



Apache Cassandra™ 2.0

Documentation

November 23, 2013

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

Contents

Welcome to Apache Cassandra.....	7
What's new in Cassandra 2.0.....	7
CQL.....	10
Understanding the architecture.....	11
Architecture in brief.....	11
Internode communications (gossip).....	12
Configuring gossip settings.....	12
Purging gossip state on a node.....	13
Failure detection and recovery.....	13
Data distribution and replication.....	13
Consistent hashing.....	14
Virtual nodes.....	15
Data replication.....	16
Partitioners.....	17
Murmur3Partitioner.....	17
RandomPartitioner.....	17
ByteOrderedPartitioner.....	18
Snitches.....	18
Dynamic snitching.....	18
SimpleSnitch.....	19
RackInferringSnitch.....	19
PropertyFileSnitch.....	19
GossipingPropertyFileSnitch.....	20
EC2Snitch.....	20
EC2MultiRegionSnitch.....	20
Client requests.....	21
Write requests.....	21
Multiple data center write requests.....	21
Read requests.....	22
Planning a cluster deployment.....	23
Selecting hardware for enterprise implementations.....	23
Planning an Amazon EC2 cluster.....	25
Calculating usable disk capacity.....	26
Calculating user data size.....	26
Anti-patterns in Cassandra.....	27
Installing DataStax Community.....	30
Installing DataStax Community on RHEL-based systems.....	30
Installing DataStax Community on Debian-based systems.....	30
Installing DataStax Community on any Linux-based platform.....	31
Installing a Cassandra cluster on Amazon EC2.....	32
Creating an EC2 security group.....	33
Launching the DataStax Community AMI.....	34
Connecting to your DataStax Community EC2 instance.....	36
Expanding a Cassandra AMI cluster.....	38

Installing the Oracle JRE and the JNA.....	39
Installing Oracle JRE on RHEL-based Systems.....	39
Installing Oracle JRE on Debian or Ubuntu Systems.....	40
Installing the JNA on RHEL or CentOS Systems.....	41
Installing the JNA on SUSE Systems.....	41
Installing the JNA on Debian or Ubuntu Systems.....	41
Installing the JNA using the binary tarball.....	41
Recommended production settings.....	41
Upgrading.....	44
Best practices.....	44
Prerequisites.....	44
Debian or Ubuntu.....	44
RHEL or CentOS.....	45
Tarball.....	45
Completing the upgrade.....	45
Changes impacting upgrade.....	46
Initializing a cluster.....	47
Initializing a multiple node cluster (single data center).....	47
Initializing a multiple node cluster (multiple data centers).....	48
Security.....	51
Securing Cassandra.....	51
SSL encryption.....	51
Client-to-node encryption.....	51
Node-to-node encryption.....	52
Using cqlsh with SSL encryption.....	52
Preparing server certificates.....	53
Internal authentication.....	53
Internal authentication.....	54
Configuring authentication.....	54
Logging in using cqlsh.....	55
Internal authorization.....	55
Object permissions.....	55
Configuring internal authorization.....	55
Configuring firewall port access.....	56
Database internals.....	57
Managing data.....	57
Throughput and latency.....	57
Separate table directories.....	57
About writes.....	58
The role of replication.....	58
How Cassandra stores data.....	59
About index updates.....	59
About inserts and updates.....	59
The write path of an update.....	60
About deletes.....	60
About hinted handoff writes.....	60
About reads.....	62
How off-heap components affect reads.....	62
Reading a clustered row.....	63

The read path.....	63
How write patterns affect reads.....	64
How the row cache affects reads.....	64
How compaction and compression affect reads.....	64
About transactions and concurrency control.....	64
Lightweight transactions.....	65
Atomicity.....	65
Consistency.....	65
Isolation.....	66
Durability.....	66
Configuring data consistency.....	66
About schema changes.....	70
Handling schema disagreements.....	70
Configuration.....	71
The cassandra.yaml configuration file.....	71
Keyspace and table storage configuration.....	81
Keyspace attributes.....	81
Table attributes.....	82
Configuring the heap dump directory.....	87
Generating tokens.....	88
Configuring virtual nodes.....	88
Enabling virtual nodes on a new cluster.....	88
Enabling virtual nodes on an existing production cluster.....	89
Logging configuration.....	89
Logging configuration.....	89
Changing the rotation and size of the Cassandra output.log.....	90
Changing the rotation and size of the Cassandra system.log.....	90
Commit log archive configuration.....	91
Operations.....	93
Monitoring Cassandra.....	93
Monitoring a Cassandra cluster.....	93
Tuning Bloom filters.....	104
Data caching.....	104
Configuring data caches.....	104
Monitoring and adjusting caching.....	106
Configuring memtable throughput.....	107
Compaction and compression.....	107
Compaction.....	107
Compression.....	109
Testing compaction and compression.....	110
Tuning Java resources.....	110
Repairing nodes.....	112
Adding or removing a node or data center.....	113
Adding nodes an existing cluster.....	113
Adding a data center to a cluster.....	114
Replacing a dead node.....	115
Decommissioning a data center.....	115
Removing a node.....	115
Backing up and restoring data.....	117
Taking a snapshot.....	117

Deleting snapshot files.....	117
Enabling incremental backups.....	118
Restoring from a Snapshot.....	118
Node restart method.....	118
Cassandra tools.....	120
The nodetool utility.....	120
Cassandra bulk loader.....	124
The cassandra utility.....	126
The cassandra-stress tool.....	127
Options for cassandra-stress.....	128
Using the Daemon Mode.....	129
Interpreting the output of cassandra-stress.....	130
The cassandra-shuffle utility.....	130
Commands and options.....	131
The sstablescrib utility.....	131
The sstable2json / json2sstable utilities.....	132
The sstable2json utility.....	132
The json2sstable utility.....	134
sstablekeys.....	135
The sstableupgrade tool.....	135
References.....	136
Starting and stopping Cassandra.....	136
Starting Cassandra as a service.....	136
Starting Cassandra as a stand-alone process.....	136
Stopping Cassandra as a service.....	136
Stopping Cassandra as a stand-alone process.....	136
Clearing the data as a service.....	136
Clearing the data as a stand-alone process.....	136
Install locations.....	137
Locations of the configuration files.....	137
Troubleshooting.....	138
DataStax Community release notes.....	142
Glossary.....	143

Welcome to Apache Cassandra

What is Apache Cassandra?

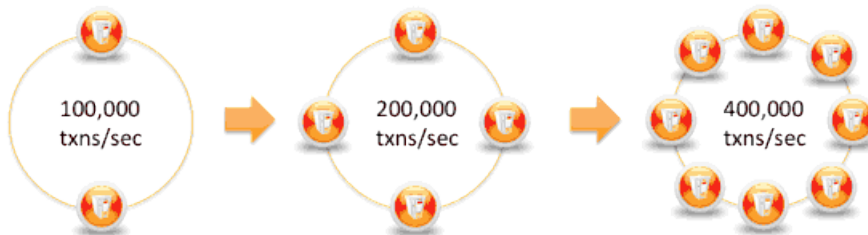
Apache Cassandra™ is a massively scalable open source NoSQL database. Cassandra is perfect for managing large amounts of structured, semi-structured, and unstructured data across multiple data centers and the cloud. Cassandra delivers continuous availability, linear scalability, and operational simplicity across many commodity servers with no single point of failure, along with a powerful dynamic data model designed for maximum flexibility and fast response times.

How does Cassandra work?

Cassandra sports a “masterless” architecture meaning all nodes are the same. Cassandra provides automatic data distribution across all nodes that participate in a “ring” or database cluster. There is nothing programmatic that a developer or administrator needs to do or code to distribute data across a cluster because data is transparently partitioned across all nodes in a cluster.

Cassandra also provides built-in and customizable replication, which stores redundant copies of data across nodes that participate in a Cassandra ring. This means that if any node in a cluster goes down, one or more copies of that node's data is available on other machines in the cluster. Replication can be configured to work across one data center, many data centers, and multiple cloud availability zones.

Cassandra supplies linear scalability, meaning that capacity may be easily added simply by adding new nodes online. For example, if 2 nodes can handle 100,000 transactions per second, 4 nodes will support 200,000 transactions/sec and 8 nodes will tackle 400,000 transactions/sec:



To gain an understanding of Cassandra's origins and where it has evolved to today, please read ["Facebook's Cassandra paper, annotated and compared to Apache Cassandra 2.0"](#), authored by Jonathan Ellis.

What's new in Cassandra 2.0

Cassandra 2.0 includes enhancements to CQL, security, and performance. The significant number of enhancements resulted in an update of the CQL specification to version 3.1.0. Key features of Cassandra 2.0 are:

- Support for **lightweight transactions**
 - Use of the **IF** keyword in CQL INSERT and UPDATE statements
 - New **SERIAL** consistency level
- **Triggers**

The first phase of support for triggers for firing an event that executes a set of programmatic logic, which runs either inside or outside a database cluster

- CQL paging support

Paging of result sets of SELECT statements executed over a **CQL native protocol 2 connection**, which eliminates the need to use the token function to page through results. For example, to page through data in this table, a simple SELECT statement after Cassandra 2.0 replaces the complex one using the token function before Cassandra 2.0.

To query this table

```
CREATE TABLE timeline (  
    user_id uuid,  
    tweet_id timeuuid,  
    tweet_author uuid,  
    tweet_body text,  
    PRIMARY KEY (userid, tweet_id)  
)
```

Before Cassandra 2.0

```
SELECT *  
FROM timeline  
WHERE (user_id = :last_key  
        AND tweet_id > :last_tweet)  
      OR token (user_id) > token (:last_key)
```

After Cassandra 2.0

```
SELECT *  
FROM timeline
```

- **Prepared statement support**

Atomic BATCH guarantees for large sets of prepared statements

- **Bind variable support**

One-shot binding of optional variables or prepared statements and variables for server-side request parsing and execution using a BATCH message containing a list of query strings--no reparsing

- Improved authentication

SASL support for easier and better authentication over prior versions of the CQL native protocol

- **Drop column support**

Re-introduction of the ALTER TABLE DROP command

- SELECT **column aliases**

Support for column aliases in a SELECT statement, similar to aliases in RDBMS SQL:

```
SELECT hdate AS hired_date  
FROM emp WHERE empid = 500
```

- Conditional DDL

Conditionally tests for the existence of a table, keyspace, or index before issuing a DROP or CREATE statement using **IF EXISTS** or **IF NOT EXISTS**

- Index enhancements

Indexing of any part, partition key or clustering columns, portion of a **compound key**

- One-off prepare and execute statements

Use of a prepared statement, even for the single execution of a query to pass binary values for a statement, for example to avoid a conversion of a blob to a string, over a native protocol version 2 connection

- Performance enhancements

- **Off-heap partition summary**
- Eager retries support

Sending the user request to other replicas before the query times out when a replica is unusually slow in delivering needed data

- Compaction improvements

Hybrid (leveled and size-tiered) compaction improvements to the leveled compaction strategy to reduce the performance overhead on read operations when compaction cannot keep pace with write-heavy workloads

Other changes in Cassandra 2.0 are:

- New commands to disable background compactions

`nodetool disableautocompaction` and `nodetool enableautocompaction`

- A change to random token selection during cluster setup

Auto_bootstrapping of a single-token node with no initial_token

- Removal of super column support

Continued support for apps that query super columns, translation of super columns on the fly into CQL constructs and results

- Removal of the cqlsh ASSUME command

Use the `blobAsType` and `typeAsBlob` conversion functions instead of ASSUME

- Cqlsh `COPY` command support for collections
- Inclusion of the native protocol version in the system.local table
- Inclusion of default_time_to_live, speculative_retry, and memtable_flush_period_in_ms in cqlsh DESCRIBE TABLE output
- Support for an empty list of values in the IN clause of SELECT, UPDATE, and DELETE commands, useful in Java Driver applications when passing empty arrays as arguments for the IN clause

CQL

Cassandra Query Language (CQL) is the default and primary interface into the Cassandra DBMS. Using CQL is similar to using SQL (Structured Query Language). The concept of a table having rows and columns is almost the same in CQL and SQL. The main difference is that Cassandra does not support joins or subqueries, except for batch analysis through Hive. Instead, Cassandra emphasizes denormalization through CQL features like **collections and clustering** specified at the schema level.

CQL is the recommended way to interact with Cassandra. The simplicity of reading and using CQL is an advantage over older Cassandra APIs.

The **CQL documentation** contains a data modeling section, examples, and reference information for each CQL command.

Understanding the architecture

Architecture in brief

An overview of Cassandra's structure.

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 **Client requests** on page 21.

Key components for configuring Cassandra

- **Gossip:** A peer-to-peer communication protocol to discover and share location and state information about the other nodes in a Cassandra cluster.
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:** 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.
You must set the **partitioner type** and assign the node a **num_tokens** value for each node. If not using virtual nodes (vnodes), use the **initial_token** setting instead.
- **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.
When you create a keyspace, you must define the **replica placement strategy** and the number of replicas you want.
- **Snitch:** A snitch defines the topology information that the replication strategy uses to place replicas and route requests efficiently.
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.
- The **cassandra.yaml** file is the main configuration file for Cassandra. In this file, you set the initialization properties for a cluster, caching parameters for tables, properties for tuning and resource utilization, timeout settings, client connections, backups, and security.

Understanding the architecture

- Cassandra stores **attributes** in the system keyspace. You set storage configuration attributes on a per-keyspace or per-table basis programmatically or using a client application, such as CQL.

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**.

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. 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.

To prevent partitions in gossip communications, use the same list of seed nodes in all nodes in a cluster. 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.

Configuring gossip settings

When a node first starts up, it looks at its `cassandra.yaml` configuration file to determine the name of the Cassandra cluster it belongs to; which nodes (called **seeds**) to contact to obtain information about the other nodes in the cluster; and other parameters for determining port and range information.

In the `cassandra.yaml` file, set the following parameters:

Property	Description
cluster_name	Name of the cluster that this node is joining. Must 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	Used in the single-node-per-token architecture, where a node owns exactly one contiguous range in the ring space.

Property	Description
num_tokens	Used for virtual nodes (vnodes). Defines the number of tokens randomly assigned to this node on the ring.

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 communications.

You may want to clear gossip history on node restart in certain cases, such as when node IP addresses have changed.

1. Edit the `cassandra-env.sh` file:
 - Packaged installs: `/usr/share/cassandra`
 - Tarball installs: `install_location/conf`
2. Add the following line to the `cassandra-env.sh` file:


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

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**.

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.

Understanding the architecture

Data is organized by table and identified by 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 you 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

Details about how the consistent hashing mechanism distributes data across a cluster in Cassandra.

Consistent hashing partitions data based on the partition key. (For an explanation of partition keys and primary keys, see the [Data modeling example](#) in *CQL for Cassandra 2.0*.)

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 partition key:

Partition 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 partition 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	Partition key	Hash value
A	-9223372036854775808	-4611686018427387903	johnny	-6723372854036780875
B	-4611686018427387904	-1	jim	-2245462676723223822
C	0	4611686018427387903	suzy	1168604627387940318

Node	Start range	End range	Partition key	Hash value
D	4611686018427387994	23372036854775307	Carol	7723358927203680754

Virtual nodes

Overview of virtual nodes (vnodes).

Vnodes simplify many tasks in Cassandra:

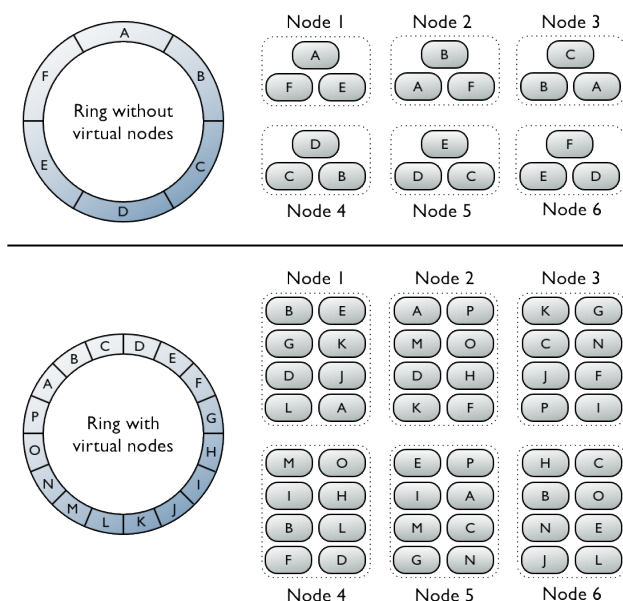
- You no longer have to calculate and assign tokens to each node.
- Rebalancing a cluster is no longer necessary when adding or removing nodes. When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster. If a node fails, the load is spread 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 vnodes to smaller and larger machines.

For more information, see the article [Virtual nodes in Cassandra 1.2](#) and [Enabling virtual nodes on an existing production cluster](#) on page 89.

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 (vnodes). Vnodes allow each node to own a large number of small **partition ranges** distributed throughout the cluster. Vnodes 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 vnodes. 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

partition 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 partition range in the ring space.

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

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 specifies 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 **NetworkTopologyStrategy**, 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 partition key. Each row of data is uniquely identified by a partition 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 [partition 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](#) on page 14.

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.

Set the partitioner in the `cassandra.yaml` file:

- `Murmur3Partitioner`: `org.apache.cassandra.dht.Murmur3Partitioner`
- `RandomPartitioner`: `org.apache.cassandra.dht.RandomPartitioner`
- `ByteOrderedPartitioner`: `org.apache.cassandra.dht.ByteOrderedPartitioner`

Note: If using virtual nodes (vnodes), you do **not** need to calculate the tokens. If not using vnodes, you **must** calculate the tokens to assign to the `initial_token` parameter in the `cassandra.yaml` file. See [Generating tokens](#) on page 88 and use the method for the type of partitioner you are using.

Murmur3Partitioner

The `Murmur3Partitioner` provides faster hashing and improved performance than the previous default partitioner (`RandomPartitioner`).

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 partition 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 query.

RandomPartitioner

The default partitioner prior to Cassandra 1.2.

Although no longer the default partitioner, you can use the `RandomPartitioner` in version 1.2, even when using virtual nodes (vnodes). However, if you don't use vnodes, 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 query.

ByteOrderedPartitioner

Use for ordered partitioning

Cassandra provides the ByteOrderedPartitioner for ordered partitioning. This partitioner orders rows lexically by key bytes. You calculate tokens by looking at the actual values of your partition 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 partition key, you can scan rows for users whose names fall between Jake and Joe. This type of query is not possible using randomly partitioned partition 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 **partition ranges** (formerly token ranges) based on their estimates of the partition 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.

Snitches

A snitch determines which data centers and racks are written to and read from.

Snitches inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into data centers and racks. All nodes must have exactly the same snitch configuration. 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.

Dynamic snitching

Monitors the performance of reads from the various replicas and chooses the best replica based on this history.

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 **Failure detection and recovery** on page 13.

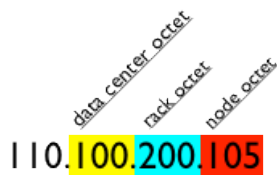
SimpleSnitch

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

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. 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 correlate 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 **Locations of the configuration files** on page 137 or **DataStax Enterprise File Locations**

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
```

Understanding the architecture

```
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 [Locations of the configuration files](#) on page 137 or [DataStax Enterprise File 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 option](#), 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 option**, use the EC2 region name, such as ``us-east``, as your data center names.

