Cassandra „Hello World“ Example

DEZEMBER 8, 2016 BY OVEITS — LEAVE A COMMENT

Today, we will introduce Cassandra, a distributed and resilient, highly scalable noSQL database. For simplicity, we will run a cluster within Docker containers. We will test the resiliency functions by killing one of two containers and verifying that all data is retained.

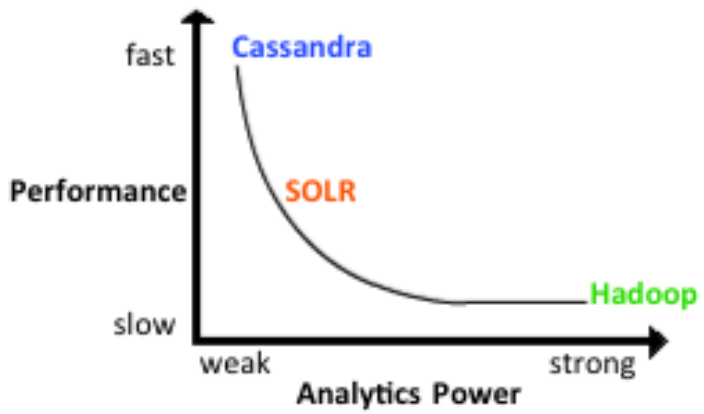
Contents [hide]

What is Cassandra?

Apache Cassandra is a fast, distributed noSQL database that can be used for big data use cases. A short comparison of Apache Cassandra with Apache Hadoop can found in this Cassandra vs Hadoop blog post:

- Hadoop is a big data framework for storing and analyzing a vast amount of unstructured, historic data. Why ‚historic'? The reason is that the search capabilities of Hadoop rely on long-running, CPU-intensive MapReduce processes that are running as batch processes.
- Cassandra is a distributed noSQL database for structured data, and is ideally suited for structured, „hot" data, i.e. Cassandra is capable of processing online workloads of a transactional nature.

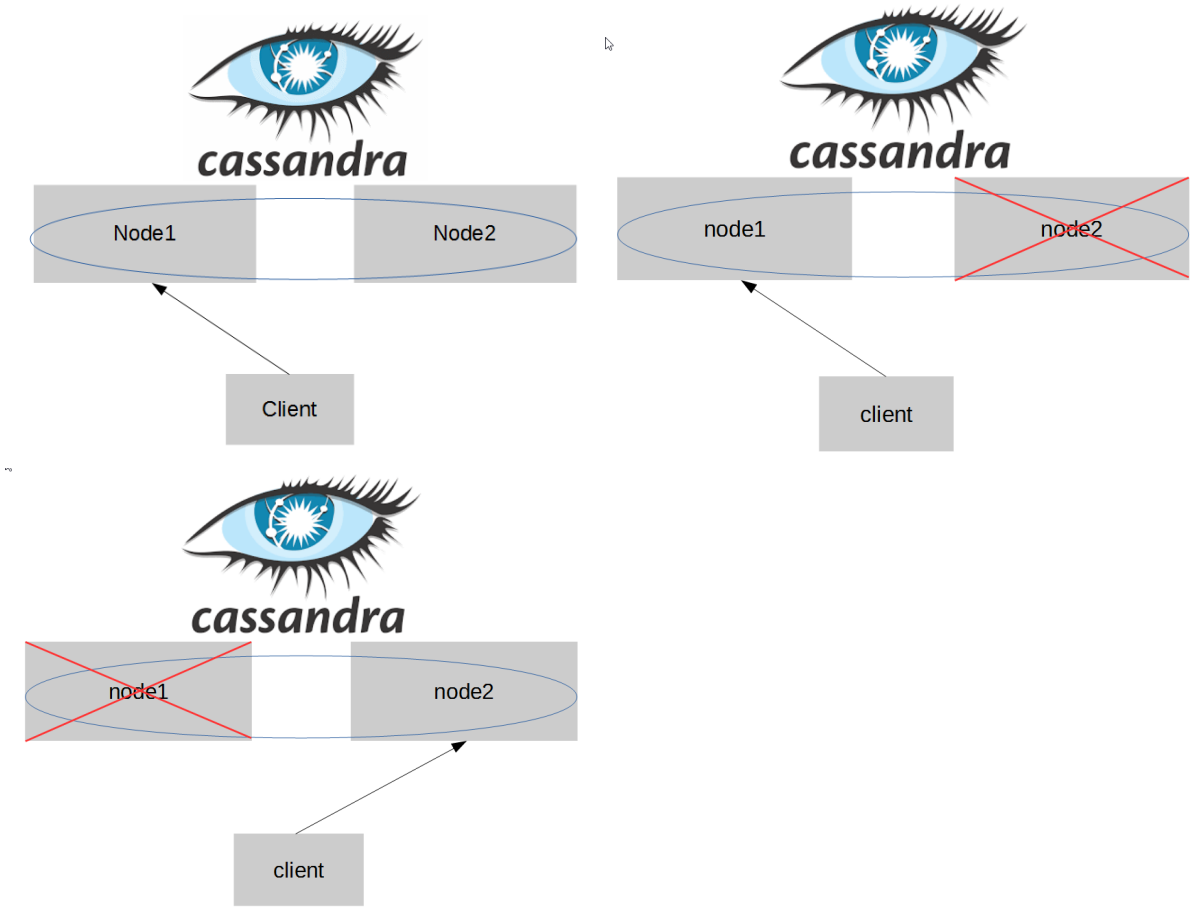
I have found following figure that compares Cassandra with SOLR/Lucene and Apache Hadoop:



Source:
https://docs.datastax.com/en/datastax_enterprise/4.5/datastax_enterprise/srch/srchIntro.html

Target Configuration for this Blog Post

In this Hello World blog post, we will create two Cassandra server containers and a Cassandra client container. For the sake of this test, we store the Cassandra databases within the containers (in a productive environment, you would most likely store the database outside the container). We will add data to the cluster and make sure the data is replicated to both servers. We can test this by shutting down one server container first, starting a new server container to restore the redundancy, shutting down a second container and make sure, that the data is still available.



Tools used

- Vagrant 1.8.6
- Virtualbox 5.0.20
- Docker 1.12.1
- Cassandra 3.9

Prerequisites:

- > 3.3 GB DRAM (Docker host: ~0.3 GB, ~1.5 GB per Cassandra node, < ~0.1 GB for Cassandra client)

Step 1: Install a Docker Host via Vagrant and Connect to the Host via SSH

If you are using an existing docker host, make sure that your host has enough memory and your own Docker ho

We will run Cassandra in Docker containers in order to allow for maximum interoperability. This way, we always can use the latest Logstash version without the need to control the java version used: e.g. Logstash v 1.4.x works with java 7, while version 5.0.x works with java 8 only, currently.

If you are new to Docker, you might want to read this blog post.

Installing Docker on Windows and Mac can be a real challenge, but no worries: we will show an easy way here, that is much quicker than the one described in Docker's official documentation:

Prerequisites of this step:

- I recommend having direct access to the Internet: via your Firewall, but without any HTTP proxy. However, if you cannot get rid of your HTTP proxy, read this blog post.
- Administration rights on you computer.

Steps to install a Docker Host VirtualBox VM:

Download and install Virtualbox (if the installation fails with error message „<to be completed> see Appendix A of this blog post: Virtualbox Installation Workaround below)

1. Download and Install Vagrant (requires a reboot)

2. Download Vagrant Box containing an Ubuntu-based Docker Host and create a VirtualBox VM like follows:

```
basesystem# mkdir ubuntu-trusty64-docker ; cd ubuntu-trusty64-docker
basesystem# vagrant init williamyeh/ubuntu-trusty64-docker
basesystem# vagrant up
basesystem# vagrant ssh
```

Now you are logged into the Docker host and we are ready for the next step: to create the Ansible Docker image.

