Operating Cassandra

Operating Cassandra

Backups

Backups

Apache Cassandra stores data in immutable SSTable files. Backups in Apache Cassandra database are backup copies of the database data that is stored as SSTable files. Backups are used for several purposes including the following:

- To store a data copy for durability
- To be able to restore a table if table data is lost due to node/partition/network failure
- To be able to transfer the SSTable files to a different machine; for portability

Types of Backups

Apache Cassandra supports two kinds of backup strategies.

- Snapshots
- Incremental Backups

A *snapshot* is a copy of a table's SSTable files at a given time, created via hard links. The DDL to create the table is stored as well. Snapshots may be created by a user or created automatically. The setting `snapshot_before_compaction` in the `cassandra.yaml` file determines if snapshots are created before each compaction. By default, `snapshot_before_compaction` is set to false. Snapshots may be created automatically before keyspace truncation or dropping of a table by setting `auto_snapshot` to true (default) in `cassandra.yaml`. Truncates could be delayed due to the auto snapshots and another setting in `cassandra.yaml` determines how long the coordinator should wait for truncates to complete. By default Cassandra waits 60 seconds for auto snapshots to complete.

An *incremental backup* is a copy of a table's SSTable files created by a hard link when memtables are flushed to disk as SSTables. Typically incremental backups are paired with snapshots to reduce the backup time as well as reduce disk space. Incremental backups are not enabled by default and must be enabled explicitly in `cassandra.yaml` (with `incremental_backups` setting) or with `nodetool`. Once enabled, Cassandra creates a hard link to each SSTable flushed or streamed locally in a `backups/` subdirectory of the keyspace data. Incremental backups of system tables are also created.

Data Directory Structure

The directory structure of Cassandra data consists of different directories for keyspaces, and tables with the data files within the table directories. Directories backups and snapshots to store backups and snapshots respectively for a particular table are also stored within the table directory. The directory structure for Cassandra is illustrated in Figure 1.

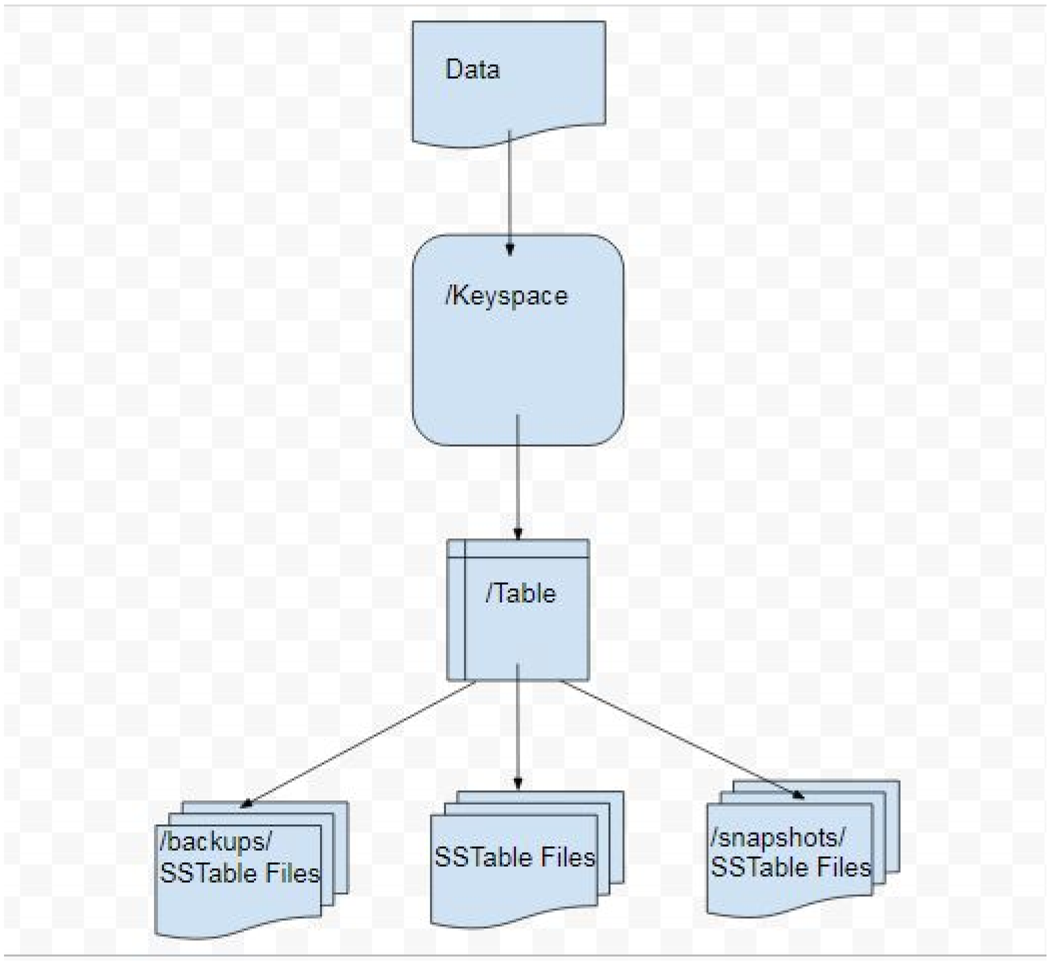


Figure 1. Directory Structure for Cassandra Data

Setting Up Example Tables for Backups and Snapshots

In this section we shall create some example data that could be used to demonstrate incremental backups and snapshots. We have used a three node Cassandra cluster. First, the keyspaces are created. Then tables are created within a keyspace and table data is added. We have used two keyspaces `cqlkeyspace` and `catalogkeyspace` with two tables within each.

Create the keyspace `cqlkeyspace`:

```
CREATE KEYSPACE cqlkeyspace
   WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};
```

Create two tables `t` and `t2` in the `cqlkeyspace` keyspace.

```
USE cqlkeyspace;
CREATE TABLE t (
   id int,
   k int,
   v text,
   PRIMARY KEY (id)
);
CREATE TABLE t2 (
   id int,
   k int,
   v text,
   PRIMARY KEY (id)
);
```

Add data to the tables:

```
INSERT INTO t (id, k, v) VALUES (0, 0, 'val0');
INSERT INTO t (id, k, v) VALUES (1, 1, 'val1');

INSERT INTO t2 (id, k, v) VALUES (0, 0, 'val0');
INSERT INTO t2 (id, k, v) VALUES (1, 1, 'val1');
INSERT INTO t2 (id, k, v) VALUES (2, 2, 'val2');
```

Query the table to list the data:

```
SELECT * FROM t;
SELECT * FROM t2;
```

results in

```
id | k | v
----+---+------
 1 | 1 | val1
 0 | 0 | val0

 (2 rows)


id | k | v
----+---+------
 1 | 1 | val1
 0 | 0 | val0
 2 | 2 | val2

 (3 rows)
```

Create a second keyspace `catalogkeyspace`:

```
CREATE KEYSPACE catalogkeyspace
   WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};
```

Create two tables `journal` and `magazine` in `catalogkeyspace`:

```
USE catalogkeyspace;
CREATE TABLE journal (
   id int,
   name text,
   publisher text,
   PRIMARY KEY (id)
);

CREATE TABLE magazine (
   id int,
   name text,
   publisher text,
   PRIMARY KEY (id)
);
```

Add data to the tables:

```
INSERT INTO journal (id, name, publisher) VALUES (0, 'Apache Cassandra Magazine', 'Apache
Cassandra');
INSERT INTO journal (id, name, publisher) VALUES (1, 'Couchbase Magazine', 'Couchbase');

INSERT INTO magazine (id, name, publisher) VALUES (0, 'Apache Cassandra Magazine',
'Apache Cassandra');
INSERT INTO magazine (id, name, publisher) VALUES (1, 'Couchbase Magazine', 'Couchbase');
```

Query the tables to list the data:

```
SELECT * FROM catalogkeyspace.journal;
SELECT * FROM catalogkeyspace.magazine;
```

results in

```
id | name                      | publisher
----+---------------------------+------------------
 1 |        Couchbase Magazine |        Couchbase
 0 | Apache Cassandra Magazine | Apache Cassandra
 (2 rows)

id | name                        | publisher
----+---------------------------+------------------
 1 |        Couchbase Magazine |        Couchbase
 0 | Apache Cassandra Magazine | Apache Cassandra

 (2 rows)
```

Snapshots

In this section, we demonstrate creating snapshots. The command used to create a snapshot is `nodetool snapshot` with the usage:

```
$ nodetool help snapshot
```

results in

```
NAME
       nodetool snapshot - Take a snapshot of specified keyspaces or a snapshot
       of the specified table

SYNOPSIS
       nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
               [(-pp | --print-port)] [(-pw <password> | --password <password>)]
               [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
               [(-u <username> | --username <username>)] snapshot
               [(-cf <table> | --column-family <table> | --table <table>)]
               [(-kt <ktlist> | --kt-list <ktlist> | -kc <ktlist> | --kc.list <ktlist>)]
               [(-sf | --skip-flush)] [(-t <tag> | --tag <tag>)] [--] [<keyspaces...>]

OPTIONS
       -cf <table>, --column-family <table>, --table <table>
           The table name (you must specify one and only one keyspace for using
           this option)

       -h <host>, --host <host>
           Node hostname or ip address

       -kt <ktlist>, --kt-list <ktlist>, -kc <ktlist>, --kc.list <ktlist>
           The list of Keyspace.table to take snapshot.(you must not specify
           only keyspace)

       -p <port>, --port <port>
```

```
            Remote jmx agent port number

        -pp, --print-port
            Operate in 4.0 mode with hosts disambiguated by port number

        -pw <password>, --password <password>
            Remote jmx agent password

        -pwf <passwordFilePath>, --password-file <passwordFilePath>
            Path to the JMX password file

        -sf, --skip-flush
            Do not flush memtables before snapshotting (snapshot will not
            contain unflushed data)

        -t <tag>, --tag <tag>
            The name of the snapshot

        -u <username>, --username <username>
            Remote jmx agent username

        --
            This option can be used to separate command-line options from the
            list of argument, (useful when arguments might be mistaken for
            command-line options

        [<keyspaces...>]
            List of keyspaces. By default, all keyspaces
```

Configuring for Snapshots

To demonstrate creating snapshots with Nodetool on the commandline we have set `auto_snapshots` setting to `false` in the `cassandra.yaml` file:

```
auto_snapshot: false
```

Also set `snapshot_before_compaction` to `false` to disable creating snapshots automatically before compaction:

```
snapshot_before_compaction: false
```

Creating Snapshots

Before creating any snapshots, search for snapshots and none will be listed:

```
$ find -name snapshots
```

We shall be using the example keyspaces and tables to create snapshots.

Taking Snapshots of all Tables in a Keyspace

Using the syntax above, create a snapshot called `catalog-ks` for all the tables in the `catalogkeyspace` keyspace:

```
$ nodetool snapshot --tag catalog-ks catalogkeyspace
```

results in

```
Requested creating snapshot(s) for [catalogkeyspace] with snapshot name [catalog-ks] and
options {skipFlush=false}
Snapshot directory: catalog-ks
```

Using the `find` command above, the snapshots and `snapshots` directories are now found with listed files similar to:

```
./cassandra/data/data/catalogkeyspace/journal-296a2d30c22a11e9b1350d927649052c/snapshots
./cassandra/data/data/catalogkeyspace/magazine-446eae30c22a11e9b1350d927649052c/snapshots
```

Snapshots of all tables in multiple keyspaces may be created similarly:

```
$ nodetool snapshot --tag catalog-cql-ks catalogkeyspace, cqlkeyspace
```

Taking Snapshots of Single Table in a Keyspace

To take a snapshot of a single table the `nodetool snapshot` command syntax becomes as follows:

```
$ nodetool snapshot --tag <tag> --table <table>  --<keyspace>
```

Using the syntax above, create a snapshot for table `magazine` in keyspace `catalogkeyspace`:

```
$ nodetool snapshot --tag magazine --table magazine  catalogkeyspace
```

results in

```
Requested creating snapshot(s) for [catalogkeyspace] with snapshot name [magazine] and
options {skipFlush=false}
Snapshot directory: magazine
```

Taking Snapshot of Multiple Tables from same Keyspace

To take snapshots of multiple tables in a keyspace the list of *Keyspace.table* must be specified with option `--kt-list`. For example, create snapshots for tables `t` and `t2` in the `cqlkeyspace` keyspace:

```
$ nodetool snapshot --kt-list cqlkeyspace.t,cqlkeyspace.t2 --tag multi-table
```

results in

```
Requested creating snapshot(s) for ["CQLKeyspace".t,"CQLKeyspace".t2] with snapshot name
[multi-
table] and options {skipFlush=false}
Snapshot directory: multi-table
```

Multiple snapshots of the same set of tables may be created and tagged with a different name. As an example, create another snapshot for the same set of tables `t` and `t2` in the `cqlkeyspace` keyspace and tag the snapshots differently:

```
$ nodetool snapshot --kt-list cqlkeyspace.t, cqlkeyspace.t2 --tag multi-table-2
```

results in

```
Requested creating snapshot(s) for ["CQLKeyspace".t,"CQLKeyspace".t2] with snapshot name
[multi-
table-2] and options {skipFlush=false}
Snapshot directory: multi-table-2
```

Taking Snapshot of Multiple Tables from Different Keyspaces

To take snapshots of multiple tables that are in different keyspaces the command syntax is the same as when multiple tables are in the same keyspace. Each <keyspace>.<table> must be specified separately in the `--kt-list` option.

For example, create a snapshot for table `t` in the `cqlkeyspace` and table `journal` in the catalogkeyspace and tag the snapshot `multi-ks`.

```
$ nodetool snapshot --kt-list catalogkeyspace.journal,cqlkeyspace.t --tag multi-ks
```

results in

```
Requested creating snapshot(s) for [catalogkeyspace.journal,cqlkeyspace.t] with snapshot
name [multi-ks] and options {skipFlush=false}
Snapshot directory: multi-ks
```

Listing Snapshots

To list snapshots use the `nodetool listsnapshots` command. All the snapshots that we created in the preceding examples get listed:

```
$ nodetool listsnapshots
```

results in

```
Snapshot Details:
Snapshot name Keyspace name   Column family name True size Size on disk
multi-table   cqlkeyspace     t2                 4.86 KiB  5.67 KiB
multi-table   cqlkeyspace     t                  4.89 KiB  5.7 KiB
multi-ks      cqlkeyspace     t                  4.89 KiB  5.7 KiB
multi-ks      catalogkeyspace journal            4.9 KiB   5.73 KiB
magazine      catalogkeyspace magazine           4.9 KiB   5.73 KiB
multi-table-2 cqlkeyspace     t2                 4.86 KiB  5.67 KiB
multi-table-2 cqlkeyspace     t                  4.89 KiB  5.7 KiB
catalog-ks    catalogkeyspace journal            4.9 KiB   5.73 KiB
catalog-ks    catalogkeyspace magazine           4.9 KiB   5.73 KiB

Total TrueDiskSpaceUsed: 44.02 KiB
```

Finding Snapshots Directories

The `snapshots` directories may be listed with `find –name snapshots` command:

```
$ find -name snapshots
```

results in

```
./cassandra/data/data/cqlkeyspace/t-d132e240c21711e9bbee19821dcea330/snapshots
./cassandra/data/data/cqlkeyspace/t2-d993a390c22911e9b1350d927649052c/snapshots
./cassandra/data/data/catalogkeyspace/journal-296a2d30c22a11e9b1350d927649052c/snapshots
./cassandra/data/data/catalogkeyspace/magazine-446eae30c22a11e9b1350d927649052c/snapshots
```

To list the snapshots for a particular table first change to the snapshots directory for that table. For example, list the snapshots for the `catalogkeyspace/journal` table:

```
$ cd ./cassandra/data/data/catalogkeyspace/journal-
296a2d30c22a11e9b1350d927649052c/snapshots && ls -l
```

results in

```
total 0
drwxrwxr-x. 2 ec2-user ec2-user 265 Aug 19 02:44 catalog-ks
drwxrwxr-x. 2 ec2-user ec2-user 265 Aug 19 02:52 multi-ks
```

A `snapshots` directory lists the SSTable files in the snapshot. A `schema.cql` file is also created in each snapshot that defines schema that can recreate the table with CQL when restoring from a snapshot:

```
$ cd catalog-ks && ls -l
```

results in

```
total 44
-rw-rw-r--. 1 ec2-user ec2-user   31 Aug 19 02:44 manifest.jsonZ
-rw-rw-r--. 4 ec2-user ec2-user   47 Aug 19 02:38 na-1-big-CompressionInfo.db
-rw-rw-r--. 4 ec2-user ec2-user   97 Aug 19 02:38 na-1-big-Data.db
-rw-rw-r--. 4 ec2-user ec2-user   10 Aug 19 02:38 na-1-big-Digest.crc32
-rw-rw-r--. 4 ec2-user ec2-user   16 Aug 19 02:38 na-1-big-Filter.db
-rw-rw-r--. 4 ec2-user ec2-user   16 Aug 19 02:38 na-1-big-Index.db
-rw-rw-r--. 4 ec2-user ec2-user 4687 Aug 19 02:38 na-1-big-Statistics.db
-rw-rw-r--. 4 ec2-user ec2-user   56 Aug 19 02:38 na-1-big-Summary.db
-rw-rw-r--. 4 ec2-user ec2-user   92 Aug 19 02:38 na-1-big-TOC.txt
-rw-rw-r--. 1 ec2-user ec2-user  814 Aug 19 02:44 schema.cql
```

Clearing Snapshots

Snapshots may be cleared or deleted with the `nodetool clearsnapshot` command. Either a specific snapshot name must be specified or the `–all` option must be specified.

For example, delete a snapshot called `magazine` from keyspace `cqlkeyspace`:

```
$ nodetool clearsnapshot -t magazine cqlkeyspace
```

or delete all snapshots from `cqlkeyspace` with the –all option:

```
$ nodetool clearsnapshot -all cqlkeyspace
```

Incremental Backups

In the following sections, we shall discuss configuring and creating incremental backups.

Configuring for Incremental Backups

To create incremental backups set `incremental_backups` to `true` in `cassandra.yaml`.

```
incremental_backups: true
```

This is the only setting needed to create incremental backups. By default `incremental_backups` setting is set to `false` because a new set of SSTable files is created for each data flush and if several CQL statements are to be run the `backups` directory could fill up quickly and use up storage that is needed to store table data. Incremental backups may also be enabled on the command line with the nodetool command `nodetool enablebackup`. Incremental backups may be disabled with `nodetool disablebackup` command. Status of incremental backups, whether they are enabled may be checked with `nodetool statusbackup`.

Creating Incremental Backups

After each table is created flush the table data with `nodetool flush` command. Incremental backups get created.

```
$ nodetool flush cqlkeyspace t
$ nodetool flush cqlkeyspace t2
$ nodetool flush catalogkeyspace journal magazine
```

Finding Incremental Backups

Incremental backups are created within the Cassandra's `data` directory within a table directory. Backups may be found with following command.

```
$ find -name backups
```

results in

```
./cassandra/data/data/cqlkeyspace/t-d132e240c21711e9bbee19821dcea330/backups
./cassandra/data/data/cqlkeyspace/t2-d993a390c22911e9b1350d927649052c/backups
./cassandra/data/data/catalogkeyspace/journal-296a2d30c22a11e9b1350d927649052c/backups
./cassandra/data/data/catalogkeyspace/magazine-446eae30c22a11e9b1350d927649052c/backups
```

Creating an Incremental Backup

This section discusses how incremental backups are created in more detail using the keyspace and table previously created.

Flush the keyspace and table:

```
$ nodetool flush cqlkeyspace t
```

A search for backups and a `backups` directory will list a backup directory, even if we have added no table data yet.

```
$ find -name backups
```

results in

```
./cassandra/data/data/cqlkeyspace/t-d132e240c21711e9bbee19821dcea330/backups
```

Checking the `backups` directory will show that there are also no backup files:

```
$ cd ./cassandra/data/data/cqlkeyspace/t-d132e240c21711e9bbee19821dcea330/backups && ls
-l
```

results in

```
total 0
```

If a row of data is added to the data, running the `nodetool flush` command will flush the table data and an incremental backup will be created:

```
$ nodetool flush cqlkeyspace t
$ cd ./cassandra/data/data/cqlkeyspace/t-d132e240c21711e9bbee19821dcea330/backups && ls
-l
```

results in

```
total 36
-rw-rw-r--. 2 ec2-user ec2-user   47 Aug 19 00:32 na-1-big-CompressionInfo.db
-rw-rw-r--. 2 ec2-user ec2-user   43 Aug 19 00:32 na-1-big-Data.db
-rw-rw-r--. 2 ec2-user ec2-user   10 Aug 19 00:32 na-1-big-Digest.crc32
-rw-rw-r--. 2 ec2-user ec2-user   16 Aug 19 00:32 na-1-big-Filter.db
-rw-rw-r--. 2 ec2-user ec2-user    8 Aug 19 00:32 na-1-big-Index.db
-rw-rw-r--. 2 ec2-user ec2-user 4673 Aug 19 00:32 na-1-big-Statistics.db
-rw-rw-r--. 2 ec2-user ec2-user   56 Aug 19 00:32 na-1-big-Summary.db
-rw-rw-r--. 2 ec2-user ec2-user   92 Aug 19 00:32 na-1-big-TOC.txt
```

NOTE The `backups` directory for any table, such as `cqlkeyspace/t` is created in the `data` directory for that table.

Adding another row of data and flushing will result in another set of incremental backup files. The SSTable files are timestamped, which distinguishes the first incremental backup from the second:

```
total 72
-rw-rw-r--. 2 ec2-user ec2-user   47 Aug 19 00:32 na-1-big-CompressionInfo.db
-rw-rw-r--. 2 ec2-user ec2-user   43 Aug 19 00:32 na-1-big-Data.db
-rw-rw-r--. 2 ec2-user ec2-user   10 Aug 19 00:32 na-1-big-Digest.crc32
-rw-rw-r--. 2 ec2-user ec2-user   16 Aug 19 00:32 na-1-big-Filter.db
-rw-rw-r--. 2 ec2-user ec2-user    8 Aug 19 00:32 na-1-big-Index.db
-rw-rw-r--. 2 ec2-user ec2-user 4673 Aug 19 00:32 na-1-big-Statistics.db
-rw-rw-r--. 2 ec2-user ec2-user   56 Aug 19 00:32 na-1-big-Summary.db
-rw-rw-r--. 2 ec2-user ec2-user   92 Aug 19 00:32 na-1-big-TOC.txt
-rw-rw-r--. 2 ec2-user ec2-user   47 Aug 19 00:35 na-2-big-CompressionInfo.db
-rw-rw-r--. 2 ec2-user ec2-user   41 Aug 19 00:35 na-2-big-Data.db
-rw-rw-r--. 2 ec2-user ec2-user   10 Aug 19 00:35 na-2-big-Digest.crc32
-rw-rw-r--. 2 ec2-user ec2-user   16 Aug 19 00:35 na-2-big-Filter.db
-rw-rw-r--. 2 ec2-user ec2-user    8 Aug 19 00:35 na-2-big-Index.db
-rw-rw-r--. 2 ec2-user ec2-user 4673 Aug 19 00:35 na-2-big-Statistics.db
-rw-rw-r--. 2 ec2-user ec2-user   56 Aug 19 00:35 na-2-big-Summary.db
```

```
-rw-rw-r--. 2 ec2-user ec2-user   92 Aug 19 00:35 na-2-big-TOC.txt
```

Restoring from Incremental Backups and Snapshots

The two main tools/commands for restoring a table after it has been dropped are:

- sstableloader
- nodetool refresh

A snapshot contains essentially the same set of SSTable files as an incremental backup does with a few additional files. A snapshot includes a `schema.cql` file for the schema DDL to create a table in CQL. A table backup does not include DDL which must be obtained from a snapshot when restoring from an incremental backup.

