Chapter 37.

January 2020

Welcome to the January 2020 edition of DataStax Developer’s Notebook (DDN). This month we answer the following question(s);

> My company maintains a lot of data on Hadoop, in Parquet and other formats, and need to perform integrated reporting with data resident inside DataStax. Can you help ?
>
> *Excellent question ! Yes. This is like a two-liner solution. We’ll detail all of the concepts and code inside this document.*

Software versions

The primary DataStax software component used in this edition of DDN is DataStax Enterprise (DSE), currently release 6.8 EAP (Early Access Program). 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.

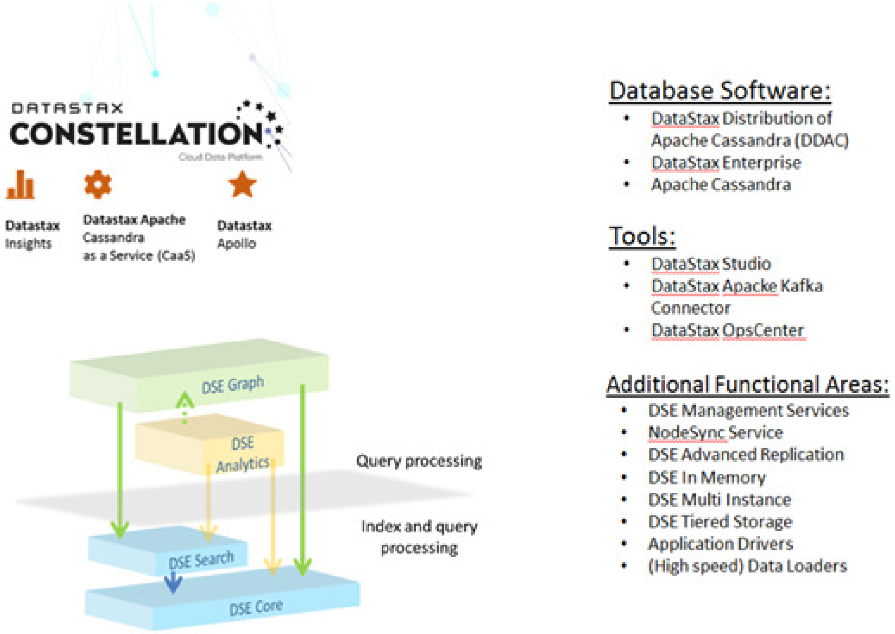
For isolation and (simplicity), we develop and test all systems inside virtual machines using a hypervisor (Oracle Virtual Box, VMWare Fusion version 8.5, or similar). The guest operating system we use is Ubuntu Desktop version 18.04, 64 bit.

37.1 Terms and core concepts

As stated above, ultimately the end goal is to perform reporting on data; some of it resident inside DataStax Enterprise (DSE), some of it resident on Hadoop/HDFS, in Parquet format, and presumably without a lift and shift. (E.g., don't make me have even more copies of the same data.)

Easy peezy.

Recall there are 4 primary functional areas inside DataStax Enterprise (DSE), as displayed in Figure 37-1. A code review follows.



Database Software:

- DataStax Distribution of Apache Cassandra (DDAC)
- DataStax Enterprise
- Apache Cassandra

Tools:

- DataStax Studio
- DataStax Apacke Kafka Connector
- DataStax OpsCenter

Additional Functional Areas:

- DSE Management Services
- NodeSync Service
- DSE Advanced Replication
- DSE In Memory
- DSE Multi Instance
- DSE Tiered Storage
- Application Drivers
- (High speed) Data Loaders