> *Note: I have experienced problems with the vi editor when running `vagrant ssh` in a Windows terminal. In case of Windows, consider to follow Appendix C of this blog post and to use putty instead.*

Step 2 (optional): Download Cassandra Image

This extra download step is optional, since the Cassandra Docker image will be downloaded automatically in step 3, if it is not already found on the system:

```
(dockerhost)$ docker pull cassandra
Using default tag: latest
latest: Pulling from library/cassandra

386a066cd84a: Already exists
e4bd24d76b78: Pull complete
5ccb1c317672: Pull complete
a7ffd548f738: Pull complete
d6f6138be804: Pull complete
756363f453c9: Pull complete
26258521e648: Pull complete
fb207e348163: Pull complete
3f9a7ac16b1d: Pull complete
49e0632fe1f1: Pull complete
ba775b0b41f4: Pull complete
Digest: sha256:f5b1391b457ead432dc05d34797212f038bd9bd4f0b0260d90ce74e53cbe7c
Status: Downloaded newer image for cassandra:latest
```

The version of the downloaded Cassandra image can be checked with following command:

```
(dockerhost)$ sudo docker run -it --rm --name cassandra cassandra -v
3.9
```

We are using version 3.9 currently. If you want to make sure that you use the exact same version as I have used in this blog, you can use the imagename `cassandra:3.9` in all docker commands instead of `cassandra` only.

Step 2: Run Cassandra in interactive Terminal Mode

In this step, we will run Cassandra interactively (with `-it` switch instead of `-d` switch) to better see, what is happening. In a productive environment, you will use the detached mode -d instead of the interactive terminal mode -it.