Client requests

Client read or write requests can go to any node in the cluster because all nodes in Cassandra are peers.

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.

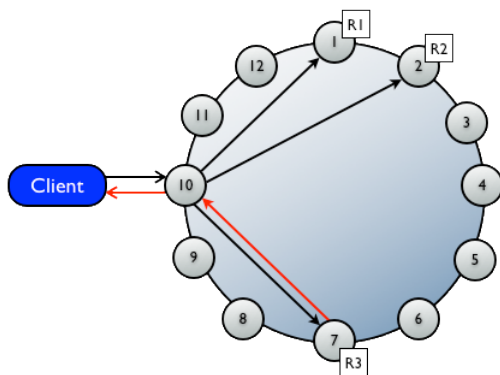
The job of the coordinator is to act as a proxy between the client application and the nodes (or replicas) that own the data being requested. The coordinator determines which nodes in the ring should get the request based on the cluster configured **partitioner** and **replica placement** strategy.

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.

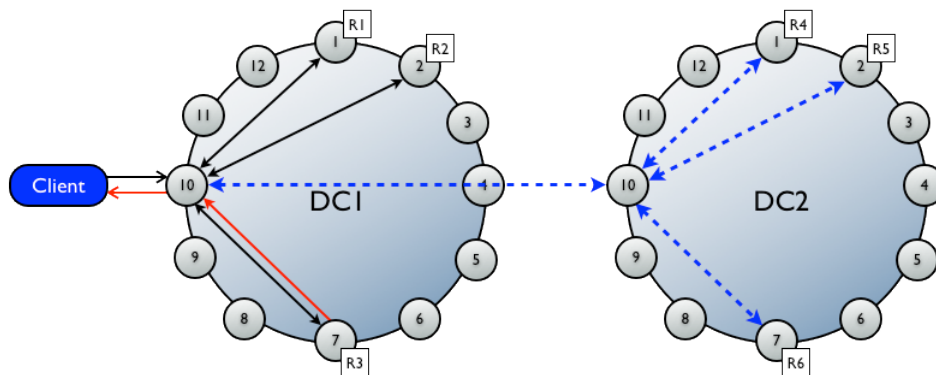
That node forwards the write to all replicas of that row. It will respond back to the client once it receives a write acknowledgment from the number of nodes specified by the consistency level. When a node writes and responds, that means it has written to the commit log and puts the mutation into a memtable.



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.



Read requests

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

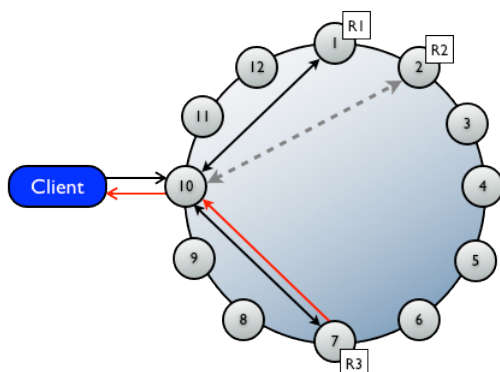
- A direct read request
- A background read repair request

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 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.

The following topics provide information for planning your cluster:

Selecting hardware for enterprise implementations

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. (All writes go to the commit log, but Cassandra is so efficient in writing that the CPU is the limiting factor.) 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](#) on page 143 for durability and when flushing [memtable](#) on page 145 to [SSTable](#) on page 146 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](#).

Recommendations:

Capacity per node

Understanding the architecture

Most workloads work best with a capacity under 500GB to 1TB per node depending on I/O. Maximum recommended capacity for Cassandra 1.2 and later is 3 to 5TB per node. For Cassandra 1.1, it is 500 to 800GB per node.

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.

Note: For SSDs it is recommended that both commit logs and SSTables are on the same mount point.

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 (vnodes), 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 that your network can handle traffic between nodes without bottlenecks. You should bind your interfaces to separate Network Interface Cards (NIC). You can use public or private depending on your requirements.

- Recommended bandwidth is 1000 Mbit/s (gigabit) or greater.
- Thrift/native protocols use the [rpc_address](#).
- Cassandra's internal storage protocol uses the [listen_address](#).

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](#) on page 56.

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:

- Only use known AMI's from a trusted source. Random AMI's pose a security risk and may perform levels slower than expected due to the way the install is configured for EC2. For example:
 - [Ubuntu Amazon EC2 AMI Locator](#)
 - [Debian AmazonEC2Image](#)
 - [CentOS-6 images on Amazon's EC2 Cloud](#)
- 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.

Understanding the architecture

- EBS volumes have unreliable performance. I/O performance can be exceptionally slow, causing the system to back load 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 [Systematic Look at EC2 I/O](#).

Calculating usable disk capacity

Determining how much data your Cassandra nodes can hold.

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.

1. Start with the raw capacity of the physical disks:

```
raw_capacity = disk_size * number_of_data_disks
```

2. Calculate the usable disk space as follows:

```
( 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.

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

```
formatted_disk_space * (0.5 to 0.8) = usable_disk_space
```

Calculating user data size

Accounting for storage overhead in determining 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.

- Determine column overhead:

```
regular_total_column_size = column_name_size + column_value_size + 15  
counter - expiring_total_column_size = column_name_size + column_value_size  
+ 23
```

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, you should add an additional 8 bytes (23 bytes total).

- Account for row overhead.

Every row in Cassandra incurs 23 bytes of overhead.

- Estimate primary key index size:

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

Every table also maintains a partition index. This estimation is in bytes.

- Determine replication overhead:

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

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.

Anti-patterns in Cassandra

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 (not maxed out)	10 ms

For information on heap sizing, see [Tuning Java resources](#) on page 110.

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.

Using the Byte Ordered Partitioner

The Byte Ordered Partitioner (BOP) is not recommended.

Use **virtual nodes** (vnodes) and use either the **Murmur3Partitioner** (default) or the **RandomPartitioner**. Vnodes allow each node to own a large number of small ranges distributed throughout the cluster. Using vnodes saves you the effort of generating tokens and assigning tokens to your nodes. If not using vnodes, 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 work flows requiring reads before writes, this small amount of latency can affect overall throughput. All write I/O in Cassandra is sequential so there is very little performance difference regardless of data size or key distribution.

Load balancers

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

Insufficient testing

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

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

Lack of familiarity with Linux

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

- Parallel SSH and Cluster SSH: The pssh and cssh tools allow SSH access to multiple nodes. This is useful for inspections and cluster wide changes.

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

Running without the recommended settings

Be sure to use the [recommended settings](#) in the Cassandra documentation.

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

More anti-patterns

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

Installing DataStax Community

Installing DataStax Community on RHEL-based systems

Install using Yum repositories on RHEL, CentOS, and Oracle Linux.

Note: To install on SUSE, use the [Cassandra binary tarball distribution](#).

For a complete list of supported platforms, see [DataStax Community – Supported Platforms](#).

- Yum Package Management application installed.
- Root or sudo access to the install machine.
- Latest version of Oracle Java SE Runtime Environment (JRE) 7. See [Installing Oracle JRE on RHEL-based Systems](#) on page 39.
- Java Native Access (JNA) is required for production installations. See [Installing the JNA on RHEL or CentOS Systems](#) on page 41.

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

In a terminal window:

1. Check which version of Java is installed by running the following command:

```
$ java -version
```

Use the latest version of Oracle Java 7 on all nodes.

2. Add the DataStax Community repository to the `/etc/yum.repos.d/datastax.repo`:

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

3. Install the packages:

```
$ sudo yum install dsc20
```

The DataStax Community distribution of Cassandra is ready for configuration.

- [Initializing a multiple node cluster \(single data center\)](#) on page 47
- [Initializing a multiple node cluster \(multiple data centers\)](#) on page 48
- [Recommended production settings](#) on page 41
- [Installing OpsCenter](#)
- [Key components for configuring Cassandra](#)

Installing DataStax Community on Debian-based systems

Install using APT repositories on Debian and Ubuntu.

For a complete list of supported platforms, see [DataStax Community – Supported Platforms](#).

- Advanced Package Tool is installed.
- Root or sudo access to the install machine.
- Latest version of Oracle Java SE Runtime Environment (JRE) 7. See [Installing Oracle JRE on Debian or Ubuntu Systems](#) on page 40.

- Java Native Access (JNA) is required for production installations. [Installing the JNA on Debian or Ubuntu Systems](#) on page 41.

Note: If you are using Ubuntu 10.04 LTS, you must update to JNA 3.4, as described in [Installing the JNA using the binary tarball](#) on page 41.

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

In a terminal window:

1. Check which version of Java is installed by running the following command:

```
$ java -version
```

Use the latest version of Oracle Java 7 on all nodes.

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

```
echo "deb http://debian.datastax.com/community stable main" | \
sudo tee -a /etc/apt/sources.list.d/cassandra.sources.list
```

3. Debian systems only:

- a) 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. For example:

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

This allows installation of the Oracle JVM instead of the OpenJDK JVM.

- b) 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 dsc20
```

This installs the DataStax Community distribution of Cassandra. 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/*
```

The DataStax Community distribution of Cassandra is ready for configuration.

- [Initializing a multiple node cluster \(single data center\)](#) on page 47
- [Initializing a multiple node cluster \(multiple data centers\)](#) on page 48
- [Recommended production settings](#) on page 41
- [Installing OpsCenter](#)
- [Key components for configuring Cassandra](#)

Installing DataStax Community on any Linux-based platform

Install on all Linux-based platforms using a binary tarball.

Use this install for Mac OS X and platforms without package support, or if you do not have or want a root installation.

For a complete list of supported platforms, see [DataStax Community – Supported Platforms](#).

- Latest version of Oracle Java SE Runtime Environment (JRE) 7. See [Installing the JRE](#).
- Java Native Access (JNA) is required for production installations. See [Installing the JNA](#).

Note: If you are using Ubuntu 10.04 LTS, you must update to JNA 3.4, as described in [Installing the JNA using the binary tarball](#) on page 41.

The binary tarball runs as a stand-alone process.

In a terminal window:

1. Check which version of Java is installed by running the following command:

```
$ java -version
```

Use the latest version of Oracle Java 7 on all nodes.

2. Download the DataStax Community:

```
$ curl -L http://downloads.datastax.com/community/dsc.tar.gz | tar xz
```

You can also download from [Planet Cassandra](#).

3. Go to the install directory:

```
$ cd dsc-cassandra-2.0.x
```

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:

```
$ 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
```

The DataStax Community distribution of Cassandra is installed and ready for configuration.

- [Initializing a multiple node cluster \(single data center\)](#) on page 47
- [Initializing a multiple node cluster \(multiple data centers\)](#) on page 48
- [Recommended production settings](#) on page 41
- [Installing OpsCenter](#)
- [Key components for configuring Cassandra](#)

Installing a Cassandra cluster on Amazon EC2

A step-by-step guide for installing the DataStax Community AMI (Amazon Machine Image).

The DataStax AMI allows you to set up a simple DataStax Community cluster using the [Amazon Web Services EC2 Management Console](#). Installing via the AMI allows you to quickly deploy a Cassandra cluster within a single availability zone. Clusters within Amazon Virtual Private Cloud (Amazon VPC) are not supported.

The AMI does the following:

- Installs the number of nodes you specify for your cluster with the latest software with 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.
- Sets the seed nodes cluster-wide.
- Installs the OpsCenter on the first node in the cluster (by default).
- Configures the Cassandra replication strategy using the EC2Snitch.

Note: 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 DataStax Community on Debian-based systems](#) on page 30, and then configure the cluster as a [multiple data center cluster](#).

To install a Cassandra cluster from the AMI, complete the following tasks:

Creating an EC2 security group

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**, under **Resources**, select **Security Groups > Create Security Group**.
2. Fill out the name and description and then click **Yes, Create**.

3. Under **Security Group**, click the **Inbound** tab and add rules for the ports:
 - a) For **Create a new rule**, select **Custom TCP rule**.
 - b) For **Port range**, see the table below.
 - c) For **Source**, see the table below.

To create rules that open a port to other nodes in the same security group, in the **Source** field, enter the letters `sg`, and then select your security group.

Table 1: Public ports

Port number	Description
22	SSH port

Port number	Description
8888	OpsCenter website. The opscenterd daemon listens on this port for HTTP requests coming directly from the browser.

Table 2: Cassandra inter-node ports

Port number	Description
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).

Table 3: Cassandra OpsCenter ports

Port number	Description
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 to connect to JMX locally or use the DataStax OpsCenter.

- After you are done adding the above port rules, 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

Caution: 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 creating the Security group, you can launch the Cassandra 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.

1. In the left panel, click **EC2 Dashboard**.
2. Click **Launch Instance**.
3. In **Step 1: Choose an Amazon Machine Image (AMI)**:
 - a) Click **Community AMIs**.
 - b) Search for **DataStax**.
 - c) Select the **DataStax Auto-Clustering AMI**.
4. In **Step 2: Choose an Instance Type**, choose the appropriate type, and then click **Next: Configure Instance Details**:
 - Extra large for production.
 - Large for development and light production.
 - Small and Medium not supported.
5. Configure the instances to suit your requirements:
 - a) Select the number of instances.
 - b) Select **Launch into EC2-Classic**. (VPC instances are not supported.)
 - c) Open **Advanced Details**.
 - d) Add the following options (as text) to the **User Data** section.

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

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

▼ Advanced Details

Kernel ID ⓘ Use default

RAM disk ID ⓘ Use default

User data ⓘ ☒ As text ☐ As file ☐ Input is already base64 encoded

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

6. Click **Next: Add Storage**, and add EBS volumes as 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. Click **Next: Tag Instance** and give a name to your DSE instance, such as `workload-dsc`.
8. Click **Next: Configure Security Group** and configure as follows:
 - a) Choose **Select an existing security group**.
 - b) Select the Security Group you created earlier.
 - c) Click **Review and Launch**.
9. In the **Step 7: Review Instance Launch** page, make any needed changes.
10. Click **Launch** and then in the **Select an existing key pair or create a new key pair** dialog, do one of the following:
 - If you need to create a new key pair, click **Choose an existing key pair** and select **Create a new key pair**. Save the key (`.pem` file) to your local machine; you will need it to log in to your DataStax Enterprise instances.
 - Select an existing key pair from the **Select a key pair** drop list and check the acknowledgment statement.
11. Click **Launch Instances**.

The **Launch Status** page is displayed.
12. Click **View Instances**.

Connecting to your DataStax Community EC2 instance

Once the cluster is launched, you can connect to it from a terminal or SSH client, such as PuTTY.

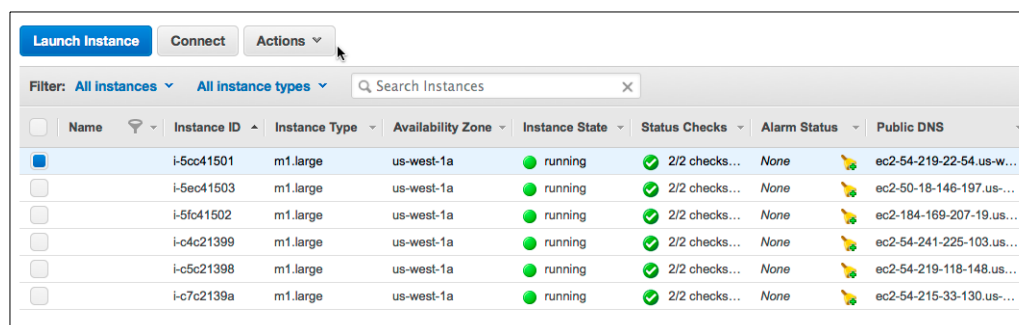
Connect as user `ubuntu` rather than as `root`.

1. 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
```

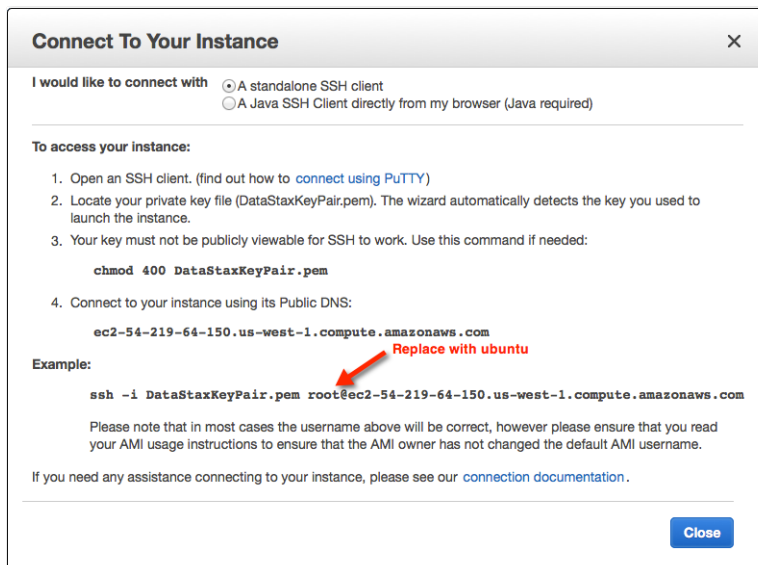
2. From the **EC2 Dashboard**, click **Running Instances**.

You can connect to any node in the cluster. However, the first node generally runs OpsCenter and is the Cassandra seed node.



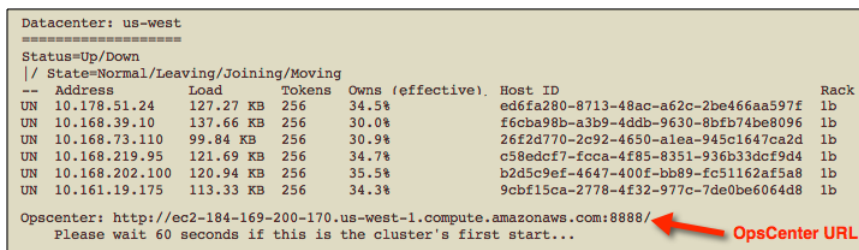
	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
<input checked="" type="checkbox"/>		i-5cc41501	m1.large	us-west-1a	running	2/2 checks...	None	ec2-54-219-22-54.us-...
<input type="checkbox"/>		i-5ec41503	m1.large	us-west-1a	running	2/2 checks...	None	ec2-50-18-146-197.us-...
<input type="checkbox"/>		i-5fc41502	m1.large	us-west-1a	running	2/2 checks...	None	ec2-184-169-207-19.us-...
<input type="checkbox"/>		i-c4c21399	m1.large	us-west-1a	running	2/2 checks...	None	ec2-54-241-225-103.us-...
<input type="checkbox"/>		i-c5c21398	m1.large	us-west-1a	running	2/2 checks...	None	ec2-54-219-118-148.us-...
<input type="checkbox"/>		i-c7c2139a	m1.large	us-west-1a	running	2/2 checks...	None	ec2-54-215-33-130.us-...

3. To get the public DNS name of a node, select the node you want to connect to, and then click **Connect**.
4. In **Connect To Your Instance**, select **A standalone SSH client**.
5. Copy the example command line and change the user from `root` to `ubuntu`, then paste it into your SSH client.



