Manual Backup and Restore, with Point-in-time and table-level restore.

Follow

Backing up and restoring data in DataStax Enterprise

Managing backups and testing the restoration of them is an essential practice for data storage operations teams, and DSE is no exception to this rule. DSE is equipped for backing up and restoring part, or all, of the body of data stored in a cluster, to a fixed point in time, down to a transactional level of restoration. This article is intended as a useful reference for general knowledge, and steps required, to take backups, and to restore data to a cluster. Datastax Worldwide Support recommends testing these steps to familiarize teams with them prior to implementing them on a production database.

There are 3 layers of backup granularity available in DSE:

1. Snapshots - a hardlink to the sstable files in the data directory for a schema table, at the moment the snapshot is executed.
2. Incremental backups - a separate hardlink to an sstable, that is created at the same time the original sstable is written to disk in the data directory.
3. Commitlog archives - a copy of, or link to, an individual commitlog. The commitlog holds the transactional log of the node it is from, and is required for transactional-level restoration.

Using the 3 layers summarized above, the idea is to take regular snapshots, take incremental backups in between snapshots, and take commitlog archives that capture the running application of mutations (writes) to the node.

The layers are additive, are inclusive, and are redundant. This is to say that the snapshots include any data in the incremental backups, and the incremental backups will be persisted records of the commits in the commitlog. So,

conceptually, when an incremental backup is taken, you could discard any earlier archived commitlogs, and when a snapshot is taken, you could discard any earlier incremental backups. Or, you could only save sets of incremental backups, and commitlogs, and get the same net result as with using snapshots as a base. But - the aim here is solely to illustrate the overlap of the backups, and not to recommend retention policy guidelines or data handling policy.

To get a good grasp on this, we can consider how Cassandra writes and stores data, in the context of the backups:

- Once a commitlog flushes, it makes an incremental backup, at the same time it makes the live sstable, so the original log segment is not needed any more - the state of the data is persisted in the sstables, which you have a live copy of, and a backup copy of, that are actually hard-links of the same data on-disk. The commitlog archive will be where the transactional application of data, the mutation, is stored, while the live commitlogs will be rolled over per configuration, as they would normally, regardless of backups being taken or not.
- Once the snapshot is taken, all data that is in the incremental backups will also be in the snapshot, so the incremental backups can be cleaned up. The sstables going into the live data directory will be the same files as the backups, and so, will be located in the subfolder holding the snapshot, while the live directory will continue to compact through sstable generations as they would normally.

Then, at restore time, you would:

- load the most recent snapshot you have prior to the desired restore time;
- also load any incremental backups from **after** the snapshot, into the live data directories for the keyspace or table that's being restored;
- finally, load the commitlog archives from after the latest point in time covered by the sstables, into the live commitlog location, where they will be replayed on node startup. This will fill the gap in time from the persistence layer, the sstables, to get the data on that node back to the same state it was in, at the point in time requested.

Important note: A point-in-time restore *requires* a cluster restart for the commitlog replay to run.

In addition to the note above, there are some things it's helpful to be clear on about these operations:

- Snapshots are your primary backup vehicle. A snapshot will contain all the data stored on disk at the time it's taken. Once the snapshot is taken it can

be moved to another location and DSE won't care.

- It's important that snapshots from different nodes do not get stored in the same folder. The sstables aren't written with unique filenames per node, meaning that they will overwrite one another if two or more files of the same filename are saved in the same location.
- Incremental backups will take copies of sstables as they are created, and can also be moved to other locations once they exist. The same rules about unique filenames applies to them.
- Incremental backups are useful to add to the collection of files included in a snapshot, to get you closer to the point in time of the restore - they make up the gap between the last snapshot you took and the very last sstable that was flushed for that keyspace.
- The commitlog will hold a replay of any transactions that were in memory when the node stopped. The commitlog replay is the most expensive part of restoring, but is how you get to point-in-time restores with millisecond precision.
- Commitlogs from one node can't be played on a different node or cluster. They are transactions specific to the node they came from.
- Commitlogs from different nodes should also be stored separately from one another. They will overwrite in the same way sstables will, and transactions would be lost.