We have found out by analyzing the Cassandra image via the online imagelayer tool, that the default command is to run `/docker-entrypoint.sh cassandra -f` and that cassandra uses the ports 7000/tcp 7001/tcp 7199/tcp 9042/tcp 9160/tcp. We keep the entrypoint and map the ports to the outside world:

```
(dockerhost)$ sudo docker run -it --rm --name cassandra-node1 -p7000:7000 -p7

INFO  15:30:17 Configuration location: file:/etc/cassandra/cassandra.yaml
INFO  15:30:17 Node configuration:[allocate_tokens_for_keyspace=null; authent
INFO  15:30:17 DiskAccessMode 'auto' determined to be mmap, indexAccessMode i
INFO  15:30:17 Global memtable on-heap threshold is enabled at 251MB
INFO  15:30:17 Global memtable off-heap threshold is enabled at 251MB
WARN  15:30:18 Only 22.856GiB free across all data volumes. Consider adding m
INFO  15:30:18 Hostname: 4ba7699e4fc2
INFO  15:30:18 JVM vendor/version: OpenJDK 64-Bit Server VM/1.8.0_111
INFO  15:30:18 Heap size: 1004.000MiB/1004.000MiB
INFO  15:30:18 Code Cache Non-heap memory: init = 2555904(2496K) used = 39068
INFO  15:30:18 Metaspace Non-heap memory: init = 0(0K) used = 15609080(15243K
INFO  15:30:18 Compressed Class Space Non-heap memory: init = 0(0K) used = 19
INFO  15:30:18 Par Eden Space Heap memory: init = 167772160(163840K) used = 7
INFO  15:30:18 Par Survivor Space Heap memory: init = 20971520(20480K) used =
INFO  15:30:18 CMS Old Gen Heap memory: init = 864026624(843776K) used = 0(0K
INFO  15:30:18 Classpath: /etc/cassandra:/usr/share/cassandra/lib/HdrHistogra
INFO  15:30:18 JVM Arguments: [-Xloggc:/var/log/cassandra/gc.log, -ea, -XX:+U
WARN  15:30:18 Unable to lock JVM memory (ENOMEM). This can result in part of
INFO  15:30:18 jemalloc seems to be preloaded from /usr/lib/x86_64-linux-gnu/
WARN  15:30:18 JMX is not enabled to receive remote connections. Please see c
WARN  15:30:18 OpenJDK is not recommended. Please upgrade to the newest Oracl
INFO  15:30:18 Initializing SIGAR library
WARN  15:30:18 Cassandra server running in degraded mode. Is swap disabled? :
WARN  15:30:18 Directory /var/lib/cassandra/data doesn't exist
WARN  15:30:18 Directory /var/lib/cassandra/commitlog doesn't exist
WARN  15:30:18 Directory /var/lib/cassandra/saved_caches doesn't exist
WARN  15:30:18 Directory /var/lib/cassandra/hints doesn't exist
INFO  15:30:18 Initialized prepared statement caches with 10 MB (native) and
INFO  15:30:19 Initializing system.IndexInfo
INFO  15:30:20 Initializing system.batches
INFO  15:30:20 Initializing system.paxos
INFO  15:30:20 Initializing system.local
INFO  15:30:20 Initializing system.peers
INFO  15:30:20 Initializing system.peer_events
```

```
INFO  15:30:20 Initializing system.range_xfers
INFO  15:30:20 Initializing system.compaction_history
INFO  15:30:20 Initializing system.sstable_activity
INFO  15:30:20 Initializing system.size_estimates
INFO  15:30:20 Initializing system.available_ranges
INFO  15:30:20 Initializing system.views_builds_in_progress
INFO  15:30:20 Initializing system.built_views
INFO  15:30:20 Initializing system.hints
INFO  15:30:20 Initializing system.batchlog
INFO  15:30:20 Initializing system.schema_keyspaces
INFO  15:30:20 Initializing system.schema_columnfamilies
INFO  15:30:20 Initializing system.schema_columns
INFO  15:30:20 Initializing system.schema_triggers
INFO  15:30:20 Initializing system.schema_usertypes
INFO  15:30:20 Initializing system.schema_functions
INFO  15:30:20 Initializing system.schema_aggregates
INFO  15:30:20 Not submitting build tasks for views in keyspace system as sto
INFO  15:30:20 Configured JMX server at: service:jmx:rmi://127.0.0.1/jndi/rmi
INFO  15:30:21 Initializing key cache with capacity of 50 MBs.
INFO  15:30:21 Initializing row cache with capacity of 0 MBs
INFO  15:30:21 Initializing counter cache with capacity of 25 MBs
INFO  15:30:21 Scheduling counter cache save to every 7200 seconds (going to
INFO  15:30:21 Global buffer pool is enabled, when pool is exhausted (max is
INFO  15:30:21 Populating token metadata from system tables
INFO  15:30:21 Token metadata:
INFO  15:30:21 Initializing system_schema.keyspaces
INFO  15:30:21 Initializing system_schema.tables
INFO  15:30:21 Initializing system_schema.columns
INFO  15:30:21 Initializing system_schema.triggers
INFO  15:30:21 Initializing system_schema.dropped_columns
INFO  15:30:21 Initializing system_schema.views
INFO  15:30:21 Initializing system_schema.types
INFO  15:30:21 Initializing system_schema.functions
INFO  15:30:21 Initializing system_schema.aggregates
INFO  15:30:21 Initializing system_schema.indexes
INFO  15:30:21 Not submitting build tasks for views in keyspace system_schema
INFO  15:30:21 Completed loading (5 ms; 1 keys) KeyCache cache
```

```
 INFO  15:30:21 No commitlog files found; skipping replay
 INFO  15:30:21 Populating token metadata from system tables
 INFO  15:30:21 Token metadata:
 INFO  15:30:22 Cassandra version: 3.9
 INFO  15:30:22 Thrift API version: 20.1.0
 INFO  15:30:22 CQL supported versions: 3.4.2 (default: 3.4.2)
 INFO  15:30:22 Initializing index summary manager with a memory pool size of
 INFO  15:30:22 Starting Messaging Service on /172.17.0.4:7000 (eth0)
 WARN  15:30:22 No host ID found, created 1c7f41f6-4513-4949-abc3-0335af298fc8
 INFO  15:30:22 Loading persisted ring state
 INFO  15:30:22 Starting up server gossip
 INFO  15:30:22 This node will not auto bootstrap because it is configured to
 INFO  15:30:22 Generated random tokens. tokens are [295917137465811607, -3021
 INFO  15:30:22 Create new Keyspace: KeyspaceMetadata{name=system_traces, para
 INFO  15:30:22 Not submitting build tasks for views in keyspace system_traces
 INFO  15:30:22 Initializing system_traces.events
 INFO  15:30:22 Initializing system_traces.sessions
 INFO  15:30:22 Create new Keyspace: KeyspaceMetadata{name=system_distributed,
 INFO  15:30:22 Not submitting build tasks for views in keyspace system_distri
 INFO  15:30:22 Initializing system_distributed.parent_repair_history
 INFO  15:30:22 Initializing system_distributed.repair_history
 INFO  15:30:22 Initializing system_distributed.view_build_status
 INFO  15:30:22 Node /172.17.0.4 state jump to NORMAL
 INFO  15:30:22 Create new Keyspace: KeyspaceMetadata{name=system_auth, params
 INFO  15:30:23 Not submitting build tasks for views in keyspace system_auth a
 INFO  15:30:23 Initializing system_auth.resource_role_permissons_index
 INFO  15:30:23 Initializing system_auth.role_members
 INFO  15:30:23 Initializing system_auth.role_permissions
 INFO  15:30:23 Initializing system_auth.roles
 INFO  15:30:23 Waiting for gossip to settle before accepting client requests.
 INFO  15:30:31 No gossip backlog; proceeding
 INFO  15:30:31 Netty using native Epoll event loop
 INFO  15:30:31 Using Netty Version: [netty-buffer=netty-buffer-4.0.39.Final.3
 INFO  15:30:31 Starting listening for CQL clients on /0.0.0.0:9042 (unencrypt
 INFO  15:30:31 Not starting RPC server as requested. Use JMX (StorageService-
 INFO  15:30:33 Scheduling approximate time-check task with a precision of 10
 INFO  15:30:33 Created default superuser role 'cassandra'
```