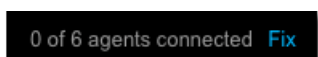
The AMI image configures your cluster and starts the Cassandra services.

6. After you have logged into a node and the AMI has completed installing and setting up the nodes, the status is displayed:



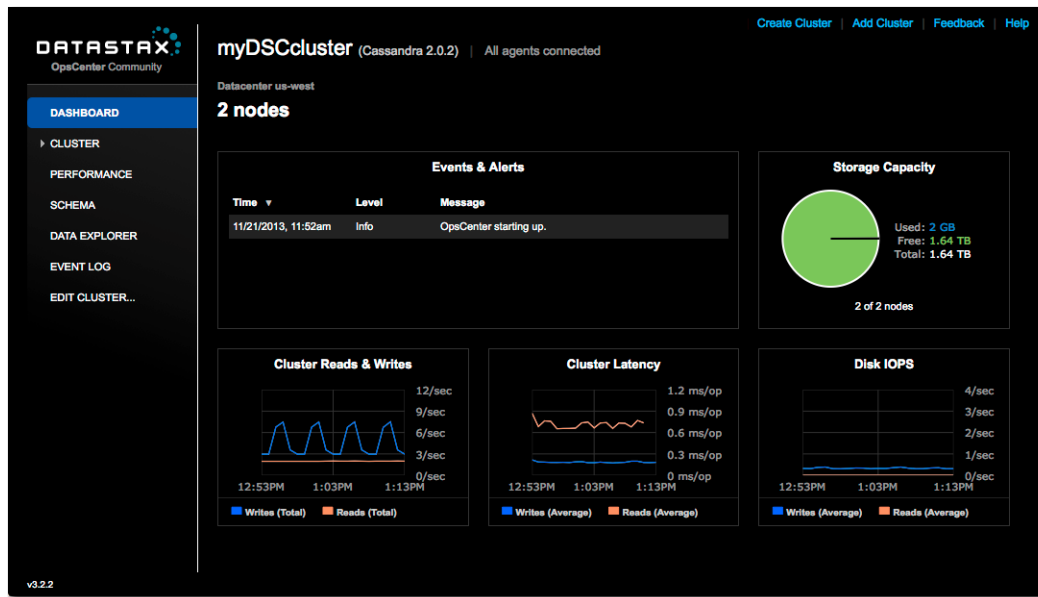
The URL for the OpsCenter is displayed when you connect to the node containing it; otherwise it is not displayed.

7. If you installed OpsCenter, 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/`
8. After the OpsCenter loads, you must install the OpsCenter agents to see the cluster performance data.
 - a) To install the agents, click the **Fix** link located near the top left of the **Dashboard**.



- 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`).

The Dashboard shows the agents are connected.



Expanding a Cassandra AMI cluster

Expanding a cluster that uses virtual nodes (created with the AMI for DataStax Community 1.2 and later).

For adding nodes to clusters that use the single-token-per-node architecture, follow the instructions in the 1.1 topic [Expanding a Cassandra AMI cluster](#).

Virtual nodes (vnodes) greatly simplify adding nodes to an existing cluster. For a detailed explanation about how this works, see [Vnodes](#).

1. In the **AWS Management Console**, create the number of nodes you need in another cluster with a temporary name, as described in [Launching the DataStax Community AMI](#).

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/*
```

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

4. For each node, set the following properties in the `cassandra.yaml` configuration 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. Set `auto_bootstrap`:

- If adding nodes to an existing data center, set `auto_bootstrap: true`.
- If adding nodes to a new data center, set `auto_bootstrap: false`.

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

```
$ sudo service cassandra start
```

You can monitor the start-up and data streaming process using `nodetool netstats`.

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

```
$ nodetool status
```

```
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
```

Each node should show `UN` (Up/Normal) not `UJ` (Up/Joining).

Installing the Oracle JRE and the JNA

Instructions for various platforms.

Installing Oracle JRE on RHEL-based Systems

You must configure your operating system to use the Oracle JRE, not the OpenJDK. The latest 64-bit version of Java 7 is recommended. The minimum supported version is 1.7.0_25.

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

```
$ export JAVA_HOME =path_to_java
```

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

```
$ java -version
```

If Oracle Java is used, the results should look like:

```
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

2. If necessary, go to [Oracle Java SE Downloads](#), accept the license agreement, and download the installer for your distribution.

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

3. From the directory where you downloaded the package, run the install:

```
$ sudo rpm -ivh jre-7uversion-linux-x64.rpm
```

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

4. Use the `alternatives` command to add a symbolic link to the Oracle JRE installation so that your system uses the Oracle JRE instead of the OpenJDK JRE:

```
$ sudo alternatives --install /usr/bin/java java /usr/java/jre1.7.0_version/bin/java 20000
```

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

Installing DataStax Community

```
$ java -version
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

6. 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-1.7.0-openjdk.x86_64/bin/java
*+ 2	/usr/java/jre1.7.0_25/bin/java

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

Installing Oracle JRE on Debian or Ubuntu Systems

You must configure your operating system to use the Oracle JRE, not the OpenJDK. The latest 64-bit version of Java 7 is recommended. The minimum supported version is 1.7.0_25.

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

```
$ export JAVA_HOME =path_to_java
```

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
```

If Oracle Java is used, the results should look like:

```
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

2. If necessary, go to [Oracle Java SE Downloads](#), accept the license agreement, and download the installer for your distribution.

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

3. Make a directory for the JRE:

```
$ sudo mkdir -p /usr/lib/jvm
```

4. Unpack the tarball and install the JRE:

```
$ sudo tar zxvf jre-7u25-linux-x64.tar.gz -C /usr/lib/jvm
```

The JRE files are installed into a directory called `/usr/lib/jvm/jre-7u_version`.

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

```
$ sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/jre1.7.0_version/bin/java" 1
```

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.

6. Set the new JRE as the default:

```
$ sudo update-alternatives --set java /usr/lib/jvm/jre1.7.0_version/bin/java
```

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

```
$ java -version
```



```
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

Installing the JNA on RHEL or CentOS Systems

Installing JNA can improve Cassandra memory usage. When installed and configured, Linux does not swap out the JVM, and thus avoids related performance issues.

Install with the following command:

```
$ sudo yum install jna
```

Installing the JNA on SUSE Systems

Installing JNA can improve Cassandra memory usage. When installed and configured, Linux does not swap out the JVM, and thus avoids related performance issues.

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
```

Installing the JNA on Debian or Ubuntu Systems

Installing JNA can improve Cassandra memory usage. When installed and configured, Linux does not swap out the JVM, and thus avoids related performance issues.

Install the JNA with the following command:

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

For Ubuntu 10.04 LTS, update to JNA 3.4 as follows:

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
```

Installing the JNA using the binary tarball

Installing JNA can improve Cassandra memory usage. When installed and configured, Linux does not swap out the JVM, and thus avoids related performance issues.

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

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

Recommended production settings

Recommendations for production environments; adjust them accordingly for your implementation.

User resource limits

You can view the current limits using the `ulimit -a` command. Although limits can also be temporarily set using this command, DataStax recommends making the changes permanent:

Packaged installs: Ensure that the following settings are included in the `/etc/security/limits.d/cassandra.conf` file:

```
cassandra - memlock unlimited
cassandra - nofile 100000
cassandra - nproc 32768
cassandra - as unlimited
```

Tarball installs: Ensure that the following settings are included in the `/etc/security/limits.conf` file:

```
* - memlock unlimited
* - nofile 100000
* - nproc 32768
* - as unlimited
```

If you run Cassandra as root, some Linux distributions such as Ubuntu, require setting the limits for root explicitly instead of using `*`:

```
root - memlock unlimited
root - nofile 100000
root - nproc 32768
root - as unlimited
```

For CentOS, RHEL, OEL systems, also set the nproc limits in `/etc/security/limits.d/90-nproc.conf`:

```
* - nproc 32768
```

For all installations, add the following line to `/etc/sysctl.conf`:

```
vm.max_map_count = 131072
```

To make the changes take effect, reboot the server or run the following command:

```
$ sudo sysctl -p
```

To confirm the limits are applied to the Cassandra process, run the following command where `<pid>` is the process ID of the currently running Cassandra process:

```
$ cat /proc/<pid>/limits
```

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
```

To make this change permanent, remove all swap file entries from `/etc/fstab`.

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 7 is recommended, not the OpenJDK.

Java Native Access

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

Upgrading

Upgrade to Cassandra 1.2.9 or later before upgrading to 2.0. Cassandra 2.0 is not network- or SSTable-compatible with versions older than 1.2.9. If you want to perform a **rolling restart**, first upgrade the entire cluster to 1.2.9, and then to 2.0

Best practices

Follow these best practices before upgrading:

- 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 NEWS.txt for any new information about upgrading.
News.txt is on the [Apache Cassandra github site](#).
- Familiarize yourself with changes and fixes in this release.
A complete list is available in **CHANGES.txt**.

Prerequisites

- Remove dead nodes.
Do not upgrade if nodes in the cluster are down. Use the **nodetool removename** command, which was called **nodetool removetoken** in earlier releases, to remove dead nodes.
- In the old cassandra.yaml, check the value of **index_interval**, which in Cassandra 2.0 is a table property instead of a cassandra.yaml configuration option.
Ensure that the value of **index_interval** is the value you want applied to your tables. During the upgrade, the value defined in the old cassandra.yaml is applied as the default property for existing tables. You can alter this property after the upgrade using CQL.
- If your cluster does not use vnodes, in each new cassandra.yaml, disable vnodes before doing the rolling restart.
In Cassandra 2.0, vnodes are enabled by default and must be disabled before upgrading to 2.0.
- Check the **product changes** that impact upgrading and take recommended actions.

Debian or Ubuntu

Follow these steps to get the new version, merge your customizations of the old cassandra.yaml file to the new one, and then complete the upgrade.

1. Save the cassandra.yaml file from the old installation to a safe place.
2. On each of your Cassandra nodes, install the new version.

```
sudo apt-get install dsc20
```
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.
Do not use the default partitioner setting in the new cassandra.yaml 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.

5. Save the file as `cassandra.yaml`.
6. Follow steps for [completing the upgrade](#).

RHEL or CentOS

Follow these steps to remove the old installation, merge your customizations of the old `cassandra.yaml` file to the new one, and then complete the upgrade.

1. On each Cassandra node, remove the old installation. For example:

```
sudo yum remove apache-cassandra11
```

2. Install the new version.

```
sudo yum install dsc20
```

The installer creates the file `cassandra.yaml.rpmnew` in `/etc/cassandra/default.conf/`.

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.
Unless your old release uses the `Murmur3Partitioner`, do not use the default partitioner setting in the new `cassandra.yaml`, `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.
5. Save the file as `cassandra.yaml`.
6. Follow steps for [completing the upgrade](#).

Tarball

Follow these steps to download and unpack the binary tarball, merge your customizations of the old `cassandra.yaml` file into the new one, and then complete the upgrade.

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.
Do not use the default partitioner setting in the new `cassandra.yaml` 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](#).

Completing the upgrade

Regardless of your operating system, you perform these steps to complete the upgrade process.

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.
7. After upgrading all nodes in the cluster, consider upgrading existing nodes to vnodes.

Upgrading to vnodes is optional but has a **number of important advantages**. See **Enabling virtual nodes on an existing production cluster** on page 89.

Changes impacting upgrade

Changes that can affect upgrading to Cassandra 2.0.x are:

- CAS and new features in CQL such as DROP COLUMN assume that cell timestamps are microseconds-since-epoch.

Do not use these features if you are using client-specified timestamps with some other source.

- **Replication and strategy options** do not accept unknown options any longer.
- Cassandra has a new version of CQL (and cqlsh) based on the CQL specification 3.1.0.
- Use lowercase property map keys in ALTER and CREATE statements.

In earlier releases, CQL 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. Lowercase property map keys are shown in this example:

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

- You might need to fix queries having loose type validation of CQL constants that now have strong validation.

See the changelog section of <http://cassandra.apache.org/doc/cql3/CQL.html>. Using blobs as strings constants is deprecated in favor of blob constants.

- auto_bootstrap of a single-token node with no initial_token now picks a random token instead of bisecting an existing token range.

Upgrading to vnodes is recommended; otherwise, specify an initial token.

- reduce_cache_sizes_at, reduce_cache_capacity_to, and flush_largest_memtables_at options have been removed from cassandra.yaml
- CacheServiceMBean.reduceCacheSizes() has been removed. Use CacheServiceMBean.set{Key,Row}CacheCapacityInMB() instead.
- authority option in cassandra.yaml has been deprecated since 1.2.0, but it has been completely removed in 2.0. Please use 'authorizer' option.
- index_interval is now a CQL table property. You can change the value of index_interval using ALTER TABLE after which SSTables will be written to disk with new value.

When upgrading, Cassandra will pick up the value defined in cassandra.yaml as the default for existing tables, until you explicitly set the value for those.

- The deprecated native_transport_min_threads option has been removed in Cassandra.yaml.
- vnodes are enabled by default in the 2.0 cassandra.yaml and must be disabled when upgrading clusters that do not use vnodes.

Initializing a cluster

Initializing a multiple node cluster (single data center)

You can use initialize a Cassandra cluster with a single data center.

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 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 [The cassandra.yaml configuration file](#) on page 71.

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

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.

1. Suppose you install Cassandra on these nodes:

```
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 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](#) on page 56.
3. If Cassandra is running:

Packaged installs:

- a) Stop Cassandra:

```
$ sudo service cassandra stop
```

- b) Clear the data:

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

Tarball installs:

- a) Stop Cassandra:

```
$ ps aux | grep cassandra
$ sudo kill <pid>
```

- b) Clear the data:

```
$ cd install_location
$ sudo rm -rf /var/lib/cassandra/*
```

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

- `num_tokens`: <recommended value: 256>
- `-seeds`: <internal IP address of each seed node>
- `listen_address`: <localhost IP address>
- `endpoint_snitch`: <name of snitch> (See [endpoint_snitch](#).)
- `auto_bootstrap`: false (Add this setting **only** when initializing a fresh cluster with no data.)

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.

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

For packaged installs, run the following command:

```
$ sudo service cassandra start
```

For binary installs, run the following commands:

```
$ cd install_location
$ bin/cassandra
```

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
```

Initializing a multiple node cluster (multiple data centers)

You can use initialize a Cassandra cluster with multiple 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.

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.

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 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 **The `cassandra.yaml` configuration file** on page 71.

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** on page 56.

3. If Cassandra is running:

Packaged installs:

a) Stop Cassandra:

```
$ sudo service cassandra stop
```

b) Clear the data:

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

Tarball installs:

a) Stop Cassandra:

```
$ ps aux | grep cassandra
$ sudo kill <pid>
```

b) Clear the data:

```
$ cd install_location
$ sudo rm -rf /var/lib/cassandra/*
```

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

- `num_tokens`: <recommended value: 256>
- `-seeds`: <internal IP address of each seed node>
- `listen_address`: <localhost IP address>
- `endpoint_snitch`: <name of snitch> (See **endpoint_snitch**.)
- `auto_bootstrap`: false (Add this setting **only** when initializing a fresh cluster with no data.)

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.

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

For packaged installs, run the following command:

```
$ sudo service cassandra start
```

For binary installs, run the following commands:

```
$ cd install_location
$ bin/cassandra
```

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
```

Security

Securing Cassandra

Cassandra provides these security features to the open source community.

- **Client-to-node encryption**

Cassandra includes an optional, secure form of communication from a client machine to a database cluster. Client to server SSL ensures data in flight is not compromised and is securely transferred back/forth from client machines.

- **Authentication based on internally controlled login accounts/passwords**

Administrators can create users who can be authenticated to Cassandra database clusters using the CREATE USER command. Internally, Cassandra manages user accounts and access to the database cluster using passwords. User accounts may be altered and dropped using the **Cassandra Query Language** (CQL).

- **Object permission management**

Once authenticated into a database cluster using either internal authentication, the next security issue to be tackled is permission management. What can the user do inside the database? Authorization capabilities for Cassandra use the familiar GRANT/REVOKE security paradigm to manage object permissions.

SSL encryption

Client-to-node encryption

Client-to-node encryption protects data in flight from client machines to a database cluster using SSL (Secure Sockets Layer). It establishes a secure channel between the client and the coordinator node.

All nodes must have all the relevant SSL certificates on all nodes. See **Preparing server certificates** on page 53.

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>
```

Node-to-node encryption

Node-to-node encryption protects data transferred between nodes in a cluster using SSL (Secure Sockets Layer).

All nodes must have all the relevant SSL certificates on all nodes. See [Preparing server certificates](#) on page 53.

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 `internode_encryption`.

The available options are:

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

```
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>
```

Using cqlsh with SSL encryption

Using a `.cqlshrc` file means you don't have to override the `SSL_CERTFILE` environmental variables every time.

To run `cqlsh`, you must create a `.cqlshrc` file in your home or client program directory. 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`
- Binary installs: `install_location/conf`

```
[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
```

Note:

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.

Preparing server certificates

Generate 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.

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.

```
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

Add new users when client certificate authentication is enabled.

The client certificate authentication must be enabled (`require_client_auth=true`).

1. Generate the certificate as described in [Client-to-node encryption](#) on page 51.
2. Import the user's certificate into every node's truststore using keytool:

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

Internal authentication

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` table.

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.

Limitations

The `dsetool` and Hadoop utilities are not supported by internal authentication.

Configuring authentication

To configure Cassandra to use internal authentication, first make a change to the `cassandra.yaml` file and increase the replication factor of the `system_auth` keyspace, as described in this procedure. Next, start up Cassandra using the default user name and password (`cassandra/cassandra`), and start `cqlsh` using the same credentials. Finally, use these CQL 3 statements to set up user accounts to authorize users to access the database objects:

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

1. Change the authenticator option in the `cassandra.yaml` to PasswordAuthenticator.

By default, the authenticator option is set to AllowAllAuthenticator.

```
authenticator: PasswordAuthenticator
```

2. Increase the **replication factor for the system_auth** keyspace.

3. Restart the Cassandra client.

The default **superuser** name and password that you use to start the client is stored in Cassandra.

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

4. Start `cqlsh` using the superuser name and password.

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

5. Create another superuser, not named `cassandra`. This step is optional but highly recommended.
6. Log in as that new superuser.
7. Change the `cassandra` user password to something long and incomprehensible, and then forget about it. It won't be used again.
8. Take away the `cassandra` user's superuser status.
9. Use the CQL 3 statements listed previously to set up user accounts and then grant permissions to access the database objects.

Logging in using cqlsh

To avoid having to pass credentials for every login using cqlsh, you can create a .cqlshrc file your home directory. When present, this file passes default login information to cqlsh.

1. Open a text editor and create a file that specifies a user name and password.

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

2. Save the file in your home directory and name it .cqlshrc
3. Set permissions on the file.

To protect database login information, ensure that the file is secure from unauthorized access.

Note: Sample .cqlshrc files are available in:

- Packaged installs
/usr/share/doc/dse-libcassandra
- Binary installs
install_location/resources/cassandra/conf

Internal authorization

Object permissions

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. Object permission management is based on internal authorization.