Bloom Filters

Bloom Filters

In the read path, Cassandra merges data on disk (in SSTables) with data in RAM (in memtables). To avoid checking every SSTable data file for the partition being requested, Cassandra employs a data structure known as a bloom filter.

Bloom filters are a probabilistic data structure that allows Cassandra to determine one of two possible states: - The data definitely does not exist in the given file, or - The data probably exists in the given file.

While bloom filters can not guarantee that the data exists in a given SSTable, bloom filters can be made more accurate by allowing them to consume more RAM. Operators have the opportunity to tune this behavior per table by adjusting the the `bloom_filter_fp_chance` to a float between 0 and 1.

The default value for `bloom_filter_fp_chance` is 0.1 for tables using LeveledCompactionStrategy and 0.01 for all other cases.

Bloom filters are stored in RAM, but are stored offheap, so operators should not consider bloom filters when selecting the maximum heap size. As accuracy improves (as the `bloom_filter_fp_chance` gets closer to 0), memory usage increases non-linearly - the bloom filter for `bloom_filter_fp_chance = 0.01` will require about three times as much memory as the same table with `bloom_filter_fp_chance = 0.1`.

Typical values for `bloom_filter_fp_chance` are usually between 0.01 (1%) to 0.1 (10%) false-positive chance, where Cassandra may scan an SSTable for a row, only to find that it does not exist on the disk. The parameter should be tuned by use case:

- Users with more RAM and slower disks may benefit from setting the `bloom_filter_fp_chance` to a numerically lower number (such as 0.01) to avoid excess IO operations
- Users with less RAM, more dense nodes, or very fast disks may tolerate a higher `bloom_filter_fp_chance` in order to save RAM at the expense of excess IO operations
- In workloads that rarely read, or that only perform reads by scanning the entire data set (such as analytics workloads), setting the `bloom_filter_fp_chance` to a much higher number is acceptable.

Changing

The bloom filter false positive chance is visible in the `DESCRIBE TABLE` output as the field `bloom_filter_fp_chance`. Operators can change the value with an `ALTER TABLE` statement: :

```
ALTER TABLE keyspace.table WITH bloom_filter_fp_chance=0.01
```

Operators should be aware, however, that this change is not immediate: the bloom filter is calculated when the file is written, and persisted on disk as the Filter component of the SSTable. Upon issuing an

`ALTER TABLE` statement, new files on disk will be written with the new `bloom_filter_fp_chance`, but existing sstables will not be modified until they are compacted - if an operator needs a change to `bloom_filter_fp_chance` to take effect, they can trigger an SSTable rewrite using `nodetool scrub` or `nodetool upgradesstables -a`, both of which will rebuild the sstables on disk, regenerating the bloom filters in the progress.

Bulk Loading

Bulk Loading

Bulk loading Apache Cassandra data is supported by different tools. The data to bulk load must be in the form of SSTables. Cassandra does not support loading data in any other format such as CSV, JSON, and XML directly. Although the cqlsh `COPY` command can load CSV data, it is not a good option for amounts of data. Bulk loading is used to:

- Restore incremental backups and snapshots. Backups and snapshots are already in the form of SSTables.
- Load existing SSTables into another cluster. The data can have a different number of nodes or replication strategy.
- Load external data to a cluster.

Tools for Bulk Loading

Cassandra provides two commands or tools for bulk loading data:

- Cassandra Bulk loader, also called `sstableloader`
- The `nodetool import` command

The `sstableloader` and `nodetool import` are accessible if the Cassandra installation `bin` directory is in the `PATH` environment variable. Or these may be accessed directly from the `bin` directory. The examples use the keyspaces and tables created in **Backups**.

Using sstableloader

The `sstableloader` is the main tool for bulk uploading data. `sstableloader` streams SSTable data files to a running cluster, conforming to the replication strategy and replication factor. The table to upload data to does need not to be empty.

The only requirements to run `sstableloader` are:

- One or more comma separated initial hosts to connect to and get ring information
- A directory path for the SSTables to load

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

Sstableloader bulk loads the SSTables found in the directory `<dir_path>` to the configured cluster. The `<dir_path>` is used as the target *keyspace/table* name. For example, to load an SSTable named `Standard1-`

g-1-Data.db into Keyspace1/Standard1, you will need to have the files Standard1-g-1-Data.db and Standard1-g-1-Index.db in a directory /path/to/Keyspace1/Standard1/.

Sstableloader Option to accept Target keyspace name

Often as part of a backup strategy, some Cassandra DBAs store an entire data directory. When corruption in the data is found, restoring data in the same cluster (for large clusters 200 nodes) is common, but with a different keyspace name.

Currently `sstableloader` derives keyspace name from the folder structure. As an option, to specify target keyspace name as part of `sstableloader`, version 4.0 adds support for the `--target-keyspace` option (CASSANDRA-13884).

The following options are supported, with `-d,--nodes <initial hosts>` required:

```
-alg,--ssl-alg <ALGORITHM>                                  Client SSL: algorithm

-ap,--auth-provider <auth provider>                         Custom
                                                            AuthProvider class name for
                                                            cassandra authentication
-ciphers,--ssl-ciphers <CIPHER-SUITES>                      Client SSL:
                                                            comma-separated list of
                                                            encryption suites to use
-cph,--connections-per-host <connectionsPerHost>            Number of
                                                            concurrent connections-per-
host.
-d,--nodes <initial hosts>                                  Required.
                                                            Try to connect to these
hosts (comma separated) initially for ring information

--entire-sstable-throttle-mib <throttle-mib>                Entire SSTable throttle
                                                            speed in MiB/s (default 0
for unlimited).

--entire-sstable-inter-dc-throttle-mib                      <inter-dc-throttle-mib>
                                                            Entire SSTable inter-
datacenter throttle
                                                            speed in MiB/s (default 0
for unlimited).

-f,--conf-path <path to config file>                        cassandra.yaml file path for
streaming throughput and client/server SSL.

-h,--help                                                   Display this help message

-i,--ignore <NODES>                                         Don't stream to this (comma
separated) list of nodes

-idct,--inter-dc-throttle <inter-dc-throttle>               (deprecated) Inter-
datacenter throttle speed in Mbits (default 0 for unlimited).
                                                            Use --inter-dc-throttle-mib
instead.

--inter-dc-throttle-mib <inter-dc-throttle-mib>             Inter-datacenter throttle
speed in MiB/s (default 0 for unlimited)

-k,--target-keyspace <target keyspace name>                 Target
                                                            keyspace name
-ks,--keystore <KEYSTORE>                                   Client SSL:
                                                            full path to keystore
-kspw,--keystore-password <KEYSTORE-PASSWORD>               Client SSL:
                                                            password of the keystore
--no-progress                                               Don't
                                                            display progress
-p,--port <native transport port>                           Port used
                                                            for native connection
(default 9042)
-prtcl,--ssl-protocol <PROTOCOL>                            Client SSL:
                                                            connections protocol to use
(default: TLS)
-pw,--password <password>                                   Password for
                                                            cassandra authentication
-sp,--storage-port <storage port>                           Port used
                                                            for internode communication
(default 7000)
-spd,--server-port-discovery <allow server port discovery>   Use ports
                                                            published by server to
decide how to connect. With SSL requires StartTLS
                                                            to be used.
-ssp,--ssl-storage-port <ssl storage port>                  Port used
                                                            for TLS internode
communication (default 7001)
-st,--store-type <STORE-TYPE>                               Client SSL:
                                                            type of store
-t,--throttle <throttle>                                    (deprecated) Throttle speed
in Mbits (default 0 for unlimited).
                                                            Use --throttle-mib instead.
--throttle-mib <throttle-mib>                               Throttle
                                                            speed in MiB/s (default 0
for unlimited)
-ts,--truststore <TRUSTSTORE>                               Client SSL:
                                                            full path to truststore
-tspw,--truststore-password <TRUSTSTORE-PASSWORD>           Client SSL:
                                                            Password of the truststore
```

```
-u,--username <username>                                 Username for
                                                         cassandra authentication
-v,--verbose                                             verbose
                                                         output
```

The `cassandra.yaml` file can be provided on the command-line with `-f` option to set up streaming throughput, client and server encryption options. Only `stream_throughput_outbound_megabits_per_sec`, `server_encryption_options` and `client_encryption_options` are read from the `cassandra.yaml` file. You can override options read from `cassandra.yaml` with corresponding command line options.

A sstableloader Demo

An example shows how to use `sstableloader` to upload incremental backup data for the table `catalogkeyspace.magazine`. In addition, a snapshot of the same table is created to bulk upload, also with `sstableloader`.

The backups and snapshots for the `catalogkeyspace.magazine` table are listed as follows:

```
$ cd ./cassandra/data/data/catalogkeyspace/magazine-446eae30c22a11e9b1350d927649052c &&
ls -l
```

results in

```
total 0
drwxrwxr-x. 2 ec2-user ec2-user 226 Aug 19 02:38 backups
drwxrwxr-x. 4 ec2-user ec2-user  40 Aug 19 02:45 snapshots
```

The directory path structure of SSTables to be uploaded using `sstableloader` is used as the target keyspace/table. You can directly upload from the `backups` and `snapshots` directories respectively, if the directory structure is in the format used by `sstableloader`. But the directory path of backups and snapshots for SSTables is `/catalogkeyspace/magazine-446eae30c22a11e9b1350d927649052c/backups` and `/catalogkeyspace/magazine-446eae30c22a11e9b1350d927649052c/snapshots` respectively, and cannot be used to upload SSTables to `catalogkeyspace.magazine` table. The directory path structure must be `/catalogkeyspace/magazine/` to use `sstableloader`. Create a new directory structure to upload SSTables with `sstableloader` located at `/catalogkeyspace/magazine` and set appropriate permissions.

```
$ sudo mkdir -p /catalogkeyspace/magazine
$ sudo chmod -R 777 /catalogkeyspace/magazine
```

Bulk Loading from an Incremental Backup

An incremental backup does not include the DDL for a table; the table must already exist. If the table was dropped, it can be created using the `schema.cql` file generated with every snapshot of a table. Prior to using `sstableloader` to load SSTables to the `magazine` table, the table must exist. The table does not need to be empty but we have used an empty table as indicated by a CQL query:

```
SELECT * FROM magazine;
```

results in

```
id | name | publisher
----+------+-----------

(0 rows)
```

After creating the table to upload to, copy the SSTable files from the `backups` directory to the `/catalogkeyspace/magazine/` directory.

```
$ sudo cp ./cassandra/data/data/catalogkeyspace/magazine-
446eae30c22a11e9b1350d927649052c/backups/* \
/catalogkeyspace/magazine/
```

Run the `sstableloader` to upload SSTables from the `/catalogkeyspace/magazine/` directory.

```
$ sstableloader --nodes 10.0.2.238  /catalogkeyspace/magazine/
```

The output from the `sstableloader` command should be similar to this listing:

```
$ sstableloader --nodes 10.0.2.238  /catalogkeyspace/magazine/
```

results in

```
Opening SSTables and calculating sections to stream
Streaming relevant part of /catalogkeyspace/magazine/na-1-big-Data.db
/catalogkeyspace/magazine/na-2-big-Data.db  to [35.173.233.153:7000, 10.0.2.238:7000,
54.158.45.75:7000]
progress: [35.173.233.153:7000]0:1/2 88 % total: 88% 0.018KiB/s (avg: 0.018KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% total: 176% 33.807KiB/s (avg: 0.036KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% total: 176% 0.000KiB/s (avg: 0.029KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% [10.0.2.238:7000]0:1/2 39 % total: 81%
0.115KiB/s
(avg: 0.024KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% [10.0.2.238:7000]0:2/2 78 % total: 108%
```

```
97.683KiB/s (avg: 0.033KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% [10.0.2.238:7000]0:2/2 78 %
[54.158.45.75:7000]0:1/2 39 % total: 80% 0.233KiB/s (avg: 0.040KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% [10.0.2.238:7000]0:2/2 78 %
[54.158.45.75:7000]0:2/2 78 % total: 96% 88.522KiB/s (avg: 0.049KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% [10.0.2.238:7000]0:2/2 78 %
[54.158.45.75:7000]0:2/2 78 % total: 96% 0.000KiB/s (avg: 0.045KiB/s)
progress: [35.173.233.153:7000]0:2/2 176% [10.0.2.238:7000]0:2/2 78 %
[54.158.45.75:7000]0:2/2 78 % total: 96% 0.000KiB/s (avg: 0.044KiB/s)
```

After the `sstableloader` has finished loading the data, run a query the `magazine` table to check:

```
SELECT * FROM magazine;
```

results in

```
id | name                      | publisher
----+---------------------------+------------------
 1 |        Couchbase Magazine |        Couchbase
 0 | Apache Cassandra Magazine | Apache Cassandra

(2 rows)
```

Bulk Loading from a Snapshot

Restoring a snapshot of a table to the same table can be easily accomplished:

If the directory structure needed to load SSTables to `catalogkeyspace.magazine` does not exist create the directories and set appropriate permissions:

```
$ sudo mkdir -p /catalogkeyspace/magazine
$ sudo chmod -R 777 /catalogkeyspace/magazine
```

Remove any files from the directory, so that the snapshot files can be copied without interference:

```
$ sudo rm /catalogkeyspace/magazine/*
$ cd /catalogkeyspace/magazine/
$ ls -l
```

results in

```
total 0
```

Copy the snapshot files to the `/catalogkeyspace/magazine` directory.

```
$ sudo cp ./cassandra/data/data/catalogkeyspace/magazine-
446eae30c22a11e9b1350d927649052c/snapshots/magazine/* \
/catalogkeyspace/magazine
```

List the files in the `/catalogkeyspace/magazine` directory. The `schema.cql` will also be listed.

```
$ cd /catalogkeyspace/magazine && ls -l
```

results in

```
total 44
-rw-r--r--. 1 root root   31 Aug 19 04:13 manifest.json
-rw-r--r--. 1 root root   47 Aug 19 04:13 na-1-big-CompressionInfo.db
-rw-r--r--. 1 root root   97 Aug 19 04:13 na-1-big-Data.db
-rw-r--r--. 1 root root   10 Aug 19 04:13 na-1-big-Digest.crc32
-rw-r--r--. 1 root root   16 Aug 19 04:13 na-1-big-Filter.db
-rw-r--r--. 1 root root   16 Aug 19 04:13 na-1-big-Index.db
-rw-r--r--. 1 root root 4687 Aug 19 04:13 na-1-big-Statistics.db
-rw-r--r--. 1 root root   56 Aug 19 04:13 na-1-big-Summary.db
-rw-r--r--. 1 root root   92 Aug 19 04:13 na-1-big-TOC.txt
-rw-r--r--. 1 root root  815 Aug 19 04:13 schema.cql
```

Alternatively create symlinks to the snapshot folder instead of copying the data:

```
$ mkdir <keyspace_name>
$ ln -s <path_to_snapshot_folder> <keyspace_name>/<table_name>
```

If the `magazine` table was dropped, run the DDL in the `schema.cql` to create the table. Run the `sstableloader` with the following command:

```
$ sstableloader --nodes 10.0.2.238  /catalogkeyspace/magazine/
```

As the output from the command indicates, SSTables get streamed to the cluster:

```
Established connection to initial hosts
Opening SSTables and calculating sections to stream
```

```
Streaming relevant part of /catalogkeyspace/magazine/na-1-big-Data.db  to
[35.173.233.153:7000, 10.0.2.238:7000, 54.158.45.75:7000]
progress: [35.173.233.153:7000]0:1/1 176% total: 176% 0.017KiB/s (avg: 0.017KiB/s)
progress: [35.173.233.153:7000]0:1/1 176% total: 176% 0.000KiB/s (avg: 0.014KiB/s)
progress: [35.173.233.153:7000]0:1/1 176% [10.0.2.238:7000]0:1/1 78 % total: 108%
0.115KiB/s
(avg: 0.017KiB/s)
progress: [35.173.233.153:7000]0:1/1 176% [10.0.2.238:7000]0:1/1 78 %
[54.158.45.75:7000]0:1/1 78 % total: 96% 0.232KiB/s (avg: 0.024KiB/s)
progress: [35.173.233.153:7000]0:1/1 176% [10.0.2.238:7000]0:1/1 78 %
[54.158.45.75:7000]0:1/1 78 % total: 96% 0.000KiB/s (avg: 0.022KiB/s)
progress: [35.173.233.153:7000]0:1/1 176% [10.0.2.238:7000]0:1/1 78 %
[54.158.45.75:7000]0:1/1 78 % total: 96% 0.000KiB/s (avg: 0.021KiB/s)
```

Some other requirements of `sstableloader` that should be kept into consideration are:

- The SSTables loaded must be compatible with the Cassandra version being loaded into.
- Repairing tables that have been loaded into a different cluster does not repair the source tables.
- Sstableloader makes use of port 7000 for internode communication.
- Before restoring incremental backups, run `nodetool flush` to backup any data in memtables.

Using nodetool import

Importing SSTables into a table using the `nodetool import` command is recommended instead of the deprecated `nodetool refresh` command. The `nodetool import` command has an option to load new SSTables from a separate directory.

The command usage is as follows:

```
nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
       [(-pp | --print-port)] [(-pw <password> | --password <password>)]
       [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
       [(-u <username> | --username <username>)] import
       [(-c | --no-invalidate-caches)] [(-e | --extended-verify)]
       [(-l | --keep-level)] [(-q | --quick)] [(-r | --keep-repaired)]
       [(-t | --no-tokens)] [(-v | --no-verify)] [--] <keyspace> <table>
       <directory> ...
```

The arguments `keyspace`, `table` name and `directory` are required.

The following options are supported:

```
-c, --no-invalidate-caches
```

```
    Don't invalidate the row cache when importing

-e, --extended-verify
    Run an extended verify, verifying all values in the new SSTables

-h <host>, --host <host>
    Node hostname or ip address

-l, --keep-level
    Keep the level on the new SSTables

-p <port>, --port <port>
    Remote jmx agent port number

-pp, --print-port
    Operate in 4.0 mode with hosts disambiguated by port number

-pw <password>, --password <password>
    Remote jmx agent password

-pwf <passwordFilePath>, --password-file <passwordFilePath>
    Path to the JMX password file

-q, --quick
    Do a quick import without verifying SSTables, clearing row cache or
    checking in which data directory to put the file

-r, --keep-repaired
    Keep any repaired information from the SSTables

-t, --no-tokens
    Don't verify that all tokens in the new SSTable are owned by the
    current node

-u <username>, --username <username>
    Remote jmx agent username

-v, --no-verify
    Don't verify new SSTables

--
    This option can be used to separate command-line options from the
    list of argument, (useful when arguments might be mistaken for
    command-line options
```

Because the keyspace and table are specified on the command line for `nodetool import`, there is not the same requirement as with `sstableloader`, to have the SSTables in a specific directory path. When importing snapshots or incremental backups with `nodetool import`, the SSTables don’t need to be copied to another directory.

Importing Data from an Incremental Backup

Using `nodetool import` to import SSTables from an incremental backup, and restoring the table is shown below.

```
DROP table t;
```

An incremental backup for a table does not include the schema definition for the table. If the schema definition is not kept as a separate backup, the `schema.cql` from a backup of the table may be used to create the table as follows:

```
CREATE TABLE IF NOT EXISTS cqlkeyspace.t (
   id int PRIMARY KEY,
   k int,
   v text)
   WITH ID = d132e240-c217-11e9-bbee-19821dcea330
   AND bloom_filter_fp_chance = 0.01
   AND crc_check_chance = 1.0
   AND default_time_to_live = 0
   AND gc_grace_seconds = 864000
   AND min_index_interval = 128
   AND max_index_interval = 2048
   AND memtable_flush_period_in_ms = 0
   AND speculative_retry = '99p'
   AND additional_write_policy = '99p'
   AND comment = ''
   AND caching = { 'keys': 'ALL', 'rows_per_partition': 'NONE' }
   AND compaction = { 'max_threshold': '32', 'min_threshold': '4',
   'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy' }
   AND compression = { 'chunk_length_in_kb': '16', 'class':
   'org.apache.cassandra.io.compress.LZ4Compressor' }
   AND cdc = false
   AND extensions = {  }
;
```

Initially the table could be empty, but does not have to be.

```
SELECT * FROM t;
```

```
id | k | v
----+---+---

(0 rows)
```

Run the `nodetool import` command, providing the keyspace, table and the backups directory. Don’t copy the table backups to another directory, as with `sstableloader`.

```
$ nodetool import -- cqlkeyspace t \
./cassandra/data/data/cqlkeyspace/t-d132e240c21711e9bbee19821dcea330/backups
```

The SSTables are imported into the table. Run a query in cqlsh to check:

```
SELECT * FROM t;
```

```
id | k | v
----+---+------
 1 | 1 | val1
 0 | 0 | val0

(2 rows)
```

Importing Data from a Snapshot

Importing SSTables from a snapshot with the `nodetool import` command is similar to importing SSTables from an incremental backup. Shown here is an import of a snapshot for table `catalogkeyspace.journal`, after dropping the table to demonstrate the restore.

```
USE CATALOGKEYSPACE;
DROP TABLE journal;
```

Use the `catalog-ks` snapshot for the `journal` table. Check the files in the snapshot, and note the existence of the `schema.cql` file.

```
$ ls -l
```

```
total 44
-rw-rw-r--. 1 ec2-user ec2-user   31 Aug 19 02:44 manifest.json
```

```
-rw-rw-r--. 3 ec2-user ec2-user   47 Aug 19 02:38 na-1-big-CompressionInfo.db
-rw-rw-r--. 3 ec2-user ec2-user   97 Aug 19 02:38 na-1-big-Data.db
-rw-rw-r--. 3 ec2-user ec2-user   10 Aug 19 02:38 na-1-big-Digest.crc32
-rw-rw-r--. 3 ec2-user ec2-user   16 Aug 19 02:38 na-1-big-Filter.db
-rw-rw-r--. 3 ec2-user ec2-user   16 Aug 19 02:38 na-1-big-Index.db
-rw-rw-r--. 3 ec2-user ec2-user 4687 Aug 19 02:38 na-1-big-Statistics.db
-rw-rw-r--. 3 ec2-user ec2-user   56 Aug 19 02:38 na-1-big-Summary.db
-rw-rw-r--. 3 ec2-user ec2-user   92 Aug 19 02:38 na-1-big-TOC.txt
-rw-rw-r--. 1 ec2-user ec2-user  814 Aug 19 02:44 schema.cql
```

Copy the DDL from the `schema.cql` and run in cqlsh to create the `catalogkeyspace.journal` table:

```
CREATE TABLE IF NOT EXISTS catalogkeyspace.journal (
   id int PRIMARY KEY,
   name text,
   publisher text)
   WITH ID = 296a2d30-c22a-11e9-b135-0d927649052c
   AND bloom_filter_fp_chance = 0.01
   AND crc_check_chance = 1.0
   AND default_time_to_live = 0
   AND gc_grace_seconds = 864000
   AND min_index_interval = 128
   AND max_index_interval = 2048
   AND memtable_flush_period_in_ms = 0
   AND speculative_retry = '99p'
   AND additional_write_policy = '99p'
   AND comment = ''
   AND caching = { 'keys': 'ALL', 'rows_per_partition': 'NONE' }
   AND compaction = { 'min_threshold': '4', 'max_threshold':
   '32', 'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy' }
   AND compression = { 'chunk_length_in_kb': '16', 'class':
   'org.apache.cassandra.io.compress.LZ4Compressor' }
   AND cdc = false
   AND extensions = {  }
;
```

Run the `nodetool import` command to import the SSTables for the snapshot:

```
$ nodetool import -- catalogkeyspace journal \
./cassandra/data/data/catalogkeyspace/journal-
296a2d30c22a11e9b1350d927649052c/snapshots/catalog-ks/
```

Subsequently run a CQL query on the `journal` table to check the imported data:

```
SELECT * FROM journal;
```

```
id | name                      | publisher
----+---------------------------+------------------
 1 |        Couchbase Magazine |        Couchbase
 0 | Apache Cassandra Magazine | Apache Cassandra

(2 rows)
```

Bulk Loading External Data

Bulk loading external data directly is not supported by any of the tools we have discussed which include `sstableloader` and `nodetool import`. The `sstableloader` and `nodetool import` require data to be in the form of SSTables. Apache Cassandra supports a Java API for generating SSTables from input data, using the `org.apache.cassandra.io.sstable.CQLSSTableWriter` Java class. Subsequently, either `sstableloader` or `nodetool import` is used to bulk load the SSTables.

