Chapter 50.

February 2021

Welcome to the February 2021 edition of DataStax Developer's Notebook (DDN). This month we answer the following question(s);

> My company is moving its operations to the cloud, including cloud native computing and Kubernetes. I believe we can run Apache Cassandra on Kubernetes. Can you help ?
>
> *Excellent question ! This is the second of four articles relative to running Apache Cassandra atop Kubernetes. Where the first article (last month, January 2021) was an (up and running, a primer), this article discusses Cassandra node failure. The third and fourth articles discuss Cassandra cluster cloning, and Kubernetes snapshots in general.*

Software versions

The primary DataStax software component used in this edition of DDN is DataStax Enterprise (DSE), currently release 6.8.*, DataStax Astra (Apache Cassandra version 4.0.0.*), or Kubernetes 1.17/1.18, as required. All of the steps outlined below can be run on one laptop with 16 GB of RAM, or if you prefer, run these steps on Amazon Web Services (AWS), Microsoft Azure, or similar, to allow yourself a bit more resource.

For isolation and (simplicity), we develop and test all systems inside virtual machines using a hypervisor (Oracle Virtual Box, VMWare Fusion version 8.5, or similar). The guest operating system we use is Ubuntu Desktop version 18.04, 64 bit. Or, we're running on one of the major cloud providers on Kubernetes 1.18.

50.1 Terms and core concepts

As stated above, this is the second in a series of four articles, each detailing some aspect of running Apache Cassandra (Cassandra) atop Kubernetes. Last month, the first article in this series (January 2021), got the Apache Cassandra distributed database server up and running. This month we detail recovery from Cassandra node failure.

Comments include:

– The top level object in Cassandra world is the cluster, a collection of nodes, all of which contain the end user data proper.

– Between the Cassandra cluster and Cassandra nodes is a logical construct titled, a keyspace. A Cassandra keyspace is a collection of nodes; all nodes or a subset of same.

  Relevant to our discussion of failure, a keyspace specifies two pieces of metadata-

  • A replication factor (RF)-

    That is, how many nodes will contain a copy of any piece of any data placed in that keyspace. Given a keyspace with 5 nodes, and an RF=3, this means that any single row of data is written, redundantly, on every 3 out of 5 nodes.

    Lose one of 5 nodes, and you still have 2 writable copies of that piece of data.

Note: Isn't RF=3 too redundant; wouldn't RF=2 suffice ?

No.

We want 3 copies of everything in the event you take a node down for maintenance. This condition would still leave you with 2 copies, and no single point of failure.

  • A replication strategy (RS), effectively SimpleStrategy or NetworkTopologyStrategy-

    SimpleStrategy places all nodes in the Cassandra cluster in one default pool. NetworkTopologyStrategy further allows for geographic placement of data; data localization. E.g., German country data must reside in Germany.

    A single Cassandra cluster can have both replication strategies in place at the same time, just on different keyspaces.

- If a node fails in Cassandra, its contents can be fully recovered, online and in full multi-user mode, by copying data from any of the 2 copies of this data; data on other nodes because of the replication factor.

  This work happens automatically and without fail.

  Further; a recent Cassandra feature titled 'zero copy streaming', makes this type of recovery very fast. (Fast, but you still have to copy 2TB of data over the wire. 2TB ? Whatever amount of data is on each node.)

The following is also held true:

- There are many directories specified in the setup of a Cassandra cluster. The most relevant to this discussion are the, (data file directories: DFDs). This single or set of directories are where the end user data proper is written to.

  While there are other directories, this/these are the only we need consider for this discussion.

- If you copy all files from the DFDs, you basically have all that is required to replace a node.

  While zero copy streaming is fast, it is not as fast as already having this data copied, and in place on a net new (replacement) node.

Note: Folks have been copying and then using the data file directories (DFDs) for a host of reasons, and for a long time, largely to accelerate QA and development efforts-

- Cassandra cluster cloning-

  If you match the entire Cassandra cluster network topology, you can copy (and then restore) each node's DFDs to new nodes and new (Cassandra clusters).

  With this technique, you can easily replicate entire Cassandra clusters **pre-populated with data** for faster application testing.

- Differentiated data-

  Same general techniques as above; you can run significant unit and system test that write-to/change the data within Cassandra, and restore from those changes by manually managing the contents under the DFDs.