Read access to these system tables 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

Configuring internal authorization

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

1. In the cassandra.yaml, comment out the default AllowAllAuthorizer and add the CassandraAuthorizer.

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

You can use any authenticator except AllowAll.

2. Configure the **replication factor** for the system_auth keyspace.
3. Adjust the validity period for permissions caching by setting the **permissions_validity_in_ms** option in the cassandra.yaml.

Alternatively, disable permission caching by setting this option to 0.

CQL supports these authorization statements:

- GRANT
- LIST PERMISSIONS
- REVOKE

Configuring firewall port access

Which ports to open when nodes are protected by a firewall.

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.

Table 4: Public ports

Port number	Description
22	SSH port
8888	OpsCenter website. The opscenterd daemon listens on this port for HTTP requests coming directly from the browser.

Table 5: Cassandra inter-node ports

Port number	Description
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).

Table 6: Cassandra OpsCenter ports

Port number	Description
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.

Database internals

Managing data

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 contiguously. Operations are parallel across 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 writes

To manage and access data in Cassandra, it is important to understand how Cassandra writes and reads data, the hinted handoff feature, areas of conformance and non-conformance to the ACID (atomic, consistent, isolated, durable) database properties. In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replicas.

Cassandra includes client utilities and application programming interfaces (APIs) for developing applications for data storage and retrieval.

The role of replication

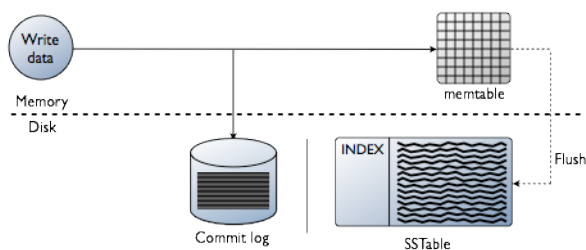
Cassandra delivers high availability for writing through its data replication strategy. Cassandra duplicates data on multiple peer nodes to ensure reliability and fault tolerance. Relational databases, on the other hand, typically structure tables to keep data duplication at a minimum. The relational 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 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:

- Partition index
A list of primary keys and the start position of rows in the data file.
- Partition summary
A subset of the partition index. By default 1 primary key out of every 128 is sampled.

How Cassandra stores data

In the memtable, data is organized in sorted order.

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:

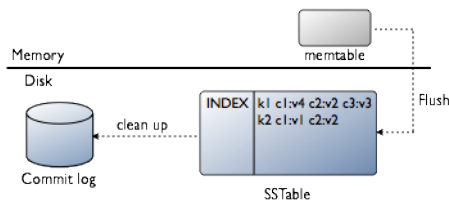
```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

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 index updates

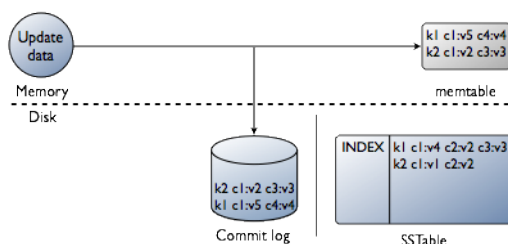
To update an index Cassandra appends data to the commit log, updates the memtable, and updates the index. Writing to a table having an index involves more work than writing to a table without an index, but the update process has been improved in Cassandra 1.2. The need for a synchronization lock to prevent concurrency issues for heavy insert loads has been removed.

When a column is updated, the 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

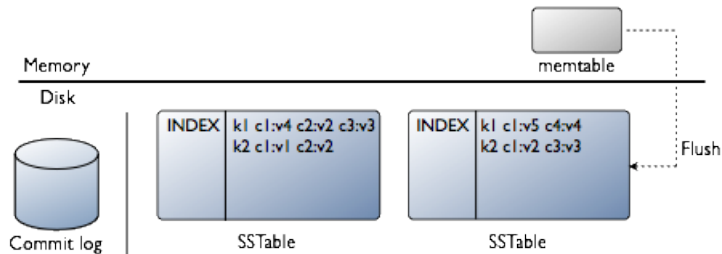
Insert and update operations are identical. As inserts/updates come in, Cassandra does not overwrite the rows in place, but instead groups inserts/updates in the **memtable**.



Any number of columns can be inserted/updated at the same time. When inserting or updating columns in a table, the client application identifies which column records to change.

The write path of an update

Inserting a duplicate primary 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).

About hinted handoff writes

Hinted handoff is a Cassandra feature that ensures high write availability when **consistency** is not required. Hinted handoff dramatically improves response consistency after temporary outages such as network failures. You **enable or disable hinted handoff** in the `cassandra.yaml` file.

How hinted handoff works

When a write is performed and a replica node for the row is either known to be down ahead of time, or does not respond to the write request, the coordinator will store a hint locally in the `system.hints` table. This hint indicates that the write needs to be replayed to the unavailable node(s).

The hint consists of:

- The location of the replica that is down
- The row that requires a replay
- The actual data being written

By default, hints are 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, run a repair to re-replicate the data before the failure occurred. You can configure this interval of time using the `max_hint_window_in_ms` property in the `cassandra.yaml` file.

After a node discovers from `gossip` that a node for which it holds hints has recovered, the node sends the data row corresponding to each hint to the target. Additionally, the node checks every ten minutes for any hints for writes that timed out during an outage too brief for the failure detector to notice through gossip.

A hinted write does not count toward `consistency level` requirements of ONE, QUORUM, or ALL. The coordinator node stores hints for dead replicas regardless of consistency level unless hinted handoff is disabled. If insufficient replica targets are alive to satisfy a requested consistency level, an `UnavailableException` is thrown with or without hinted handoff. This is an important difference from `Dynamo's` replication model; Cassandra does not default to sloppy quorum.

For example, in a cluster of two nodes, A and B, having a replication factor (RF) of 1: each row is stored on one node. Suppose node A is down while we write row K to it with consistency level of one. The write fails because reads always reflect the most recent write when:

$W\text{-nodes} + R > \text{replication factor}$

where W is the number of nodes to block for writes and R is the number of nodes to block for reads. Cassandra does not write a hint to B and call the write good because Cassandra cannot read the data at any consistency level until A comes back up and B forwards the data to A.

Extreme write availability

For applications that want Cassandra to accept writes even when all the normal replicas are down, when not even consistency level ONE can be satisfied, Cassandra provides consistency level ANY. ANY guarantees that the write is durable and will be readable after an appropriate replica target becomes available and receives the hint replay.

Performance

By design, hinted handoff inherently forces Cassandra to continue performing the same number of writes even when the cluster is operating at reduced capacity. Pushing your cluster to maximum capacity with no allowance for failures is a bad idea. Hinted handoff is designed to minimize the extra load on the cluster.

All hints for a given replica are stored under a single `partition key`, so replaying hints is a simple sequential read with minimal performance impact.

If a replica node is overloaded or unavailable, and the failure detector has not yet marked it down, then expect most or all writes to that node to fail after the timeout triggered by `write_request_timeout_in_ms`, which defaults to 10 seconds. During that time, Cassandra writes the hint when the timeout is reached.

If this happens on many nodes at once this could become substantial memory pressure on the coordinator. So the coordinator tracks how many hints it is currently writing, and if this number gets too high it will temporarily refuse writes (`withUnavailableException`) whose replicas include the misbehaving nodes.

Removal of hints

When removing a node from the cluster by decommissioning the node or by using the `nodetool removenode` command, Cassandra automatically removes hints targeting the node that no longer exists. Cassandra also removes hints for dropped tables.

Scheduling repair weekly

At first glance, it may appear that hinted handoff lets you safely get away without needing repair. This is only true if you never have hardware failure. Hardware failure has the following ramifications:

- Loss of historical data about which writes have finished. There is no information about what data has gone missing to convey to the rest of the cluster.
- Loss of hints-not-yet-replayed from requests that the failed node coordinated.

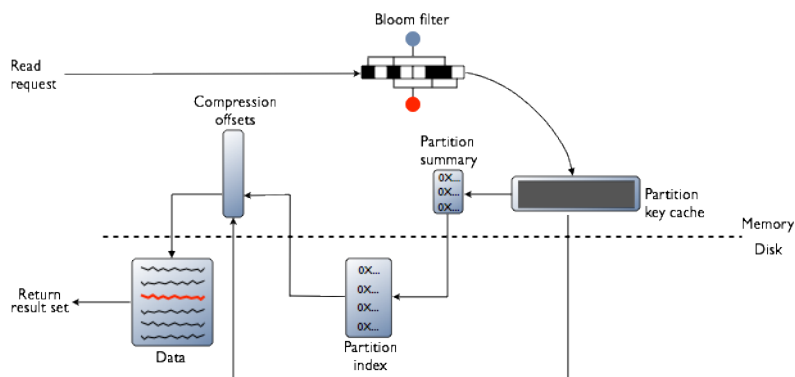
About reads

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 partition 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 the probability of having any data for the requested partition key in the SSTable before doing any disk I/O.

If the probability is good, Cassandra checks the **partition key cache** and takes one of these courses of action:

- If an index entry is found in the cache:
 - Cassandra goes to the compression offset map to find the compressed block having the data.
 - Fetches the compressed data on disk and returns the result set.
- If an index entry is not found in the cache:
 - Cassandra searches the **partition summary** to determine the approximate location on disk of the index entry.
 - Next, to fetch the index entry, Cassandra hits the disk for the first time, performing a single seek and a sequential read of columns (a range read) in the SSTable if the columns are contiguous.
 - Cassandra goes to the compression offset map to find the compressed block having the data.
 - Fetches the compressed data on disk and returns the result set.



How off-heap components affect reads

In Cassandra 1.2 and later, the Bloom filter and compression offset map, shown in the previous figure, are off-heap to increase the data handling capacity per node. In Cassandra 2.0 and later, the partition

summary is also off-heap. Of the components in memory, only the partition key cache is a fixed size. Other components grow as the data set grows.

- The Bloom filter grows to approximately 1-2 GB per billion partitions. In the extreme case, you can have one partition per row, so you can easily have billions of these entries on a single machine. The Bloom filter is tunable if you want to trade memory for performance.
- By default, the partition summary is a sample of the partition index. You configure sample frequency by changing the `index_interval` property in the **table definition**. You can probably increase the `index_interval` to 512 without seeing degradation. Cassandra 1.2.5 reduced the size of the partition summary by using raw longs instead of boxed numbers inside jvm.
- The compression offset map grows to 1-3 GB per terabyte compressed. The more you compress data, the greater number of compressed blocks you have and the larger the compression offset table.

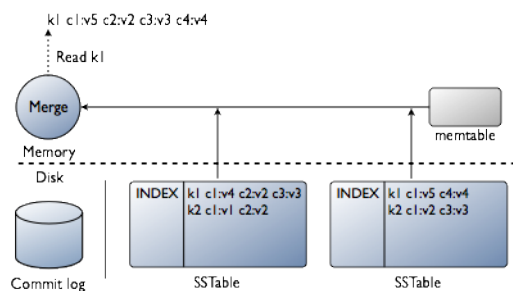
Compression is enabled by default even though going through the compression offset map consumes CPU resources. Having compression enabled makes the page cache more effective, and typically, almost always pays off.

Reading a clustered row

Using a CQL 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.

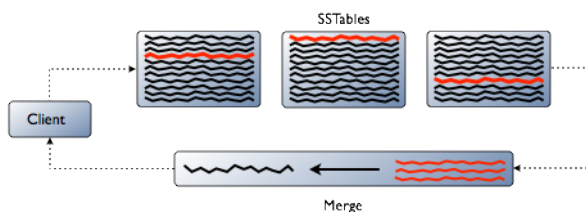
The read path

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 of 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.



Database internals

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 more information about client read and write requests, see the ["Client requests" section](#).

How write patterns affect reads

The type of [compaction strategy](#) Cassandra performs on your data is configurable and can significantly affect read performance. 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 partition 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 use RDBMS ACID transactions with rollback or locking mechanisms, but instead offers atomic, isolated, and durable transactions with eventual/tunable consistency that lets the user decide how strong or eventual they want each transaction's consistency to be.

- 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.

Lightweight transactions

While durable transactions with eventual/tunable consistency is quite satisfactory for many use cases, situations do arise where more is needed. Lightweight transactions, also known as compare and set, that use **linearizable consistency** can probably fulfill those needs.

For example, two users attempting to create a unique user account in the same cluster could overwrite each other's work with neither user knowing about it. To avoid this situation, lightweight transactions (or 'compare and set') have been added to version 2.0 of Cassandra.

Using and extending the Paxos consensus protocol (which allows a distributed system to agree on proposed data additions/modifications with a quorum-based algorithm, and without the need for any one 'master' database or two-phase commit), Cassandra now offers a way to ensure a transaction isolation level similar to the serializable level offered by RDBMS's. Extensions to CQL enable an easy way to carry out such operations.

A new IF clause has been introduced for both the INSERT and UPDATE commands that lets the user invoke lightweight transactions. For example, if a user wants to ensure an insert they are about to make into a new accounts table is unique for a new customer, they would use the IF NOT EXISTS clause:

```
INSERT INTO customer_account (customerID, customer_email)
VALUES ('LauraS', 'lauras@gmail.com')
IF NOT EXISTS;
```

DML modifications via UPDATE can also make use of the new IF clause by comparing one or more columns to various values:

```
UPDATE customer_account
SET customer_email='laurass@gmail.com'
IF customer_email='lauras@gmail.com';
```

Behind the scenes, Cassandra is making four round trips between a node proposing a lightweight transaction and any needed replicas in the cluster to ensure proper execution so performance is affected. Consequently, reserve lightweight transactions for those situations where they are absolutely necessary; Cassandra's normal eventual consistency can be used for everything else.

A **SERIAL consistency level** allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit it as part of the read.

Atomicity

In Cassandra, a write is atomic at the row-level, meaning inserting or updating columns in a row is 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 replicate the write to all nodes in the cluster and wait for acknowledgement from two nodes. If the write fails on one of the nodes but succeeds on the other, Cassandra reports a failure to replicate the write on that node. However, the replicated write that succeeds on the other node is not automatically rolled back.

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.

Consistency

Cassandra 2.0 offers two types of consistency:

- **Tuneable consistency**

Availability and consistency can be tuned, and can be strong in the **CAP** sense--data is made consistent across all the nodes in a distributed database cluster.

- Linearizable consistency

In ACID terms, linearizable consistency is a serial (immediate) isolation level for lightweight (compare-and-set, CAS) transactions.

In Cassandra, there are no locking or transactional dependencies when concurrently updating multiple rows or tables. Tuning availability and consistency always gives you partition tolerance. 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.

Linearizable consistency is used in rare cases when a strong version of tunable consistency in a distributed, masterless Cassandra with quorum reads and writes is not enough. Such cases might be encountered when performing uninterrupted sequential operations or when producing the same results when running an operation concurrently or not. For example, an application that registers new accounts needs to ensure that only one user can claim a given account. The challenge is handling a race condition analogous to two threads attempting to make an insertion into a non-concurrent Map. Checking for the existence of the account before performing the insert in thread A does not guarantee that thread X will not insert the account between the check time and A's insert. Linearizable consistency meets these challenges.

Cassandra 2.0 uses the Paxos consensus protocol, which resembles 2-phase commit, to support linearizable consistency. All operations are quorum-based and updates will incur a performance hit, effectively a degradation to one-third of normal. For in-depth information about this new consistency level, see the article, *Lightweight transactions in Cassandra*.

To support linearizable consistency, a **consistency level of SERIAL** has been added to Cassandra. **Additions to CQL** have been made to support lightweight transactions.

Isolation

Prior early versions of Cassandra, 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 in place, which means 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.

Durability

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.

Configuring data consistency

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

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

Tunable consistency for client requests

Consistency levels in Cassandra can be configured to manage response time versus data accuracy. You can configure consistency on a cluster, data center, or individual I/O operation basis. Very strong or eventual consistency among 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

The consistency level specifies the number of replicas on which the write must succeed before returning an acknowledgment to the client application. In Cassandra 2.0 and later, the SERIAL consistency level is available for use with **linearizable consistency**.

Table 7: Write Consistency Levels

Level	Description	Usage
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 after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that row have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability compared to other levels.
ONE	A write must be written to the commit log and memory table of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent. The replica node closest to the coordinator node that received the request serves the request (unless the dynamic snitch determines that the node is performing poorly and routes it elsewhere).
TWO	A write must be written to the commit log and memory table of at least two replica nodes.	Similar to ONE.
THREE	A write must be written to the commit log and memory table of at least three replica nodes.	Similar to TWO.
QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes.	Provides strong consistency if you can tolerate some level of failure.
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.	Used in multiple data center clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy. Use to maintain consistency at locally (within the single data center) .

Level	Description	Usage
LOCAL_ONE	Available in Cassandra 1.2.11 and 2.0.2 and later. A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.	In a multiple data center clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other data centers if an offline node goes down.
EACH_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in <i>all</i> data centers.	Used in multiple data center clusters to strictly maintain consistency at the same level in each data center. For example, choose this level if you want a read to fail when a data center is down and the QUORUM cannot be reached on that data center.
ALL	A write must be written to the commit log and memory table on all replica nodes in the cluster for that row.	Provides the highest consistency and the lowest availability of any other level.
SERIAL	A write must be written to the commit log and memory table on a quorum of replica nodes.	Used to achieve linearizable consistency for lightweight transactions.

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.

About read consistency

The consistency level specifies how many replicas must respond to a read request before returning data to the client application.

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

Table 8: Read Consistency Levels

Level	Description	Usage
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.	Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.
TWO	Returns the most recent data from two of the closest replicas.	Similar to ONE.

Level	Description	Usage
THREE	Returns the most recent data from three of the closest replicas.	Similar to TWO.
QUORUM	Returns the record with the most recent timestamp after a quorum of replicas has responded.	Ensures strong consistency if you can tolerate some level of failure.
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.	Used in multiple data center clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy.
LOCAL_ONE	Available in Cassandra 1.2.11 and 2.0.2 and later. Returns a response from the closest replica, as determined by the snitch, but only if the replica is in the local data center.	Same usage as described in the table about write consistency levels.
EACH_QUORUM	Returns the record with the most recent timestamp once a quorum of replicas in each data center of the cluster has responded.	Same as LOCAL_QUORUM
ALL	Returns the record with the most recent timestamp after all replicas have responded. The read operation will fail if a replica does not respond.	Provides the highest consistency of all levels and the lowest availability of all levels.

About the QUORUM levels

The QUORUM level writes to the number of nodes that make up a quorum. A quorum is calculated, and then rounded down to a whole number, as follows:

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

For example, using a replication factor of 3, a quorum is 2 nodes—the cluster can tolerate 1 replica down. Using a replication factor of 6, a quorum is 4—the cluster can tolerate 2 replicas down.

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

$$(\text{nodes_written} + \text{nodes_read}) > \text{replication_factor}$$

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

Configuring 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 commands. You set the consistency level programmatically (at the driver level). For example, call `QueryBuilder.insertInto` with a `setConsistencyLevel` argument. The consistency level defaults to ONE for all write and read operations.

About built-in consistency repair features

You can use these built-in repair utilities to ensure that data remains consistent across replicas.

- [Read repair](#)
- [Hinted handoff](#)
- [Anti-entropy node repair](#)

About schema changes

In Cassandra 1.2 and later, 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 other action will be instantly agreed upon. Which action is agreed upon 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.

Handling schema disagreements

In the event that a schema disagreement occurs, check for and resolve schema disagreements as follows:

1. Using the Command Line Interface (CLI), run the DESCRIBE CLUSTER command.

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