Generating SSTables with CQLSSTableWriter Java API

To generate SSTables using the `CQLSSTableWriter` class the following are required:

- An output directory to generate the SSTable in
- The schema for the SSTable
- A prepared statement for the `INSERT`
- A partitioner

The output directory must exist before starting. Create a directory (`/sstables` as an example) and set appropriate permissions.

```
$ sudo mkdir /sstables
$ sudo chmod  777 -R /sstables
```

To use `CQLSSTableWriter` in a Java application, create a Java constant for the output directory.

```
public static final String OUTPUT_DIR = "./sstables";
```

`CQLSSTableWriter` Java API can create a user-defined type. Create a new type to store `int` data:

```
String type = "CREATE TYPE CQLKeyspace.intType (a int, b int)";
```

```
// Define a String variable for the SSTable schema.
String schema = "CREATE TABLE CQLKeyspace.t ("
                 + "  id int PRIMARY KEY,"
                 + "  k int,"
                 + "  v1 text,"
                 + "  v2 intType,"
                 + ")";
```

Define a `String` variable for the prepared statement to use:

```
String insertStmt = "INSERT INTO CQLKeyspace.t (id, k, v1, v2) VALUES (?, ?, ?, ?)";
```

The partitioner to use only needs setting if the default partitioner `Murmur3Partitioner` is not used.

All these variables or settings are used by the builder class `CQLSSTableWriter.Builder` to create a `CQLSSTableWriter` object.

Create a File object for the output directory.

```
File outputDir = new File(OUTPUT_DIR + File.separator + "CQLKeyspace" + File.separator +
"t");
```

Obtain a `CQLSSTableWriter.Builder` object using `static` method `CQLSSTableWriter.builder()`. Set the following items:

- output directory `File` object
- user-defined type
- SSTable schema
- buffer size
- prepared statement
- optionally any of the other builder options

and invoke the `build()` method to create a `CQLSSTableWriter` object:

```
CQLSSTableWriter writer = CQLSSTableWriter.builder()
                                             .inDirectory(outputDir)
                                             .withType(type)
                                             .forTable(schema)
                                             .withBufferSizeInMB(256)
                                             .using(insertStmt).build();
```

Set the SSTable data. If any user-defined types are used, obtain a `UserType` object for each type:

```
UserType userType = writer.getUDType("intType");
```

Add data rows for the resulting SSTable:

```
writer.addRow(0, 0, "val0", userType.newValue().setInt("a", 0).setInt("b", 0));
   writer.addRow(1, 1, "val1", userType.newValue().setInt("a", 1).setInt("b", 1));
   writer.addRow(2, 2, "val2", userType.newValue().setInt("a", 2).setInt("b", 2));
```

Close the writer, finalizing the SSTable:

```
writer.close();
```

Other public methods the `CQLSSTableWriter` class provides are:

Method	**Description**
addRow(java.util.List<java.lang.Object> values)	Adds a new row to the writer. Returns a CQLSSTableWriter object. Each provided value type should correspond to the types of the CQL column the value is for. The correspondence between java type and CQL type is the same one than the one documented at www.datastax.com/drivers/java/2.0/apidocs/com/datastax/driver/core/DataType.Name.html#asJavaClass().

Method	Description
addRow(java.util.Map<java.lang.String,java.lang.Object> values)	Adds a new row to the writer. Returns a CQLSSTableWriter object. This is equivalent to the other addRow methods, but takes a map whose keys are the names of the columns to add instead of taking a list of the values in the order of the insert statement used during construction of this SSTable writer. The column names in the map keys must be in lowercase unless the declared column name is a case-sensitive quoted identifier in which case the map key must use the exact case of the column. The values parameter is a map of column name to column values representing the new row to add. If a column is not included in the map, it's value will be null. If the map contains keys that do not correspond to one of the columns of the insert statement used when creating this SSTable writer, the corresponding value is ignored.
addRow(java.lang.Object… values)	Adds a new row to the writer. Returns a CQLSSTableWriter object.
CQLSSTableWriter.builder()	Returns a new builder for a CQLSSTableWriter.
close()	Closes the writer.
rawAddRow(java.nio.ByteBuffer… values)	Adds a new row to the writer given already serialized binary values. Returns a CQLSSTableWriter object. The row values must correspond to the bind variables of the insertion statement used when creating by this SSTable writer.
rawAddRow(java.util.List<java.nio.ByteBuffer> values)	Adds a new row to the writer given already serialized binary values. Returns a CQLSSTableWriter object. The row values must correspond to the bind variables of the insertion statement used when creating by this SSTable writer.
rawAddRow(java.util.Map<java.lang.String, java.nio.ByteBuffer> values)	Adds a new row to the writer given already serialized binary values. Returns a CQLSSTableWriter object. The row values must correspond to the bind variables of the insertion statement used when creating by this SSTable writer.

Method	Description
getUDType(String dataType)	Returns the User Defined type used in this SSTable Writer that can be used to create UDTValue instances.

Other public methods the `CQLSSTableWriter.Builder` class provides are:

Method	Description
inDirectory(String directory)	The directory where to write the SSTables. This is a mandatory option. The directory to use should already exist and be writable.
inDirectory(File directory)	The directory where to write the SSTables. This is a mandatory option. The directory to use should already exist and be writable.
forTable(String schema)	The schema (CREATE TABLE statement) for the table for which SSTable is to be created. The provided CREATE TABLE statement must use a fully-qualified table name, one that includes the keyspace name. This is a mandatory option.
withPartitioner(IPartitioner partitioner)	The partitioner to use. By default, Murmur3Partitioner will be used. If this is not the partitioner used by the cluster for which the SSTables are created, the correct partitioner needs to be provided.
using(String insert)	The INSERT or UPDATE statement defining the order of the values to add for a given CQL row. The provided INSERT statement must use a fully-qualified table name, one that includes the keyspace name. Moreover, said statement must use bind variables since these variables will be bound to values by the resulting SSTable writer. This is a mandatory option.
withBufferSizeInMiB(int size)	The size of the buffer to use. This defines how much data will be buffered before being written as a new SSTable. This corresponds roughly to the data size that will have the created SSTable. The default is 128MB, which should be reasonable for a 1GB heap. If OutOfMemory exception gets generated while using the SSTable writer, should lower this value.

Method	Description
withBufferSizeInMB(int size)	Deprecated, and it will be available at least until next major release. Please use withBufferSizeInMiB(int size) which is the same method with a new name.
sorted()	Creates a CQLSSTableWriter that expects sorted inputs. If this option is used, the resulting SSTable writer will expect rows to be added in SSTable sorted order (and an exception will be thrown if that is not the case during row insertion). The SSTable sorted order means that rows are added such that their partition keys respect the partitioner order. This option should only be used if the rows can be provided in order, which is rarely the case. If the rows can be provided in order however, using this sorted might be more efficient. If this option is used, some option like withBufferSizeInMB will be ignored.
build()	Builds a CQLSSTableWriter object.

Change Data Capture

Change Data Capture

Overview

Change data capture (CDC) provides a mechanism to flag specific tables for archival as well as rejecting writes to those tables once a configurable size-on-disk for the CDC log is reached. An operator can enable CDC on a table by setting the table property `cdc=true` (either when **`creating the table`** or **`altering it`**). Upon CommitLogSegment creation, a hard-link to the segment is created in the directory specified in `cassandra.yaml`. On segment fsync to disk, if CDC data is present anywhere in the segment a <segment_name>_cdc.idx file is also created with the integer offset of how much data in the original segment is persisted to disk. Upon final segment flush, a second line with the human-readable word "COMPLETED" will be added to the _cdc.idx file indicating that Cassandra has completed all processing on the file.

We use an index file rather than just encouraging clients to parse the log realtime off a memory mapped handle as data can be reflected in a kernel buffer that is not yet persisted to disk. Parsing only up to the listed offset in the _cdc.idx file will ensure that you only parse CDC data for data that is durable.

Please note that in rare chances, e.g. slow disk, it is possible for the consumer to read an empty value from the _cdc.idx file because update is achieved with first truncating the file then write to the file. In such case, the consumer should retry read the index file.

A threshold of total disk space allowed is specified in the yaml at which time newly allocated CommitLogSegments will not allow CDC data until a consumer parses and removes files from the specified cdc_raw directory.

Configuration

Enabling or disabling CDC on a table

CDC is enable or disable through the cdc table property, for instance:

```
CREATE TABLE foo (a int, b text, PRIMARY KEY(a)) WITH cdc=true;

ALTER TABLE foo WITH cdc=true;

ALTER TABLE foo WITH cdc=false;
```

cassandra.yaml parameters

The following cassandra.yaml options are available for CDC:

`cdc_enabled` **(default: false)**

Enable or disable CDC operations node-wide.

`cdc_raw_directory` **(default: `$CASSANDRA_HOME/data/cdc_raw`)**

Destination for CommitLogSegments to be moved after all corresponding memtables are flushed.

`cdc_total_space`**: (default: min of 4096MiB and 1/8th volume space)**

Calculated as sum of all active CommitLogSegments that permit CDC
all flushed CDC segments in `cdc_raw_directory`.

`cdc_free_space_check_interval` **(default: 250ms)**

When at capacity, we limit the frequency with which we re-calculate the space taken up by `cdc_raw_directory` to prevent burning CPU cycles unnecessarily. Default is to check 4 times per second.

Reading CommitLogSegments

Use a CommitLogReader.java. Usage is fairly straightforward with a variety of signatures available for use. In order to handle mutations read from disk, implement CommitLogReadHandler.

Warnings

Do not enable CDC without some kind of consumption process in-place.

If CDC is enabled on a node and then on a table, the `cdc_free_space_in_mb` will fill up and then writes to CDC-enabled tables will be rejected unless some consumption process is in place.

Further Reading

- JIRA ticket
- JIRA ticket

Compression

Compression

Cassandra offers operators the ability to configure compression on a per-table basis. Compression reduces the size of data on disk by compressing the SSTable in user-configurable compression `chunk_length_in_kb`. As Cassandra SSTables are immutable, the CPU cost of compressing is only necessary when the SSTable is written - subsequent updates to data will land in different SSTables, so Cassandra will not need to decompress, overwrite, and recompress data when UPDATE commands are issued. On reads, Cassandra will locate the relevant compressed chunks on disk, decompress the full chunk, and then proceed with the remainder of the read path (merging data from disks and memtables, read repair, and so on).

Compression algorithms typically trade off between the following three areas:

- **Compression speed**: How fast does the compression algorithm compress data. This is critical in the flush and compaction paths because data must be compressed before it is written to disk.
- **Decompression speed**: How fast does the compression algorithm de-compress data. This is critical in the read and compaction paths as data must be read off disk in a full chunk and decompressed before it can be returned.
- **Ratio**: By what ratio is the uncompressed data reduced by. Cassandra typically measures this as the size of data on disk relative to the uncompressed size. For example a ratio of `0.5` means that the data on disk is 50% the size of the uncompressed data. Cassandra exposes this ratio per table as the `SSTable Compression Ratio` field of `nodetool tablestats`.

Cassandra offers five compression algorithms by default that make different tradeoffs in these areas. While benchmarking compression algorithms depends on many factors (algorithm parameters such as compression level, the compressibility of the input data, underlying processor class, etc …), the following table should help you pick a starting point based on your application's requirements with an extremely rough grading of the different choices by their performance in these areas (A is relatively good, F is relatively bad):

Compression Algorithm	Cassandra Class	Compression	Decompression	Ratio	C* Version
LZ4	`LZ4Compressor`	A+	A+	C+	`>=1.2.2`
LZ4HC	`LZ4Compressor`	C+	A+	B+	`>= 3.6`
Zstd	`ZstdCompressor`	A-	A-	A+	`>= 4.0`
Snappy	`SnappyCompressor`	A-	A	C	`>= 1.0`
Deflate (zlib)	`DeflateCompressor`	C	C	A	`>= 1.0`

Generally speaking for a performance critical (latency or throughput) application LZ4 is the right choice as it gets excellent ratio per CPU cycle spent. This is why it is the default choice in Cassandra.

For storage critical applications (disk footprint), however, Zstd may be a better choice as it can get significant additional ratio to LZ4.

Snappy is kept for backwards compatibility and LZ4 will typically be preferable.

Deflate is kept for backwards compatibility and Zstd will typically be preferable.

Configuring Compression

Compression is configured on a per-table basis as an optional argument to CREATE TABLE or ALTER TABLE. Three options are available for all compressors:

- class (default: LZ4Compressor): specifies the compression class to use. The two "fast" compressors are LZ4Compressor and SnappyCompressor and the two "good" ratio compressors are ZstdCompressor and DeflateCompressor.
- chunk_length_in_kb (default: 16KiB): specifies the number of kilobytes of data per compression chunk. The main tradeoff here is that larger chunk sizes give compression algorithms more context and improve their ratio, but require reads to deserialize and read more off disk.

The LZ4Compressor supports the following additional options:

- lz4_compressor_type (default fast): specifies if we should use the high (a.k.a LZ4HC) ratio version or the fast (a.k.a LZ4) version of LZ4. The high mode supports a configurable level, which can allow operators to tune the performance <→ ratio tradeoff via the lz4_high_compressor_level option. Note that in 4.0 and above it may be preferable to use the Zstd compressor.
- lz4_high_compressor_level (default 9): A number between 1 and 17 inclusive that represents how much CPU time to spend trying to get more compression ratio. Generally lower levels are "faster" but they get less ratio and higher levels are slower but get more compression ratio.

The ZstdCompressor supports the following options in addition:

- compression_level (default 3): A number between -131072 and 22 inclusive that represents how much CPU time to spend trying to get more compression ratio. The lower the level, the faster the speed (at the cost of ratio). Values from 20 to 22 are called "ultra levels" and should be used with caution, as they require more memory. The default of 3 is a good choice for competing with Deflate ratios and 1 is a good choice for competing with LZ4.

Users can set compression using the following syntax:

```
CREATE TABLE keyspace.table (id int PRIMARY KEY)
   WITH compression = {'class': 'LZ4Compressor'};
```

Or

```
ALTER TABLE keyspace.table
   WITH compression = {'class': 'LZ4Compressor', 'chunk_length_in_kb': 64};
```

Once enabled, compression can be disabled with `ALTER TABLE` setting `enabled` to `false`:

```
ALTER TABLE keyspace.table
   WITH compression = {'enabled':'false'};
```

Operators should be aware, however, that changing compression is not immediate. The data is compressed when the SSTable is written, and as SSTables are immutable, the compression will not be modified until the table is compacted. Upon issuing a change to the compression options via `ALTER TABLE`, the existing SSTables will not be modified until they are compacted - if an operator needs compression changes to take effect immediately, the operator can trigger an SSTable rewrite using `nodetool scrub` or `nodetool upgradesstables -a`, both of which will rebuild the SSTables on disk, re-compressing the data in the process.

Other options

- `crc_check_chance` (default: `1.0`): determines how likely Cassandra is to verify the checksum on each compression chunk during reads to protect against data corruption. Unless you have profiles indicating this is a performance problem it is highly encouraged not to turn this off as it is Cassandra's only protection against bitrot. In earlier versions of Cassandra a duplicate of this option existed in the compression configuration. The latter was deprecated in Cassandra 3.0 and removed in Cassandra 5.0.

Benefits and Uses

Compression's primary benefit is that it reduces the amount of data written to disk. Not only does the reduced size save in storage requirements, it often increases read and write throughput, as the CPU overhead of compressing data is faster than the time it would take to read or write the larger volume of uncompressed data from disk.

Compression is most useful in tables comprised of many rows, where the rows are similar in nature. Tables containing similar text columns (such as repeated JSON blobs) often compress very well. Tables containing data that has already been compressed or random data (e.g. benchmark datasets) do not typically compress well.

Operational Impact

- Compression metadata is stored off-heap and scales with data on disk. This often requires 1-3GB of off-heap RAM per terabyte of data on disk, though the exact usage varies with `chunk_length_in_kb` and compression ratios.
- Streaming operations involve compressing and decompressing data on compressed tables - in some code paths (such as non-vnode bootstrap), the CPU overhead of compression can be a limiting factor.
- To prevent slow compressors (`Zstd`, `Deflate`, `LZ4HC`) from blocking flushes for too long, all three flush with the default fast `LZ4` compressor and then rely on normal compaction to re-compress the data into the desired compression strategy. See CASSANDRA-15379 for more details.
- The compression path checksums data to ensure correctness - while the traditional Cassandra read path does not have a way to ensure correctness of data on disk, compressed tables allow the user to set `crc_check_chance` (a float from 0.0 to 1.0) to allow Cassandra to probabilistically validate chunks on read to verify bits on disk are not corrupt.

Advanced Use

Advanced users can provide their own compression class by implementing the interface at `org.apache.cassandra.io.compress.ICompressor`.

Hardware Choices

Hardware Choices

Like most databases, Cassandra throughput improves with more CPU cores, more RAM, and faster disks. While Cassandra can be made to run on small servers for testing or development environments (including Raspberry Pis), a minimal production server requires at least 2 cores, and at least 8GB of RAM. Typical production servers have 8 or more cores and at least 32GB of RAM.

CPU

Cassandra is highly concurrent, handling many simultaneous requests (both read and write) using multiple threads running on as many CPU cores as possible. The Cassandra write path tends to be heavily optimized (writing to the commitlog and then inserting the data into the memtable), so writes, in particular, tend to be CPU bound. Consequently, adding additional CPU cores often increases throughput of both reads and writes.

Memory

Cassandra runs within a Java VM, which will pre-allocate a fixed size heap (java's Xmx system parameter). In addition to the heap, Cassandra will use significant amounts of RAM offheap for compression metadata, bloom filters, row, key, and counter caches, and an in process page cache. Finally, Cassandra will take advantage of the operating system's page cache, storing recently accessed portions files in RAM for rapid re-use.

For optimal performance, operators should benchmark and tune their clusters based on their individual workload. However, basic guidelines suggest:

- ECC RAM should always be used, as Cassandra has few internal safeguards to protect against bit level corruption
- The Cassandra heap should be no less than 2GB, and no more than 50% of your system RAM
- Heaps smaller than 12GB should consider ParNew/ConcurrentMarkSweep garbage collection
- Heaps larger than 12GB should consider either:
    - 16GB heap with 8-10GB of new gen, a survivor ratio of 4-6, and a maximum tenuring threshold of 6
    - G1GC

Disks

Cassandra persists data to disk for two very different purposes. The first is to the commitlog when a new write is made so that it can be replayed after a crash or system shutdown. The second is to the data directory when thresholds are exceeded and memtables are flushed to disk as SSTables.

Commitlogs receive every write made to a Cassandra node and have the potential to block client operations, but they are only ever read on node start-up. SSTable (data file) writes on the other hand occur asynchronously, but are read to satisfy client look-ups. SSTables are also periodically merged and rewritten in a process called compaction. The data held in the commitlog directory is data that has not been permanently saved to the SSTable data directories - it will be periodically purged once it is flushed to the SSTable data files.

Cassandra performs very well on both spinning hard drives and solid state disks. In both cases, Cassandra's sorted immutable SSTables allow for linear reads, few seeks, and few overwrites, maximizing throughput for HDDs and lifespan of SSDs by avoiding write amplification. However, when using spinning disks, it's important that the commitlog (`commitlog_directory`) be on one physical disk (not simply a partition, but a physical disk), and the data files (`data_file_directories`) be set to a separate physical disk. By separating the commitlog from the data directory, writes can benefit from sequential appends to the commitlog without having to seek around the platter as reads request data from various SSTables on disk.

In most cases, Cassandra is designed to provide redundancy via multiple independent, inexpensive servers. For this reason, using NFS or a SAN for data directories is an antipattern and should typically be avoided. Similarly, servers with multiple disks are often better served by using RAID0 or JBOD than RAID1 or RAID5 - replication provided by Cassandra obsoletes the need for replication at the disk layer, so it's typically recommended that operators take advantage of the additional throughput of RAID0 rather than protecting against failures with RAID1 or RAID5.

Common Cloud Choices

Many large users of Cassandra run in various clouds, including AWS, Azure, and GCE - Cassandra will happily run in any of these environments. Users should choose similar hardware to what would be needed in physical space. In EC2, popular options include:

- i2 instances, which provide both a high RAM:CPU ratio and local ephemeral SSDs
- i3 instances with NVMe disks
    - EBS works okay if you want easy backups and replacements
- m4.2xlarge / c4.4xlarge instances, which provide modern CPUs, enhanced networking and work well with EBS GP2 (SSD) storage

Generally, disk and network performance increases with instance size and generation, so newer generations of instances and larger instance types within each family often perform better than their smaller or older alternatives.

Hints

Hints

Hinting is a data repair technique applied during write operations. When replica nodes are unavailable to accept a mutation, either due to failure or more commonly routine maintenance, coordinators attempting to write to those replicas store temporary hints on their local filesystem for later application to the unavailable replica. Hints are an important way to help reduce the duration of data inconsistency. Coordinators replay hints quickly after unavailable replica nodes return to the ring. Hints are best effort, however, and do not guarantee eventual consistency like `anti-entropy repair` does.

