTable. A major compaction merges all SSTables into one. A minor compaction merges from 4 to 32 SSTables for a table.

compound primary key

A *column* having a name and value like a standard column except that the value, which is a byte array, is stuffed with more bytes that can be converted to multiple values by built-in handlers.

consistency

The synchronization of data on replicas in a cluster. Consistency is categorized as *weak* or *strong*.

consistency level

A setting that defines a successful write or read by the number of cluster replicas that acknowledge the write or respond to the read request, respectively.

coordinator node

The node that determines which nodes in the ring should get the request based on the cluster configured snitch.

cross-data center forwarding

A technique for optimizing replication across data centers. To replicate data in a node in one data center to nodes in another data center, the data is sent to one node in the other data center, and that node forwards the data to other nodes in its data center.

data center

Synonymous with replication group. A group of related nodes configured together within a cluster for replication purposes. It is not necessarily a *physical* data center.

gossip

A peer-to-peer communication protocol for exchanging location and state information between nodes.

HDFS

Hadoop Distributed File System that stores data on nodes to improve performance. A necessary component in addition to MapReduce in a Hadoop distribution.

idempotent

An operation that can occur multiple times without changing the result, such as Cassandra performing the same update multiple times without affecting the outcome.

index summary

A subset of the *primary index*. By default, 1 row key out of every 128 is sampled.

keyspace

A namespace container that defines how data is replicated on nodes.

MapReduce

Hadoop's parallel processing engine that can process large data sets relatively quickly. A necessary component in addition to MapReduce in a Hadoop distribution.

memtable

A Cassandra table-specific, in-memory data structure that resembles a write-back cache. See also *About writes*.

mutation

1) An *upsert*. *2)* A Thrift base class that has abstract methods for reading and writing data input and output.

node repair

A process that makes all data on a replica consistent.

partitioner

Distributes the data across the cluster. The types of partitioners are Murmur3Partitioner (default), RandomPartitioner, and OrderPreservingPartitioner.

partition key

The first column declared in the PRIMARY KEY definition, which is the row key, or in the case of a compound primary key, multiple columns can declare those columns that form the compound primary key.

primary index

A list of row keys and the start position of rows in the data file

primary key

One or more columns that uniquely identify a row in a *table*.

read repair

A process that updates Cassandra replicas with the most recent version of frequently-read data.

replication group

See *data center*.

replica placement strategy

A specification that determines the replicas for each row of data.

rolling restart

A procedure that is performed during upgrading nodes in a cluster for zero downtime. Nodes are upgraded and restarted one at a time, while other nodes continue to operate online.

row

1) Columns that have the same row key. *2)* A collection of cells per combination of columns in the storage engine.

row key

A unique identifier of a row in a table that is similar to a primary key in a relational database table.

secondary index

A native Cassandra capability for finding a row in the database that does not involve using the row key.

slice

A Thrift API term for a set of columns from a single row, described either by name or as a contiguous run of columns from a starting point.

snitch

The mapping from the IP addresses of nodes to physical and virtual locations, such as racks and data centers. There are several types of snitches. The type of snitch affects the request routing mechanism.

SSTable

A sorted string table (SSTable) is an immutable data file to which Cassandra writes memtables periodically. SSTables are stored on disk sequentially and maintained for each Cassandra table. See also *About writes*.

strong consistency

When reading data, Cassandra performs *read repair* before returning results.

superuser

By default, each installation of Cassandra includes a superuser account named `cassandra` and whose password is also `cassandra`. This account allows a user to perform any action on the database cluster and create new login accounts. It is recommended that the password be changed from the default.

table

In CQL 3, a collection of ordered (by name) columns. In previous versions of CQL, the *column family* was synonymous, in many respects, to a table. In CQL 3 a table is sparse, including only columns that rows have been assigned a value.

token

An element on the ring that depends on the partitioner. A token determines the node's position on the ring and the portion of data it is responsible for. The range for the Murmur3Partitioner (default) is -2^{63} to $+2^{63}$. The range for the RandomPartitionerIntegers is 0 to $2^{127}-1$.

tombstone

A marker in a row that indicates a column was deleted. During compaction, marked column are deleted.

TTL

Time-to-live. An optional expiration date for values inserted into a column. See also *Expiring columns*.

weak consistency

When reading data, Cassandra performs *read repair* after returning results.

upsert

A change in the database that updates a specified column in a row if the column exists or inserts the column if it does not exist.

CQL Command Table of Contents

CQL Commands	cqlsh Commands
ALTER KEYSPACE	*ASSUME*
ALTER TABLE	*CAPTURE*
ALTER USER	*CONSISTENCY*
BATCH	*COPY*
CREATE INDEX	*DESCRIBE*
CREATE KEYSPACE	*EXIT*
CREATE TABLE	*SHOW*
CREATE USER	*SOURCE*
DELETE	*TRACING*
DROP INDEX	
DROP KEYSPACE	
DROP TABLE	
DROP USER	
GRANT	
INSERT	
LIST PERMISSIONS	
LIST USERS	
REVOKE	
SELECT	
TRUNCATE	
UPDATE	
USE	