```
[default@unknown] DESCRIBE cluster;
```

If any node is UNREACHABLE, you see output something like this:

```
[default@unknown] describe cluster;
Cluster Information:
Snitch: com.datastax.bdp.snitch.DseDelegateSnitch
Partitioner: org.apache.cassandra.dht.RandomPartitioner
Schema versions:
UNREACHABLE: [10.202.205.203, 10.80.207.102, 10.116.138.23]
```

2. Restart unreachable nodes.
3. Repeat steps 1 and 2 until the DESCRIBE cluster command shows that all nodes have the same schema version number—only one schema version appears in the output of DESCRIBE cluster.

Configuration

The `cassandra.yaml` configuration file

The `cassandra.yaml` file is the main configuration file for Cassandra.

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.

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

The configuration properties are grouped into the following sections:

- **Initialization properties**
Controls how a node is configured within a cluster, including inter-node communication, data partitioning, and replica placement.
- **Global row and key caches properties**
Caching parameters for tables.
- **Performance tuning properties**
Tuning performance and system resource utilization, including memory, disk I/O, CPU, reads, and writes.
- **Binary and RPC protocol timeout properties**
Timeout settings for the binary protocol.
- **Remote procedure call tuning (RPC) properties**
Settings for configuring and tuning RPCs (client connections).
- **Fault detection properties**
Settings to handle poorly performing or failing nodes.
- **Automatic backup properties**
Automatic backup settings.
- **Security properties**
Server and client security settings.

Initialization properties

Controls how a node is configured within a cluster, including inter-node communication, data partitioning, and replica placement.

Note: It is recommended that you carefully evaluate your requirements and make any changes before starting a node for the first time.

auto_bootstrap

Configuration

(Default: true) This setting has been removed from default configuration. It makes new (non-seed) nodes automatically migrate the right data to themselves. When initializing a fresh cluster with no data, add `auto_bootstrap: false`.

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 [Snitches](#) on page 18.

initial_token

(Default: disabled) Used in the single-node-per-token architecture, where a node owns exactly one contiguous range in the ring space. 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 [Cassandra 1.1 Node and Cluster Configuration](#) documentation.

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: 256) Used for [virtual nodes](#) (vnodes). 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 [The cassandra-shuffle utility](#) on page 130.

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). See [Initializing a multiple node cluster \(single data center\)](#) on page 47 and [Initializing a multiple node cluster \(multiple data centers\)](#) on page 48.

start_native_transport

(Default: `false`) Enable or disable the native transport server. 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`. "Currently, only the Thrift server is started by default because the native transport is considered beta." - native proto does start by default, and it's not beta

native_transport_port

(Default: 9042) Port on which the CQL native transport listens for clients.

native_transport_min_threads

(Default: 16**) The minimum number of thread handling requests. The meaning is the same as `rpc_min_threads`.

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.

Configuration

Global row and key caches properties

Caching parameters for tables.

When creating or modifying tables, you enable or disable the key cache (partition key cache) or row cache 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.ILogRowCacheProvider`.
- `ConcurrentLinkedHashMapCacheProvider`: 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 `ConcurrentLinkedHashMapCacheProvider` 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 `ConcurrentLinkedHashMapCacheProvider` does.

Performance tuning properties

Tuning performance and system resource utilization, including memory, disk I/O, CPU, 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](#) on page 91.

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 specified 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 $(16 \times \text{number_of_drives})$ allows operations to queue low enough in the stack so that the OS and drives can reorder them.

concurrent_writes

(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 $(8 \times \text{number_of_cpu_cores})$.

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.

Caution: 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

Configuration

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.

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 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`. Enable this parameter to avoid sudden dirty buffer flushing from impacting read latencies. Recommended to use on SSDs, but not on HDDs.

trickle_fsync_interval_in_kb

(Default: 10240)]. Sets the size of the fsync in kilobytes.

Binary and RPC protocol timeout properties

Timeout settings for 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.

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

Settings for configuring and tuning 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 $((\text{concurrent_reads} + \text{concurrent_writes}) \times 2)$.
- `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.

Configuration

rpc_rcv_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

Settings to handle poorly performing or failing nodes.

dynamic_snitch_badness_threshold

(Default: 0.0) Sets the performance threshold for dynamically routing requests away from a poorly performing node. A value of 0.2 means Cassandra 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) Maximum throttle per delivery thread in kilobytes per second. This rate reduces proportionally to the number of nodes in the cluster. For example, if there are two nodes in the cluster, each delivery thread will use the maximum rate; if there are three, each node will throttle to half of the maximum, since the two nodes are expected to deliver hints simultaneously.

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

Automatic backup settings.

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

Server and client security settings.

authenticator

(Default: `org.apache.cassandra.auth.AllowAllAuthenticator`) The authentication backend. It implements `IAuthenticator`, which is used to identify users. The available authenticators are:

- `org.apache.cassandra.auth.AllowAllAuthenticator`: Disables authentication; no checks are performed.
- `org.apache.cassandra.auth.PasswordAuthenticator`: Authenticates users with usernames and hashed passwords stored in the `system_auth.credentials` table. If you use this authenticator, increase the **system_auth keyspace replication factor**.

authorizer

(Default: `org.apache.cassandra.auth.AllowAllAuthorizer`) The authorization backend. It implements `IAuthenticator`, which limits access and provides permissions. The available authorizers are:

Configuration

- `org.apache.cassandra.auth.AllowAllAuthorizer`: Disables authorization; allows any action to any user.
- `org.apache.cassandra.auth.CassandraAuthorizer`: Stores permissions in `system_auth.permissions` table. If you use this authenticator, increase the `system_auth keyspace replication factor`.

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. The available options are:

- `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-table basis programmatically or using a client application, such as CQL.

The attribute names documented in this section are the names used by Thrift, the Command Line Interface (CLI), and other legacy APIs and utilities. A few attributes have slightly different names in CQL. See [CQL keyspace and table properties](#).

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
<code>name</code>	NA
<code>placement_strategy</code>	SimpleStrategy
<code>strategy_options</code>	N/A (container attribute)
<code>durable_writes</code>	N/A (container attribute)

`name`

Required. The name for the keyspace.

`placement_strategy`

Configuration

Required. 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 **snitch** to be able to determine rack and data center locations of a node. For more information about replication placement strategy, see [Data replication](#) on page 16.

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](#) on page 16.

durable_writes

(Default: true) When set to false, data written to the keyspace bypasses the commit log. Be careful using this option because you risk losing data. Change the `durable_writes` attribute using CQL.

Table attributes

The following attributes can be declared per table.

Option	Default value
<code>bloom_filter_fp_chance</code>	0.01 or 0.1 (Value depends on the compaction strategy.)
<code>bucket_high</code>	1.5
<code>bucket_low</code>	0.5
<code>caching</code>	keys_only
<code>column_metadata</code>	N/A (container attribute)
<code>column_type</code>	Standard
<code>comment</code>	N/A
<code>compaction_strategy</code>	SizeTieredCompactionStrategy
<code>compaction_strategy_options</code>	N/A (container attribute)
<code>comparator</code>	BytesType
<code>compare_subcolumns_with</code>	BytesType*
<code>compression_options</code>	sstable_compression='SnappyCompressor'
<code>default_time_to_live</code>	0
<code>default_validation_class</code>	N/A

Option	Default value
<code>dclocal_read_repair_chance</code>	0.0
<code>gc_grace</code>	864000 (10 days)
<code>index_interval</code>	128
<code>key_validation_class</code>	N/A
<code>max_compaction_threshold</code>	32
<code>min_compaction_threshold</code>	4
<code>memtable_flush_after_mins</code>	N/A*
<code>memtable_flush_period_in_ms</code>	0
<code>memtable_operations_in_millions</code>	N/A*
<code>memtable_throughput_in_mb</code>	N/A*
<code>min_sstable_size</code>	50MB
<code>name</code>	N/A
<code>populate_io_cache_on_flush</code>	False
<code>read_repair_chance</code>	0.1 or 1 (See description below.)
<code>replicate_on_write</code>	true
<code>speculative_retry</code>	NONE
<code>sstable_size_in_mb</code>	160MB
<code>tombstone_compaction_interval</code>	1 day
<code>tombstone_threshold</code>	0.2

* Deprecated as of Cassandra 1.0, but can still be declared for backward compatibility.

compaction_strategy_options

(Default: N/A - container attribute) Sets attributes related to the chosen compaction-strategy. Attributes are:

- `bucket_high`
- `bucket_low`
- `max_compaction_threshold`
- `min_compaction_threshold`
- `min_sstable_size`
- `sstable_size_in_mb`
- `tombstone_compaction_interval`
- `tombstone_threshold`

CQL examples show how to set and update compaction properties.

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.

Configuration

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.

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:

- name: Binds a validation_class and (optionally) an index to a column.
- validation_class: Type used to check the column value.
- index_name: Name of the 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 an 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_compaction_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 leveledb](#) implementation. For more information, see the articles [When to Use Leveled Compaction](#) and [Leveled Compaction in Apache Cassandra](#).

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. Deprecated as of Cassandra 1.0, but can still 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_time_to_live

(Default: 0) The default expiration time in seconds for a table. Used in MapReduce/Hive scenarios when you have no control of TTL.

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

(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.

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

Configuration

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.

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) In `SizeTieredCompactionStrategy`, sets the maximum number of SSTables to allow in a minor compaction. In `LeveledCompactionStrategy`, it applies to L0 when L0 gets behind, that is, when L0 accumulates more than `MAX_COMPACTING_L0` SSTables.

mix_compaction_threshold

(Default: 4) In `SizeTieredCompactionStrategy` sets the minimum number of SSTables to trigger a minor compaction.

memtable_flush_after_mins

Deprecated as of Cassandra 1.0, but can still be declared for backward compatibility. Use `commitlog_total_space_in_mb`.

memtable_flush_period_in_ms

(Default: 0) Forces flushing of the memtable after the specified time in milliseconds elapses.

memtable_operations_in_millions

Deprecated as of Cassandra 1.0, but can still be declared for backward compatibility. Use `commitlog_total_space_in_mb`.

memtable_throughput_in_mb

Deprecated as of Cassandra 1.0, but can still be declared for backward compatibility. Use `commitlog_total_space_in_mb`.

min_sstable_size

(Default: 50MB) The `SizeTieredCompactionStrategy` 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 define 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.

populate_io_cache_on_flush

(Default: false) Populates the page cache on memtable flush and compaction. Enable only when all data on the node fits within memory. Use for fast reading of SSTables from IO cache (memory).

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.

speculative_retry

(Default: NONE) Overrides normal read timeout when `read_repair_chance` is not 1.0, sending another request to read. Choices are:

- ALWAYS: Retry reads of all replicas.
- Xpercentile: Retry reads based on the effect on throughput and latency.

- Yms: Retry reads after specified milliseconds.
- NONE: Do not retry reads.

sstable_size_in_mb

(Default: 160MB) 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 `LZ4Compressor` available in Cassandra 1.2.2 and later), `SnappyCompressor`, and `DeflateCompressor`. Use an empty string (") to disable compression. Choosing the right compressor depends on your requirements for space savings over read performance. LZ4 is fastest to decompress, followed by Snappy, then by Deflate. Compression effectiveness is inversely correlated with decompression speed. The extra compression from Deflate or Snappy is not enough to make up for the decreased performance for general-purpose workloads, but for archival data they may be worth considering. 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 minimum 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

Analyzing the heap dump file can help troubleshoot memory problems.

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. 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

This file is located in:

- Packaged installs: `/etc/dse/cassandra`
- Tarball installs: `install_location/resources/cassandra/conf`

Base the size of the directory on the value of the Java `-mx` option.

1. Open the `cassandra-env.sh` file for editing.

```
# set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
```

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:

Configuration

```
# set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
CASSANDRA_HEAPDUMP_DIR =<path>
```

4. Save the `cassandra-env.sh` file and restart.

Generating tokens

If not using virtual nodes (vnodes), you still need to calculate tokens for your cluster.

The following topics in the Cassandra 1.1 documentation provide conceptual information about tokens:

- [Data Distribution in the Ring](#)
- [Replication Strategy](#)

About calculating tokens for single or multiple data centers in Cassandra 1.2 and later

- Single data center deployments: calculate tokens by dividing the hash range by the number of nodes in the cluster.
- 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 Murmur3Partitioner

Use this method for generating tokens when you are **not** using virtual nodes (vnodes) 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' ]
```

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](#).

Configuring virtual nodes

Enabling virtual nodes on a new cluster

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

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.

Enabling virtual nodes on an existing production cluster

For production clusters, enabling virtual nodes (vnodes) has less impact on performance if you bring up a another data center configured with vnodes already enabled and let Cassandra automatic mechanisms distribute the existing data into the new nodes. Using the [shuffle utility](#) can be an expensive process on a running production system because the data to be shuffled around. Bootstrapping a new data center is a much safer way to enable vnodes.

1. [Add a new data center to the cluster](#).
2. Once the new data center with vnodes enabled is up, switch your clients to use the new data center.
3. Run a full repair with [nodetool repair](#).

This step ensures that after you move the client to the new data center that any previous writes are added to the new data center and that nothing else, such as hints, is dropped when you remove the old data center.

4. Update your schema to no longer reference the old data center.
5. Remove the old data center from the cluster.

See [Decommissioning a data center](#) on page 115.

Logging configuration

Logging configuration

To get 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.

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.

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.

Changing Logging Levels

Changing logging levels using the log4j-server.properties file.

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
```

Configuration

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
```

To determine which category a particular message in the log belongs to, you 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`.

Changing the rotation and size of the Cassandra output.log

Controlling the rotation and size of the `output.log`.

Cassandra's `output.log` logging configuration is controlled by the `log4j-server.properties` file in the following directories:

- Packaged installs: `/etc/dse/cassandra`
- Tarball installs: `install_location/resources/cassandra/conf`

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.

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.

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
}
```

Changing the rotation and size of the Cassandra system.log

Controlling the rotation and size of the `system.log`.

Cassandra's `system.log` logging configuration is controlled by the `log4j-server.properties` file in the following directories:

- Packaged installs: `/etc/dse/cassandra`
- Tarball installs: `install_location/resources/cassandra/conf`

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.

- 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.
- 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. Make sure you have enough space to store the additional logs.

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`

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 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:

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>	

- Set the restore directory location:

Command	<code>restore_directories=</code>
Format	<code>restore_directories=<restore_directory location></code>

- Restore mutations created up to and including the specified timestamp:

Command	<code>restore_point_in_time=</code>
Format	<code><timestamp></code>
Example	<code>restore_point_in_time=2012-08-16 20:43:12</code>

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 Cassandra

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:

- The Cassandra **nodetool** utility
- **DataStax OpsCenter** management console
- JConsole

Using the same tools, you can perform certain administrative commands and operations such as flushing caches or doing a **node repair**.

Monitoring using 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%    b8e6748f-ec11-410d-c94f-9b67d88a28e7 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.

The nodetool cfstats command

The nodetool cfstats command displays statistics for each table and keyspace. This example shows an excerpt of the output of the command after flushing a table of wikipedia data to disk.

```
-----
Keyspace: wiki
```

```

Read Count: 3589
Read Latency: 1.9554240735580943 ms.
Write Count: 3579
Write Latency: 0.32189075160659397 ms.
Pending Tasks: 0
Column Family: solr
SSTable count: 1
Space used (live): 9592399
Space used (total): 9592399
SSTable Compression Ratio: 0.5250859637980083
Number of Keys (estimate): 3584
Memtable Columns Count: 0
Memtable Data Size: 0
Memtable Switch Count: 1
Read Count: 3589
Read Latency: 1.955 ms.
Write Count: 3579
Write Latency: 0.322 ms.
Pending Tasks: 0
Bloom Filter False Positives: 0
Bloom Filter False Ratio: 0.00000
Bloom Filter Space Used: 4488
Compacted row minimum size: 180
Compacted row maximum size: 219342
Compacted row mean size: 5559
-----

```

This table describes the nodetool cfstats output.

Table 9: nodetool cfstats output

Name of statistic	Example value	Brief description	Related information
Keyspace	wiki	Name of the keyspace	Keyspace and table storage configuration
Read latency	1.9554240735580943 ms.	Read latency of tables in the keyspace	OpsCenter alert metrics
Write count	3579	Write count of tables in the keyspace	Same as above
Write latency	0.32189075160659397 ms.	Write latency of tables in the keyspace	Same as above
Pending tasks	0	Tasks in the queue for reads, writes, and cluster operations of tables in the keyspace	OpsCenter pending task metrics
Column family	solr	Name of the Cassandra table	
SSTable count	1	Number of SSTables	How to use the SSTable counts metric and OpsCenter alert metrics

Name of statistic	Example value	Brief description	Related information
		containing data from the table	
Space used (live)	9592399	Bytes, the space that is measured depends on operating system	Advanced system alert metrics
Space used (total)	9592399	Same as above	Same as above
SSTable compression ratio	0.5250859	Percentage of reduction in data-representation size resulting from compression	Types of compression (sstable_compression option)
Number of keys (estimate)	3584	Sum of storage engine rows (containers for columns) in each SSTable, approximate (within 128)	Counting keys in Cassandra article
Memtable columns count	0	Number of cells (storage engine rows x columns) of data in the memtable	Cassandra memtable structure in memory
Memtable data size	0	Size in bytes of the memtable data	Same as above
Memtable switch	1	Number of times a full memtable was swapped for an empty one that increases each time the memtable	How memtables are measured article

Name of statistic	Example value	Brief description	Related information
		for a table is flushed to disk	
Read count	3589	Number of pending read requests	OpsCenter alert documentation
Read latency	1.955 ms.	Round trip time to complete a read request in milliseconds	Factors that affect read latency
Write count	3579	Number of pending write requests	OpsCenter alert documentation
Write latency	0.322 ms.	Round trip time to complete a read request in milliseconds	Factors that affect write latency
Pending tasks	0	Number of read, write, and cluster operations that are pending	OpsCenter pending task metrics documentation
Bloom filter false positives	0	Number of false positives, which occur when the bloom filter said the row existed, but it actually did not exist in absolute numbers	Tuning bloom filters
Bloom filter false ratio	0.00000	Fraction of all bloom filter checks resulting in a false positive	Same as above
Bloom filter space used	4488	Bytes of bloom filter data	Same as above

Name of statistic	Example value	Brief description	Related information
Compacted row minimum size	180	Lower size limit in MB for table rows being compacted in memory	Used to calculate what the approximate row cache size should be. Multiply the reported row cache size, which is the number of rows in the cache, by the compacted row mean size for every table and sum them.
Compacted row maximum size	219342	Upper size limit in bytes for compacted table rows, configurable in the <code>cassandra.yaml</code> (<code>in_memory_compaction_limit_in_mb</code>)	Same as above
Compacted row mean size	5559	The average size in bytes of compacted table rows	Same as above

The `nodetool cfhistograms` command

The `nodetool cfhistograms` command provides statistics about a table, including read/write latency, row size, column count, and number of SSTables.

The syntax of the command is:

```
nodetool cfhistograms keyspace table
```

For example, to get statistics about the `solr` table in the `wiki` keyspace on Linux, use this command:

```
cd install_location/bin
nodetool cfhistograms wiki solr
```

An example of output is (explained below):

```
wiki/solr histograms
Offset      SSTables    Write Latency    Read Latency    Row Size
Column Count
1           3579        0                0                0
0
2           0          0                0                0
0
...
35          0          0                0                0
0
42          0          0                27               0
0
50          0          0                187              0
0
60          0          10               460              0
0
72          0          200              689              0
0
86          0          663              552              0
0
```

Operations

103		0	796	367	0
	0				
124		0	297	736	0
	0				
149		0	265	243	0
	0				
179		0	460	263	0
	0				
25109160		0	0	0	0
	0				

The Offset column corresponds to the x-axis in a histogram. It represents buckets of values. More precisely, it is a series of ranges where each offset includes the range of values greater than the previous offset and less than or equal to the current offset. The offsets start at 1 and each subsequent offset is calculated by multiplying the previous offset by 1.2, rounding up, and removing duplicates. The offsets can range from 1 to approximately 25 million, with less precision as the offsets get larger.

Columns of metrics represent the number of values that fall into a particular offset's range. Latencies are shown in terms of microseconds (μ s), Row Size is in bytes, and SSTables and Column Count are counts.

To illustrate:

- Offset 1 shows that 3579 requests only had to look at one SSTable. The SSTables column corresponds to how many SSTables were involved in a read request.
- Offset 86 shows that there were 663 requests with a write latency between 73 and 86 μ s. The range falls into the 73 to 86 bucket.

OpsCenter, described later on this page, displays the same information in a better format for understanding the statistics.

The netstats command

The `nodetool netstats` command provides statistics about network operations and connections.

The syntax of the command is:

```
nodetool cfnetstats host
```

An example of output is:

```
Mode: NORMAL
Not sending any streams.
Not receiving any streams.
Read Repair Statistics:
Attempted: 1
Mismatch (Blocking): 0
Mismatch (Background): 0
Pool Name           Active   Pending   Completed
Commands            n/a      0         0
Responses           n/a      0         0
```

- **Attempted**
The number of successfully completed **read repair operations**
- **Mismatch (blocking)**
The number of read repair operations since server restart that blocked a query.
- **Mismatch (background)**
The number of read repair operations since server restart performed in the background.
- **Pool name**
Information about **client read and write requests** by thread pool.

nodetool tpstats command

The `nodetool tpstats` command provides statistics about the number of active, pending, and completed tasks for each stage of Cassandra operations by thread pool. A high number of pending tasks for any pool can indicate performance problems.

The syntax of the command is:

```
nodetool tpstats
```

An example of output is:

Pool Name	Active	Pending	Completed	Blocked	All
time blocked					
ReadStage	0	0	15	0	
0					
RequestResponseStage	0	0	0	0	
0					
MutationStage	0	0	3674	0	
0					
ReadRepairStage	0	0	0	0	
0					
ReplicateOnWriteStage	0	0	0	0	
0					
GossipStage	0	0	0	0	
0					
AntiEntropyStage	0	0	0	0	
0					
MigrationStage	0	0	0	0	
0					
MemoryMeter	0	0	1	0	
0					
MemtablePostFlusher	0	0	267	0	
0					
FlushWriter	0	0	9	0	
1					
MiscStage	0	0	0	0	
0					
commitlog_archiver	0	0	0	0	
0					
InternalResponseStage	0	0	0	0	
0					
HintedHandoff	0	0	0	0	
0					

Message type	Dropped
RANGE_SLICE	0
READ_REPAIR	0
BINARY	0
READ	0
MUTATION	0
_TRACE	0
_REQUEST_RESPONSE	0

This table describes key indicators:

Table 10: nodetool tpstats output

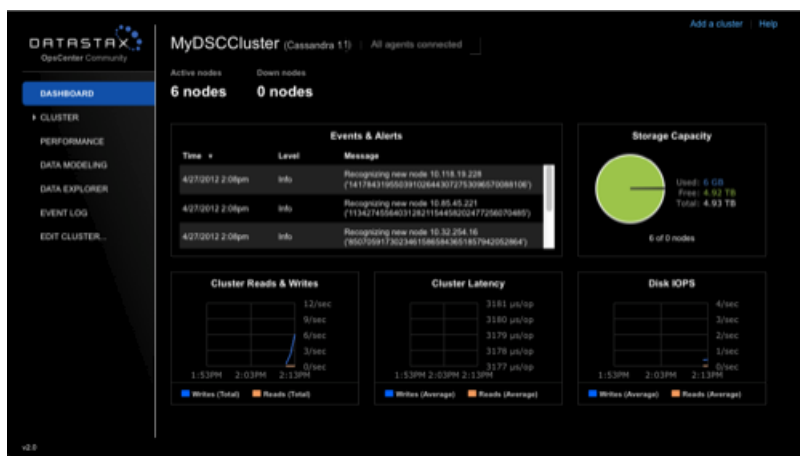
Name of statistic	Brief description	Related information
Mutation stage	Status of write requests. A high number of pending write requests indicates a problem handling them.	Tune hardware or Cassandra configuration.

Name of statistic	Brief description	Related information
Flush writer	Status of the sort and write-to-disk operations.	Tune hardware or Cassandra configuration.

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 customize 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 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:

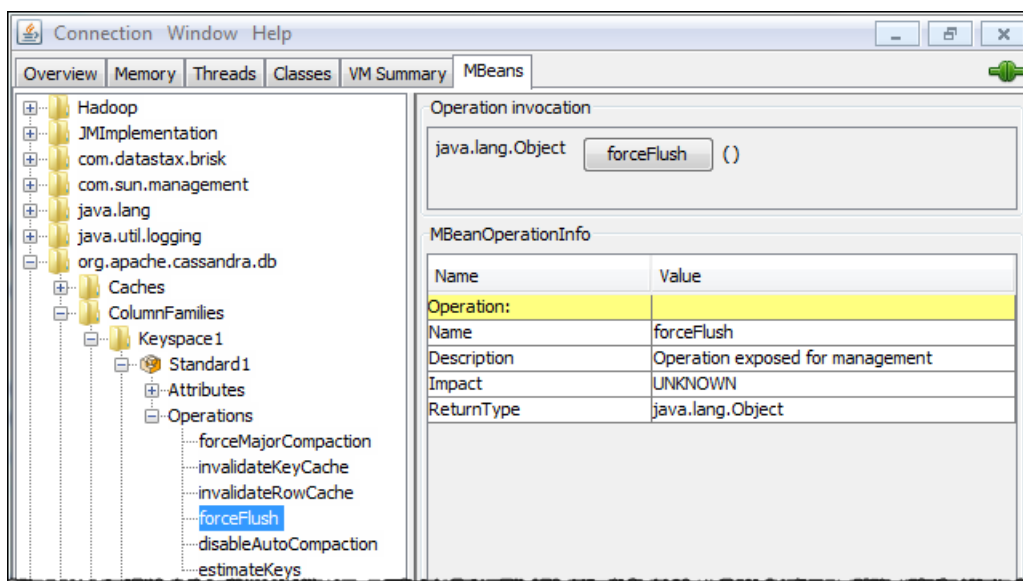
- Overview
Displays overview information about the Java VM and monitored values.
- Memory
Displays information about memory use.
- Threads
Displays information about thread use.
- Classes
Displays information about class loading.
- VM Summary
Displays information about the Java Virtual Machine (VM).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

Table 11: Compaction Metrics

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 and read/write latency 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.

Table 12: Compaction Metrics

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:

Table 13: Compaction Metrics

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.

Attribute	Description
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 table, 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.

Data caching

Configuring data caches

Cassandra includes integrated caching and 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 partition key cache

The partition key cache is a cache of the **partition 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 partition 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.

Enabling and configuring caching

Enable or disable caching and using CQL to configure the **caching** storage parameter. Set parameters in the `cassandra.yaml` file to configure caching properties:

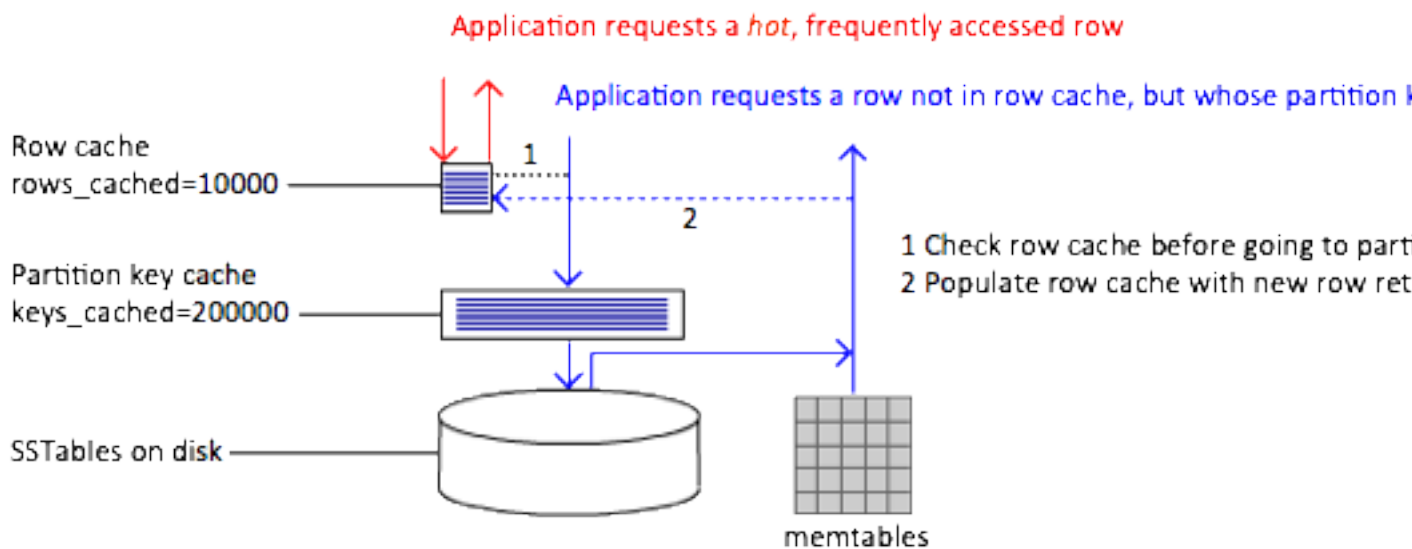
- **Partition key cache size**
- **Row cache size**
- How often Cassandra **saves partition key caches to disk**
- How often Cassandra **saves row caches to disk**
- The **row cache implementation**

Set the caching attribute in the `WITH` clause of the table definition. For example, configure Cassandra to configure the partition key cache and row cache.

```
CREATE TABLE users (
    userid text PRIMARY KEY,
    first_name text,
    last_name text,
)
WITH caching = 'all';
```

How caching works

When both row cache and partition key cache are configured, the row cache returns results whenever possible. In the event of a row cache miss, the partition 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 partition 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 **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 caching

Make changes to cache options in small, incremental adjustments, then monitor the effects of each change using **DataStax Opscenter** or the **nodetool** utility. The output of the `nodetool info` command shows the following row cache and key cache metrics, which are configured in the `cassandra.yaml` file:

- Cache size in bytes
- Capacity in bytes
- Number of hits
- Number of requests
- Recent hit rate
- Duration in seconds after which Cassandra saves the key cache.

For example, on start-up, the information from `nodetool info` might look something like this:

```
Token           : (invoke with -T/--tokens to see all 256 tokens)
ID              : 387d15ba-7103-491b-9327-1a691dbb504a
Gossip active   : true
Thrift active   : true
Native Transport active: true
Load            : 11.35 KB
Generation No   : 1384180190
```

```

Uptime (seconds) : 437
Heap Memory (MB) : 136.33 / 1996.81
Data Center      : datacenter1
Rack             : rack1
Exceptions       : 0
Key Cache        : size 360 (bytes), capacity 103809024 (bytes), 15 hits, 19
                  requests, 0.789 recent hit rate, 14400 save period in seconds
Row Cache        : size 0 (bytes), capacity 0 (bytes), 0 hits, 0 requests, NaN
                  recent hit rate, 0 save period in seconds

```

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.

Compaction and compression

Compaction

During compaction, Cassandra combines multiple data files to improve the performance of partition scans and to reclaim space from deleted data.

Compaction is a periodic background process. During compaction Cassandra merges SSTables by combining row fragments, evicting expired **tombstones**, and rebuilding indexes. Because the SSTables are sorted, the merge is efficient (no random disk I/O). After a newly merged SSTable is complete, the input SSTables are reference counted and removed as soon as possible. During compaction, there is a temporary spike in disk space usage and disk I/O.

Cassandra provides two compaction strategies:

- **SizeTieredCompactionStrategy**: The default compaction strategy. This strategy gathers SSTables of similar size and compacts them together into a larger SSTable. You can configure this strategy with **CQL-configurable thresholds**.
- **LeveledCompactionStrategy**: Introduced in Cassandra 1.0, this 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. To enable this strategy, set the **compaction_strategy** for the table.

For more information about compaction strategies and which strategy to choose, see **compaction strategies** and the articles **When to Use Leveled Compaction** and **Leveled Compaction in Apache Cassandra**.

Compaction can impact **reads** that are not fulfilled by the cache because temporary increases in disk I/O and utilization occur. However, after a compaction completes, off-cache read performance improves since there are fewer SSTable files on disk that need to be checked.

In Cassandra 1.2 and later, you can configure when compaction (eviction) occurs for tombstones for TTL-configured and deleted columns with the tombstone parameters. Setting these parameters helps you avoid having to manually performing compaction to recover disk space.

The `compaction_strategy_options` parameter sets attributes related to the chosen compaction-strategy:

Note: The attribute names documented here are the names as they are stored in the system keyspace within Cassandra. A few attributes have slightly different names in CQL. See [CQL keyspace and table properties](#).

Parameter	SizeTieredCompactionStrategy	LeveledCompactionStrategy
<code>bucket_high</code>	Yes	No
<code>bucket_low</code>	Yes	No
<code>max_compaction_threshold</code>	Yes	Yes
<code>min_compaction_threshold</code>	Yes	Yes
<code>min_sstable_size</code>	Yes	No
<code>sstable_size_in_mb</code>	No	Yes
<code>tombstone_compaction_interval</code>	Yes	Yes
<code>tombstone_threshold</code>	Yes	Yes

You can specify a number of compaction parameters in the `cassandra.yaml` file:

- `snapshot_before_compaction`
- `in_memory_compaction_limit_in_mb`
- `multithreaded_compaction`
- `compaction_preheat_key_cache`
- `concurrent_compactors`
- `compaction_throughput_mb_per_sec`

Starting with Cassandra 1.2.5, the `compaction_throughput_mb_per_sec` parameter works better with large [partitions](#) because compaction is throttled to the specified total throughput across the entire system. In older releases, Cassandra only checked the compaction throughput between partitions, so large partitions could cause spikes in I/O demand.

Cassandra provides a start-up option for [testing compaction strategies](#) without affecting the production workload.

For information about compaction metrics, see [Compaction metrics](#) on page 102.

About full compactions

A full compaction applies only to `SizeTieredCompactionStrategy`. It merges all SSTables into one large SSTable. However, a full compaction is not recommended because the large SSTable that is created will not be compacted until the amount of actual data increases four-fold (or `min_compaction_threshold`). Additionally, during runtime, full compaction is I/O and CPU intensive and can temporarily double disk space usage when no old versions or tombstones are evicted.

To initiate a full compaction for all tables in a keyspace use the `nodetool compact` command.

Configuring compaction

In addition to consolidating SSTables, the compaction process merges keys, combines columns, evicts 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 configure compaction, construct a property map in the CQL collection format:

```
compaction = { name : value, name : value, ...}
```

In this string, *italics* indicates optional.

1. Create or update a table to set the compaction strategy using the ALTER or CREATE TABLE statements.

```
ALTER TABLE users WITH
  compaction = { 'class' : 'LeveledCompactionStrategy' }
```

2. Control the frequency and scope of a minor compaction of a table that uses the default size-tiered compaction strategy by setting the CQL `min_threshold` attribute.

```
ALTER TABLE users
  WITH compaction =
    { 'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 }
```

For the list of options and more information, see [CQL keyspace and table properties](#).

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.

When to compress data

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.

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](#).

Configuring compression

Compression is enabled by default in Cassandra 1.1 and later.

1. Disable compression, using 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' : '' };
```

2. Enable compression on an existing table, using ALTER TABLE to set the compression algorithm sstable_compression to LZ4Compressor (Cassandra 1.2.2 and later), SnappyCompressor, or DeflateCompressor.

```
CREATE TABLE DogTypes (
    block_id uuid,
    species text,
    alias text,
    population varint,
    PRIMARY KEY (block_id)
)
WITH compression = { 'sstable_compression' : 'LZ4Compressor' };
```

3. Change compression on an existing table, using ALTER TABLE and setting the compression algorithm sstable_compression to DeflateCompressor.

```
ALTER TABLE CatTypes
WITH compression = { 'sstable_compression' : 'DeflateCompressor',
    'chunk_length_kb' : 64 }
```

You tune data compression on a per-table basis using CQL to alter a table.

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.

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.

Enable write survey mode by starting a Cassandra node using the write_survey option.

```
bin/cassandra - Dcassandra.write_survey=true
```

This example shows how to start a tarball installation of Cassandra.

Tuning Java resources

Consider tuning Java resources in the event of a performance degradation or high memory consumption.

There are two files that control environment settings for Cassandra:

- conf/cassandra-env.sh
Java Virtual Machine (JVM) configuration settings
- bin/cassandra-in.sh

Sets up Cassandra environment variables such as CLASSPATH and JAVA_HOME.

Heap sizing options

If you decide to change the Java heap sizing, both MAX_HEAP_SIZE and HEAP_NEWSIZE should be set together in conf/cassandra-env.sh.

- **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

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.

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.

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

How Cassandra uses memory

Using a java-based system like Cassandra, you can typically allocate about 8GB of memory on the heap before garbage collection pause time starts to become a problem. Modern machines have much more memory than that and Cassandra can make use of additional memory as page cache when files on disk are accessed. Allocating more than 8GB of memory on the heap poses a problem due to the amount of Cassandra metadata about data on disk. The Cassandra metadata resides in memory and is proportional to total data. Some of the components **grow proportionally to the size of total memory**.

In Cassandra 1.2 and later, the Bloom filter and compression offset map that store this metadata reside off-heap, greatly increasing the capacity per node of data that Cassandra can handle efficiently. In Cassandra 2.0, the partition summary also resides off-heap.

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

Running node repair.

To understand how repair works and the information described in this topic, please read the blog article [Advanced repair techniques](#).

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.

Guidelines for running routine node repair include:

- The hard requirement for routine repair frequency is the value of `gc_grace`. 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.
- Use caution when running routine node repair on more than one node at a time and schedule regular repair operations for low-usage hours.
- In systems that seldom delete or overwrite data, you can raise the value of `gc_grace` with minimal impact to disk space. This allows wider intervals for scheduling repair operations with the `nodetool` utility.