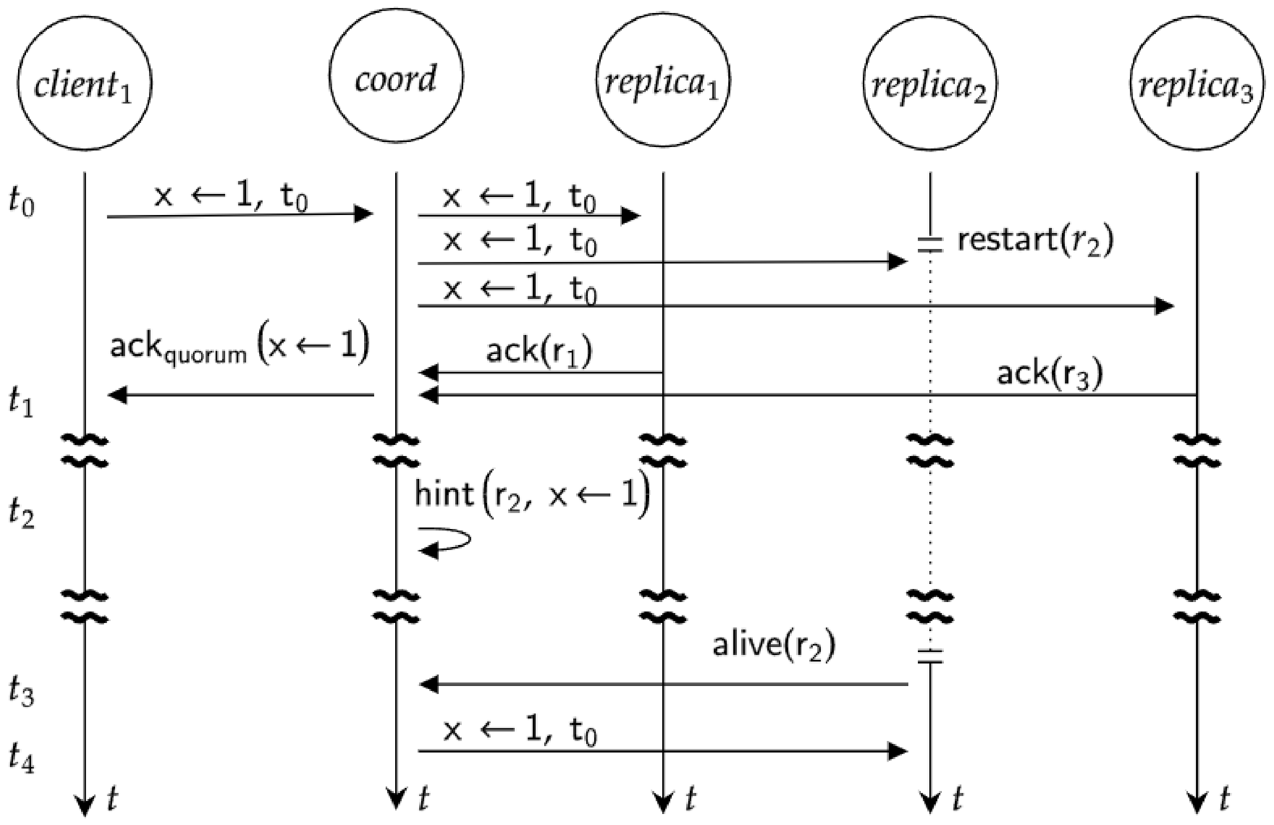
Hints are useful because of how Apache Cassandra replicates data to provide fault tolerance, high availability and durability. Cassandra `partitions data across the cluster` using consistent hashing, and then replicates keys to multiple nodes along the hash ring. To guarantee availability, all replicas of a key can accept mutations without consensus, but this means it is possible for some replicas to accept a mutation while others do not. When this happens an inconsistency is introduced.

Hints are one of the three ways, in addition to read-repair and full/incremental anti-entropy repair, that Cassandra implements the eventual consistency guarantee that all updates are eventually received by all replicas. Hints, like read-repair, are best effort and not an alternative to performing full repair, but they do help reduce the duration of inconsistency between replicas in practice.

Hinted Handoff

Hinted handoff is the process by which Cassandra applies hints to unavailable nodes.

For example, consider a mutation is to be made at `Consistency Level` `LOCAL_QUORUM` against a keyspace with `Replication Factor` of `3`. Normally the client sends the mutation to a single coordinator, who then sends the mutation to all three replicas, and when two of the three replicas acknowledge the mutation the coordinator responds successfully to the client. If a replica node is unavailable, however, the coordinator stores a hint locally to the filesystem for later application. New hints will be retained for up to `max_hint_windowin_ms` of downtime (defaults to `3 h`). If the unavailable replica does return to the cluster before the window expires, the coordinator applies any pending hinted mutations against the replica to ensure that eventual consistency is maintained.



- (`t0`): The write is sent by the client, and the coordinator sends it to the three replicas. Unfortunately `replica_2` is restarting and cannot receive the mutation.
- (`t1`): The client receives a quorum acknowledgement from the coordinator. At this point the client believe the write to be durable and visible to reads (which it is).
- (`t2`): After the write timeout (default `2s`), the coordinator decides that `replica_2` is unavailable and stores a hint to its local disk.
- (`t3`): Later, when `replica_2` starts back up it sends a gossip message to all nodes, including the coordinator.
- (`t4`): The coordinator replays hints including the missed mutation against `replica_2`.

If the node does not return in time, the destination replica will be permanently out of sync until either read-repair or full/incremental anti-entropy repair propagates the mutation.

Application of Hints

Hints are streamed in bulk, a segment at a time, to the target replica node and the target node replays them locally. After the target node has replayed a segment it deletes the segment and receives the next segment. This continues until all hints are drained.

Storage of Hints on Disk

Hints are stored in flat files in the coordinator node's `$CASSANDRA_HOME/data/hints` directory. A hint includes a hint id, the target replica node on which the mutation is meant to be stored, the serialized mutation (stored as a blob) that couldn't be delivered to the replica node, the mutation timestamp, and the Cassandra version used to serialize the mutation. By default hints are compressed using `LZ4Compressor`. Multiple hints are appended to the same hints file.

Since hints contain the original unmodified mutation timestamp, hint application is idempotent and cannot overwrite a future mutation.

Hints for Timed Out Write Requests

Hints are also stored for write requests that time out. The `write_request_timeout` setting in `cassandra.yaml` configures the timeout for write requests.

```
write_request_timeout: 2000ms
```

The coordinator waits for the configured amount of time for write requests to complete, at which point it will time out and generate a hint for the timed out request. The lowest acceptable value for `write_request_timeout` is 10 ms.

Configuring Hints

Hints are enabled by default as they are critical for data consistency. The `cassandra.yaml` configuration file provides several settings for configuring hints:

Table 1. Settings for Hints

Setting	Description	Default Value
`hinted_handoff_enabled`	Enables/Disables hinted handoffs	`true`
`hinted_handoff_disabled_datacenters`	A list of data centers that do not perform hinted handoffs even when handoff is otherwise enabled. Example: `hinted_handoff_disabled_datacenters:` `- DC1` `- DC2`	`unset`

Setting	Description	Default Value
`max_hint_window`	Defines the maximum amount of time a node shall have hints generated after it has failed.	`3h`
`hinted_handoff_throttle`	Maximum throttle in KiBs per second, per delivery thread. This will be reduced proportionally to the number of nodes in the cluster. (If there are two nodes in the cluster, each delivery thread will use the maximum rate; if there are 3, each will throttle to half of the maximum,since it is expected for two nodes to be delivering hints simultaneously.)	`1024KiB`
`max_hints_delivery_threads`	Number of threads with which to deliver hints; Consider increasing this number when you have multi-dc deployments, since cross-dc handoff tends to be slower	`2`
`hints_directory`	Directory where Cassandra stores hints.	`$CASSANDRA_HOME/data/hints`
`hints_flush_period`	How often hints should be flushed from the internal buffers to disk. Will *not* trigger fsync.	`10000ms`
`max_hints_file_size	Maximum size for a single hints file, in megabytes.	`128MiB`
`hints_compression`	Compression to apply to the hint files. If omitted, hints files will be written uncompressed. LZ4, Snappy, and Deflate compressors are supported.	`LZ4Compressor`

Configuring Hints at Runtime with `nodetool`

`nodetool` provides several commands for configuring hints or getting hints related information. The nodetool commands override the corresponding settings if any in `cassandra.yaml` for the node running the command.

Table 2. Nodetool Commands for Hints

Command	Description
`nodetool disablehandoff`	Disables storing and delivering hints
`nodetool disablehintsfordc`	Disables storing and delivering hints to a data center
`nodetool enablehandoff`	Re-enables future hints storing and delivery on the current node
`nodetool enablehintsfordc`	Enables hints for a data center that was previously disabled
`nodetool getmaxhintwindow`	Prints the max hint window in ms. New in Cassandra 4.0.
`nodetool handoffwindow`	Prints current hinted handoff window
`nodetool pausehandoff`	Pauses hints delivery process
`nodetool resumehandoff`	Resumes hints delivery process
`nodetool sethintedhandoffthrottlekb`	Sets hinted handoff throttle in kb per second, per delivery thread
`nodetool setmaxhintwindow`	Sets the specified max hint window in ms
`nodetool statushandoff`	Status of storing future hints on the current node
`nodetool truncatehints`	Truncates all hints on the local node, or truncates hints for the endpoint(s) specified.

Make Hints Play Faster at Runtime

The default of `1024 kbps` handoff throttle is conservative for most modern networks, and it is entirely possible that in a simple node restart you may accumulate many gigabytes hints that may take hours to play back. For example if you are ingesting `100 Mbps` of data per node, a single 10 minute long restart will create `10 minutes * (100 megabit / second) ~= 7 GiB` of data which at `(1024 KiB / second)` would take `7.5 GiB / (1024 KiB / second) = 2.03 hours` to play back. The exact math depends on the load balancing strategy (round robin is better than token aware), number of tokens per node (more tokens is better than fewer), and naturally the cluster's write rate, but regardless you may find yourself wanting to increase this throttle at runtime.

If you find yourself in such a situation, you may consider raising the `hinted_handoff_throttle` dynamically via the `nodetool sethintedhandoffthrottlekb` command.

Allow a Node to be Down Longer at Runtime

Sometimes a node may be down for more than the normal `max_hint_window`, (default of three hours), but the hardware and data itself will still be accessible. In such a case you may consider raising the `max_hint_window` dynamically via the `nodetool setmaxhintwindow` command added in Cassandra 4.0 (CASSANDRA-11720). This will instruct Cassandra to continue holding hints for the down endpoint for a longer amount of time.

This command should be applied on all nodes in the cluster that may be holding hints. If needed, the setting can be applied permanently by setting the `max_hint_window` setting in `cassandra.yaml` followed by a rolling restart.

Monitoring Hint Delivery

Cassandra 4.0 adds histograms available to understand how long it takes to deliver hints which is useful for operators to better identify problems (CASSANDRA-13234).

There are also metrics available for tracking `Hinted Handoff <handoff-metrics>` and `Hints Service <hintsservice-metrics>` metrics.

Repair

Repair

Cassandra is designed to remain available if one of it’s nodes is down or unreachable. However, when a node is down or unreachable, it needs to eventually discover the writes it missed. Hints attempt to inform a node of missed writes, but are a best effort, and aren’t guaranteed to inform a node of 100% of the writes it missed. These inconsistencies can eventually result in data loss as nodes are replaced or tombstones expire.

These inconsistencies are fixed with the repair process. Repair synchronizes the data between nodes by comparing their respective datasets for their common token ranges, and streaming the differences for any out of sync sections between the nodes. It compares the data with merkle trees, which are a hierarchy of hashes.

Incremental and Full Repairs

There are 2 types of repairs: full repairs, and incremental repairs. Full repairs operate over all of the data in the token range being repaired. Incremental repairs only repair data that’s been written since the previous incremental repair.

Incremental repairs are the default repair type, and if run regularly, can significantly reduce the time and io cost of performing a repair. However, it’s important to understand that once an incremental repair marks data as repaired, it won’t try to repair it again. This is fine for syncing up missed writes, but it doesn’t protect against things like disk corruption, data loss by operator error, or bugs in Cassandra. For this reason, full repairs should still be run occasionally.

Usage and Best Practices

Since repair can result in a lot of disk and network io, it’s not run automatically by Cassandra. It is run by the operator via nodetool.

Incremental repair is the default and is run with the following command:

```
nodetool repair
```

A full repair can be run with the following command:

```
nodetool repair --full
```

Additionally, repair can be run on a single keyspace:

```
nodetool repair [options] <keyspace_name>
```

Or even on specific tables:

```
nodetool repair [options] <keyspace_name> <table1> <table2>
```

The repair command repairs token ranges only on the node being repaired; it does not repair the whole cluster. By default, repair operates on all token ranges replicated by the node on which repair is run, causing duplicate work when running it on every node. Avoid duplicate work by using the `-pr` flag to repair only the "primary" ranges on a node. Do a full cluster repair by running the `nodetool repair -pr` command on each node in each datacenter in the cluster, until all of the nodes and datacenters are repaired.

The specific frequency of repair that’s right for your cluster, of course, depends on several factors. However, if you’re just starting out and looking for somewhere to start, running an incremental repair every 1-3 days, and a full repair every 1-3 weeks is probably reasonable. If you don’t want to run incremental repairs, a full repair every 5 days is a good place to start.

At a minimum, repair should be run often enough that the gc grace period never expires on unrepaired data. Otherwise, deleted data could reappear. With a default gc grace period of 10 days, repairing every node in your cluster at least once every 7 days will prevent this, while providing enough slack to allow for delays.

Other Options

`-pr, --partitioner-range`

Restricts repair to the 'primary' token ranges of the node being repaired. A primary range is just a token range for which a node is the first replica in the ring.

`-prv, --preview`

Estimates the amount of streaming that would occur for the given repair command. This builds the merkle trees, and prints the expected streaming activity, but does not actually do any streaming. By default, incremental repairs are estimated, add the `--full` flag to estimate a full repair.

`-vd, --validate`

Verifies that the repaired data is the same across all nodes. Similiar to `--preview`, this builds and compares merkle trees of repaired data, but doesn’t do any streaming. This is useful for troubleshooting. If this shows that the repaired data is out of sync, a full repair should be run.

`nodetool repair docs <nodetool_repair>`

Full Repair Example

Full repair is typically needed to redistribute data after increasing the replication factor of a keyspace or after adding a node to the cluster. Full repair involves streaming SSTables. To demonstrate full repair start with a three node cluster.

```
[ec2-user@ip-10-0-2-238 ~]$ nodetool status
Datacenter: us-east-1
=====================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load        Tokens  Owns  Host ID                               Rack
UN  10.0.1.115  547 KiB     256    ?  b64cb32a-b32a-46b4-9eeb-e123fa8fc287  us-east-1b
UN  10.0.3.206  617.91 KiB  256    ?  74863177-684b-45f4-99f7-d1006625dc9e  us-east-1d
UN  10.0.2.238  670.26 KiB  256    ?  4dcdadd2-41f9-4f34-9892-1f20868b27c7  us-east-1c
```

Create a keyspace with replication factor 3:

```
cqlsh> DROP KEYSPACE cqlkeyspace;
cqlsh> CREATE KEYSPACE CQLKeyspace
  ... WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};
```

Add a table to the keyspace:

```
cqlsh> use cqlkeyspace;
cqlsh:cqlkeyspace> CREATE TABLE t (
           ...   id int,
           ...   k int,
           ...   v text,
           ...   PRIMARY KEY (id)
           ... );
```

Add table data:

```
cqlsh:cqlkeyspace> INSERT INTO t (id, k, v) VALUES (0, 0, 'val0');
cqlsh:cqlkeyspace> INSERT INTO t (id, k, v) VALUES (1, 1, 'val1');
cqlsh:cqlkeyspace> INSERT INTO t (id, k, v) VALUES (2, 2, 'val2');
```

A query lists the data added:

```
cqlsh:cqlkeyspace> SELECT * FROM t;
```

```
id | k | v
----+---+------
 1 | 1 | val1
 0 | 0 | val0
 2 | 2 | val2
(3 rows)
```

Make the following changes to a three node cluster:

1. Increase the replication factor from 3 to 4.
2. Add a 4th node to the cluster

When the replication factor is increased the following message gets output indicating that a full repair is needed as per (CASSANDRA-13079):

```
cqlsh:cqlkeyspace> ALTER KEYSPACE CQLKeyspace
           ... WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
Warnings :
When increasing replication factor you need to run a full (-full) repair to distribute
the
data.
```

Perform a full repair on the keyspace `cqlkeyspace` table `t` with following command:

```
nodetool repair -full cqlkeyspace t
```

Full repair completes in about a second as indicated by the output:

```
[ec2-user@ip-10-0-2-238 ~]$ nodetool repair -full cqlkeyspace t
[2019-08-17 03:06:21,445] Starting repair command #1 (fd576da0-c09b-11e9-b00c-
1520e8c38f00), repairing keyspace cqlkeyspace with repair options (parallelism: parallel,
primary range: false, incremental: false, job threads: 1, ColumnFamilies: [t],
dataCenters: [], hosts: [], previewKind: NONE, # of ranges: 1024, pull repair: false,
force repair: false, optimise streams: false)
[2019-08-17 03:06:23,059] Repair session fd8e5c20-c09b-11e9-b00c-1520e8c38f00 for range
[(-8792657144775336505,-8786320730900698730], (-5454146041421260303,-
5439402053041523135], (4288357893651763201,4324309707046452322], ... ,
(4350676211955643098,4351706629422088296]] finished (progress: 0%)
[2019-08-17 03:06:23,077] Repair completed successfully
[2019-08-17 03:06:23,077] Repair command #1 finished in 1 second
[ec2-user@ip-10-0-2-238 ~]$
```

The `nodetool tpstats` command should list a repair having been completed as `Repair-Task` > `Completed` column value of 1:

```
[ec2-user@ip-10-0-2-238 ~]$ nodetool tpstats
Pool Name Active   Pending Completed   Blocked  All time blocked
ReadStage  0           0           99       0              0
...
Repair-Task 0       0           1        0              0
RequestResponseStage                  0        0        2078        0               0
```

Read repair

Read repair

Read Repair is the process of repairing data replicas during a read request. If all replicas involved in a read request at the given read consistency level are consistent the data is returned to the client and no read repair is needed. But if the replicas involved in a read request at the given consistency level are not consistent a read repair is performed to make replicas involved in the read request consistent. The most up-to-date data is returned to the client. The read repair runs in the foreground and is blocking in that a response is not returned to the client until the read repair has completed and up-to-date data is constructed.

Expectation of Monotonic Quorum Reads

Cassandra uses a blocking read repair to ensure the expectation of "monotonic quorum reads" i.e. that in 2 successive quorum reads, it's guaranteed the 2nd one won't get something older than the 1st one, and this even if a failed quorum write made a write of the most up to date value only to a minority of replicas. "Quorum" means majority of nodes among replicas.

Table level configuration of monotonic reads

Cassandra 4.0 adds support for table level configuration of monotonic reads (CASSANDRA-14635). The `read_repair` table option has been added to table schema, with the options `blocking` (default), and `none`.

The `read_repair` option configures the read repair behavior to allow tuning for various performance and consistency behaviors. Two consistency properties are affected by read repair behavior.

- Monotonic Quorum Reads: Provided by `BLOCKING`. Monotonic quorum reads prevents reads from appearing to go back in time in some circumstances. When monotonic quorum reads are not provided and a write fails to reach a quorum of replicas, it may be visible in one read, and then disappear in a subsequent read.
- Write Atomicity: Provided by `NONE`. Write atomicity prevents reads from returning partially applied writes. Cassandra attempts to provide partition level write atomicity, but since only the data covered by a `SELECT` statement is repaired by a read repair, read repair can break write atomicity when data is read at a more granular level than it is written. For example read repair can break write atomicity if you write multiple rows to a clustered partition in a batch, but then select a single row by specifying the clustering column in a `SELECT` statement.

The available read repair settings are:

Blocking

The default setting. When `read_repair` is set to `BLOCKING`, and a read repair is started, the read will block on writes sent to other replicas until the CL is reached by the writes. Provides monotonic quorum reads, but not partition level write atomicity.

None

When `read_repair` is set to `NONE`, the coordinator will reconcile any differences between replicas, but will not attempt to repair them. Provides partition level write atomicity, but not monotonic quorum reads.

An example of using the `NONE` setting for the `read_repair` option is as follows:

```
CREATE TABLE ks.tbl (k INT, c INT, v INT, PRIMARY KEY (k,c)) with read_repair='NONE'");
```

Read Repair Example

To illustrate read repair with an example, consider that a client sends a read request with read consistency level `TWO` to a 5-node cluster as illustrated in Figure 1. Read consistency level determines how many replica nodes must return a response before the read request is considered successful.

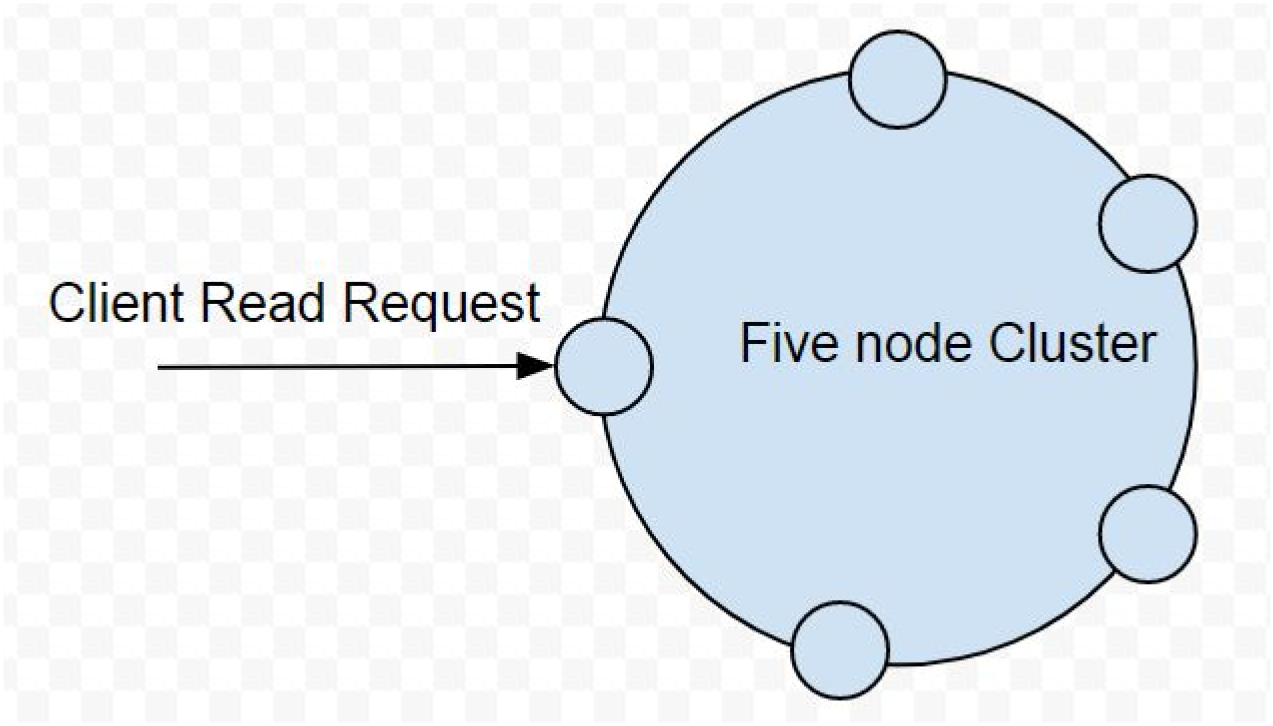


Figure 1. Client sends read request to a 5-node Cluster

Three nodes host replicas for the requested data as illustrated in Figure 2. With a read consistency level of `TWO` two replica nodes must return a response for the read request to be considered successful. If the node the client sends request to hosts a replica of the data requested only one other replica node needs to be sent a read request to. But if the receiving node does not host a replica for the requested data the node becomes a coordinator node and forwards the read request to a node that hosts a replica. A direct read request is forwarded to the fastest node (as determined by dynamic snitch) as shown in Figure 2. A direct read request is a full read and returns the requested data.