Taking snapshots

Using the `nodetool snapshot` command on a node will result in a new folder being created in the `../snapshots/` folder in any live data directories on that node for any tables in the snapshot. The new folder will be named with Unix timestamp by default, but can be tagged as an option to the command. The tag is useful for organizing the snapshots.

Inside that folder, hardlinks to the sstable files will be created. So, this snapshot subfolder will contain a complete image of the persisted state of the data, at the point in time of the snapshot.

If you want to get the complete state of the data persisted before the snapshot is taken, you can run `nodetool flush` to force a flush on one or more tables, emptying the commitlogs and making any data in them into sstables. These folders or the sstables in them can then be copied to remote storage easily using your favorite copy tool such as `rsync` or `scp`.

As mentioned previously, make sure that snapshots from different nodes are stored separately. A structure in remote storage with a parent folder for each

node, with branches for keyspaces and tables going downward from there, in each node's folder in the remote directory, is a good way to keep them separate.

Taking incremental backups

Incremental backups can be turned on in Cassandra. Enabling it means whenever a commitlog is flushed, through normal operations or through a command, which results in a new sstable in the live data directory for a table, a duplicate of that sstable will be created in the `../<keyspace>/backups/` folder for each keyspace. You must maintain and clean up these backups, since the database doesn't know about them, just like snapshots.

The incremental backups and the snapshots are both hardlinks to the live data sstables, so, once you have a snapshot of a keyspace, you can clear the contents of the incremental backups folder for that keyspace - all the files in it will be in the keyspace, and it might also have older files if you don't clean it out regularly.

You really only should retain the incrementals until your next snapshot is taken, in other words. Again, this is not a retention guideline recommendation, only a practical note about the redundancy of the backup layers.

Commitlog archives

This is managed by the `commitlog_archiving.properties` file. When you set the file up for archiving, you should set the `archive_command` - this can either run a command from the OS directly, or invoke a shell and run a script.. In place of the `%path` variable, it will echo out the absolute path of the commitlog file it's archiving, so, if it's about to archive `/var/lib/cassandra/commitlog/CommitLog-4- 1471286005846.log` then that's what the archive operation will echo out. In place of the `%name` variable, it will just echo the file name, so, `CommitLog-4- 1471286005846.log` for example.

There is an example in the `commitlog_archiving.properties` file of the `archive_command` being set to `/bin/ln %path /backup/%name`. Continuing with the same file from above, this would result in the literal command `/bin/ln/ /var/lib/cassandra/commitlog/CommitLog-4-1471286005846.log /backup/CommitLog-4- 1471286005846.log` being run when the `archive_command` executes. The commitlog is archived at node startup, when a log is written to disk by the system, and when you force a flush.

The command set here should be considered carefully. The `ln` command can't work across storage devices, for example. What OpsCenter does on your behalf when you use OpsCenter to enable commitlog archiving, is to instruct all agents to edit the `commitlog_archiving.properties` file for you, and sets the `archive_command` to `/usr/share/datastax-agent/bin/archive_commitlog.sh %path %name`, and `archive_commitlog.sh` contains `cp $1 /var/lib/cassandra/commitlog/backups/$2`.

Continuing with the same example, this makes the node call the `archive_commitlog.sh` with the arguments `/var/lib/cassandra/commitlog/CommitLog-4- 1471286005846.log` and `CommitLog-4- 1471286005846.log`, so then the system runs the `cp` command with them, which is means it runs `cp /var/lib/cassandra/commitlog/CommitLog-4- 1471286005846.log /var/lib/cassandra/commitlog/backups/CommitLog-4-1471286005846.log` in our example. Once a new commitlog is written, the command is executed against it, and so on, indefinitely.