Figure 37-1 DSE product

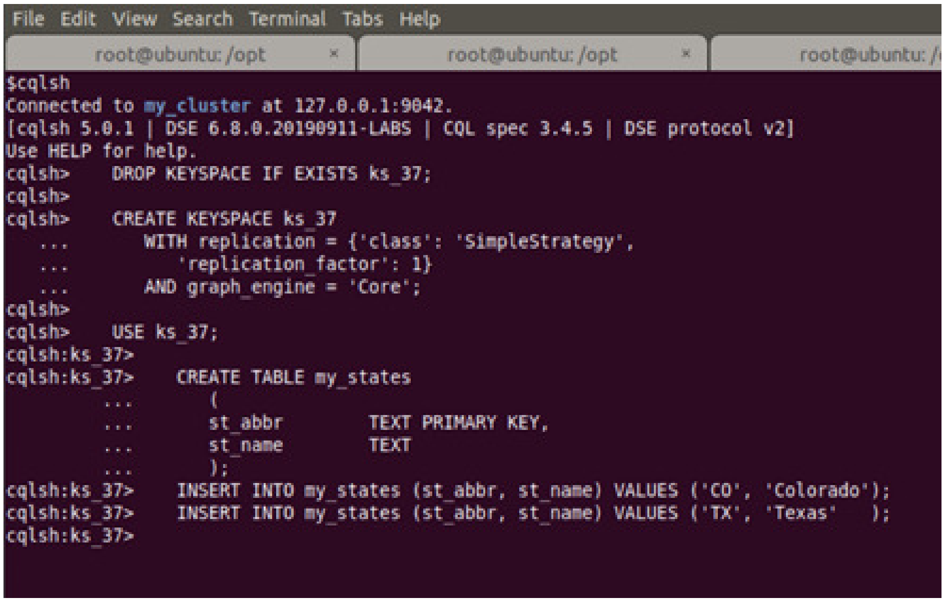
Relative to Figure 37-1, the following is offered:

- The 4 primary functional areas inside DataStax Enterprise (DSE) are located in the bottom left of the graphic above and are titled; DSE Core, DSE Search, DSE Analytics, and DSE Graph.
- DSE Core is largely a hardened version of Apache Cassandra.
- DSE Search gives DSE Core integrated Apache Solr/Lucene indexes and query predicates.

- DSE Graph is largely Apache TinkerPop/Gremlin.
- And DSE Analytics is largely an integrated Apache Spark, and gives DSE built in Parquet support, as well as joins, as well as query parallelism.

If we're going to join

If we're going to join data between DSE (Core) and Hadoop/HDFS, we'll need at least two tables, one resident inside DSE. Figure 37-2 displays the DSE (Core) assets we'll use. A code review follows.



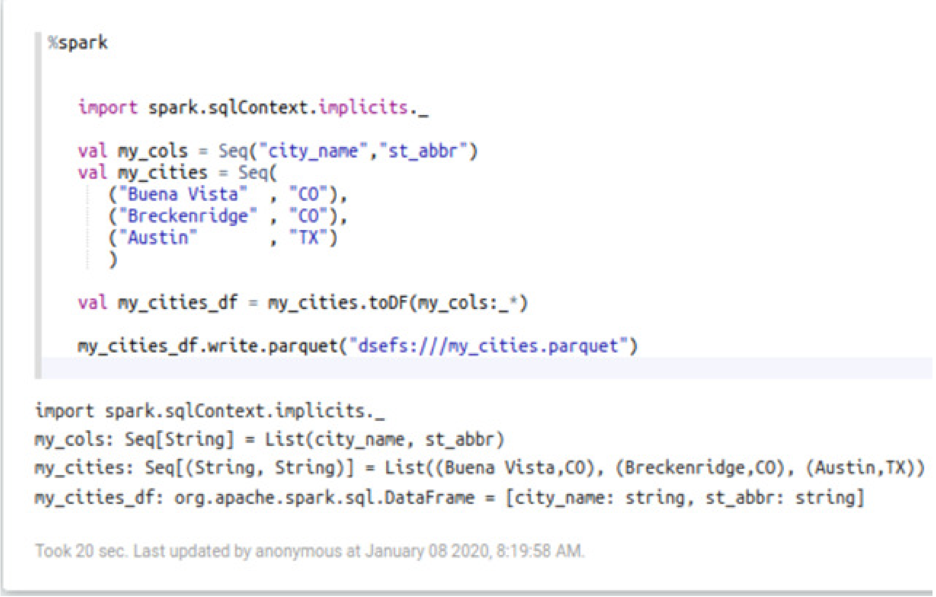
```
$cqlsh
Connected to my_cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | DSE 6.8.0.20190911-LABS | CQL spec 3.4.5 | DSE protocol v2]
Use HELP for help.
cqlsh>    DROP KEYSPACE IF EXISTS ks_37;
cqlsh>
cqlsh>    CREATE KEYSPACE ks_37
   ...       WITH replication = {'class': 'SimpleStrategy',
   ...          'replication_factor': 1}
   ...       AND graph_engine = 'Core';
cqlsh>
cqlsh>    USE ks_37;
cqlsh:ks_37>
cqlsh:ks_37>    CREATE TABLE my_states
         ...       (
         ...       st_abbr        TEXT PRIMARY KEY,
         ...       st_name        TEXT
         ...       );
cqlsh:ks_37>    INSERT INTO my_states (st_abbr, st_name) VALUES ('CO', 'Colorado');
cqlsh:ks_37>    INSERT INTO my_states (st_abbr, st_name) VALUES ('TX', 'Texas'   );
cqlsh:ks_37>
```