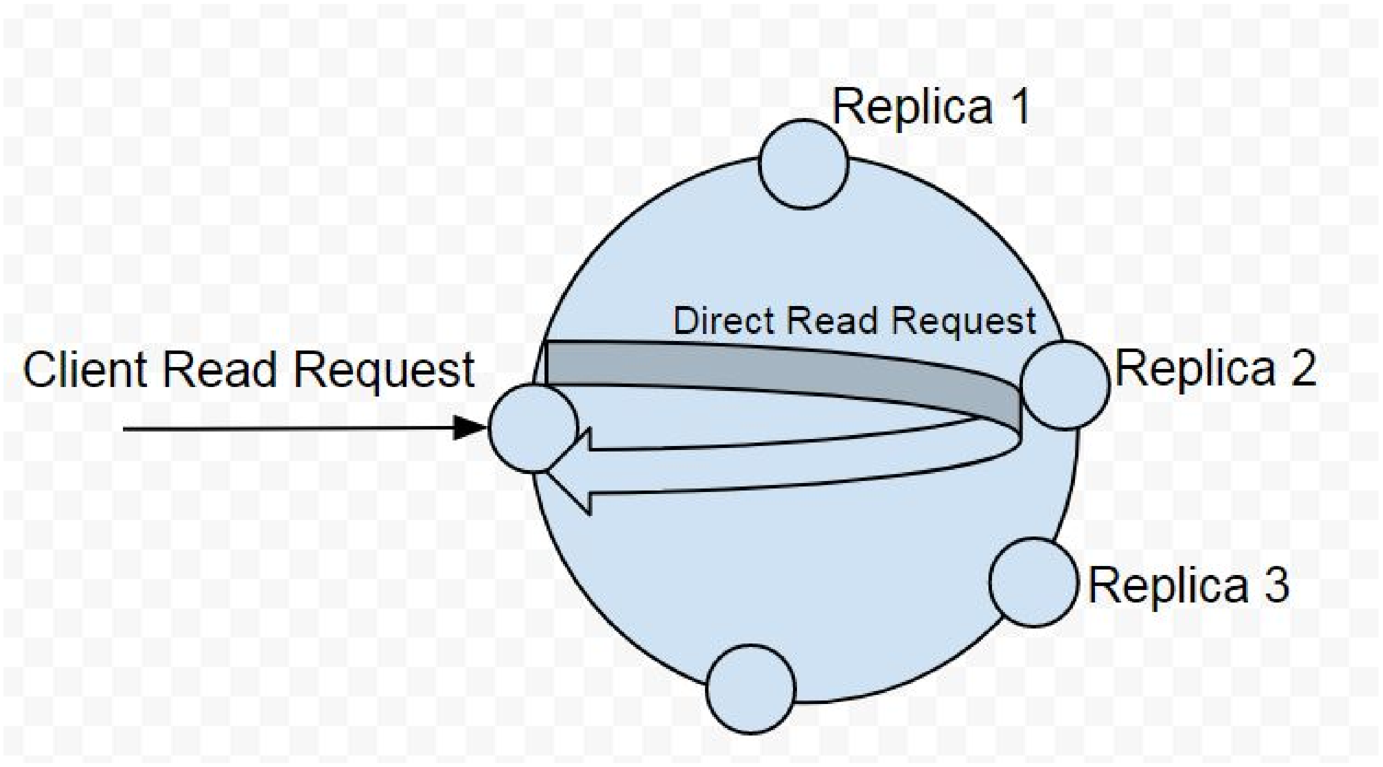


Figure 2. Direct Read Request sent to Fastest Replica Node

Next, the coordinator node sends the requisite number of additional requests to satisfy the consistency level, which is `TWO`. The coordinator node needs to send one more read request for a total of two. All read requests additional to the first direct read request are digest read requests. A digest read request is not a full read and only returns the hash value of the data. Only a hash value is returned to reduce the network data traffic. In the example being discussed the coordinator node sends one digest read request to a node hosting a replica as illustrated in Figure 3.

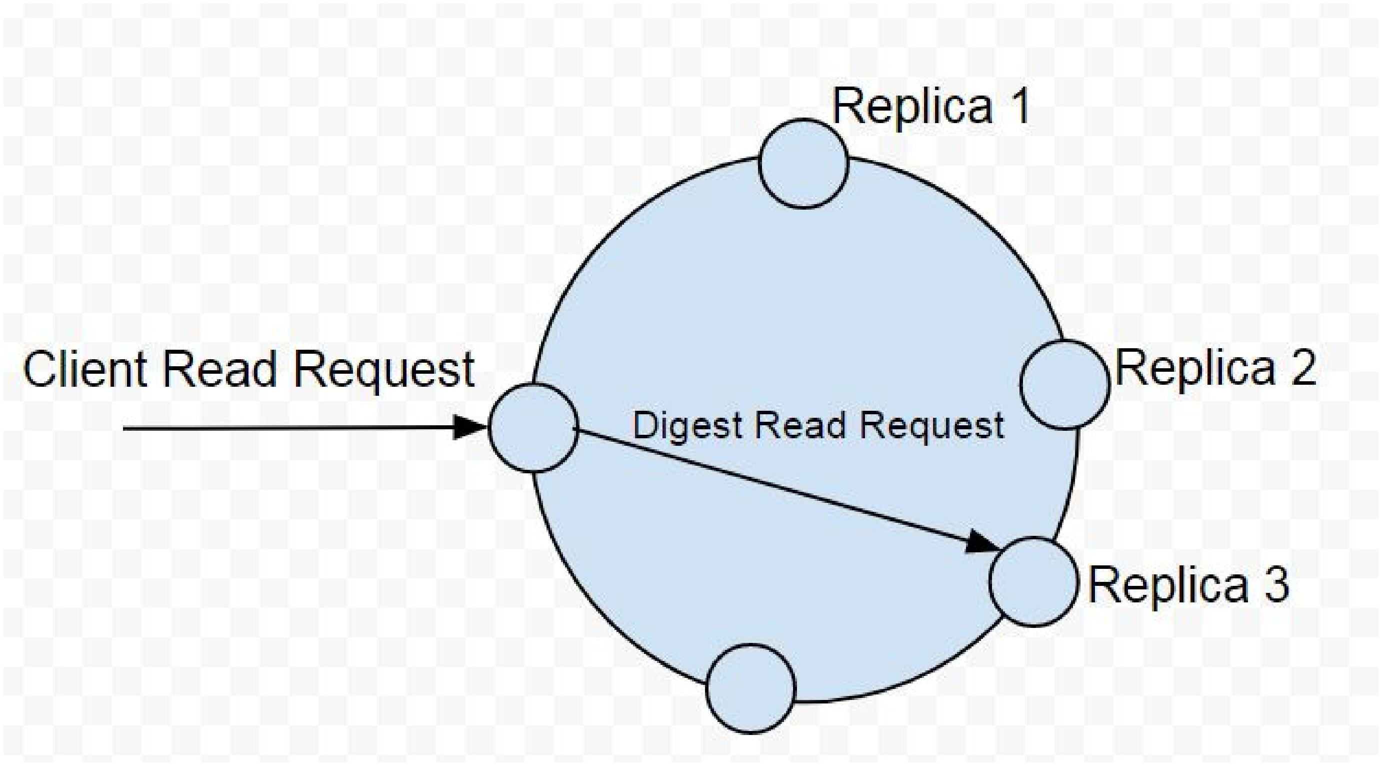


Figure 3. Coordinator Sends a Digest Read Request

The coordinator node has received a full copy of data from one node and a hash value for the data from another node. To compare the data returned a hash value is calculated for the full copy of data. The two hash values are compared. If the hash values are the same no read repair is needed and the full copy of requested data is returned to the client. The coordinator node only performed a total of two replica read request because the read consistency level is `TWO` in the example. If the consistency level were higher such as `THREE`, three replica nodes would need to respond to a read request and only if all digest or hash values were to match with the hash value of the full copy of data would the read request be considered successful and the data returned to the client.

But, if the hash value/s from the digest read request/s are not the same as the hash value of the data from the full read request of the first replica node it implies that an inconsistency in the replicas exists. To fix the inconsistency a read repair is performed.

For example, consider that that digest request returns a hash value that is not the same as the hash value of the data from the direct full read request. We would need to make the replicas consistent for which the coordinator node sends a direct (full) read request to the replica node that it sent a digest read request to earlier as illustrated in Figure 4.

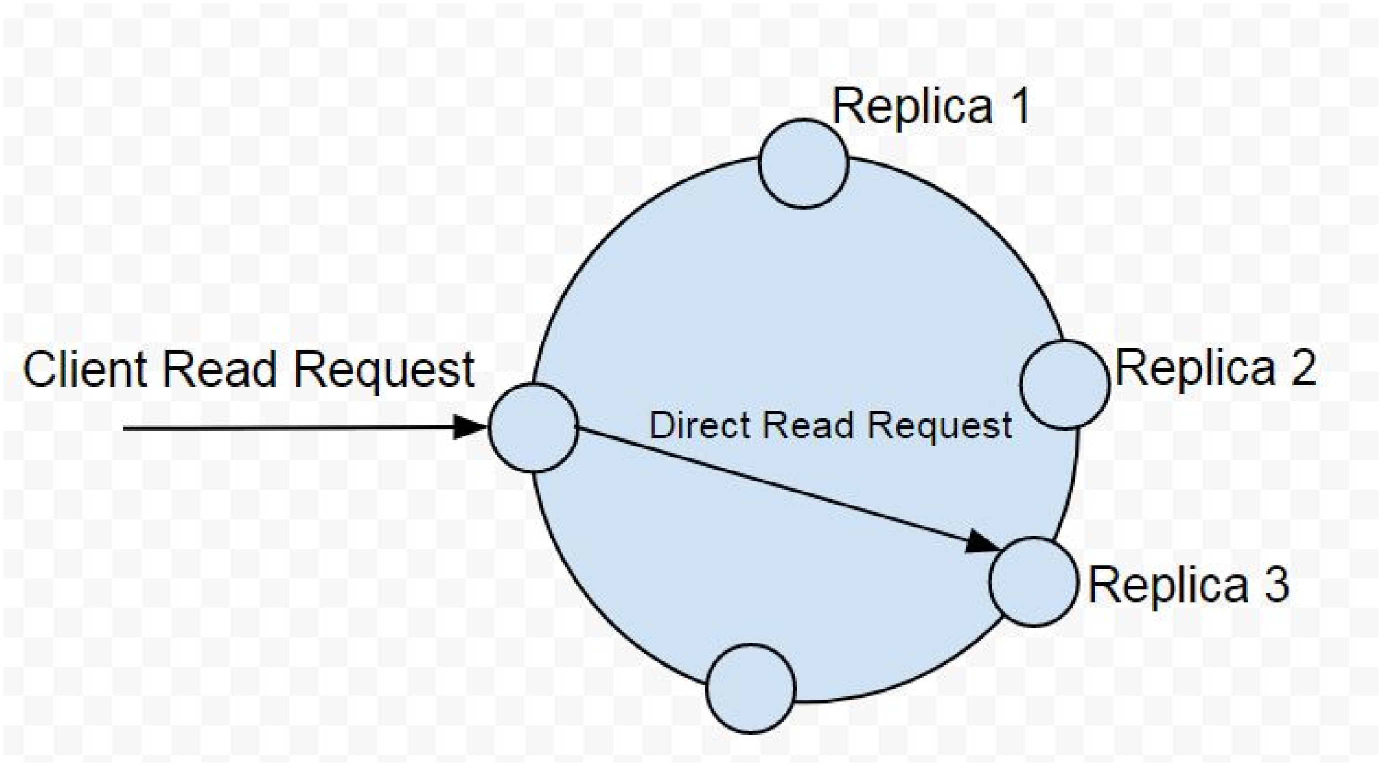


Figure 4. Coordinator sends Direct Read Request to Replica Node it had sent Digest Read Request to

After receiving the data from the second replica node the coordinator has data from two of the replica nodes. It only needs two replicas as the read consistency level is `TWO` in the example. Data from the two replicas is compared and based on the timestamps the most recent replica is selected. Data may need to be merged to construct an up-to-date copy of data if one replica has data for only some of the columns. In the example, if the data from the first direct read request is found to be outdated and the data from the second full read request to be the latest read, repair needs to be performed on Replica 2. If a new up-to-date data is constructed by merging the two replicas a read repair would be needed on both the replicas involved. For example, a read repair is performed on Replica 2 as illustrated in Figure 5.

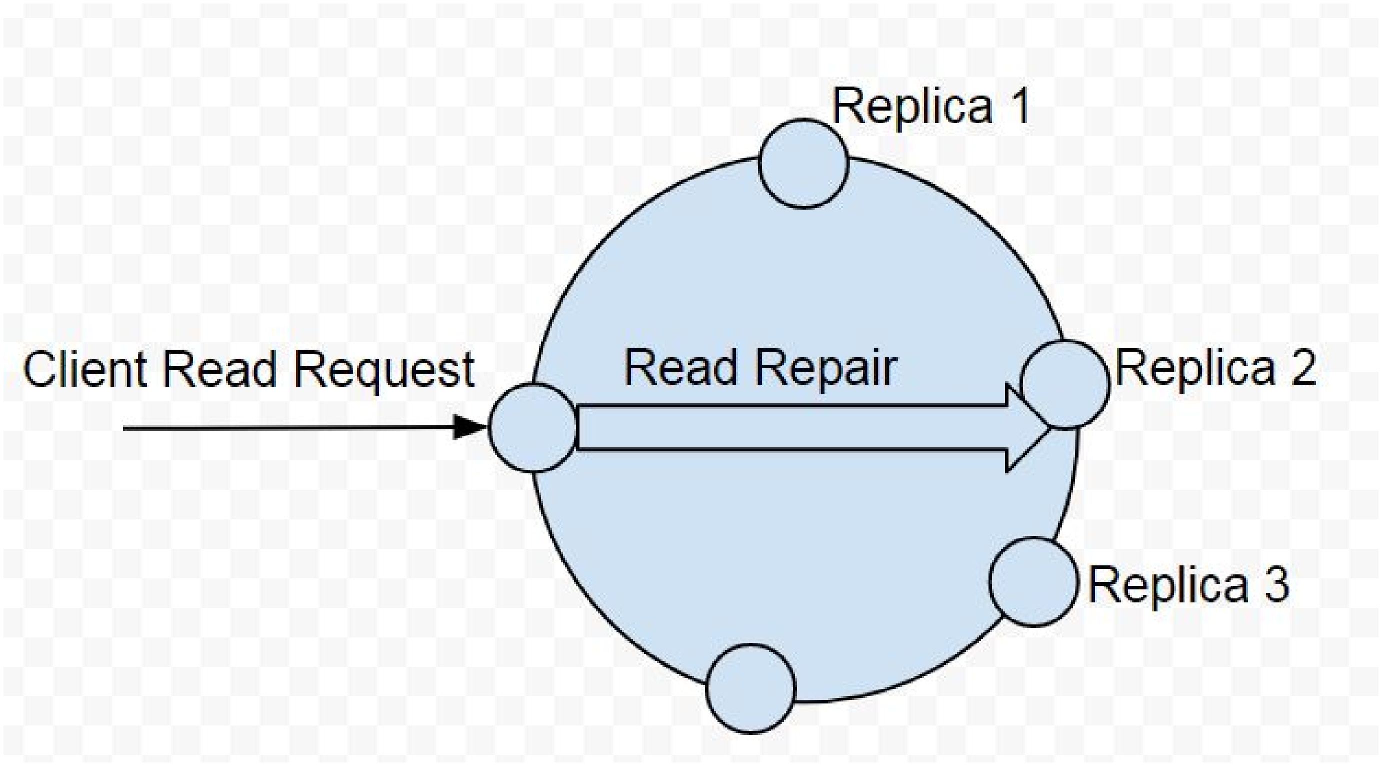


Figure 5. Coordinator performs Read Repair

The most up-to-date data is returned to the client as illustrated in Figure 6. From the three replicas Replica 1 is not even read and thus not repaired. Replica 2 is repaired. Replica 3 is the most up-to-date and returned to client.

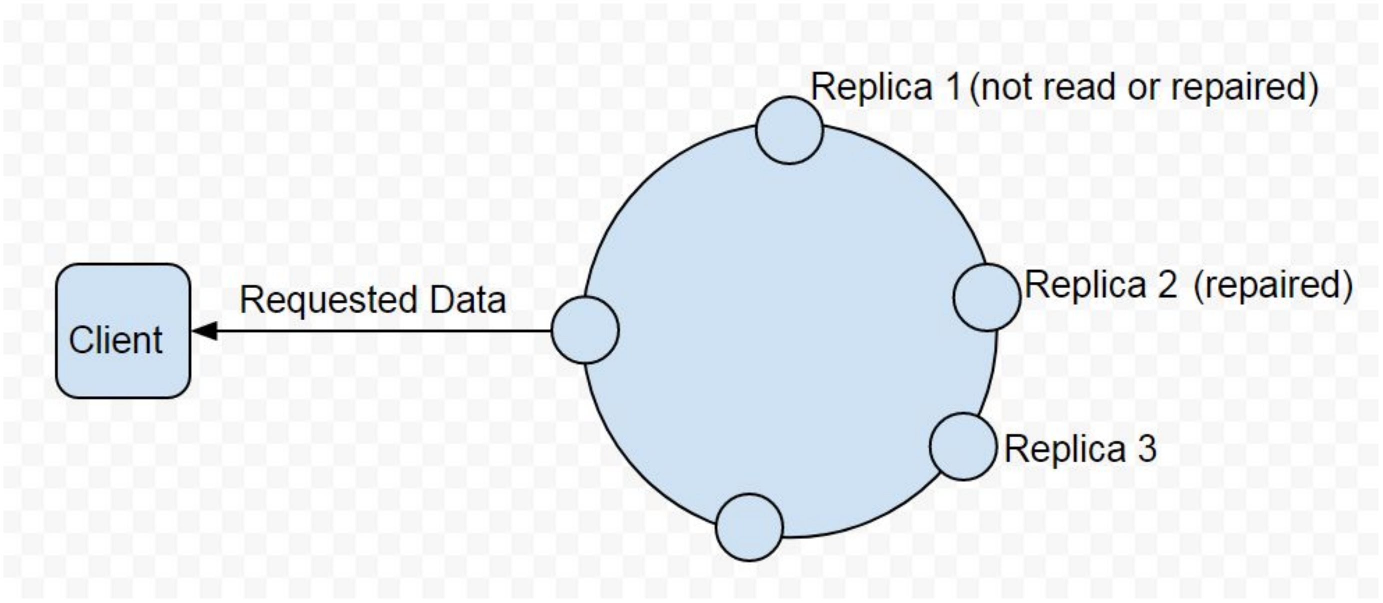


Figure 6. Most up-to-date Data returned to Client

Read Consistency Level and Read Repair

The read consistency is most significant in determining if a read repair needs to be performed. As discussed in Table 1 a read repair is not needed for all of the consistency levels.

Table 1. Read Repair based on Read Consistency Level

Read Consistency Level	**Description**
ONE	Read repair is not performed as the data from the first direct read request satisfies the consistency level ONE. No digest read requests are involved for finding mismatches in data.
TWO	Read repair is performed if inconsistencies in data are found as determined by the direct and digest read requests.
THREE	Read repair is performed if inconsistencies in data are found as determined by the direct and digest read requests.
LOCAL_ONE	Read repair is not performed as the data from the direct read request from the closest replica satisfies the consistency level LOCAL_ONE.No digest read requests are involved for finding mismatches in data.
LOCAL_QUORUM	Read repair is performed if inconsistencies in data are found as determined by the direct and digest read requests.
QUORUM	Read repair is performed if inconsistencies in data are found as determined by the direct and digest read requests.

If read repair is performed it is made only on the replicas that are not up-to-date and that are involved in the read request. The number of replicas involved in a read request would be based on the read consistency level; in the example it is two.

Improved Read Repair Blocking Behavior in Cassandra 4.0

Cassandra 4.0 makes two improvements to read repair blocking behavior (CASSANDRA-10726).

1. Speculative Retry of Full Data Read Requests. Cassandra 4.0 makes use of speculative retry in sending read requests (full, not digest) to replicas if a full data response is not received, whether in the initial full read request or a full data read request during read repair. With speculative retry if it looks like a response may not be received from the initial set of replicas Cassandra sent messages to, to satisfy the consistency level, it speculatively sends additional read request to un-contacted replica/s. Cassandra 4.0 will also speculatively send a repair mutation to a minority of nodes not involved in the read repair data read / write cycle with the combined contents of all un-acknowledged mutations if it looks like one may not respond. Cassandra accepts acks from them in

lieu of acks from the initial mutations sent out, so long as it receives the same number of acks as repair mutations transmitted.

2. Only blocks on Full Data Responses to satisfy the Consistency Level. Cassandra 4.0 only blocks for what is needed for resolving the digest mismatch and wait for enough full data responses to meet the consistency level, no matter whether it’s speculative retry or read repair chance. As an example, if it looks like Cassandra might not receive full data requests from everyone in time, it sends additional requests to additional replicas not contacted in the initial full data read. If the collection of nodes that end up responding in time end up agreeing on the data, the response from the disagreeing replica that started the read repair is not considered, and won’t be included in the response to the client, preserving the expectation of monotonic quorum reads.

Diagnostic Events for Read Repairs

Cassandra 4.0 adds diagnostic events for read repair (CASSANDRA-14668) that can be used for exposing information such as:

- Contacted endpoints
- Digest responses by endpoint
- Affected partition keys
- Speculated reads / writes
- Update oversized

Background Read Repair

Background read repair, which was configured using `read_repair_chance` and `dclocal_read_repair_chance` settings in `cassandra.yaml` is removed Cassandra 4.0 (CASSANDRA-13910).

Read repair is not an alternative for other kind of repairs such as full repairs or replacing a node that keeps failing. The data returned even after a read repair has been performed may not be the most up-to-date data if consistency level is other than one requiring response from all replicas.