Cassandra node failure under Kubernetes

In the previous edition of the document, we detailed the following:

- A Kubernetes cluster contains one or more 'worker nodes', where we can create 'pods', the smallest unit of execution when using Kubernetes.

- Pods contain one or more containers.
- There is a one to one relationship between a pod and a Cassandra node. Cassandra (pods) have two containers; one with Cassandra proper, and a second (sidecar patterned) container which maintains the Cassandra message log file (a node-local ASCII text file reporting Cassandra server events).
- With Kubernetes, pods are ephemeral, that is; they die and go away, and any contents inside the pod are lost forever. This would be bad for a Kubernetes hosted database application.

  But, Kubernetes has a plan.
- The DataStax Cassandra Kubernetes Operator is a Kubernetes CRD (custom resource definition file), that we provision. While provisioned (a verb), the operator exists as a Kubernetes 'deployment'.

  To see the Cassandra operator (and all deployed operators in the Kubernetes cluster), run a,

```
#kubectl get deployments -n cass-operator
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
cass-operator   1/1     1            1           111s
```

Note: If you run the (get deployments) without the (-n cass-operator) namespace modifier, you wont see the Cassandra operator.

To see all deployments in all namespaces, drop the (-n cass-operator), and replace it with a (-A), for all namespaces.

Generally (across all Kubernetes object types), you can replace the get verb with a 'describe', to get much more detail.

If you replace 'get' with 'explain', you get more of the object model (metadata model) for said (object).

- But where's my Cassandra cluster; isn't that deployed ?

  Because Cassandra clusters desire non-ephemeral disks, Cassandra is (deployed: we're overloading that term), as a Kubernetes 'stateful set'. Kubernetes has 'replica set's and other similar (objects), but databases, in general, are (deployed) as stateful sets.
- To see your Cassandra cluster (deployment), run a,

```
#kubectl get statefulsets -n cass-operator
NAME                            READY   AGE
```

```
cluster1-system1-default-sts    3/3     10m
```

And again, you can run a 'describe' or 'explain' instead of get, to get more detail.

Note: There are dozens of reports you can run to get the (state, other) of your Cassandra cluster running inside a Kubernetes cluster.

At the moment, we are giving focus to the object type that Cassandra exists as; a stateful set.

Why are we doing this ?

Because Kubernetes stateful sets are largely differentiated from other Kubernetes (objects) by their need and use of non-ephemeral disks, in the form of 'persistent volumes' (PVs), and persistent volume claims' (PVCs).

We were talking about recovery from node failure

A Kubernetes cluster has one or more 'worker nodes', which is where our applications reside and operate. To further divide/organize these Kubernetes nodes, we have Kubernetes pods and containers. A Cassandra node exists, within Kubernetes, as a pod with 2 containers. The 2 containers are; the Cassandra node as you would expect, and a second container with just the Cassandra message log file.

But recall, Kubernetes pods expect ephemeral disk. When a Kubernetes pod desires non-ephemeral disk (persistent disk), it requests and makes use of a Kubernetes 'persistent volume'.

Note: There are many types of uses of Kubernetes volumes, and persistent volumes are just one type.

While there are many modifiers to persistent volumes, the following is held true (by default) for Cassandra pods and their persistent volumes:

- A 'persistent volume' is like the filesystem (device, storage) proper. Kubernetes supports having multiple pods access the same persistent volume concurrently.

  Cassandra does not use volumes in this manner; it doesn't need to.
- When a container (attaches/mounts, our words for it) to a persistent volume (PV), it does so via a persistent volume claim (PVC).
- When a pod fails, a new replacement pod will be created. This new pod can mount the persistent volume that was in previously use by the now failed pod.

Note: This works because persistent volumes are managed/exist outside of the pod.

You could think of this as 'network attached storage', versus 'locally attached storage'.

Wait; isn't network attached storage labeled as an anti-pattern on the open source Cassandra version 2.x and 3.x documentation pages ?

> Yes, it is. But, that's only because locally attached storage is expected to be faster than network attached storage, and Cassandra is all about speed.

Network attached storage may or may not be slower, hopefully, the performance difference is negligible. Given the very highly productive and resilient nature of Kubernetes, hopefully any possible performance differences are offset by the many, other benefits to using Kubernetes.

Note: But are Kubernetes persistent volumes actually NFS mounted drives ?

They could be.

What the underlying storage actually is, is determined by the Kubernetes storage class, in place for any persistent volumes created.