Repair requires intensive disk I/O. This occurs because of the validation compaction used for building the Merkle tree. To mitigate heavy disk usage:

- Use the `nodetool` compaction throttling options (`setcompactionthroughput` and `setcompactionthreshold`).
- Use `nodetool snapshot` and *sequentially* repair from the snapshot. Recall that snapshots are just hardlinks to existing SSTables, immutable, and require almost no disk space. This means that for any given replica set, only one replica at a time performs the validation compaction. This allows the dynamic snitch to maintain performance for your application via the other replicas.

Note: Using the `nodetool repair -pr` (`-partitioner-range`) option repairs only the primary range for that node, the other replicas for that range still have to perform the Merkle tree calculation, causing a validation compaction. Because all the replicas are compacting at the same time, all the nodes may be slow to respond for that portion of the data.

Repair can result in overstreaming. Overstreaming occurs, for example, when there is a single damaged partition, but many more streams are sent.

This happens because the Merkle trees don't have infinite resolution and Cassandra makes a tradeoff between the size and space. Currently, Cassandra uses a fixed depth of 15 for the tree (32K leaf nodes). For a node containing a million partitions with one damaged partition, about 30 partitions are streamed, which is the number that fall into each of the *leaves* of the tree. Of course, the problem gets worse when more partitions exist per node, and results in a lot of disk space usage and needless compaction.

To mitigate overstreaming, you can use subrange repair (available starting in Cassandra 1.1.11). Subrange repair allows for repairing only a portion of the data belonging to the node. Because the Merkle tree precision is fixed, this effectively increases the overall precision.

To use subrange repair:

1. Use the Java `describe_splits` call to ask for a split containing 32K partitions.
2. Iterate throughout the entire range incrementally or in parallel. This completely eliminates the overstreaming behavior and wasted disk usage overhead.
3. Pass the tokens you received for the split to the `nodetool repair -st` (`-start-token`) and `-et` (`-end-token`) options.
4. Pass the `-local` (`-in-local-dc`) option to `nodetool` to repair only within the local data center. This reduces the cross data-center transfer load.

Adding or removing a node or data center

Adding nodes an existing cluster

Adding nodes when using virtual nodes.

Virtual nodes (vnodes) 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](#) on page 15.

Note: If you do not use vnodes, follow the instructions in the 1.1 topic [Adding Capacity to an Existing 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` configuration 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 vnodes to assign to the node. If the hardware capabilities vary among the nodes in your cluster, you can assign a proportional number of vnodes 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

Adding a data center to an existing cluster.

1. Ensure that you are using **NetworkTopologyStrategy** for all of your keyspaces.
2. For each node, set the following properties in the `cassandra.yaml` file:

- a) 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 prevents 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.

- b) Set other properties, such as -seeds and listen_address, to match the cluster settings.

For more guidance, see [Initializing a multiple node cluster \(multiple data centers\)](#) on page 48.

- c) If you want to enable vnodes, set **num_tokens**.

The recommended value is 256. Do not set the **initial_token** parameter.

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.
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, use `setHostConfig.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 Cassandra on 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

Replace a node that has died for some reason, such as hardware failure.

Note: This procedure applies to clusters using vnodes. If not using vnodes, use the **instructions** in the Cassandra 1.1 documentation.

You must prepare and start the replacement node, integrate it into the cluster, and then remove the dead node.

1. Confirm that the node is dead:
 - Run **nodetool ring** if not using vnodes.
 - Run **nodetool status** if using vnodes.

The nodetool command shows a down status for the dead node (DV:

```
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%    b8e6748f-ec11-410d-c94f-9b67d88a28e7 rack1
```

2. Note Host ID of the dead node; it is used in the last step.
3. Add and start the replacement node as described in **Adding nodes an existing cluster** on page 113.
4. Using the Host ID of the dead node, remove the dead node from the cluster using the **nodetool removemode** command. You may need to use the force option.

Decommissioning a data center

Properly removing a data center ensures that no information is lost.

1. Make sure no clients are still writing to any nodes in the data center.
2. Run a full repair with **nodetool repair**.
This ensures that all data is propagated from the data center being decommissioned.
3. **Change all keyspaces** so they no longer reference the data center being removed.
4. Run **nodetool decommission** on every node in the data center being removed.

Removing a node

Reduce the size of a data center.

Use these instructions when you want to remove up nodes to reduce the size of your cluster, not for **replacing a dead node**.

Attention: If you are not using **virtual nodes** (vnodes), you must rebalance the cluster.

1. Run a **repair** on each keyspace:

```
$ nodetool repair -h ip_address_of_node keyspace_name
```

The **nodetool repair** command repairs inconsistencies across all of the replicas for a given range of data.

2. If the node is up, run **nodetool decommission**.

This assigns the ranges that the node was responsible for to other nodes and replicates the data appropriately.

Use `nodetool netstats` to monitor the progress.

3. If the node is down, following the steps in [Replacing a dead node](#) on page 115 except don't replace the node.
4. If using vnodes, remove the node using the `nodetool removemode` command.
5. If not using vnodes, removing a node from the cluster will create a hotspot. Adjust your tokens to evenly distribute the data across the remaining nodes before running the `nodetool removemode` command.

For information on how to do this, see the following topics in the Cassandra 1.1 documentation:

- [About Data Partitioning in Cassandra](#)
- [Generating Tokens](#)

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.

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

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

The snapshot is created in `<data_directory_location>/<keyspace_name>/<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, run the `nodetool clearsnapshot` command. For example:

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

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

Enabling incremental backups

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

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.

Edit the `cassandra.yaml` configuration file on each node in the cluster and change the value of `incremental_backups` to `true`.

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.

Generally, before restoring a snapshot, you should **truncate** the table. If the backup occurs before the **delete** and you restore the backup after the delete without first truncating, you do not get back the original data (row). Until compaction, the tombstone is in a different SSTable than the original row, so restoring the SSTable containing the original row does not remove the tombstone and the data still appears to be deleted.

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 shut down 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.

1. Shut down the node.
2. Clear all files in `/var/lib/cassandra/commitlog`.
3. Delete all `*.db` files in the 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>
```

5. Copy its contents into this directory:

```
<data_directory_location>/<keyspace_name>/<table_name> directory.
```

6. If using incremental backups, copy all contents of this directory:

```
<data_directory_location>/<keyspace_name>/<table_name>/backups
```

7. Paste it into this directory:

```
<data_directory_location>/<keyspace_name>/<table_name>
```

8. Restart the node.

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

9. Run `nodetool repair`.

Cassandra tools

The nodetool utility

A command line interface for Cassandra for managing a cluster.

Command format

- Packaged installs: `nodetool -h HOSTNAME [-p JMX_PORT] COMMAND`
- Tarball installs: `install_location/bin/nodetool -h HOSTNAME [-p JMX_PORT] COMMAND`
- Remote Method Invocation: `nodetool -h HOSTNAME [-p JMX_PORT -u JMX_USERNAME -pw JMX_PASSWORD] COMMAND`

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

Options

Flag	Option	Description
-a	--include-all-sstables	Rewrite/upgrade all SSTables including the most recent when rebuilding SSTables .
-et	--end-token arg	Token at which repair range ends.
-h	--host arg	Hostname of node or IP address.
-local	--in-local-dc	Only repair against nodes in the same data center.
-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.
-st	--start-token arg	Token at which repair range starts.
-T	--tokens	Display all tokens.
-u	--username arg	Remote JMX agent username.
Snapshot options only		
-cf	--column-family arg	Only take a snapshot of the specified table.
-ns	--no-snapshot	Disables snapshot during scrub to save disk space. If the disk utilization is more than 50%, scrub fails.
-snapshot	--with-snapshot	Repair one node at a time using snapshots.
-t	--tag arg	Optional name to give a snapshot.

Commands

Square brackets indicate optional parameters.

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. Also see <http://wiki.apache.org/cassandra/NodeProbe#Decommission> and http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely.

describing keyspace

Shows the `partition ranges` (formerly token ranges) for a given keyspace.

disableautocompaction *[keyspace] [cfnames]*

Disable autocompaction for the given keyspace and column family.

disablebackup

Disable incremental backup.

disablebinary

Disable native transport (binary protocol).

disablegossip

Disable Gossip. Effectively marks the node dead.

disablehandoff

Disable storing of future hints on the current node.

disablethrift

Disable 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.

enableautocompaction *[keyspace] [cfnames]*

Enable autocompaction.

enablebackup

Enable incremental backup.

enablebinary

Re-enable native transport (binary protocol).

enablegossip

Re-enables Gossip.

enablehandoff

Re-enable storing future hints on the current node.

enablethrift

Re-enable 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>.

getcompactionthroughput

Display the system's maximum throughput in megabytes for compaction.

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 file names that own the key.

getstreamthroughput

Displays the maximum streaming throughput in the system in megabytes per second.

gossipinfo

Shows the gossip information for the cluster.

info [-T or --tokens]

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 on the token ring to a new token. For negative tokens, escape using `\\`. For example: `move \\-123`. 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.

pausehandoff

Pause the hints delivery process.

predictconsistency *replication_factor time [versions] [latency_percentile]*

Predict the latency and consistency "t" milliseconds after writes.

proxyhistograms

Print statistic histograms for network operations.

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 the 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.

resetlocalschema

Reset the node's local schema and resync.

resumehandoff

Resume the hints delivery process.

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 (vnodes), 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. If scrub can't validate the column value against the column definition's data type, it logs the row key and skips to the next row. If the -ns option is specified, snapshot creation is disabled.

setcachecapacity key-cache-capacity row-cache-capacity

Set the global key and row cache capacities in megabytes.

setcompactionthreshold keyspace cfname

Set minimum and maximum compaction thresholds for a table. This 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. Also see <http://wiki.apache.org/cassandra/MemtableSSTable>.

setcompactionthroughput value_in_mb

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

setstreamthroughput value_in_mb

Set the maximum streaming throughput in the system in megabytes per second. To disable throttling, set to 0.

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 [*keyspaces...*] -cf [*tableName*] -t [*snapshotName*]

Takes an online snapshot of Cassandra's data. You can specify the table for particular keyspaces using the *snapshotName* option. 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](http://wiki.apache.org/cassandra/Operations#Backing_up_data) and http://wiki.apache.org/cassandra/Operations#Backing_up_data.

status

Display cluster information, such as state, load, host ID, and token.

statusbinary

Status of native transport (binary protocol).

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 [-a] [*keyspace*] [*tables*]

Rebuilds SSTables on a node for the named tables that are not on the current version. Use when upgrading your server or changing compression options (available from Cassandra 1.0.4 and later).

Use `-a` to include all SSTables, even those already on the current version.

version

Displays the Cassandra release version for the node being queried.

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.

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 `CQL` or Thrift.

To get the best throughput from SStable loading, you can use multiple instances of `sstableloader` to stream across multiple machines. No hard limit exists on the number of SSTables that `sstableloader` can run at the same time, so you can add additional loaders until you see no further improvement.

If you use `sstableloader` on the same machine as the Cassandra node, you can't use the same network interface as the Cassandra node. However, you can use the JMX `StorageService > bulkload()` call from that node. This method takes the absolute path to the directory where the SSTables are located, and loads them just as `sstableloader` does. However, because the node is both source and destination for the streaming, it increases the load on that node. This means that you should load data from machines that are not Cassandra nodes when loading into a live cluster.

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/`.

Packaged installs:

```
bash sstableloader [options ] <dir_path>
```

Tarball installs:

```
cd install_location/bin
bash sstableloader [options ] <dir_path>
```

For example:

```
$ ls -l 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.

Table 14: sstableloader

Short option	Long option	Description
-d <initial hosts>	--nodes <initial hosts>	Connect to comma separated list of hosts for initial ring information.
	-debug	Display stack traces.
-h	--help	Do not stream to this comma separated list of nodes.
-i <NODES>	--ignore <NODES>	Display help.
	--no-progress	Do not display progress.
-p <rpc port>	--port <rpc port>	RPC port (default 9160).
-t <throttle>	--throttle <throttle>	Throttle speed in Mbits (default unlimited).
-v	--verbose	Verbose output.

The cassandra utility

Starts the Cassandra Java server process.

Usage:

```
cassandra [OPTIONS]
```

Set the following environment variables:

- **JAVA_HOME**: path location of your Java Virtual Machine (JVM) installation.
- **CLASSPATH**: 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.

Table 15: Options

Option	Description
<code>-f</code>	Start the cassandra process in foreground (default is to start as a background process).
<code>-p <filename></code>	Log the process ID in the named file. Useful for stopping Cassandra by killing its PID.
<code>-v</code>	Print the version and exit.
<code>-D <parameter></code>	<p>Passes in one of the following startup parameters:</p> <p>access.properties=<filename> The file location of the <code>access.properties</code> file.</p> <p>cassandra-pidfile=<filename> Log the Cassandra server process ID in the named file. Useful for stopping Cassandra by killing its PID.</p> <p>cassandra.config=<directory> The directory location of the Cassandra configuration files.</p> <p>cassandra.initial_token=<token> Sets the initial partitioner token for a node the first time the node is started.</p> <p>cassandra.join_ring=<true false> Set to false to start Cassandra on a node but not have the node join the cluster.</p> <p>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 <code>cassandra.yaml</code> (such as <code>listen_address</code>).</p> <p>cassandra.renew_counter_id=<true false></p>

Option	Description
	<p>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 tables on the node, then restart the node with <code>-Dcassandra.renew_counter_id=true</code>, then run <code>nodetool repair</code> once the node is up again.</p> <p>cassandra.write_survey=true</p> <p>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 on page 110.</p>

```

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>

```

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
```

The `cassandra-stress` tool

A Java-based stress testing utility for benchmarking and load testing a Cassandra cluster.

The binary installation of the tool also includes a daemon, which in larger-scale testing can prevent potential skews in the test results by keeping the JVM warm.

Modes of operation:

- **Inserting:** Loads test data.
- **Reading:** Reads test data.
- **Indexed range slicing:** Works with `RandomPartitioner` on indexed tables.

The `cassandra-stress` tool creates a keyspace called `Keyspace1` and within that, tables named `Standard1`, `Super1`, `Counter1`, and `SuperCounter1`, depending on what type of table is being tested. These are automatically created the first time you run the stress test and will be reused on subsequent runs unless you drop the keyspace using [CQL](#). It is not possible to change the names; they are hard-coded.

Commands:

Cassandra tools

- Packaged installs: `cassandra-stress [options]`
- Tarball installs: `install_location/tools/bin/cassandra-stress [options]`

You can use these modes with or without the **cassandra-stress daemon** running (binary installs only).

Options for cassandra-stress

Short option	Long option	Description
-V	--average-size-values	Generate column values of average rather than specific size.
-C <CARDINALITY>	--cardinality <CARDINALITY>	Number of unique values stored in columns. Default is 50.
-c <COLUMNS>	--columns <COLUMNS>	Number of columns per key. Default is 5.
-S <COLUMN-SIZE>	--column-size <COLUMN-SIZE>	Size of column values in bytes. Default is 34.
-Z <COMPACTION-STRATEGY>	--compaction-strategy <COMPACTION-STRATEGY>	Specifies which compaction strategy to use.
-U <COMPARATOR>	--comparator <COMPARATOR>	Specifies which column comparator to use. Supported types are: TimeUUIDType, AsciiType, and UTF8Type.
-I <COMPRESSION>	--compression <COMPRESSION>	Specifies the compression to use for SSTables. Default is no compression.
-e <CONSISTENCY-LEVEL>	--consistency-level <CONSISTENCY-LEVEL>	Consistency level to use (ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL, ANY). Default is ONE.
-x <CREATE-INDEX>	--create-index <CREATE-INDEX>	Type of index to create on columns (KEYS).
-L I	--enable-cql	Perform queries using CQL (Cassandra Query Language).
-y <TYPE>	--family-type <TYPE>	Sets the table type .
-f <FILE>	--file <FILE>	Write output to a given file.
-h	--help	Show help.
-k	--keep-going	Ignore errors when inserting or reading. When set, --keep-trying has no effect. Default is false.
-K <KEEP-TRYING>	--keep-trying <KEEP-TRYING>	Retry on-going operation N times (in case of failure). Use a positive integer. The default is 10.
-g <KEYS-PER-CALL>	-g, --keys-per-call <KEYS-PER-CALL>	Number of keys to per call. Default is 1000.
-d <NODES>	--nodes <NODES>	Nodes to perform the test against. Must be comma separated with no spaces. Default is localhost.
-D <NODESFILE>	--nodesfile <NODESFILE>	File containing host nodes (one per line).
-W	--no-replicate-on-write	Set replicate_on_write to false for counters. Only for counters with a consistency level of ONE (CL=ONE).

Short option	Long option	Description
-F <NUM-DIFFERENT-KEYS>	--num-different-keys <NUM-DIFFERENT-KEYS>	Number of different keys. If less than NUM-KEYS, the same key is re-used multiple times. Default is NUM-KEYS.
-n <NUMKEYS>	--num-keys <NUMKEYS>	Number of keys to write or read. Default is 1,000,000.
-o <OPERATION>	--operation <OPERATION>	Operation to perform: INSERT, READ, INDEXED_RANGE_SLICE, MULTI_GET, COUNTER_ADD, COUNTER_GET. Default is INSERT.
-p <PORT>	--port <PORT>	Thrift port. Default is 9160.
-i <PROGRESS-INTERVAL>	--progress-interval <PROGRESS-INTERVAL>	The interval, in seconds, at which progress is output. Default is 10 seconds.
-Q <QUERY-NAMES>	--query-names <QUERY-NAMES>	Comma-separated list of column names to retrieve from each row.
-r	--random	Use random key generator. When used --stdev has no effect. Default is false.
-l <REPLICATION-FACTOR>	--replication-factor <REPLICATION-FACTOR>	Replication Factor to use when creating tables. Default is 1.
-R <REPLICATION-STRATEGY>	--replication-strategy <REPLICATION-STRATEGY>	Replication strategy to use (only on insert and when a keyspace does not exist.) The default is SimpleStrategy .
-T <SEND-TO>	--send-to <SEND-TO>	Sends the command as a request to the cassandra-stressd daemon at the specified IP address. The daemon must already be running at that address.
-N <SKIP-KEYS>	--skip-keys <SKIP-KEYS>	Fraction of keys to skip initially. Default is 0.
-s <STDEV>	--stdev <STDEV>	Standard deviation. Default is 0.1.
-O <STRATEGY-PROPERTIES>	--strategy-properties <STRATEGY-PROPERTIES>	Replication strategy properties in the following format: <dc_name>:<num>,<dc_name>:<num>, For use with NetworkTopologyStrategy .
-t <THREADS>	--threads <THREADS>	Number of threads to use. Default is 50.
-m	--unframed	Use unframed transport. Default is false.
-P	--use-prepared-statements	(CQL only) Perform queries using prepared statements.

Using the Daemon Mode

Usage for the daemon mode in binary installs.

Run the daemon from:

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

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

- Insert 1,000,000 rows to given host:

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

When the number of rows is not specified, one million rows are inserted.

- Read 1,000,000 rows from given host:

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

- Insert 10,000,000 rows across two nodes:

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

- Insert 10,000,000 rows across two nodes using the daemon mode:

```
/tools/bin/cassandra-stress -d 192.168.1.101,192.168.1.102 -n 10000000 --send-to 54.0.0.1
```

Interpreting the output of cassandra-stress

About the period output from the running tests.

Each line reports data for the interval between the last elapsed time and current elapsed time, which is set by the `--progress-interval` option (default 10 seconds).