Security

Security

There are three main components to the security features provided by Cassandra:

- TLS/SSL encryption for client and inter-node communication
- Client authentication
- Authorization

By default, these features are disabled as Cassandra is configured to easily find and be found by other members of a cluster. In other words, an out-of-the-box Cassandra installation presents a large attack surface for a bad actor. Enabling authentication for clients using the binary protocol is not sufficient to protect a cluster. Malicious users able to access internode communication and JMX ports can still:

- Craft internode messages to insert users into authentication schema
- Craft internode messages to truncate or drop schema
- Use tools such as `sstableloader` to overwrite `system_auth` tables
- Attach to the cluster directly to capture write traffic

Correct configuration of all three security components should negate theses vectors. Therefore, understanding Cassandra's security features is crucial to configuring your cluster to meet your security needs.

TLS/SSL Encryption

Cassandra provides secure communication between a client machine and a database cluster and between nodes within a cluster. Enabling encryption ensures that data in flight is not compromised and is transferred securely. The options for client-to-node and node-to-node encryption are managed separately and may be configured independently.

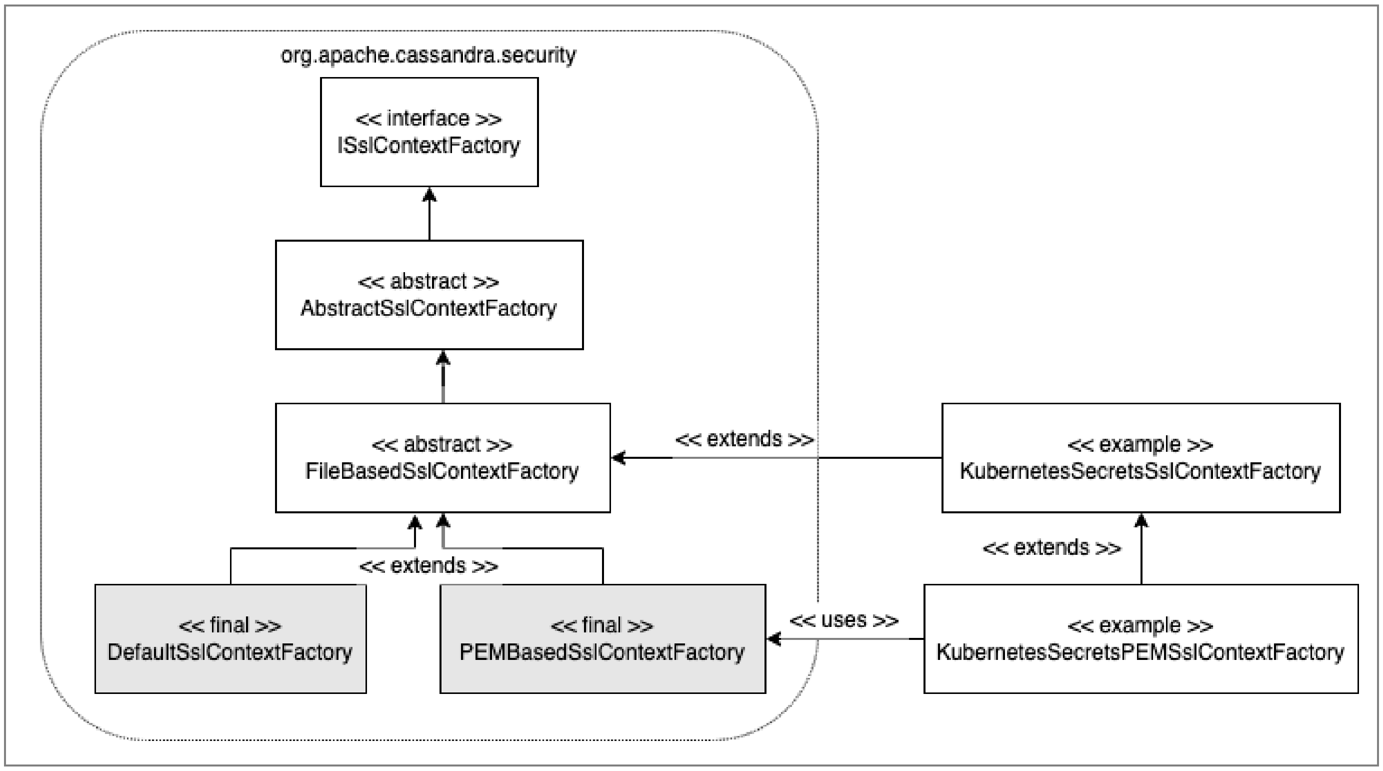
In both cases, the JVM defaults for supported protocols and cipher suites are used when encryption is enabled. These can be overidden using the settings in `cassandra.yaml`, but this is not recommended unless there are policies in place which dictate certain settings or a need to disable vulnerable ciphers or protocols in cases where the JVM cannot be updated.

FIPS compliant settings can be configured at the JVM level and should not involve changing encryption settings in cassandra.yaml. See the java document on FIPS for more details.

Cassandra provides flexibility of using Java based key material or completely customizing the SSL context. You can choose any keystore format supported by Java (JKS, PKCS12 etc) as well as other standards like PEM. You can even customize the SSL context creation to use Cloud Native technologies like Kuberenetes Secrets for storing the key material or to integrate with your in-house Key Management System.

For information on generating the keystore and truststore files required with the Java supported keystores used in SSL communications, see the java documentation on creating keystores.

For customizing the SSL context creation you can implement ISslContextCreationFactory interface or extend one of its public subclasses appropriately. You can then use the `ssl_context_factory` setting for `server_encryption_options` or `client_encryption_options` sections appropriately. See ssl-factory examples for details. Refer to the below class diagram to understand the class hierarchy.


org.apache.cassandra.security
<< interface >> ISslContextFactory
<< abstract >> AbstractSslContextFactory
<< abstract >> FileBasedSslContextFactory
<< extends >>
<< example >> KubernetesSecretsSslContextFactory
<< extends >>
<< extends >>
<< final >> DefaultSslContextFactory
<< final >> PEMBasedSslContextFactory
<< uses >>
<< example >> KubernetesSecretsPEMSslContextFactory

Using PEM based key material

You can use the in-built class `PEMBasedSSLContextFactory` as the `ssl_context_factory` setting for the PEM based key material.

You can configure this factory with either inline PEM data or with the files having the required PEM data as shown below,

- Configuration: PEM keys/certs defined in-line (mind the spaces in the YAML!)

```
client/server_encryption_options:
  ssl_context_factory:
     class_name: org.apache.cassandra.security.PEMBasedSslContextFactory
     parameters:
         private_key: |
          -----BEGIN ENCRYPTED PRIVATE KEY----- OR -----BEGIN PRIVATE KEY-----
          <your base64 encoded private key>
          -----END ENCRYPTED PRIVATE KEY----- OR -----END PRIVATE KEY-----
```

```
              -----BEGIN CERTIFICATE-----
              <your base64 encoded certificate chain>
              -----END CERTIFICATE-----

             private_key_password: "<your password if the private key is encrypted with a
password>"

             trusted_certificates: |
               -----BEGIN CERTIFICATE-----
               <your base64 encoded certificate>
               -----END CERTIFICATE-----
```

- Configuration: PEM keys/certs defined in files

```
    client/server_encryption_options:
     ssl_context_factory:
        class_name: org.apache.cassandra.security.PEMBasedSslContextFactory
     keystore: <file path to the keystore file in the PEM format with the private key and
the certificate chain>
     keystore_password: "<your password if the private key is encrypted with a password>"
     truststore: <file path to the truststore file in the PEM format>
```

SSL Certificate Hot Reloading

Beginning with Cassandra 4, Cassandra supports hot reloading of SSL Certificates. If SSL/TLS support is enabled in Cassandra and you are using default file based key material, the node periodically (every 10 minutes) polls the Trust and Key Stores specified in cassandra.yaml. When the files are updated, Cassandra will reload them and use them for subsequent connections. Please note that the Trust & Key Store passwords are part of the yaml so the updated files should also use the same passwords.

If you are customizing the SSL configuration via `ssl_context_factory` setting, Cassandra polls (at the same periodic interval mentioned above) your implementation to check if the SSL certificates need to be reloaded. See the ISslContextFactory documentation for more details. If you are using one of the Cassandra's in-built SSL context factory class (example: PEMBasedSslContextFactory) with file based key material, it supports the hot reloading of the SSL certificates like mentioned above.

Certificate Hot reloading may also be triggered using the `nodetool reloadssl` command. Use this if you want to Cassandra to immediately notice the changed certificates.

Inter-node Encryption

The settings for managing inter-node encryption are found in `cassandra.yaml` in the `server_encryption_options` section. To enable inter-node encryption, change the `internode_encryption` setting from its default value of `none` to one value from: `rack`, `dc` or `all`.

Client to Node Encryption

The settings for managing client to node encryption are found in `cassandra.yaml` in the `client_encryption_options` section. There are two primary toggles here for enabling encryption, `enabled` and `optional`.

- If neither is set to `true`, client connections are entirely unencrypted.
- If `enabled` is set to `true` and `optional` is set to `false`, all client connections must be secured.
- If both options are set to `true`, both encrypted and unencrypted connections are supported using the same port. Client connections using encryption with this configuration will be automatically detected and handled by the server.

As an alternative to the `optional` setting, separate ports can also be configured for secure and unsecure connections where operational requirements demand it. To do so, set `optional` to false and use the `native_transport_port_ssl` setting in `cassandra.yaml` to specify the port to be used for secure client communication.

Roles

Cassandra uses database roles, which may represent either a single user or a group of users, in both authentication and permissions management. Role management is an extension point in Cassandra and may be configured using the `role_manager` setting in `cassandra.yaml`. The default setting uses `CassandraRoleManager`, an implementation which stores role information in the tables of the `system_auth` keyspace.

See also the **`CQL documentation on roles`**.

Authentication

Authentication is pluggable in Cassandra and is configured using the `authenticator` setting in `cassandra.yaml`. Cassandra ships with two options included in the default distribution.

By default, Cassandra is configured with `AllowAllAuthenticator` which performs no authentication checks and therefore requires no credentials. It is used to disable authentication completely. Note that authentication is a necessary condition of Cassandra’s permissions subsystem, so if authentication is disabled, effectively so are permissions.

The default distribution also includes `PasswordAuthenticator`, which stores encrypted credentials in a system table. This can be used to enable simple username/password authentication.

Enabling Password Authentication

Before enabling client authentication on the cluster, client applications should be pre-configured with their intended credentials. When a connection is initiated, the server will only ask for credentials once authentication is enabled, so setting up the client side config in advance is safe. In contrast, as soon as a server has authentication enabled, any connection attempt without proper credentials will be rejected which may cause availability problems for client applications. Once clients are setup and ready for authentication to be enabled, follow this procedure to enable it on the cluster.

Pick a single node in the cluster on which to perform the initial configuration. Ideally, no clients should connect to this node during the setup process, so you may want to remove it from client config, block it at the network level or possibly add a new temporary node to the cluster for this purpose. On that node, perform the following steps:

1. Open a `cqlsh` session and change the replication factor of the `system_auth` keyspace. By default, this keyspace uses `SimpleReplicationStrategy` and a `replication_factor` of 1. It is recommended to change this for any non-trivial deployment to ensure that should nodes become unavailable, login is still possible. Best practice is to configure a replication factor of 3 to 5 per-DC.

```
ALTER KEYSPACE system_auth WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1':
3, 'DC2': 3};
```

2. Edit `cassandra.yaml` to change the `authenticator` option like so:

```
authenticator: PasswordAuthenticator
```

3. Restart the node.
4. Open a new `cqlsh` session using the credentials of the default superuser:

```
$ cqlsh -u cassandra -p cassandra
```

5. During login, the credentials for the default superuser are read with a consistency level of `QUORUM`, whereas those for all other users (including superusers) are read at `LOCAL_ONE`. In the interests of performance and availability, as well as security, operators should create another superuser and disable the default one. This step is optional, but highly recommended. While logged in as the default superuser, create another superuser role which can be used to bootstrap further configuration.

```
# create a new superuser
CREATE ROLE dba WITH SUPERUSER = true AND LOGIN = true AND PASSWORD = 'super';
```

6. Start a new cqlsh session, this time logging in as the new_superuser and disable the default superuser.

```
ALTER ROLE cassandra WITH SUPERUSER = false AND LOGIN = false;
```

7. Finally, set up the roles and credentials for your application users with **`CREATE ROLE`** statements.

At the end of these steps, the one node is configured to use password authentication. To roll that out across the cluster, repeat steps 2 and 3 on each node in the cluster. Once all nodes have been restarted, authentication will be fully enabled throughout the cluster.

Note that using `PasswordAuthenticator` also requires the use of **`CassandraRoleManager`**.

See also: `setting-credentials-for-internal-authentication`, **`CREATE ROLE`**, **`ALTER ROLE`**, **`ALTER KEYSPACE`** and **`GRANT PERMISSION`**.

Authorization

Authorization is pluggable in Cassandra and is configured using the `authorizer` setting in `cassandra.yaml`. Cassandra ships with two options included in the default distribution.

By default, Cassandra is configured with `AllowAllAuthorizer` which performs no checking and so effectively grants all permissions to all roles. This must be used if `AllowAllAuthenticator` is the configured authenticator.

The default distribution also includes `CassandraAuthorizer`, which does implement full permissions management functionality and stores its data in Cassandra system tables.

Enabling Internal Authorization

Permissions are modelled as a whitelist, with the default assumption that a given role has no access to any database resources. The implication of this is that once authorization is enabled on a node, all requests will be rejected until the required permissions have been granted. For this reason, it is strongly recommended to perform the initial setup on a node which is not processing client requests.

The following assumes that authentication has already been enabled via the process outlined in `password-authentication`. Perform these steps to enable internal authorization across the cluster:

1. On the selected node, edit `cassandra.yaml` to change the `authorizer` option like so:

```
authorizer: CassandraAuthorizer
```

2. Restart the node.
3. Open a new `cqlsh` session using the credentials of a role with superuser credentials:

```
$ cqlsh -u dba -p super
```

4. Configure the appropriate access privileges for your clients using **GRANT PERMISSION** statements. On the other nodes, until configuration is updated and the node restarted, this will have no effect so disruption to clients is avoided.

```
GRANT SELECT ON ks.t1 TO db_user;
```

5. Once all the necessary permissions have been granted, repeat steps 1 and 2 for each node in turn. As each node restarts and clients reconnect, the enforcement of the granted permissions will begin.

See also: `GRANT PERMISSION`, `GRANT ALL` and `REVOKE PERMISSION`.

Caching

Enabling authentication and authorization places additional load on the cluster by frequently reading from the `system_auth` tables. Furthermore, these reads are in the critical paths of many client operations, and so has the potential to severely impact quality of service. To mitigate this, auth data such as credentials, permissions and role details are cached for a configurable period. The caching can be configured (and even disabled) from `cassandra.yaml` or using a JMX client. The JMX interface also supports invalidation of the various caches, but any changes made via JMX are not persistent and will be re-read from `cassandra.yaml` when the node is restarted.

Each cache has 3 options which can be set:

Validity Period

Controls the expiration of cache entries. After this period, entries are invalidated and removed from the cache.

Refresh Rate

Controls the rate at which background reads are performed to pick up any changes to the underlying data. While these async refreshes are performed, caches will continue to serve (possibly) stale data. Typically, this will be set to a shorter time than the validity period.

Max Entries

Controls the upper bound on cache size.

The naming for these options in `cassandra.yaml` follows the convention:

- `<type>_validity_in_ms`
- `<type>_update_interval_in_ms`
- `<type>_cache_max_entries`

Where `<type>` is one of `credentials`, `permissions`, or `roles`.

As mentioned, these are also exposed via JMX in the mbeans under the `org.apache.cassandra.auth` domain.

JMX access

Access control for JMX clients is configured separately to that for CQL. For both authentication and authorization, two providers are available; the first based on standard JMX security and the second which integrates more closely with Cassandra's own auth subsystem.

The default settings for Cassandra make JMX accessible only from localhost. To enable remote JMX connections, edit `cassandra-env.sh` to change the `LOCAL_JMX` setting to `no`. Under the standard configuration, when remote JMX connections are enabled, `standard JMX authentication <standard-jmx-auth>` is also switched on.

Note that by default, local-only connections are not subject to authentication, but this can be enabled.

If enabling remote connections, it is recommended to also use **`SSL`** connections.

Finally, after enabling auth and/or SSL, ensure that tools which use JMX, such as **`nodetool`** are correctly configured and working as expected.

Standard JMX Auth

Users permitted to connect to the JMX server are specified in a simple text file. The location of this file is set in `cassandra-env.sh` by the line:

```
JVM_OPTS="$JVM_OPTS
-Dcom.sun.management.jmxremote.password.file=/etc/cassandra/jmxremote.password"
```

Edit the password file to add username/password pairs:

```
jmx_user jmx_password
```

Secure the credentials file so that only the user running the Cassandra process can read it :

```
$ chown cassandra:cassandra /etc/cassandra/jmxremote.password
$ chmod 400 /etc/cassandra/jmxremote.password
```

Optionally, enable access control to limit the scope of what defined users can do via JMX. Note that this is a fairly blunt instrument in this context as most operational tools in Cassandra require full read/write access. To configure a simple access file, uncomment this line in `cassandra-env.sh`:

```
#JVM_OPTS="$JVM_OPTS
-Dcom.sun.management.jmxremote.access.file=/etc/cassandra/jmxremote.access"
```

Then edit the access file to grant your JMX user readwrite permission:

```
jmx_user readwrite
```

Cassandra must be restarted to pick up the new settings.

See also : Using File-Based Password Authentication In JMX

Cassandra Integrated Auth

An alternative to the out-of-the-box JMX auth is to useeCassandra's own authentication and/or authorization providers for JMX clients. This is potentially more flexible and secure but it come with one major caveat. Namely that it is not available until after a node has joined the ring, because the auth subsystem is not fully configured until that point However, it is often critical for monitoring purposes to have JMX access particularly during bootstrap. So it is recommended, where possible, to use local only JMX auth during bootstrap and then, if remote connectivity is required, to switch to integrated auth once the node has joined the ring and initial setup is complete.

With this option, the same database roles used for CQL authentication can be used to control access to JMX, so updates can be managed centrally using just `cqlsh`. Furthermore, fine grained control over exactly which operations are permitted on particular MBeans can be acheived via **`GRANT PERMISSION`**.

To enable integrated authentication, edit `cassandra-env.sh` to uncomment these lines:

```
#JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.remote.login.config=CassandraLogin"
#JVM_OPTS="$JVM_OPTS -Djava.security.auth.login.config=$CASSANDRA_HOME/conf/cassandra
-jaas.config"
```

And disable the JMX standard auth by commenting this line:

```
JVM_OPTS="$JVM_OPTS
-Dcom.sun.management.jmxremote.password.file=/etc/cassandra/jmxremote.password"
```

To enable integrated authorization, uncomment this line:

```
#JVM_OPTS="$JVM_OPTS
```

```
-Dcassandra.jmx.authorizer=org.apache.cassandra.auth.jmx.AuthorizationProxy"
```

Check standard access control is off by ensuring this line is commented out:

```
#JVM_OPTS="$JVM_OPTS
-Dcom.sun.management.jmxremote.access.file=/etc/cassandra/jmxremote.access"
```

With integrated authentication and authorization enabled, operators can define specific roles and grant them access to the particular JMX resources that they need. For example, a role with the necessary permissions to use tools such as jconsole or jmc in read-only mode would be defined as:

```
CREATE ROLE jmx WITH LOGIN = false;
GRANT SELECT ON ALL MBEANS TO jmx;
GRANT DESCRIBE ON ALL MBEANS TO jmx;
GRANT EXECUTE ON MBEAN 'java.lang:type=Threading' TO jmx;
GRANT EXECUTE ON MBEAN 'com.sun.management:type=HotSpotDiagnostic' TO jmx;

# Grant the role with necessary permissions to use nodetool commands (including nodetool
status) in read-only mode
GRANT EXECUTE ON MBEAN 'org.apache.cassandra.db:type=EndpointSnitchInfo' TO jmx;
GRANT EXECUTE ON MBEAN 'org.apache.cassandra.db:type=StorageService' TO jmx;

# Grant the jmx role to one with login permissions so that it can access the JMX tooling
CREATE ROLE ks_user WITH PASSWORD = 'password' AND LOGIN = true AND SUPERUSER = false;
GRANT jmx TO ks_user;
```

Fine grained access control to individual MBeans is also supported:

```
GRANT EXECUTE ON MBEAN
'org.apache.cassandra.db:type=Tables,keyspace=test_keyspace,table=t1' TO ks_user;
GRANT EXECUTE ON MBEAN
'org.apache.cassandra.db:type=Tables,keyspace=test_keyspace,table=*' TO ks_owner;
```

This permits the `ks_user` role to invoke methods on the MBean representing a single table in `test_keyspace`, while granting the same permission for all table level MBeans in that keyspace to the `ks_owner` role.

Adding/removing roles and granting/revoking of permissions is handled dynamically once the initial setup is complete, so no further restarts are required if permissions are altered.

See also: `Permissions`.

JMX With SSL

JMX SSL configuration is controlled by a number of system properties, some of which are optional. To turn on SSL, edit the relevant lines in `cassandra-env.sh` to uncomment and set the values of these properties as required:

`com.sun.management.jmxremote.ssl`
: set to true to enable SSL

`com.sun.management.jmxremote.ssl.need.client.auth`
: set to true to enable validation of client certificates

`com.sun.management.jmxremote.registry.ssl`
: enables SSL sockets for the RMI registry from which clients obtain the JMX connector stub

`com.sun.management.jmxremote.ssl.enabled.protocols`
: by default, the protocols supported by the JVM will be used, override with a comma-separated list. Note that this is not usually necessary and using the defaults is the preferred option.

`com.sun.management.jmxremote.ssl.enabled.cipher.suites`
: by default, the cipher suites supported by the JVM will be used, override with a comma-separated list. Note that this is not usually necessary and using the defaults is the preferred option.

`javax.net.ssl.keyStore`
: set the path on the local filesystem of the keystore containing server private keys and public certificates

`javax.net.ssl.keyStorePassword`
: set the password of the keystore file

`javax.net.ssl.trustStore`
: if validation of client certificates is required, use this property to specify the path of the truststore containing the public certificates of trusted clients

`javax.net.ssl.trustStorePassword`
: set the password of the truststore file

See also: Oracle Java7 Docs, Monitor Java with JMX

Crypto providers

The ability to specify a custom Java Crypto Provider was done as part of CASSANDRA-18624

The default configuration of `crypto_provider` in `cassandra.yaml` looks like this:

```
# Configures Java crypto provider. By default, it will use
# DefaultCryptoProvider which will install Amazon Correto
# Crypto Provider.
#
# Amazon Correto Crypto Provider works currently for
# x86_64 and aarch_64 platforms. If this provider fails it will
# fall back to the default crypto provider in the JRE.
#
# To force failure when the provider was not installed properly,
# set the property "fail_on_missing_provider" to "true".
#
# To bypass the installation of a crypto provider use
# class 'org.apache.cassandra.security.JREProvider'
#
crypto_provider:
  - class_name: org.apache.cassandra.security.DefaultCryptoProvider
    parameters:
      - fail_on_missing_provider: "false"
```

For older nodes, when they upgrade to Cassandra 5.0 with the same `cassandra.yaml` where `crypto_provider` section is not set yet, they will default to `JREProvider` which does not install any provider, and it will use the one which is in a JRE Cassandra runs with.