This isn't a good location for the backups to stay, since it's right alongside the live directory. If you need to update it, and you'd like to use the same file, you can, by just pointing to it in the `archive_command` parameter in the `commitlog_archiving.properties` file. To change the location it makes the copy to, just change the filepath in `cp $1 /var/lib/cassandra/commitlog/backups/$2` from `/var/lib/cassandra/commitlog/backups/` to whatever you want.

Once you make changes to the `commitlog_archiving.properties` you need to restart the node. You'll see OpsCenter run a `drain` and restart when you turn the commitlog archiving on in the UI. This is because it made edits to this file and needs to restart the nodes for the changes to be applied.

So, restoring to a point in time, and filtering to a keyspace or table, can be fairly easy, or not practical at all, depending on your execution of archiving, and understanding the stages of the restore process.

Restoring snapshots

Restoring from a snapshot is pretty straightforward. When a node starts, it reads all the sstables in its data directories, and begins serving requests based on the token ranges it owns. It won't care much about the sstables, beyond making sure it is an sstable that matches the data directory subfolder it's in, so it knows it's for that table. If the name matches the table, the files will become the initial

loading point for the data set. For responses to be returned, though, the tokenized keys in the sstable must in a range the node owns.

To restore snapshots, there are 3 ways. They all assume the data directory has been emptied, either with the truncate command, or by manually deleting the files when the node is down.

You can:

- load any sstables from a snapshot into the data directory, and start/restart the node. The sstables will be validated and loaded on startup, regardless of the source. The node loading them has no way of knowing anything about the source of the sstables.
- with an empty data directory you can load a snapshot into it, and run nodetool refresh to force the sstables to be read into the node.
- you can run the `sstableloader` to stream sstables in a folder to a target cluster. The `sstableloader` will read any directory containing sstables that it's passed as an argument, then contact the destination ring, determine the node token ranges, and stream sstables to the correct nodes. It's a way to bypass complexity of managing tokens when loading a data set.

Snapshots are a view of the data set on that node at that point in time. So, a snapshot taken at 10 AM can be used to restore the node to the state it was in at 10 AM that day.

If you take a snapshot every day at 10 AM, you know you can restore to 10 AM without a problem. But what if you need to restore to 10:30 AM? You need the changes that were applied to the data set in the time that has passed. There are two ways to acquire those changes and have the state of the data changed: incremental backups, and commitlog archives.

Restoring incremental backups

Actually, restoring incremental backups is no different from restoring snapshots. They're both just sstables, and whatever is in the data directory will be validated and read when the node restarts, when you run `nodetool refresh <keyspace> <table>`, or the sstables can be streamed with the sstableloader.

Since an incremental backup is a copy of an sstable that is created as they are flushed, and a snapshot is a collection of the sstables making up a data set at

the point in time the snapshot was taken. you can apply them together to get the data set rolled forward closer to the point in time you want to restore to.

This means you can take a snapshot at 10 AM, have incremental backups generated every time a table flushes and generates an sstable, and then combine the two sets of file to move closer to a point you want to recover to. The rate at which the incremental backups are created depend on several factors but in a busy database, you will probably see them generated every couple of seconds at least. This gets reasonably fine granularity for restoring. If you need transactional levels of restoration, you need to replay the transactions for the nodes, on a per-node basis. To do this, you use the commitlog archives.

Restoring commitlogs

Restoring commitlogs is the process of clearing the live commitlog directory, loading one or more commitlog files from the backup location you put it in with the `archive_command:` or taking any other copy of the commitlogs you might have made, and restarting the node.

During node startup, the normal behavior of a node with regards to commitlogs is to check the `commitlog_archiving.properties` for instructions, load the appropriate tooling for archiving or apply any restore instructions, and then replay whatever is in the commitlog as transactions against the data set that is in the sstables in the live data directory.

If you have filled out the restore section of the `commitlog_archiving.properties` and start the node, the node will follow those instructions before moving on to the commitlog replay.

These are very similar to the `archive_command:`, in that the command passes variables, `%from` and `%to`, echoing them into whatever command or script is pointed to in `restore_command:`.