```
7251,725,725,56.1,95.1,191.8,10
19523,1227,1227,41.6,86.1,189.1,21
41348,2182,2182,22.5,75.7,176.0,31
...
```

Data	Description
total	Total number of operations since the start of the test.
interval_op_rate	Number of operations performed during the interval.
interval_key_rate	Number of keys/rows read or written during the interval (normally be the same as <code>interval_op_rate</code> unless doing range slices).
latency	Average latency for each operation during that interval.
95th	95% of the time the latency was less than the number displayed in the column (Cassandra 1.2 or later).
99th	99% of the time the latency was less than the number displayed in the column (Cassandra 1.2 or later).
elapsed	Number of seconds elapsed since the beginning of the test.

The cassandra-shuffle utility

Shift a single-token-per-node architecture to virtual nodes (vnodes) without downtime.

The `cassandra-shuffle` utility splits up all the contiguous **partition ranges** (formerly token ranges) for each node and then randomly distributes them into **virtual nodes** throughout the cluster. Shuffling is a two-phase operation. The utility first schedules the range transfers and then begins transferring the scheduled ranges. You can shuffle on a per-data center basis and mix virtual node-enabled and non-virtual node data centers.

For a complete description of how it works, see the blog [Upgrading an existing cluster to vnodes](#).

In a terminal window:

1. In the `cassandra.yaml` file, set the `num_tokens` parameter.

A good starting point for this parameter is 256.

2. Restart the node.

The node sleeps for RING_DELAY to make sure its view of the ring is accurate, and then splits its current range into the number of specified tokens. However, while the range is split into many tokens, the range remains *contiguous*; it is still equivalent to what it was before, but with more tokens.

3. To distribute the tokens, initialize the shuffle operation:

```
shuffle create
```

4. Starts the transfers:

```
shuffle enable
```

5. To see what transfers remain at any point:

```
shuffle ls
```

Commands and options

Table 16: Commands

shuffle [options] <sub-command>	
create	Initialize the shuffle operation.
ls	Lists pending relocations.
clear	Clears pending relocations.
en[able]	Enables shuffling.
dis[able]	Disables shuffling.

Table 17: Options

Flag	Option	Description
-dc	--only-dc	Apply only to named DC (create only).
-u	--username	JMX username.
-tp	--thrift-port	Thrift port number (Default: 9160).
-p	--port	JMX port number (Default: 7199).
-tf	--thrift-framed	Enable framed transport for Thrift (Default: false).
-en	--and-enable	Immediately enable shuffling (create only).
pw	--password	JMX password.
-H	--help	Print help information.
-h	--host	JMX hostname or IP address (Default: localhost).
-th	--thrift-host	Thrift hostname or IP address (Default: JMX host).

The sstablescrib utility

Scrub the all the SSTables for the specified table.

Use this tool to fix (throw away) corrupted tables. Before using this tool, try rebuild the tables using **nodetool scrub**. Because corrupted rows are thrown away, run a **repair** after running this tool.

Usage:

- Packaged installs: `sstablescrub [options] <keyspace> <table>`
- Tarball installs: `install_location/bin/sstablescrub [options] <keyspace> <table>`

Table 18: Options

Flag	Option	Description
	<code>--debug</code>	Display stack traces.
<code>-h</code>	<code>--help</code>	Display help.
<code>-m</code>	<code>--manifest-check</code>	Only check and repair the leveled manifest, without actually scrubbing the SSTables.
<code>-v</code>	<code>--verbose</code>	Verbose output.

The sstable2json / json2sstable utilities

The sstable2json utility

Converts the on-disk SSTable representation of a table into a JSON formatted document.

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>.

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.

The output 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.

Tracking data expiration

The output of the `sstable2json` command reveals the life cycle of Cassandra data. In this procedure, you use the `sstable2json` to view data in a row that is not scheduled to expire, data that has been evicted and marked with a tombstone, and a row that has had data removed from it.

1. Create the `playlists` table in the `music` keyspace as shown in [Data modeling](#).
2. **Insert** the row of data about ZZ Top in playlists:

```
INSERT INTO music.playlists (id, song_order, song_id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204,
1,
a3e64f8f-bd44-4f28-b8d9-6938726e34d4,
'La Grange',
'ZZ Top',
'Tres Hombres');
```

3. Flush the data to disk.

```
sudo ./nodetool flush music playlists
```

You need to have access permission to the data directories to flush data to disk.

4. Look at the json representation of the SSTable data, for example, use `sstable2json` followed by the path to the data. Substitute the path to data for your Cassandra version and configuration:

```
sudo ./sstable2json
/var/lib/cassandra/data/music/playlists/music-playlists-ic-1-Data.db
```

Output is:

```
[
{"key": "62c3609282a13a0093d146196ee77204", "columns": [[
"1:", "", 1370179611971000], [
"1:album", "Tres Hombres", 1370179611971000], [
"1:artist", "ZZ Top", 1370179611971000], [
"1:song_id", "a3e64f8f-bd44-4f28-b8d9-6938726e34d4", 1370179611971000], [
"1:title", "La Grange", 1370179611971000]]}
]
```

5. Specify the time-to-live (TTL) for the ZZ Top row, for example 300 seconds.

```
INSERT INTO music.playlists
(id, song_order, song_id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204,
1,
a3e64f8f-bd44-4f28-b8d9-6938726e34d4,
'La Grange',
'ZZ Top',
'Tres Hombres')
USING TTL 300;
```

After inserting the TTL property on the row to expire the data, Cassandra marks the row with tombstones. You need to list all columns when re-inserting data if you want Cassandra to remove the entire row.

6. Flush the data to disk again.

Do this while the data is evicted, but before the time-to-live elapses and data is removed.

7. Run the `sstable2json` command again.

```
sudo ./sstable2json
/var/lib/cassandra/data/music/playlists/music-playlists-ic-2-Data.db
```

The tombstone markers--"e" followed by the TTL value, 300--are visible in the json representation of the data.

```
[
{"key": "62c3609282a13a0093d146196ee77204", "columns": [[
"1:", "", 1370179816450000, "e", 300, 1370180116], [
```

```
"1:album", "Tres Hombres", 1370179816450000, "e", 300, 1370180116], [
"1:artist", "ZZ Top", 1370179816450000, "e", 300, 1370180116], [
"1:song_id", "a3e64f8f-bd44-4f28-b8d9-6938726e34d4", 1370179816450000, "e", 300, 1370180116], [
"1:title", "La Grange", 1370179816450000, "e", 300, 1370180116]]}
]
```

8. After the TTL elapses, flush the data to disk again.
9. Run the `sstable2json` command again.

```
sudo ./sstable2json
/var/lib/cassandra/data/music/playlists/music-playlists-ic-2-Data.db
```

The json representation of the column data shows that the tombstones and data values for the ZZ Top row have been deleted from the SSTable. The values are now marked with "d":

```
[
{"key": "62c3609282a13a0093d146196ee77204", "columns": [[
"1:", "51ab4a14", 1370179816450000, "d"], [
"1:album", "51ab4a14", 1370179816450000, "d"], [
"1:artist", "51ab4a14", 1370179816450000, "d"], [
"1:song_id", "51ab4a14", 1370179816450000, "d"], [
"1:title", "51ab4a14", 1370179816450000, "d"]]}
]
```

Tracking counter columns

You can use the `sstable2json` command to get information about a counter column.

1. Run steps 1-3 of the [counter example](#), which loads data into a counter column. Flush the data to disk.

```
nodetool flush counterks page_view_counts
```

The counter is initialized to 1.

2. Run the `sstable2json` command followed by the path to the data. Substitute the path to data for your Cassandra version and configuration.

```
sudo ./sstable2json /var/lib/cassandra/data/counterks/page_view_counts/
page_view_counts-ic-1-Data.db
```

```
[
{"key": "7777772e64617461737461782e636f6d", "columns": [[
"home:", "", 1370187164256000], [
"home:counter_value", "0001000058852cd0cb9311e2940971f75c7d0641000000000000000100000000
]
```

3. Increase the counter column by 2 and flush the data to disk again.
4. Run the `sstable2json` command again.

```
sudo ./sstable2json
/var/lib/cassandra/data/counterks/page_view_counts/counterks-
page_view_counts-ib-2-Data.db
```

```
[
{"key": "7777772e64617461737461782e636f6d", "columns": [[
"home:", "", 1370187315683000], [
"home:counter_value", "0001000058852cd0cb9311e2940971f75c7d0641000000000000000100000000
]
```

-9223372036854775808 is the timestamp of the last delete.

The `json2sstable` utility

Converts a JSON representation of a table (aka column family) to a Cassandra usable SSTable format.

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>.

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.db` file in Cassandra's data directory. For example, `/var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db`.

sstablekeys

The `sstablekeys` utility is shorthand for `sstable2json` with the `-e` option.

Instead of dumping all of a 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`.

The sstableupgrade tool

Upgrade the SSTables in the specified table (or snapshot) to match the current version of Cassandra.

This tool rewrites the SSTables in the specified table to match the currently installed version of Cassandra.

If restoring with `sstableloader`, you must upgrade your snapshots before restoring for any snapshot taken in a major version older than the major version that Cassandra is currently running.

Usage:

- Packaged installs: `sstableupgrade [options] <keyspace> <cf> [snapshot]`
- Tarball installs: `install_location/bin/sstableupgrade [options] <keyspace> <cf> [snapshot]`

The snapshot option only upgrades the specified snapshot.

Table 19: Options

Flag	Option	Description
	<code>--debug</code>	Display stack traces.
<code>-h</code>	<code>--help</code>	Display help.

References

Starting and stopping Cassandra

Starting Cassandra as a service

Start the Cassandra Java server process for packaged installations.

Startup scripts are provided in `/etc/init.d`. The service runs as the *cassandra* user.

You must have root or sudo permissions to start Cassandra as a service.

On initial start-up, each node must be started one at a time, starting with your seed nodes:

```
$ sudo service cassandra start
```

On Enterprise Linux systems, the Cassandra service runs as a java process. On Debian systems, the Cassandra service runs as a jsvc process.

Starting Cassandra as a stand-alone process

Start the Cassandra Java server process for binary installations.

On initial start-up, each node must be started one at a time, starting with your seed nodes.

- To start Cassandra in the background:

```
$ cd install_location
$ bin/cassandra
```

- To start Cassandra in the foreground:

```
$ cd install_location
$ bin/cassandra -f
```

Stopping Cassandra as a service

Stop the Cassandra Java server process on packaged installations.

You must have root or sudo permissions to stop the Cassandra service:

```
$ sudo service cassandra stop
```

Stopping Cassandra as a stand-alone process

Stop the Cassandra Java server process on binary installations.

Find the Cassandra Java process ID (PID), and then kill the process using its PID number:

```
$ ps aux | grep cassandra
$ sudo kill <pid>
```

Clearing the data as a service

Remove all data from a packaged installation.

After **stopping** the service, run the following command:

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

This command clears the data from the **default** directories.

Clearing the data as a stand-alone process

Remove all data from a binary installations.

After **stopping** the process, run the following command from the install directory:

```
$ cd install_location
$ sudo rm -rf /var/lib/cassandra/*
```

This command clears the data from the **default** directories.

Install locations

Locations of the configuration files

Cassandra 2.0 unpacks files into the directories listed here.

Locations of the configuration files

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

- Packaged installs: `/etc/cassandra/conf`
- Tarball installs: `install_location/conf`

For DataStax Enterprise installs, see [Configuration File Locations](#).

Packaged install directories

The packaged releases install into these directories.

Directories	Description
<code>/var/lib/cassandra</code>	Data directories
<code>/var/log/cassandra</code>	Log directory
<code>/var/run/cassandra</code>	Runtime files
<code>/usr/share/cassandra</code>	Environment settings
<code>/usr/share/cassandra/lib</code>	JAR files
<code>/usr/bin</code>	Binary files
<code>/usr/sbin</code>	
<code>/etc/cassandra</code>	Configuration files
<code>/etc/init.d</code>	Service startup script
<code>/etc/security/limits.d</code>	Cassandra user limits
<code>/etc/default</code>	

Binary install directories

The binary tarball releases install into these directories.

Directories	Description
<code>bin</code>	Utilities and start scripts
<code>conf</code>	Configuration files and environment settings
<code>interface</code>	Thrift and Avro client APIs
<code>javadoc</code>	Cassandra Java API documentation
<code>lib</code>	JAR and license files

Troubleshooting

This section contains the following topics:

Reads are getting slower while writes are still fast

The cluster's IO capacity is not enough to handle the write load it is receiving.

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

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

Nodes seem to freeze after some period of time

Some portion of the JVM is being swapped out by the operating system (OS).

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 high probability that some portion of the JVM is being swapped out by the OS.

One way this might happen is if the `mmap DiskAccessMode` is used without JNA support. The address space will be exhausted by `mmap`, and the OS will decide to swap out some portion of the JVM that isn't in use, but eventually the JVM will try to GC this space. Adding the JNA libraries will solve this (they cannot be shipped with Cassandra due to carrying a GPL license, but are freely available) or the `DiskAccessMode` can be switched to `mmap_index_only`, which as the name implies will only `mmap` the indices, using much less address space.

DataStax recommends that Cassandra nodes disable swap entirely (`sudo swapoff --all`), since it is better to have the OS `OutOfMemory` (OOM) killer kill the Java process entirely than it is to have the JVM buried in swap and responding poorly. To make this change permanent, remove all swap file entries from `/etc/fstab`.

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

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 tables. 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

Nodetool commands can be run locally but not on other nodes in the cluster.

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

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

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

View of ring differs between some nodes

Indicates that the ring is in a bad state.

This situation can happen when not using virtual nodes (vnodes) 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 may not have open enough file descriptors.

Cassandra generally needs more than the default (1024) amount of file descriptors. 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 `lssof -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.

Cassandra errors

Insufficient as (address space) or memlock setting

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

Insufficient memlock settings

```
WARN [main ] 2011-06-15 09:58:56,861 CLibrary.java (line 118 ) Unable to lock
JVM memory (ENOMEM ).
```

Troubleshooting

This can result in part of the JVM being swapped out, especially with mmaped 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
```

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 making the changes permanent:

Packaged installs: Ensure that the following settings are included in the `/etc/security/limits.d/cassandra.conf` file:

```
cassandra - memlock unlimited  
cassandra - nofile 100000  
cassandra - nproc 32768  
cassandra - as unlimited
```

Tarball installs: Ensure that the following settings are included in the `/etc/security/limits.conf` file:

```
* - memlock unlimited  
* - nofile 100000  
* - nproc 32768  
* - as unlimited
```

If you run Cassandra as root, some Linux distributions such as Ubuntu, require setting the limits for root explicitly instead of using `*`:

```
root - memlock unlimited  
root - nofile 100000  
root - nproc 32768  
root - as unlimited
```

For CentOS, RHEL, OEL systems, also set the nproc limits in `/etc/security/limits.d/90-nproc.conf`:

```
* - nproc 32768
```

For all installations, add the following line to `/etc/sysctl.conf`:

```
vm.max_map_count = 131072
```

To make the changes take effect, reboot the server or run the following command:

```
$ sudo sysctl -p
```

To confirm the limits are applied to the Cassandra process, run the following command where `<pid>` is the process ID of the currently running Cassandra process:

```
$ cat /proc/<pid>/limits
```

OpsCenter errors

See the [OpsCenter Troubleshooting](#) documentation.

Cannot initialize class org.xerial.snappy.Snappy

An 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-libsnapappyjava.so` for Snappy compression is included in the `snappy-java-1.0.4.1.jar` file. When the JVM initializes the JAR, the library is added to the default temp directory. If the default temp directory is mounted with a `noexec` option, it results in the above exception.

One solution is to specify a different temp directory that has already been mounted without the `noexec` option, as follows:

- If you use the DSE/Cassandra command `$_BIN/dse cassandra` or `$_BIN/cassandra`, simply append the command line:
 - DSE: `bin/dse cassandra -t -Dorg.xerial.snappy.tmpdir=/path/to/newtmp`
 - Cassandra: `bin/cassandra -Dorg.xerial.snappy.tmpdir=/path/to/newtmp`
- If starting from a package using `service dse start` or `service cassandra start`, add a system environment variable `JVM_OPTS` with the value:

```
JVM_OPTS=-Dorg.xerial.snappy.tmpdir=/path/to/newtmp
```

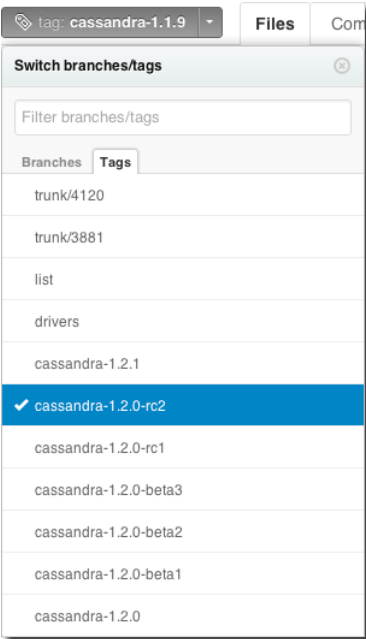
The default `cassandra-env.sh` looks for the variable and appends to it when starting the JVM.

DataStax Community release notes

Fixes and New Features in Cassandra.

Current DataStax Community version: 2.0.x

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

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 that 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. Called **table** in CQL.

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.

composite partition key

Stores columns of a row on more than one node using partition keys declared in nested parentheses of the PRIMARY KEY definition of a table.

compound primary key

A primary key consisting of the partition key, which determines on which node data is stored, and one or more additional **columns** that determine clustering.

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 that sends data from one data center to a node in another data center, and that node forwards the data to other nodes in its data center.

data center

A group of related nodes configured together within a cluster for replication and workload-segregation purposes. 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

A native Cassandra capability for finding a column in the database that does not involve using the primary key.

partition summary

A subset of the [partition index](#). By default, 1 partition 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.

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, or in the case of a compound key, multiple columns can declare those columns that form the primary key.

partition range

The limits of the partition that differ depending on the configured partitioner. Murmur3Partitioner (default) range is -2^{63} to $+2^{63}$ and RandomPartitioner range is 0 to $2^{127}-1$.

partition index

A list of primary keys and the start position of data.

primary key

The partition 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 primary key. 2) A collection of cells per combination of columns in the storage engine.

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.

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` whose password is also `cassandra`. A superuser grants initial permissions to access Cassandra data, and subsequently a user may or may not be given the permission to grant/revoke permissions.

table

A collection of ordered (by name) columns fetched by row. A row consists of columns and have a primary key. The first part of the key is a column name. Subsequent parts of a compound key are other column names that define the order of columns in the table.

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 `RandomPartitioner` 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. Also see [Expiring columns in Removing data](#).

weak consistency

When reading data, Cassandra performs [read repair](#) after returning results.

wide row

A data partition, which CQL transposes into familiar row-based resultsets.

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.