Figure 37-2   DSE keyspaces, table, and data

Relative to Figure 37-2, the following is offered:

- A classic DSE; make keyspaces, make table, insert two rows of data.
- Our data is two rows, US states; Colorado and Texas.
- We'll make a "cities" data set below, and join to that.

Figure 37-3 displays our "cities" data. A code review follows.



```
%spark

   import spark.sqlContext.implicits._

   val my_cols = Seq("city_name","st_abbr")
   val my_cities = Seq(
      ("Buena Vista"  , "CO"),
      ("Breckenridge" , "CO"),
      ("Austin"       , "TX")
      )

   val my_cities_df = my_cities.toDF(my_cols:_*)

   my_cities_df.write.parquet("dsefs:///my_cities.parquet")
```

```
import spark.sqlContext.implicits._
my_cols: Seq[String] = List(city_name, st_abbr)
my_cities: Seq[(String, String)] = List((Buena Vista,CO), (Breckenridge,CO), (Austin,TX))
my_cities_df: org.apache.spark.sql.DataFrame = [city_name: string, st_abbr: string]
```

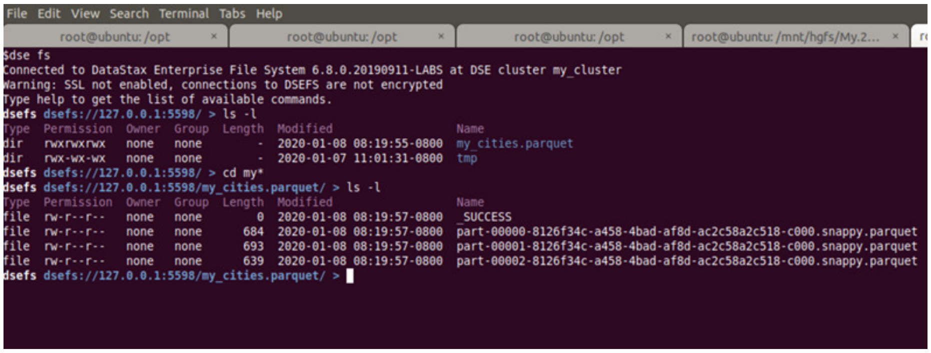
Took 20 sec. Last updated by anonymous at January 08 2020, 8:19:58 AM.