As the above snippet shows, `DefaultCryptoProvider` is installing Amazon Corretto Crypto provider which is proven to be way more performant than default crypto providers in a JRE installation.

If you want to use other crypto provider, you have two options.

The first one is to configure your JRE, specifically `java.security` file, to instruct JRE what crypto provider to use. You would need to put respective implementation of such crypto provider to the class path of JRE as well.

The second option is to implement your own crypto provider by extending `org.apache.cassandra.security.AbstractCryptoProvider` and implementing four methods:

getProviderName

Returns name of your provider

getProviderClassAsString

Returns FQCN of your actual crypto provider which extends `java.security.Provider`.

installator

Returns `Runnable` which installs your `java.security.Provider` in runtime.

isHealthyInstallation

Returns `true` if the installation is *healthy*, false otherwise. This serves as a way to check if the installation of your provider was successful or not.

Upon installation of a crypto provider, `AbstractCryptoProvider` checks whether the provider you want to install is already installed or not. If it is installed and its installation position is `1` (as providers are installed in an order), a message will be logged about this fact. This case may happen if you configured your JRE directly via `java.security` and you try to install same provider by Cassandra itself as well.

If it is installed already but not on the first position, if `fail_on_missing_provider` is set to `true`, an exception will be thrown and a node will fail to start. Same happens if the installation of a provider is not successful as such.

Platform-specific libraries are added to Cassandra's class path automatically by `cassandra.in.sh` script. Currently, there are `lib/aarch64` and `lib/x86_64` directories with JAR files for each respective architecture. A platform is determined by the output of `uname -m` command.

Snitch

Snitch

In Cassandra, the snitch has two functions:

- it teaches Cassandra enough about your network topology to route requests efficiently.
- it allows Cassandra to spread replicas around your cluster to avoid correlated failures. It does this by grouping machines into "datacenters" and "racks." Cassandra will do its best not to have more than one replica on the same "rack" (which may not actually be a physical location).

Dynamic snitching

The dynamic snitch monitors read latencies to avoid reading from hosts that have slowed down. The dynamic snitch is configured with the following properties on `cassandra.yaml`:

- `dynamic_snitch`: whether the dynamic snitch should be enabled or disabled.
- `dynamic_snitch_update_interval`: 100ms, controls how often to perform the more expensive part of host score calculation.
- `dynamic_snitch_reset_interval`: 10m, if set greater than zero, this will allow 'pinning' of replicas to hosts in order to increase cache capacity.
- `dynamic_snitch_badness_threshold:`: The badness threshold will control how much worse the pinned host has to be before the dynamic snitch will prefer other replicas over it. This is expressed as a double which represents a percentage. Thus, a value of 0.2 means Cassandra would continue to prefer the static snitch values until the pinned host was 20% worse than the fastest.

Snitch classes

The `endpoint_snitch` parameter in `cassandra.yaml` should be set to the class that implements `IEndpointSnitch` which will be wrapped by the dynamic snitch and decide if two endpoints are in the same data center or on the same rack. Out of the box, Cassandra provides the snitch implementations:

GossipingPropertyFileSnitch

This should be your go-to snitch for production use. The rack and datacenter for the local node are defined in cassandra-rackdc.properties and propagated to other nodes via gossip. If `cassandra-topology.properties` exists, it is used as a fallback, allowing migration from the PropertyFileSnitch.

SimpleSnitch

Treats Strategy order as proximity. This can improve cache locality when disabling read repair. Only appropriate for single-datacenter deployments.

PropertyFileSnitch

Proximity is determined by rack and data center, which are explicitly configured in `cassandra-`

topology.properties.

RackInferringSnitch

Proximity is determined by rack and data center, which are assumed to correspond to the 3rd and 2nd octet of each node's IP address, respectively. Unless this happens to match your deployment conventions, this is best used as an example of writing a custom Snitch class and is provided in that spirit.

Cloud-based snitches

These snitches are used in cloud environment for various cloud vendors. All cloud-based snitch implementations are currently extending `AbstractCloudMetadataServiceSnitch` (which in turn extends `AbstractNetworkTopologySnitch`).

Each cloud-based snitch has its own way how to resolve what rack and datacenter a respective node belongs to. `AbstractCloudMetadataServiceSnitch` encapsulates the most common apparatus for achieving it. All cloud-based snitches are calling an HTTP service, specific to a respective cloud. The constructor of `AbstractCloudMetadataServiceSnitch` accepts implementations of `AbstractCloudMetadataServiceConnector` which implement a method `apiCall` which, by default, executes an HTTP `GET` request against a predefined HTTP URL, sending no HTTP headers, and it expects a response with HTTP code `200`. It is possible to send various HTTP headers as part of the request if an implementator wants that.

Currently, the only implementation of `AbstractCloudMetadataServiceConnector` is `DefaultCloudMetadataServiceConnector`. If a user has a need to override the behavior of `AbstractCloudMetadataServiceConnector`, a user is welcome to do so by implementing its own connector and propagating it to the constructor of `AbstractCloudMetadataServiceSnitch`.

All cloud-based snitches are accepting these properties in `cassandra-rackdc.properties`:

metadata_url

URL of cloud service to retrieve topology information from, this is cloud-specific.

metadata_request_timeout

Default value of `30s` (30 seconds) sets connect timeout upon calls by `apiCall`. In other words, request against `metadata_url` will time out if no response arrives in that period.

dc_suffix

A string, by default empty, which will be appended to resolved datacenter.

In-built cloud-based snitches are:

Ec2Snitch

Appropriate for EC2 deployments in a single Region, or in multiple regions with inter-region VPC enabled (available since the end of 2017, see AWS announcement). Loads Region and Availability

Zone information from the EC2 API. The Region is treated as the datacenter, and the Availability Zone as the rack. Only private IPs are used, so this will work across multiple regions only if inter-region VPC is enabled.

Ec2MultiRegionSnitch

Uses public IPs as broadcast_address to allow cross-region connectivity (thus, you should set seed addresses to the public IP as well). You will need to open the `storage_port` or `ssl_storage_port` on the public IP firewall (For intra-Region traffic, Cassandra will switch to the private IP after establishing a connection).

For Ec2 snitches, since CASSANDRA-16555, it is possible to choose version of AWS IMDS. By default, IMDSv2 is used. The version of IMDS is driven by property `ec2_metadata_type` and can be of value either `v1` or `v2`. It is possible to specify custom URL of IMDS by `ec2_metadata_url` (or by `metadata_url`) which is by default `http://169.254.169.254` and then a query against `/latest/meta-data/placement/availability-zone` endpoint is executed.

IMDSv2 is secured by a token which needs to be fetched from IDMSv2 first, and it has to be passed in a header for the actual queries to IDMSv2. `Ec2Snitch` and `Ec2MultiRegionSnitch` are doing this automatically. The only configuration parameter exposed to a user is `ec2_metadata_token_ttl_seconds` which is by default set to `21600`. TTL has to be an integer from the range `[30, 21600]`.

AlibabaCloudSnitch

A snitch that assumes an ECS region is a DC and an ECS availability_zone is a rack. This information is available in the config for the node. the format of the zone-id is like `cn-hangzhou-a` where `cn` means China, `hangzhou` means the Hangzhou region, `a` means the az id. We use `cn-hangzhou` as the dc, and `a` as the zone-id. `metadata_url` for this snitch is, by default, `http://100.100.100.200/` and it will execute an HTTP request against endpoint `/latest/meta-data/zone-id`.

AzureSnitch

Azure Snitch will resolve datacenter and rack by calling `/metadata/instance/compute?api-version=%s&format=json` endpoint against `metadata_url` of `http://169.254.169.254` returning the response in JSON format for, by default, API version `2021-12-13`. The version of API is configurable via property `azure_api_version` in `cassandra-rackdc.properties`. A datacenter is resolved from `location` field of the response and a rack is resolved by looking into `zone` field first. When `zone` is not set, or it is an empty string, it will look into `platformFaultDomain` field. Such resolved value is prepended by `rack-` string.

GoogleCloudSnitch

Google snitch will resolve datacenter and rack by calling `/computeMetadata/v1/instance/zone` endpoint against `metadata_url` of `http://metadata.google.internal`.

Adding, replacing, moving and removing nodes

Adding, replacing, moving and removing nodes

Bootstrap

Adding new nodes is called "bootstrapping". The `num_tokens` parameter will define the amount of virtual nodes (tokens) the joining node will be assigned during bootstrap. The tokens define the sections of the ring (token ranges) the node will become responsible for.

Token allocation

With the default token allocation algorithm the new node will pick `num_tokens` random tokens to become responsible for. Since tokens are distributed randomly, load distribution improves with a higher amount of virtual nodes, but it also increases token management overhead. The default of 256 virtual nodes should provide a reasonable load balance with acceptable overhead.

On 3.0+ a new token allocation algorithm was introduced to allocate tokens based on the load of existing virtual nodes for a given keyspace, and thus yield an improved load distribution with a lower number of tokens. To use this approach, the new node must be started with the JVM option `-Dcassandra.allocate_tokens_for_keyspace=<keyspace>`, where `<keyspace>` is the keyspace from which the algorithm can find the load information to optimize token assignment for.

Manual token assignment

You may specify a comma-separated list of tokens manually with the `initial_token` `cassandra.yaml` parameter, and if that is specified Cassandra will skip the token allocation process. This may be useful when doing token assignment with an external tool or when restoring a node with its previous tokens.

Range streaming

After the tokens are allocated, the joining node will pick current replicas of the token ranges it will become responsible for to stream data from. By default it will stream from the primary replica of each token range in order to guarantee data in the new node will be consistent with the current state.

In the case of any unavailable replica, the consistent bootstrap process will fail. To override this behavior and potentially miss data from an unavailable replica, set the JVM flag `-Dcassandra.consistent.rangemovement=false`.

Resuming failed/hanged bootstrap

On 2.2+, if the bootstrap process fails, it’s possible to resume bootstrap from the previous saved state by calling `nodetool bootstrap resume`. If for some reason the bootstrap hangs or stalls, it may also be resumed by simply restarting the node. In order to cleanup bootstrap state and start fresh, you may set the JVM startup flag `-Dcassandra.reset_bootstrap_progress=true`.

On lower versions, when the bootstrap proces fails it is recommended to wipe the node (remove all the data), and restart the bootstrap process again.

Manual bootstrapping

It's possible to skip the bootstrapping process entirely and join the ring straight away by setting the hidden parameter `auto_bootstrap: false`. This may be useful when restoring a node from a backup or creating a new data-center.

Removing nodes

You can take a node out of the cluster with `nodetool decommission` to a live node, or `nodetool removenode` (to any other machine) to remove a dead one. This will assign the ranges the old node was responsible for to other nodes, and replicate the appropriate data there. If decommission is used, the data will stream from the decommissioned node. If removenode is used, the data will stream from the remaining replicas.

No data is removed automatically from the node being decommissioned, so if you want to put the node back into service at a different token on the ring, it should be removed manually.

Moving nodes

When `num_tokens: 1` it's possible to move the node position in the ring with `nodetool move`. Moving is both a convenience over and more efficient than decommission + bootstrap. After moving a node, `nodetool cleanup` should be run to remove any unnecessary data.

Replacing a dead node

In order to replace a dead node, start cassandra with the JVM startup flag `-Dcassandra.replace_address_first_boot=<dead_node_ip>`. Once this property is enabled the node starts in a hibernate state, during which all the other nodes will see this node to be DOWN (DN), however this node will see itself as UP (UN). Accurate replacement state can be found in `nodetool netstats`.

The replacing node will now start to bootstrap the data from the rest of the nodes in the cluster. A replacing node will only receive writes during the bootstrapping phase if it has a different ip address to the node that is being replaced. (See CASSANDRA-8523 and CASSANDRA-12344)

Once the bootstrapping is complete the node will be marked "UP".

NOTE

If any of the following cases apply, you **MUST** run repair to make the replaced node consistent again, since it missed ongoing writes during/prior to bootstrapping. The *replacement* timeframe refers to the period from when the node initially dies to when a new node completes the replacement process.

1. The node is down for longer than `max_hint_window` before being replaced.
2. You are replacing using the same IP address as the dead node **and** replacement takes longer than `max_hint_window`.

Monitoring progress

Bootstrap, replace, move and remove progress can be monitored using `nodetool netstats` which will show the progress of the streaming operations.

Cleanup data after range movements

As a safety measure, Cassandra does not automatically remove data from nodes that "lose" part of their token range due to a range movement operation (bootstrap, move, replace). Run `nodetool cleanup` on the nodes that lost ranges to the joining node when you are satisfied the new node is up and working. If you do not do this the old data will still be counted against the load on that node.

Transient Replication

Transient Replication

IMPORTANT

Transient Replication (CASSANDRA-14404) is an experimental feature designed for expert Apache Cassandra users who are able to validate every aspect of the database for their application and deployment. That means being able to check that operations like reads, writes, decommission, remove, rebuild, repair, and replace all work with your queries, data, configuration, operational practices, and availability requirements. Apache Cassandra 4.0 has the initial implementation of transient replication. Future releases of Cassandra will make this feature suitable for a wider audience. It is anticipated that a future version will support monotonic reads with transient replication as well as LWT, logged batches, and counters. Being experimental, Transient replication is **not** recommended for production use.

Objective

The objective of transient replication is to decouple storage requirements from data redundancy (or consensus group size) using incremental repair, in order to reduce storage overhead. Certain nodes act as full replicas (storing all the data for a given token range), and some nodes act as transient replicas, storing only unrepaired data for the same token ranges.

The optimization that is made possible with transient replication is called "Cheap quorums", which implies that data redundancy is increased without corresponding increase in storage usage.

Transient replication is useful when sufficient full replicas are unavailable to receive and store all the data. Transient replication allows you to configure a subset of replicas to only replicate data that hasn't been incrementally repaired. As an optimization, we can avoid writing data to a transient replica if we have successfully written data to the full replicas.

After incremental repair, transient data stored on transient replicas can be discarded.

Enabling Transient Replication

Transient replication is not enabled by default. Transient replication must be enabled on each node in a cluster separately by setting the following configuration property in `cassandra.yaml`.

```
transient_replication_enabled: true
```

Transient replication may be configured with both `SimpleStrategy` and `NetworkTopologyStrategy`. Transient replication is configured by setting replication factor as `<total_replicas>/<transient_replicas>`.

As an example, create a keyspace with replication factor (RF) 3.

```
CREATE KEYSPACE CassandraKeyspaceSimple WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 4/1};
```

As another example, `some_keysopace keyspace` will have 3 replicas in DC1, 1 of which is transient, and 5 replicas in DC2, 2 of which are transient:

```
CREATE KEYSPACE some_keysopace WITH replication = {'class': 'NetworkTopologyStrategy',
'DC1' : '3/1'', 'DC2' : '5/2'};
```

Transiently replicated keyspaces only support tables with `read_repair` set to `NONE`.

Important Restrictions:

- RF cannot be altered while some endpoints are not in a normal state (no range movements).
- You can't add full replicas if there are any transient replicas. You must first remove all transient replicas, then change the # of full replicas, then add back the transient replicas.
- You can only safely increase number of transients one at a time with incremental repair run in between each time.

Additionally, transient replication cannot be used for:

- Monotonic Reads
- Lightweight Transactions (LWTs)
- Logged Batches
- Counters
- Keyspaces using materialized views
- Secondary indexes (2i)

Cheap Quorums

Cheap quorums are a set of optimizations on the write path to avoid writing to transient replicas unless sufficient full replicas are not available to satisfy the requested consistency level. Hints are never written for transient replicas. Optimizations on the read path prefer reading from transient replicas. When writing at quorum to a table configured to use transient replication the quorum will always prefer available full replicas over transient replicas so that transient replicas don't have to process writes. Tail latency is reduced by rapid write protection (similar to rapid read protection) when full replicas are slow or unavailable by sending writes to transient replicas. Transient replicas can serve reads faster as they don't have to do anything beyond bloom filter checks if they have no data. With vnodes and large cluster sizes they will not have a large quantity of data even for failure of one or more full replicas where transient replicas start to serve a steady amount of write traffic for some of their transiently replicated ranges.

Speculative Write Option

The `CREATE TABLE` adds an option `speculative_write_threshold` for use with transient replicas. The option is of type `simple` with default value as `99PERCENTILE`. When replicas are slow or unresponsive `speculative_write_threshold` specifies the threshold at which a cheap quorum write will be upgraded to include transient replicas.

Pending Ranges and Transient Replicas

Pending ranges refers to the movement of token ranges between transient replicas. When a transient range is moved, there will be a period of time where both transient replicas would need to receive any write intended for the logical transient replica so that after the movement takes effect a read quorum is able to return a response. Nodes are *not* temporarily transient replicas during expansion. They stream data like a full replica for the transient range before they can serve reads. A pending state is incurred similar to how there is a pending state for full replicas. Transient replicas also always receive writes when they are pending. Pending transient ranges are sent a bit more data and reading from them is avoided.

Read Repair and Transient Replicas

Read repair never attempts to repair a transient replica. Reads will always include at least one full replica. They should also prefer transient replicas where possible. Range scans ensure the entire scanned range performs replica selection that satisfies the requirement that every range scanned includes one full replica. During incremental & validation repair handling, at transient replicas anti-compaction does not output any data for transient ranges as the data will be dropped after repair, and transient replicas never have data streamed to them.

Transitioning between Full Replicas and Transient Replicas

The additional state transitions that transient replication introduces requires streaming and `nodetool cleanup` to behave differently. When data is streamed it is ensured that it is streamed from a full replica and not a transient replica.

Transitioning from not replicated to transiently replicated means that a node must stay pending until the next incremental repair completes at which point the data for that range is known to be available at full replicas.

Transitioning from transiently replicated to fully replicated requires streaming from a full replica and is identical to how data is streamed when transitioning from not replicated to replicated. The transition is managed so the transient replica is not read from as a full replica until streaming completes. It can be used immediately for a write quorum.

Transitioning from fully replicated to transiently replicated requires cleanup to remove repaired data from the transiently replicated range to reclaim space. It can be used immediately for a write quorum.

Transitioning from transiently replicated to not replicated requires cleanup to be run to remove the formerly transiently replicated data.

When transient replication is in use ring changes are supported including add/remove node, change RF, add/remove DC.

Transient Replication supports EACH_QUORUM

(CASSANDRA-14727) adds support for Transient Replication support for `EACH_QUORUM`. Per (CASSANDRA-14768), we ensure we write to at least a `QUORUM` of nodes in every DC, regardless of how many responses we need to wait for and our requested consistency level. This is to minimally surprise users with transient replication; with normal writes, we soft-ensure that we reach `QUORUM` in all DCs we are able to, by writing to every node; even if we don’t wait for ACK, we have in both cases sent sufficient messages.

Virtual Tables

Virtual Tables

Apache Cassandra 4.0 implements virtual tables (CASSANDRA-7622). Virtual tables are tables backed by an API instead of data explicitly managed and stored as SSTables. Apache Cassandra 4.0 implements a virtual keyspace interface for virtual tables. Virtual tables are specific to each node.

Some of the features of virtual tables are the ability to:

- expose metrics through CQL
- expose YAML configuration information

Virtual keyspaces and tables are quite different from regular tables and keyspaces:

- Virtual tables are created in special keyspaces and not just any keyspace.
- Virtual tables are managed by Cassandra. Users cannot run DDL to create new virtual tables or DML to modify existing virtual tables.
- Virtual tables are currently read-only, although that may change in a later version.
- Virtual tables are local only, non-distributed, and thus not replicated.
- Virtual tables have no associated SSTables.
- Consistency level of the queries sent to virtual tables are ignored.
- All existing virtual tables use `LocalPartitioner`. Since a virtual table is not replicated the partitioner sorts in order of partition keys instead of by their hash.
- Making advanced queries using `ALLOW FILTERING` and aggregation functions can be executed in virtual tables, even though it is not recommended in normal tables.

Virtual Keyspaces

Apache Cassandra 4.0 has added two new keyspaces for virtual tables:

- `system_virtual_schema`
- `system_views`.

The `system_virtual_schema` keyspace has three tables: `keyspaces`, `columns` and `tables` for the virtual keyspace, table, and column definitions, respectively. These tables contain schema information for the virtual tables. It is used by Cassandra internally and a user should not access it directly.

The `system_views` keyspace contains the actual virtual tables.

Virtual Table Limitations

Before discussing virtual keyspaces and tables, note that virtual keyspaces and tables have some limitations. These limitations are subject to change. Virtual keyspaces cannot be altered or dropped. In fact, no operations can be performed against virtual keyspaces.

Virtual tables cannot be created in virtual keyspaces. Virtual tables cannot be altered, dropped, or truncated. Secondary indexes, types, functions, aggregates, materialized views, and triggers cannot be created for virtual tables. Expiring time-to-live (TTL) columns cannot be created. Virtual tables do not support conditional updates or deletes. Aggregates may be run in SELECT statements.

Conditional batch statements cannot include mutations for virtual tables, nor can a virtual table statement be included in a logged batch. In fact, mutations for virtual and regular tables cannot occur in the same batch table.

Virtual Tables

Each of the virtual tables in the `system_views` virtual keyspace contain different information.

The following table describes the virtual tables:

Virtual Table	**Description**
caches	Displays the general cache information including cache name, capacity_bytes, entry_count, hit_count, hit_ratio double, recent_hit_rate_per_second, recent_request_rate_per_second, request_count, and size_bytes.
cidr_filtering_metrics_counts	Counts metrics specific to CIDR filtering.
cidr_filtering_metrics_latencies	Latencies metrics specific to CIDR filtering.
clients	Lists information about all connected clients.
coordinator_read_latency	Records counts, keyspace_name, table_name, max, median, and per_second for coordinator reads.
coordinator_scan	Records counts, keyspace_name, table_name, max, median, and per_second for coordinator scans.
coordinator_write_latency	Records counts, keyspace_name, table_name, max, median, and per_second for coordinator writes.
disk_usage	Records disk usage including disk_space, keyspace_name, and table_name, sorted by system keyspaces.
internode_inbound	Lists information about the inbound internode messaging.

Virtual Table	Description
internode_outbound	Information about the outbound internode messaging.
local_read_latency	Records counts, keyspace_name, table_name, max, median, and per_second for local reads.
local_scan	Records counts, keyspace_name, table_name, max, median, and per_second for local scans.
local_write_latency	Records counts, keyspace_name, table_name, max, median, and per_second for local writes.
max_partition_size	A table metric for maximum partition size.
rows_per_read	Records counts, keyspace_name, tablek_name, max, and median for rows read.
settings	Displays configuration settings in cassandra.yaml.
sstable_tasks	Lists currently running tasks and progress on SSTables, for operations like compaction and upgrade.
system_logs	Displays Cassandra logs if logged via CQLLOG appender in logback.xml
system_properties	Displays environmental system properties set on the node.
thread_pools	Lists metrics for each thread pool.
tombstones_per_read	Records counts, keyspace_name, tablek_name, max, and median for tombstones.

For improved usability, from CASSANDRA-18238, all tables except `system_logs` have `ALLOW FILTERING` implicitly added to a query when required by CQL specification.