It could be NFS. It could be iSCSI. The storage could be dozens of choices; some cheaper, some faster.

Proving automatic pod recovery of stateful sets

It took umpteen pages to get to this point, and the ability to state; on Kubernetes, Cassandra pods make use of external storage, that is automatically (re-used) when a given pod fails and is replaced.

But how would you prove this ?

The procedure for this is detailed in the next section of this document, which we use as a vehicle to learn more about Kubernetes.

50.2 Complete the following

At this point in this document we are going to test automatic recovery of Cassandra persistent volumes (data) across Kubernetes pod failures. To save time, we rely on procedures documented in the first article in this series of

articles. (This document you are reading now/here, is the second of four articles in this series.) The following then, is assumed:

- You ran file 31 from the toolkit, and made a Kubernetes cluster.
- You ran file 34 and provisioned the DataStax Cassandra Kubernetes Operator.
- You ran file 40, passing an argument (or selecting within the UI), a YAML file titled D1, to make a one node Cassandra cluster.
- Run the file 40 again, and use YAML file D4, which moves the previously existing 1 node Cassandra cluster to a 3 node Cassandra cluster.
- You are ready to proceed when a

  ```
  #kubectl get statefulsets -n cass-operator
  ```

  outputs,

  ```
  NAME                            READY   AGE
  cluster1-system1-default-sts    3/3     56m
  ```

50.2.1 Make some data in the Cassandra cluster for tracing

There are several programs in the toolkit that allow you to run CQLSH;

- File 61 runs Bash through each of the Cassandra pods in turn.

  We wont use this program at this time, but know it's there.

  CQLSH is installed on each of the Cassandra pods, so we could access CQLSH through this means.
- File 63 runs CQLSH, from one of the Cassandra pods.

  We wont use this program either, but know it's there.
- We will run file 64, which runs the CQLSH that is local on our workstation (our local MAC, Linux, other), into the Kubernetes cluster hosting our Cassandra cluster.

  Why run this program, and not the other options ?

  Because the toolkit includes a number of (local) CQL command files, and by using file 64, we can pass those files to our local CQLSH.

Run the following,

```
64* T1*
```

The above will make a keyspace, table, and 10 rows of data in the Cassandra cluster. CQLSH will exit when the above CQL script is complete.

Confirm that the data is now present by running CQLSH again, using file 63 or 64. You are looking for 10 rows in the table titled, ms_cluster1.t1.

Note: Notice that the keyspaces is created with SimpleStrategy, and replication factor (RF=1).

With an RF=1; if we destroy a node, and its data comes back, then the data had to exist outside of the node, and was magically recovered somehow.

Note: File/program 64 is pretty cool-

This program uses the (port forwarding) feature of kubectl.

In effect, the remote Kubernetes/Cassandra system's necessary ports are accessible through our laptop's localhost.

50.2.2 Flush the node, kill the node

By default, Apache Cassandra flushes data in volatile RAM to disk every 10 seconds. So, we shouldn't have to manually flush, but why not-

3 Cassandra nodes (pods), let's flush node 3, then kill node 3:

- You already ran a,

  64* T1*
- Now run a,

  68*

  This will run 'nodetool flush' on each Cassandra node, in turn.
- Kill a Cassandra node, or the last node-

  65* # kills the last C* node in the list

  66* # allows you to choose which node to kill
- Watch the node get replaced with a,

  ```
  watch kubectl -n cass-operator get pods
  ```

  CONTROL-C to get out of watch, when the replacement pod is up.
- Run CQLSH, to prove that your data is present.

Cool. Automatic. Fast. Free. (Built-in.)

Note: Want to learn more ?

Don't forget to look inside each of the program we ran in this exercise; see what each program is doing.

50.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- We proved automatic Cassandra node recovery without needing to copy any data.
- Along the way, we learned how to; run CQLSH, nodetool, and more.

Persons who help this month.

Kiyu Gabriel, Joshua Norrid, Dave Bechberger, and Jim Hatcher.

Additional resources:

Free DataStax Enterprise training courses,

```
https://academy.datastax.com/courses/
```

Take any class, any time, for free. If you complete every class on DataStax Academy, you will actually have achieved a pretty good mastery of DataStax Enterprise, Apache Spark, Apache Solr, Apache TinkerPop, and even some programming.

This document is located here,

```
https://github.com/farrell0/DataStax-Developers-Notebook
```

```
https://tinyurl.com/ddn3000
```