Step 3: Create a second Cassandra Node

We want to start a second Cassandra container on the same Docker host for simple testing. We will connect to the container running on the first node via IP. For that we need to find out the IP address as follows:

```
(dockerhost)$ sudo docker inspect --format='{{ .NetworkSettings.IPAddress }}'
172.17.0.2e
```

This information can be used in the next command by setting the **CASSANDRA_SEEDS** variable accordingly:

Note also that we have changed the port mapping in order to avoid port conflicts with the first Cassandra node:

```
(dockerhost)$ sudo docker run -it --rm --entrypoint="bash" --name cassandra-n
              -p27000:7000 -p27001:7001 -p29042:9042 -p29160:9160 \
              -e CASSANDRA_SEEDS="$(docker inspect --format='{{ .NetworkSetti
              cassandra
```

Note that we have overridden the default entrypoint, so we get access to the terminal.

We now start Cassandra on the second node:

```
(container):/# /docker-entrypoint.sh cassandra -f
...
INFO 17:37:03 Starting listening for CQL clients on /0.0.0.0:9042 (unencrypte
INFO 17:37:03 Not starting RPC server as requested. Use JMX (StorageService->
INFO 17:37:05 Created default superuser role 'cassandra'
```

On the first Cassandra node, we will see following additional log lines:

```
INFO 17:36:21 Handshaking version with /172.17.0.3
INFO 17:36:21 Handshaking version with /172.17.0.3
INFO 17:36:22 InetAddress /172.17.0.3 is now DOWN
INFO 17:36:22 Handshaking version with /172.17.0.3
INFO 17:36:23 Handshaking version with /172.17.0.3
INFO 17:36:54 [Stream #beb912f0-bca3-11e6-a935-4b019c4b758d ID#0] Creating ne
INFO 17:36:54 [Stream #beb912f0-bca3-11e6-a935-4b019c4b758d, ID#0] Received s
INFO 17:36:54 [Stream #beb912f0-bca3-11e6-a935-4b019c4b758d, ID#0] Received s
INFO 17:36:55 [Stream #beb912f0-bca3-11e6-a935-4b019c4b758d] Session with /17
INFO 17:36:55 [Stream #beb912f0-bca3-11e6-a935-4b019c4b758d] All sessions com
INFO 17:36:55 Node /172.17.0.3 state jump to NORMAL
INFO 17:36:55 InetAddress /172.17.0.3 is now UP
```