We shall discuss some of the virtual tables in more detail next.

Clients Virtual Table

The `clients` virtual table lists all active connections (connected clients) including their ip address, port, client_options, connection stage, driver name, driver version, hostname, protocol version, request count, ssl enabled, ssl protocol and user name:

```
cqlsh> SELECT * FROM system_views.clients;

@ Row 1
------------------+
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------
 address          | 127.0.0.1
 port             | 50687
 client_options   | {'CQL_VERSION': '3.4.7', 'DRIVER_NAME': 'DataStax Python Driver',
'DRIVER_VERSION': '3.25.0'}
 connection_stage | ready
 driver_name      | DataStax Python Driver
 driver_version   | 3.25.0
 hostname         | localhost
 protocol_version | 5
 request_count    | 16
 ssl_cipher_suite | null
 ssl_enabled      | False
 ssl_protocol     | null
 username         | anonymous

@ Row 2
------------------+
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-------------------------------------------------------
 address          | 127.0.0.1
 port             | 50688
 client_options   | {'CQL_VERSION': '3.4.7', 'DRIVER_NAME': 'DataStax Python Driver',
'DRIVER_VERSION': '3.25.0'}
 connection_stage | ready
 driver_name      | DataStax Python Driver
 driver_version   | 3.25.0
 hostname         | localhost
 protocol_version | 5
 request_count    | 4
 ssl_cipher_suite | null
 ssl_enabled      | False
 ssl_protocol     | null
 username         | anonymous

@ Row 3
------------------+
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-------------------------------------------------------
 address          | 127.0.0.1
 port             | 50753
 client_options   | {'APPLICATION_NAME': 'TestApp', 'APPLICATION_VERSION': '1.0.0',
'CLIENT_ID': '55b3efbd-c56b-469d-8cca-016b860b2f03', 'CQL_VERSION': '3.0.0',
'DRIVER_NAME': 'DataStax Java driver for Apache Cassandra(R)', 'DRIVER_VERSION':
'4.13.0'}
 connection_stage | ready
 driver_name      | DataStax Java driver for Apache Cassandra(R)
 driver_version   | 4.13.0
 hostname         | localhost
 protocol_version | 5
 request_count    | 18
 ssl_cipher_suite | null
 ssl_enabled      | False
 ssl_protocol     | null
 username         | anonymous

@ Row 4
------------------+
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-------------------------------------------------------
 address          | 127.0.0.1
 port             | 50755
 client_options   | {'APPLICATION_NAME': 'TestApp', 'APPLICATION_VERSION': '1.0.0',
'CLIENT_ID': '55b3efbd-c56b-469d-8cca-016b860b2f03', 'CQL_VERSION': '3.0.0',
'DRIVER_NAME': 'DataStax Java driver for Apache Cassandra(R)', 'DRIVER_VERSION':
'4.13.0'}
 connection_stage | ready
 driver_name      | DataStax Java driver for Apache Cassandra(R)
 driver_version   | 4.13.0
 hostname         | localhost
 protocol_version | 5
 request_count    | 7
 ssl_cipher_suite | null
 ssl_enabled      | False
 ssl_protocol     | null
 username         | anonymous

(4 rows)
```

Some examples of how `clients` can be used are:

- To find applications using old incompatible versions of drivers before upgrading and with `nodetool enableoldprotocolversions` and `nodetool disableoldprotocolversions` during upgrades.
- To identify clients sending too many requests.
- To find if SSL is enabled during the migration to and from ssl.

The virtual tables may be described with `DESCRIBE` statement. The DDL listed however cannot be run to create a virtual table. As an example describe the `system_views.clients` virtual table:

```
cqlsh> DESCRIBE TABLE system_views.clients;

/*
Warning: Table system_views.clients is a virtual table and cannot be recreated with CQL.
Structure, for reference:
```

```
VIRTUAL TABLE system_views.clients (
  address inet,
  port int,
  client_options frozen<map<text, text>>,
  connection_stage text,
  driver_name text,
  driver_version text,
  hostname text,
  protocol_version int,
  request_count bigint,
  ssl_cipher_suite text,
  ssl_enabled boolean,
  ssl_protocol text,
  username text,
    PRIMARY KEY (address, port)
) WITH CLUSTERING ORDER BY (port ASC)
    AND comment = 'currently connected clients';
*/
```

Caches Virtual Table

The `caches` virtual table lists information about the caches. The four caches presently created are chunks, counters, keys and rows. A query on the `caches` virtual table returns the following details:

```
cqlsh:system_views> SELECT * FROM system_views.caches;
name     | capacity_bytes | entry_count | hit_count | hit_ratio |
recent_hit_rate_per_second | recent_request_rate_per_second | request_count | size_bytes
---------+----------------+-------------+-----------+-----------+
----------------------------+--------------------------------+---------------+
------------
  chunks |      229638144 |          29 |       166 |      0.83 |
5 |                              6 |           200 |     475136
counters |       26214400 |           0 |         0 |       NaN |
0 |                              0 |             0 |          0
    keys |       52428800 |          14 |       124 |  0.873239 |
4 |                              4 |           142 |       1248
    rows |              0 |           0 |         0 |       NaN |
0 |                              0 |             0 |          0

(4 rows)
```

CIDR filtering metrics Virtual Tables

The `cidr_filtering_metrics_counts` virtual table lists counts metrics specific to CIDR filtering. A query on `cidr_filtering_metrics_counts` virtual table lists metrics similar to below.

```
cqlsh> select * from system_views.cidr_filtering_metrics_counts;
 name                                                    | value
---------------------------------------------------------+-------
                          CIDR groups cache reload count |     2
 Number of CIDR accesses accepted from CIDR group - aws |    15
 Number of CIDR accesses accepted from CIDR group - gcp |    30
 Number of CIDR accesses rejected from CIDR group - gcp |     6
```

The `cidr_filtering_metrics_latencies` virtual table lists latencies metrics specific to CIDR filtering. A query on `cidr_filtering_metrics_latencies` virtual table lists below metrics.

```
cqlsh> select * from system_views.cidr_filtering_metrics_latencies;
 name                                         | max    | p50th | p95th | p999th | p99th
----------------------------------------------+-------+-------+-------+--------+-------
                     CIDR checks latency (ns) | 24601 |     1 | 11864 |  24601 | 24601
       CIDR groups cache reload latency (ns) | 42510 | 42510 | 42510 |  42510 | 42510
 Lookup IP in CIDR groups cache latency (ns) |     1 |     1 |     1 |      1 |     1
```

CQL metrics Virtual Table

The `cql_metrics` virtual table lists metrics specific to CQL prepared statement caching. A query on `cql_metrics` virtual table lists below metrics.

```
cqlsh> select * from system_views.cql_metrics ;

 name                          | value
-------------------------------+-------
    prepared_statements_count |     0
  prepared_statements_evicted |     0
 prepared_statements_executed |     0
    prepared_statements_ratio |     0
  regular_statements_executed |    17
```

CIDR filtering metrics Virtual Tables

The `cidr_filtering_metrics_counts` virtual table lists counts metrics specific to CIDR filtering. A query on `cidr_filtering_metrics_counts` virtual table lists metrics similar to below.

```
cqlsh> select * from system_views.cidr_filtering_metrics_counts;
 name                                                    | value
```

```
-------------------------------------------------------+-------
                         CIDR groups cache reload count |     2
 Number of CIDR accesses accepted from CIDR group - aws |    15
 Number of CIDR accesses accepted from CIDR group - gcp |    30
 Number of CIDR accesses rejected from CIDR group - gcp |     6
```

The `cidr_filtering_metrics_latencies` virtual table lists latencies metrics specific to CIDR filtering. A query on `cidr_filtering_metrics_latencies` virtual table lists below metrics.

```
cqlsh> select * from system_views.cidr_filtering_metrics_latencies;
 name                                         | max    | p50th | p95th | p999th | p99th
---------------------------------------------+-------+-------+-------+--------+-------
                    CIDR checks latency (ns) | 24601 |     1 | 11864 |  24601 | 24601
      CIDR groups cache reload latency (ns) | 42510 | 42510 | 42510 |  42510 | 42510
 Lookup IP in CIDR groups cache latency (ns) |     1 |     1 |     1 |      1 |     1
```

CQL metrics Virtual Table

The `cql_metrics` virtual table lists metrics specific to CQL prepared statement caching. A query on `cql_metrics` virtual table lists below metrics.

```
cqlsh> select * from system_views.cql_metrics ;

 name                         | value
------------------------------+-------
    prepared_statements_count |     0
  prepared_statements_evicted |     0
 prepared_statements_executed |     0
    prepared_statements_ratio |     0
  regular_statements_executed |    17
```

Settings Virtual Table

The `settings` table is rather useful and lists all the current configuration settings from the `cassandra.yaml`. The encryption options are overridden to hide the sensitive truststore information or passwords. The configuration settings however cannot be set using DML on the virtual table presently: :

```
cqlsh:system_views> SELECT * FROM system_views.settings;

name                                 | value
-------------------------------------+--------------------
```

```
  allocate_tokens_for_keyspace       | null
  audit_logging_options_enabled      | false
  auto_snapshot                      | true
  automatic_sstable_upgrade          | false
  cluster_name                       | Test Cluster
  enable_transient_replication       | false
  hinted_handoff_enabled             | true
  hints_directory                    | /home/ec2-user/cassandra/data/hints
  incremental_backups                | false
  initial_token                      | null
                           ...
                           ...
                           ...
  rpc_address                        | localhost
  ssl_storage_port                   | 7001
  start_native_transport             | true
  storage_port                       | 7000
  stream_entire_sstables             | true
  (224 rows)
```

The `settings` table can be really useful if yaml file has been changed since startup and don't know running configuration, or to find if they have been modified via jmx/nodetool or virtual tables.

Thread Pools Virtual Table

The `thread_pools` table lists information about all thread pools. Thread pool information includes active tasks, active tasks limit, blocked tasks, blocked tasks all time, completed tasks, and pending tasks. A query on the `thread_pools` returns following details:

```
cqlsh:system_views> select * from system_views.thread_pools;

name                         | active_tasks | active_tasks_limit | blocked_tasks |
blocked_tasks_all_time | completed_tasks | pending_tasks
------------------------------+--------------+--------------------+---------------+
------------------------+-----------------+---------------
            AntiEntropyStage |            0 |                  1 |             0 |
0 |               0 |             0
        CacheCleanupExecutor |            0 |                  1 |             0 |
0 |               0 |             0
          CompactionExecutor |            0 |                  2 |             0 |
0 |             881 |             0
        CounterMutationStage |            0 |                 32 |             0 |
0 |               0 |             0
                 GossipStage |            0 |                  1 |             0 |
0 |               0 |             0
             HintsDispatcher |            0 |                  2 |             0 |
0 |             0 |           0
      InternalResponseStage |           0 |                 2 |             0 |
0 |             0 |           0
        MemtableFlushWriter |           0 |                 2 |             0 |
0 |             1 |           0
          MemtablePostFlush |           0 |                 1 |             0 |
0 |             2 |           0
      MemtableReclaimMemory |           0 |                 1 |             0 |
0 |             1 |           0
             MigrationStage |           0 |                 1 |             0 |
0 |             0 |           0
                  MiscStage |           0 |                 1 |             0 |
0 |             0 |           0
              MutationStage |           0 |                32 |             0 |
0 |             0 |           0
    Native-Transport-Requests |           1 |               128 |             0 |
0 |           130 |           0
      PendingRangeCalculator |           0 |                 1 |             0 |
0 |             1 |           0
PerDiskMemtableFlushWriter_0 |           0 |                 2 |             0 |
0 |             1 |           0
                  ReadStage |           0 |                32 |             0 |
0 |            13 |           0
                Repair-Task |           0 |        2147483647 |             0 |
0 |             0 |           0
       RequestResponseStage |           0 |                 2 |             0 |
0 |             0 |           0
                    Sampler |           0 |                 1 |             0 |
0 |             0 |           0
     SecondaryIndexManagement |           0 |                 1 |             0 |
0 |             0 |           0
         ValidationExecutor |           0 |        2147483647 |             0 |
0 |             0 |           0
          ViewBuildExecutor |           0 |                 1 |             0 |
0 |             0 |           0
          ViewMutationStage |           0 |                32 |             0 |
0 |             0 |           0

(24 rows)
```

Internode Inbound Messaging Virtual Table

The `internode_inbound` virtual table is for the internode inbound messaging. Initially no internode inbound messaging may get listed. In addition to the address, port, datacenter and rack information includes corrupt frames recovered, corrupt frames unrecovered, error bytes, error count, expired bytes, expired count, processed bytes, processed count, received bytes, received count, scheduled bytes, scheduled count, throttled count, throttled nanos, using bytes, using reserve bytes. A query on the `internode_inbound` returns following details:

```
cqlsh:system_views> SELECT * FROM system_views.internode_inbound;
address | port | dc | rack | corrupt_frames_recovered | corrupt_frames_unrecovered |
error_bytes | error_count | expired_bytes | expired_count | processed_bytes |
processed_count | received_bytes | received_count | scheduled_bytes | scheduled_count |
throttled_count | throttled_nanos | using_bytes | using_reserve_bytes
---------+------+----+------+--------------------------+----------------------------+-
(0 rows)
```

SSTables Tasks Virtual Table

The `sstable_tasks` could be used to get information about running tasks. It lists following columns:

```
cqlsh:system_views> SELECT * FROM sstable_tasks;
keyspace_name | table_name | task_id                              | kind       | progress
| total    | unit
---------------+------------+--------------------------------------+------------+
----------+----------+-------
       basic |      wide2 | c3909740-cdf7-11e9-a8ed-0f03de2d9ae1 | compaction | 60418761
| 70882110 | bytes
       basic |      wide2 | c7556770-cdf7-11e9-a8ed-0f03de2d9ae1 | compaction |  2995623
| 40314679 | bytes
```

As another example, to find how much time is remaining for SSTable tasks, use the following query:

```
SELECT total - progress AS remaining
FROM system_views.sstable_tasks;
```

Other Virtual Tables

Some examples of using other virtual tables are as follows.

Find tables with most disk usage:

```
cqlsh> SELECT * FROM disk_usage WHERE mebibytes > 1 ALLOW FILTERING;

keyspace_name | table_name | mebibytes
---------------+------------+-----------
   keyspace1 |  standard1 |       288
```

```
  tlp_stress |   keyvalue |      3211
```

Find queries on table/s with greatest read latency:

```
cqlsh> SELECT * FROM  local_read_latency WHERE per_second > 1 ALLOW FILTERING;

keyspace_name | table_name | p50th_ms | p99th_ms | count    | max_ms  | per_second
---------------+------------+----------+----------+----------+---------+------------
  tlp_stress |   keyvalue |    0.043 |    0.152 | 49785158 | 186.563 |  11418.356
```

Example

1. To list the keyspaces, enter `cqlsh` and run the CQL command `DESCRIBE KEYSPACES`:

   ```
   cqlsh> DESC KEYSPACES;
   system_schema  system          system_distributed  system_virtual_schema
   system_auth    system_traces   system_views
   ```

2. To view the virtual table schema, run the CQL commands `USE system_virtual_schema` and `SELECT * FROM tables`:

   ```
   cqlsh> USE system_virtual_schema;
   cqlsh> SELECT * FROM tables;
   ```

   results in:

   ```
    keyspace_name         | table_name                | comment
   -----------------------+---------------------------+
   --------------------------------------
             system_views |                    caches |                        system
   caches
             system_views |                   clients |          currently connected
   clients
             system_views |  coordinator_read_latency |
             system_views |  coordinator_scan_latency |
             system_views | coordinator_write_latency |
             system_views |                disk_usage |
             system_views |          internode_inbound |
             system_views |         internode_outbound |
             system_views |         local_read_latency |
             system_views |         local_scan_latency |
             system_views |        local_write_latency |
   ```

```
          system_views |        max_partition_size |
          system_views |             rows_per_read |
          system_views |                  settings |                     current
settings
          system_views |             sstable_tasks |                current sstable
tasks
          system_views |         system_properties | Cassandra relevant system
properties
          system_views |              thread_pools |
          system_views |       tombstones_per_read |
 system_virtual_schema |                   columns |           virtual column
definitions
 system_virtual_schema |                 keyspaces |         virtual keyspace
definitions
 system_virtual_schema |                    tables |            virtual table
definitions

(21 rows)
```

3. To view the virtual tables, run the CQL commands `USE system_view` and `DESCRIBE tables`:

```
cqlsh> USE system_view;;
cqlsh> DESCRIBE tables;
```

results in:

```
sstable_tasks       clients                    coordinator_write_latency
disk_usage          local_write_latency        tombstones_per_read
thread_pools        internode_outbound         settings
local_scan_latency  coordinator_scan_latency   system_properties
internode_inbound   coordinator_read_latency   max_partition_size
local_read_latency  rows_per_read              caches
```

4. To look at any table data, run the CQL command `SELECT`:

```
cqlsh> USE system_view;;
cqlsh> SELECT * FROM clients LIMIT 2;
```

results in:

```
 address    | port  | connection_stage | driver_name            | driver_version |
hostname  | protocol_version | request_count | ssl_cipher_suite | ssl_enabled |
ssl_protocol | username
```

```
-----------+-------+------------------+-----------------------+----------------+
-----------|||+------------------+--------------+-----------------+------------+
--------------+-----------
 127.0.0.1 | 37308 |            ready | DataStax Python Driver |   3.21.0.post0 |
localhost |                4 |             17 |              null |       False |
null | anonymous
 127.0.0.1 | 37310 |            ready | DataStax Python Driver |   3.21.0.post0 |
localhost |                4 |              8 |              null |       False |
null | anonymous

(2 rows)
```

Denylisting Partitions

Denylisting Partitions

Due to access patterns and data modeling, sometimes there are specific partitions that are "hot" and can cause instability in a Cassandra cluster. This often occurs when your data model includes many update or insert operations on a single partition, causing the partition to grow very large over time and in turn making it very expensive to read and maintain.

Cassandra supports "denylisting" these problematic partitions so that when clients issue point reads (`SELECT` statements with the partition key specified) or range reads (`SELECT *`, etc that pull a range of data) that intersect with a blocked partition key, the query will be immediately rejected with an `InvalidQueryException`.

How to denylist a partition key

The `system_distributed.denylisted_partitions` table can be used to denylist partitions. There are a couple of ways to interact with and mutate this data. First: directly via CQL by inserting a record with the following details:

- Keyspace name (ks_name)
- Table name (table_name)
- Partition Key (partition_key)

The partition key format needs to be in the same form required by `nodetool getendpoints`.

Following are several examples for denylisting partition keys in keyspace `ks` and table `table1` for different data types on the primary key `Id`:

- Id is a simple type - `INSERT INTO system_distributed.denylisted_partitions (ks_name, table_name, partition_key) VALUES ('ks','table1','1');`
- Id is a blob - `INSERT INTO system_distributed.denylisted_partitions (ks_name, table_name, partition_key) VALUES ('ks','table1','12345f');`
- Id has a colon - `INSERT INTO system_distributed.denylisted_partitions (ks_name, table_name, partition_key) VALUES ('ks','table1','1\:2');`

In the case of composite column partition keys (Key1, Key2):

- `INSERT INTO system_distributed.denylisted_partitions (ks_name, table_name, partition_key) VALUES ('ks', 'table1', 'k11:k21')`

Special considerations

The denylist has the property in that you want to keep your cache (see below) and CQL data on a replica set as close together as possible, so you don't have different nodes in your cluster denying or allowing different keys. To best achieve this, the workflow for a denylist change (addition or deletion) should always be as follows:

JMX PATH (preferred for single changes):

1. Call the JMX hook for `denylistKey()` with the desired key
2. Double-check the cache reloaded with `isKeyDenylisted()`
3. Check for warnings about unrecognized keyspace/table combinations, limits, or consistency level. If you get a message about nodes being down and not hitting CL for denylist, recover the downed nodes and then trigger a re-load of the cache on each node with `loadPartitionDenylist()`

CQL PATH (preferred for bulk changes):

1. Mutate the denylisted partition lists via CQL
2. Trigger a re-load of the denylist cache on each node via JMX `loadPartitionDenylist()` (see below)
3. Check for warnings about lack of availability for a denylist refresh. In the event nodes are down, recover them, then go to 2.

Due to conditions on known unavailable range slices leading to alert storming on startup, the denylist cache will not load on node start unless it can achieve the configured consistency level in `cassandra.yaml` - `denylist_consistency_level`. The JMX call to `loadPartitionDenylist` will, however, load the cache regardless of the number of nodes available. This leaves the control for denylisting or not denylisting during degraded cluster states in the hands of the operator.

Denylisted Partitions Cache

Cassandra internally maintains an on-heap cache of denylisted partitions loaded from `system_distributed.denylisted_partitions`. The values for a table will be automatically repopulated every `denylist_refresh` as specified in the `conf/cassandra.yaml` file, defaulting to `600s`, or 10 minutes. Invalid records (unknown keyspaces, tables, or keys) will be ignored and not cached on load.

The cache can be refreshed in the following ways:

- During Cassandra node startup
- Via the automatic on-heap cache refresh mechanisms. Note: this will occur asynchronously on query after the `denylist_refresh` time is hit.
- Via the JMX command: `loadPartitionDenylist` in `the` `org.apache.cassandra.service.StorageProxyMBean` invocation point.

The Cache size is bounded by the following two config properties

- denylist_max_keys_per_table
- denylist_max_keys_total

On cache load, if a table exceeds the value allowed in `denylist_max_keys_per_table` (defaults to 1000), a warning will be printed to the logs and the remainder of the keys will not be cached. Similarly, if the total allowed size is exceeded, subsequent ks_name + table_name combinations (in clustering / lexicographical order) will be skipped as well, and a warning logged to the server logs.

NOTE Given the required workflow of 1) Mutate, 2) Reload cache, the auto-reload property seems superfluous. It exists to ensure that, should an operator make a mistake and denylist (or undenylist) a key but forget to reload the cache, that intent will be captured on the next cache reload.

JMX Interface

Command	Effect
loadPartitionDenylist()	Reloads cached denylist from CQL table
getPartitionDenylistLoadAttempts()	Gets the count of cache reload attempts
getPartitionDenylistLoadSuccesses()	Gets the count of cache reload successes
setEnablePartitionDenylist(boolean enabled)	Enables or disables the partition denylisting functionality
setEnableDenylistWrites(boolean enabled)	Enables or disables write denylisting functionality
setEnableDenylistReads(boolean enabled)	Enables or disables read denylisting functionality
setEnableDenylistRangeReads(boolean enabled)	Enables or disables range read denylisting functionality
denylistKey(String keyspace, String table, String partitionKeyAsString)	Adds a specific keyspace, table, and partition key combo to the denylist
removeDenylistKey(String keyspace, String cf, String partitionKeyAsString)	Removes a specific keyspace, table, and partition key combo from the denylist
setDenylistMaxKeysPerTable(int value)	Limits count of allowed keys per table in the denylist
setDenylistMaxKeysTotal(int value)	Limits the total count of allowable denylisted keys in the system
isKeyDenylisted(String keyspace, String table, String partitionKeyAsString)	Indicates whether the keyspace.table has the input partition key denied