Figure 37-3   Our cities data, stored on HDFS in Parquet format

Relative to Figure 37-3, the following is offered:

- We used the HDFS compatible filesystem that comes with DataStax Enterprise. If your data is resident in HDFS, remotely, you'd need to add a connection string/resource to said system.
- The above is written in Scala. Other languages work too.
- If you use the "Always-On SQL" feature to DSE, the SQL query we run below returns in very sub-second performance.
- The code above-
    - I can't actually recall if we needed the import. With any serious SQL, you would need the import.
    - my_cols basically forms our column header metadata.
    - my_cities is first a Spark RDD, having been read from a simple Scala data type of Sequence. The toDF() method then casts this data set as an Apache Spark DataFrame, so that we may inherit the write() to Parquet method.
    - And we write to HDFS in Parquet binary format.

Figure 37-4 displays the method to confirm that we just wrote to HDFS in Parquet format. A code review follows.


File Edit View Search Terminal Tabs Help
root@ubuntu: /opt
root@ubuntu: /opt
root@ubuntu: /opt
root@ubuntu: /mnt/hgfs/My.2...

```
$dse fs
Connected to DataStax Enterprise File System 6.8.0.20190911-LABS at DSE cluster my_cluster
Warning: SSL not enabled, connections to DSEFS are not encrypted
Type help to get the list of available commands.
dsefs dsefs://127.0.0.1:5598/ > ls -l
Type  Permission  Owner  Group  Length  Modified                  Name
dir   rwxrwxrwx   none   none        -  2020-01-08 08:19:55-0800  my_cities.parquet
dir   rwx-wx-wx   none   none        -  2020-01-07 11:01:31-0800  tmp
dsefs dsefs://127.0.0.1:5598/ > cd my*
dsefs dsefs://127.0.0.1:5598/my_cities.parquet/ > ls -l
Type  Permission  Owner  Group  Length  Modified                  Name
file  rw-r--r--   none   none        0  2020-01-08 08:19:57-0800  _SUCCESS
file  rw-r--r--   none   none      684  2020-01-08 08:19:57-0800  part-00000-8126f34c-a458-4bad-af8d-ac2c58a2c518-c000.snappy.parquet
file  rw-r--r--   none   none      693  2020-01-08 08:19:57-0800  part-00001-8126f34c-a458-4bad-af8d-ac2c58a2c518-c000.snappy.parquet
file  rw-r--r--   none   none      639  2020-01-08 08:19:57-0800  part-00002-8126f34c-a458-4bad-af8d-ac2c58a2c518-c000.snappy.parquet
dsefs dsefs://127.0.0.1:5598/my_cities.parquet/ >
```

Figure 37-4 Parquet data on disk, HDFS

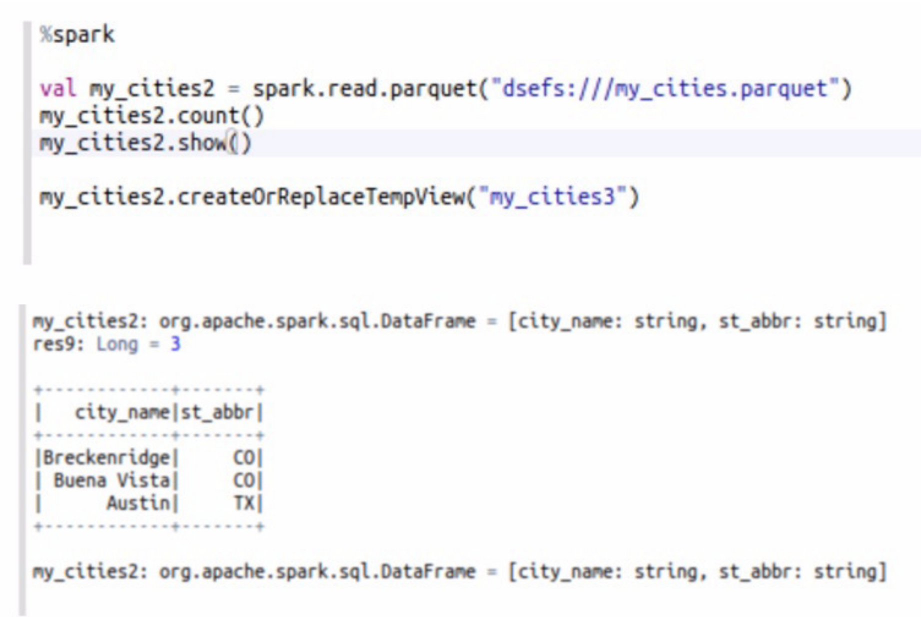
Relative to Figure 37-4, the following is offered:

- Just like when using Hadoop ("hadoop hdfs"), DSE ships with a, "dse fs" command; a shell prompt into this HDFS compatible filesystem.
- Recall that files created from Spark jobs are striped/segmented into multiple physical parts, allowing for placement across nodes. We see 3 physical files were created above.
- Because these are in binary (Parquet) format, we can't just 'cat' the files and see anything of value.

All of our prep is done

At this point, we have two data sources; one DSE, USA states data, and one Parquet, USA cities data. We are ready to continue.

Figure 37-5 shows the code to read Parquet from HDFS. A code review follows.



```
%spark

val my_cities2 = spark.read.parquet("dsefs:///my_cities.parquet")
my_cities2.count()
my_cities2.show()

my_cities2.createOrReplaceTempView("my_cities3")
```

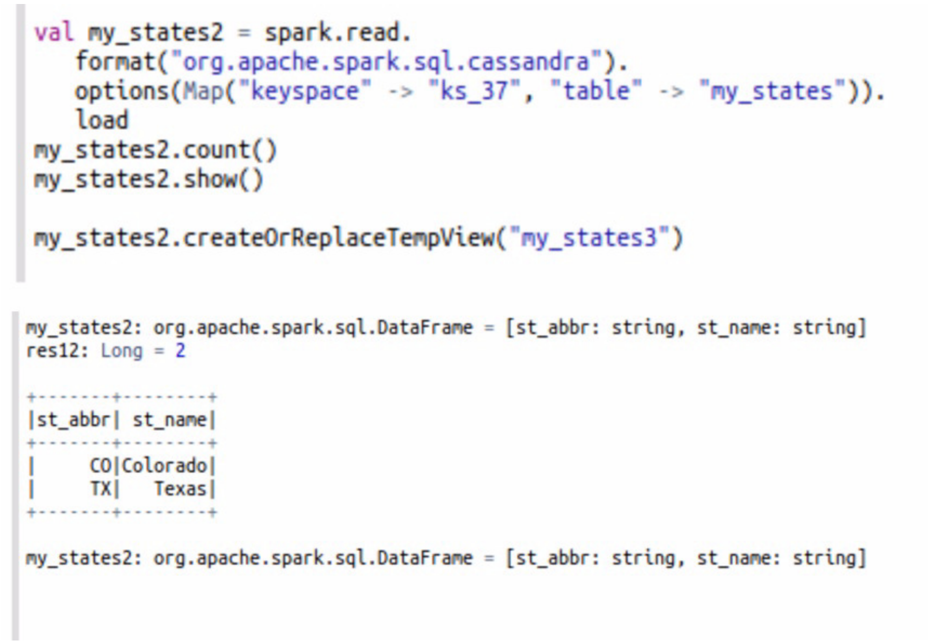
```
my_cities2: org.apache.spark.sql.DataFrame = [city_name: string, st_abbr: string]
res9: Long = 3

+------------+-------+
|   city_name|st_abbr|
+------------+-------+
|Breckenridge|     CO|
| Buena Vista|     CO|
|      Austin|     TX|
+------------+-------+

my_cities2: org.apache.spark.sql.DataFrame = [city_name: string, st_abbr: string]
```

Figure 37-5   Reading Parquet from HDFS

Relative to Figure 37-5, the following is offered:

- Again we wrote in Scala. Other languages work too.
- The count() and show() are just to check our work. We really only need the read(), and then the createOrReplaceTempView() to enable SQL operations.