The `%from` variable will be whatever you put in the `restore_directories:` in the properties file. Note that the restore directory does not have to be related to the archive directory at all. The `%to` variable will be set to the live commitlog directory automatically and cannot be changed.

So, by executing a command to copy everything from the/a backup location to the live commitlog location, you can have various archived commitlogs replayed.

Using this, along with the snapshots and incremental sstables, you can restore with millisecond precision, at a transaction level. You do this by loading the sstables from your snapshot and any incremental backups between the snapshot

and the restore window, then setting the point in time you want the commitlogs replayed up until. The mutations in the commitlog will be played against the data, bringing the node up to the timestamp indicated in the `restore_point_in_time:` parameter, which should be set as a string in YYYY:MM:DD HH:MM:SS format, and in GMT.

So, with a snapshot at 10 AM, incremental backups being created every ~30 seconds due to natural flushing in the write path, and commitlog archives from 10 AM to 10:30 AM, we can get to 10:30 AM with millisecond precision. The process would be, ideally, to load the snapshot and the incremental backups up until the last incremental before 10:30, so, 10:29:30 for example, and then the commitlogs covering the last 30 seconds at least, but maybe a couple of minutes on either side would be good, and restart the node. The node will load the data from the sstables when it starts, then scan the `commitlog_archiving.properties`, and would end up with any commitlogs that are in the `restore_directories:` being placed in the live commitlog directory, and replayed, until the `restore_point_in_time:` in epoch time is reached, or until the commitlogs are all replayed if it is not set.

Restoring Specific Schema Tables

If you want to filter the tables that the replay is applied to, you can, to achieve point-in-time recovery on a subset of tables. This is accomplished by setting the following line in the `dse-env.sh` or in the `cassandra-env.sh` file before the next start of the node: `JVM_OPTS="$JVM_OPTS -Dcassandra.replayList=<keyspace>.<table>, <keyspace>.<table>, ..."` for the list of tables you want the replay applied to. Be sure to remove it once the node is restored, or else only the tables specified will have their commitlogs replayed on the next startup.

If you are not running DSE as a service, you can pass the option in on the command line when you start DSE.

This is discussed on CASSANDRA-4809.

Some Helpful Notes for Planning

* Keep in mind that the node will replay all commitlogs in the directory, and that the commitlog replay is a fairly expensive process during the startup. The transactions are all played back by timestamp, and should be idempotent, so

applying them twice should not be a problem. But, because it takes a while, and the node won't be "UP" until it's done, you want to avoid playing back much more than you need to. For example, don't load 2 days of commitlog archives when you only need to restore the last 20 minutes. You can place only the commitlogs you want in the `restore_directories:` if you only want a recent subset to be restored.

* Commitlogs will only replay for the node they are created on. They are transactions unique to the node and won't work on other nodes. So, the changes to the `commitlog_archiving.properties` would need to be made on all nodes for a restore to be successful.

* Depending on requirements, you may not need the precision that comes from the commitlog restore step. The incremental backups of sstables are created quite rapidly in most cases, so, depending on how close you need the restore point to be to the specified point in time, you may be able to reduce the complexity of restoring somewhat.

Devin Suiter

Created: 2017-03-27 19:43:35 UTC
Last Update: 2019-12-05 20:49:54 UTC

Was this article helpful?

✓ Yes ✕ No

10 out of 10 found this helpful

Have more questions? Submit a request

Return to top

Recently viewed articles

Switch to subrange repair for Opscenter tables: settings and backup_reports

How to performance tune data streaming activities like repair and bootstrap

Related articles

Handling schema disagreements and "Schema version mismatch detected" on node restart

DSE startup fails with "Failed to dispatch hints file", "file is corrupted"

What does "prepared statements discarded in the last minute because cache limit reached" mean?

Configuring Logging in Apache Cassandra

HOW TO - Mirror DSE packages internally

Comments

0 comments

Please sign in to leave a comment.