Note: if you get following error message:

```
Exception (java.lang.RuntimeException) encountered during startup.
```

you need to start the service using the following line instead:

```
(container):/# /docker-entrypoint.sh cassandra -f -Dcassandra.repl
```

The error will show up, if you have connected a Cassandra node to the cluster, then you destroy the node (by stopping the container) and re-start a new container. The new container will re-claim the now unused IP address of the destroyed container. However, this address is marked as unreachable within the cluster. We would like to re-use the IP address in the cluster, which requires the ***-Dcassandra.replace_address*** *option.*

The term

```
(container):/# ip addr show | grep eth0 | grep -v '@' | awk '{pri
172.17.0.3
```

will return the current IP address of eth0 of the docker container and helps to feed in the correct IP address to the `-Dcassandra.replace_address` *option.*

Step 4: Start a CQL Client Container

Now we want to add some data to the distributed noSQL database. For that, we start a third container that can be used as CQL Client (CQL=Cassandra Query Language similar to SQL). We can start a CQL shell like follows:

```
(dockerhost)$ sudo docker run -it --rm -e CQLSH_HOST=$(docker inspect --forma
Connected to Test Cluster at 172.17.0.2:9042.
[cqlsh 5.0.1 | Cassandra 3.9 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Step 5: Create Keyspace

Now let us create a keyspace. A keyspace is the pendant for a database in SQL databases:

```
cqlsh> create keyspace mykeyspace with replication = {'class':'SimpleStrategy
cqlsh>
```

Upon successful creation, the prompt will be printed without error.

Step 6: Create Table

For adding data, we need to enter the keyspace and

```
cqlsh> use mykeyspace;
cqlsh:mykeyspace> create table usertable (userid int primary key, usergivenna
cqlsh:mykeyspace>
```

Step 7: Add Data

Now we can add our data:

```
cqlsh:mykeyspace> insert into usertable (userid, usergivenname, userfamilynam
cqlsh:mykeyspace>
```

The CQL INSERT command has the same syntax as an SQL INSERT command.

Step 8 (optional): Update Data

We now can update a single column as well:

```
cqlsh:mykeyspace> update usertable set userprofession = 'IT Consultant' where
```

Now let us read the entry:

```
qlsh:mykeyspace> select * from usertable where userid = 1;

 userid | userfamilyname | usergivenname | userprofession
--------+----------------+---------------+----------------
      1 |          Veits |        Oliver |  IT Consultant

(1 rows)
cqlsh:mykeyspace>
```

Step 9 (optional): Query on Data other than the primary Index

In Cassandra, we need to enable data filtering, if we try to retrieve data based on a column that has no index:

```
cqlsh:mykeyspace> select * from usertable where userprofession = 'IT Consulta
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot
```

Since data filtering will cost a lot of performance, we will add a secondary index instead. That helps us running the query without such a performance impact:

```
cqlsh:mykeyspace> create index idx_dept on usertable(userprofession);
```

Now the same query should be successful:

```
cqlsh:mykeyspace> select * from usertable where userprofession = 'IT Consulta
```

```
 userid | userfamilyname | usergivenname | userprofession
--------+----------------+---------------+----------------
      1 |          Veits |        Oliver |  IT Consultant


(1 rows)
```

Yes, perfect.

Step 10: Test Resiliency

In the moment, we have following topology (with all nodes and the client being Docker containers on the same Docker host):

Now, we will test, whether the data is retained, if the Cassandra application on node2 is stopped first. For that we stop the application on node2 by pressing `Ctrl-C` .

On node1 we see:

```
INFO 18:06:50 InetAddress /172.17.0.5 is now DOWN
INFO 18:06:51 Handshaking version with /172.17.0.5
```

On the client we see that the data is still there:

```
cqlsh:mykeyspace> select * from usertable where userprofession = 'IT Consulta
```

```
 userid | userfamilyname | usergivenname | userprofession
--------+----------------+---------------+----------------
      1 |          Veits |        Oliver |  IT Consultant



(1 rows)
```

Now let us start the Cassandra application on node 2 again and wait some time until the nodes are synchronized. On node1 we will get a log similar to:

```
INFO  17:36:35 Handshaking version with /172.17.0.4
INFO  17:38:58 Handshaking version with /172.17.0.4
INFO  17:38:58 Handshaking version with /172.17.0.4
INFO  17:39:00 Node /172.17.0.4 has restarted, now UP
INFO  17:39:00 Node /172.17.0.4 state jump to NORMAL
INFO  17:39:00 InetAddress /172.17.0.4 is now UP
INFO  17:39:00 Updating topology for /172.17.0.4
INFO  17:39:00 Updating topology for /172.17.0.4
```

Now we can stop Cassandra on node1 by pressing `Ctrl-C` on terminal 1. On node2, we will get a message similar to:

```
INFO  17:41:32 InetAddress /172.17.0.3 is now DOWN
INFO  17:41:32 Handshaking version with /172.17.0.3
```

At the same time, the node1 container is destroyed, since we have not changed the entrypoint for node1 and we have given the `--rm` option in the `docker run` command in step 2.

Now, we verify that the data is still retained:

```
cqlsh:mykeyspace> select * from usertable where userprofession = 'IT Consulta
NoHostAvailable:
```

Oh, yes, that is clear: we have used node1's IP address and port, when we have started the client.

Let us now connect to node2 by entering „ `exit"` and starting a new client container like follows:

```
(dockerhost)$ sudo docker run -it --rm -e CQLSH_HOST=$(docker inspect --forma
Connected to Test Cluster at 172.17.0.4:9042.
[cqlsh 5.0.1 | Cassandra 3.9 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
cqlsh>
```

To be honest, I was a little bit confused here and would have expected that I need to connect to port 29042 instead, since I have started node2 with a port mapping from 29042 (outside port) to 9042 (container port). But this is wrong: from the Docker host, we can directly access the node2 container IP address with all its ports, including port 9042. Only, if we want to access container from outside the Docker host, we need to access port 29042 of the Docker host IP address instead of port 9042 of the node2 container:

```
(dockerhost)$ netstat -an | grep 9042
tcp6       0      0 :::29042                :::*                    LISTEN
```

Now let us check on the second node that the data is retained:

Anyway, we are connected to the second node now and can check the data:

```
cqlsh> use mykeyspace;
cqlsh:mykeyspace> select * from usertable where userid = 1;


 userid | userfamilyname | usergivenname | userprofession
--------+----------------+---------------+----------------
      1 |          Veits |        Oliver |  IT Consultant


(1 rows)
```

Perfect! The data is still there.

Appendix A: No keyspace has been specified.

If you get an error message like follows,

```
cqlsh> select * from usertable where userprofession = 'IT Consultant';
InvalidRequest: Error from server: code=2200 [Invalid query] message="No keys
```

Then you have forgotten to prepend the „use mykeyspace; command:

```
cqlsh> use mykeyspace;
cqlsh:mykeyspace> select * from usertable where userid = 1;


 userid | userfamilyname | usergivenname | userprofession
--------+----------------+---------------+----------------
      1 |          Veits |        Oliver |  IT Consultant



(1 rows)
```

Summary

In this blog post we have performed the following tasks:

1. Introduced Cassandra with a little comparison with Hadoop
2. We have started a Cassandra node in a Docker container
3. Spun up a second Cassandra node and build a Cassandra cluster
4. Started a Cassandra Client in a container
5. Added Data with replication factor 2 and performed some CQL commands for a warm-up
6. shut down node 2 and verified that the data is still available
7. started node 2 again and wait some seconds
8. shut down node1 and verified that the data is still available on node2

With this test, we could verify that the data replication between nodes in a Cassandra cluster works and no data is lost if a node fails.