Figure 37-6 displays enabling SQL reads against our DSE table. A code review follows.



```
val my_states2 = spark.read.
   format("org.apache.spark.sql.cassandra").
   options(Map("keyspace" -> "ks_37", "table" -> "my_states")).
   load
my_states2.count()
my_states2.show()

my_states2.createOrReplaceTempView("my_states3")
```

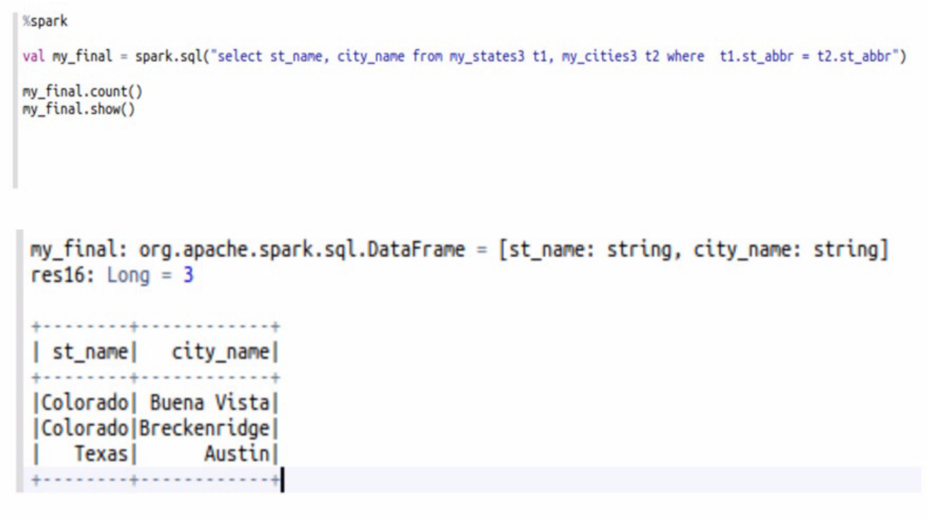
```
my_states2: org.apache.spark.sql.DataFrame = [st_abbr: string, st_name: string]
res12: Long = 2

+-------+--------+
|st_abbr| st_name|
+-------+--------+
|     CO|Colorado|
|     TX|   Texas|
+-------+--------+

my_states2: org.apache.spark.sql.DataFrame = [st_abbr: string, st_name: string]
```

Figure 37-6   USA states data inside DSE

Relative to Figure 37-6, the following is offered:

- Again, don’t need the count() or show(); just checking our work.
- The read() makes this DSE table data available.
- And the createOrReplaceTempView() enables SQL.

Figure 37-7 shows the actual SQL join. A code review follows.



```
%spark

val my_final = spark.sql("select st_name, city_name from my_states3 t1, my_cities3 t2 where  t1.st_abbr = t2.st_abbr")

my_final.count()
my_final.show()
```

```
my_final: org.apache.spark.sql.DataFrame = [st_name: string, city_name: string]
res16: Long = 3

+--------+-----------+
| st_name|  city_name|
+--------+-----------+
|Colorado|Buena Vista|
|Colorado|Breckenridge|
|   Texas|     Austin|
+--------+-----------+
```

Figure 37-7   SQL join; DSE and Parquet

Relative to Figure 37-7, the following is offered:

- We ran a simple two table SQL join with no other query predicates, but any Apache Hive 1.1 query language syntax is supported.
- Output as shown; data from DSE joined to data from HDFS/Parquet.

37.2 Complete the following

At this point in this document we have shown you how to register DataStax Enterprise (DSE) and also HDFS resident data for use inside DSE Analytics (Apache Spark) SQL queries. All valid Hive query language (SQL) syntax is valid; go forth.

Add your own tables, and write your own analytics queries. Save the data on HDFS, or inside DSE.

37.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- A rather simple primer to join DSE and HDFS/Parquet data, and perform analytics.
- Reading and writing to and from DSE and HDFS.

Persons who help this month.

Kiyu Gabriel, Dave Bechberger, Alex Ott, 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`