Chapter 56.

August 2021

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

> My company enjoyed the last article on K8ssandra and StarGate. We are highly interested in the Document API that this document referred to; use and some of the design elements. Can you help ?
>
> *Excellent question ! Last month we installed DataStax K8ssandra, which includes the StarGate component. Further, we did a (Hello World) using REST, to serve as an install/validate of StarGate. This month we dive deeper not into just REST, but now the Document API area of StarGate; make a table, insert documents, run a query.*

Software versions

The primary DataStax software component used in this edition of DDN is DataStax Enterprise (DSE), currently release 6.8.*, or DataStax Astra (Apache Cassandra version 4.0.0.*), as required. When running Kubernetes, we are running Kubernetes version 1.17 locally, or on a major cloud provider. All of the steps outlined below can be run on one laptop with 32GB of RAM, or if you prefer, run these steps on Google GCP/GKE, 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.

56.1 Terms and core concepts

As stated above, last month in this series of documents we installed and used DataStax K8ssandra, with specific focus on using the StarGate component of same. Where last month we issued REST service requests to get an authorization token and read data, this month we give focus on the Document API to StarGate.

What and why of documents

Years ago we were satisfied to load a native client side driver in an application program, and run SQL/CQL statements to complete our database work. Why should a Node.js developer (Python, Java, whatever) also have to learn an additional language (SQL/CQL), just to run data persistence routines ? This is the reason DataStax K8ssandra, and specifically the StarGate area of K8ssandra, automatically provision and maintain service API endpoints for each of; REST (Http/REST), GraphQL, (CQL), and also the Document API. Automatically, there are service endpoints running (listening) for each of; INSERT, DELETE, (read), etcetera.

So that's one improvement, less code to write and maintain; faster time to value.

But, when programming a Web application, the Web browser, and the data it sends and receives, is almost certainly JSON formatted. Why should the developer have to shred JSON to make it (rows and columns) formatted, ready to insert into a (relational database) ?

A document data store (data model) accepts (reads and write) natively formatted JSON; even more, faster time to value.

Besides the data being read and written in a JSON format, the remainder is largely semantics:

- Tables are called collections, rows are called documents.
- Data modelling is largely the same, allowing deeply nested elements, structures, collections, arrays, sorted lists, other. Apache Cassandra has long supported this modelling capability.
- But is there a schema ?

  Yes. There is always a schema. Whether you have to pre-define the schema before you can use a table/collection is another question.

  To say that MongoDB, a document database, or the K8ssandra/StarGate/Document-API system are schemaless is false. A better title would be, polymorphic schema (capable).

Note: In a document store, every single row/document in a collection (table) can have a schema unique from every other row in that same table. You could, technically, store every row, no matter how different, in a single table. (Not a great idea, but do-able.)

How MongoDB overcomes not having a data dictionary, is to embed a private copy of the schema on every single row, (because every row in a table can be different from any other row). With MongoDB, when writing a 2K row, you might actually be writing 6K, because every row is also accompanied by its own private schema.

How to have a schema when schemaless

When using the DataStax K8ssandra, StarGate, Document API, you are reading and writing to Cassandra tables, with a special data model. As such, the table is not created using a standard CQL/CREATE-TABLE method, but via a method provided for by the Document-API. Using CQL (or a number of other means), you can discover that the schema to this (Document API collection) looks like,

```
key          -  the generated, leading partition col
p0           -  the actual column name
...
p63          -
//
text_value   -
bool_value   -
dbl_value    -
```

Relative to this example, the following is offered:

- Consider that every document in a collection could have a different schema. How then do you have knowledge of the primary key ?

  By convention, document data stores will generally force at least one standard element to schemaless, that is; having at least a primary key with a label similar to, 'key', or 'id', other. 'key' could be a single column value, or a structure, or other elements more complex.

- Above, key could be generated, a UUID, or similar.
- The columns titled, p0 through p63 are Apache Cassandra partition keys. More on this though example, below.
- And the columns titled, text_value, bool_value, etcetera, store the actual column/element value.

Two examples

Below in Example 56-1, we have two curl commands that insert two documents. A code review follows.

Example 56-1

```
curl --location \
      --request POST
"localhost:8082/v2/namespaces/${l_ks}/collections/${l_table}" \
      --header "X-Cassandra-Token: $l_token" \
      --header 'Content-Type: application/json' \
      --data '{
        "col1": "222",
        "colx": { "col3" : "Bob", "col4" : "Tom" }
      }'

   curl --location \
      --request POST
"localhost:8082/v2/namespaces/${l_ks}/collections/${l_table}" \
      --header "X-Cassandra-Token: $l_token" \
      --header 'Content-Type: application/json' \
      --data '{
        "col1": "222",
        "colx": { "col3" : "Bob", "col4" : "Tom" },
        "coly": { "col3" : "Sal", "col4" : "Rob" }
      }'
   #
   #  p0   | p1   | text_value
   #  ------+------+------------
   #  col1 |      |        222
   #  colx | col3 |        Bob
   #  colx | col4 |        Tom
   #  coly | col3 |        Sal
   #  coly | col4 |        Rob
```

Relative to Example 56-1, the follow is offered:

- The two curl commands insert two new documents. Both of the inserts will each generate a new UUID, 'key' value, not displayed.
- The first insert inserts rows in the Cassandra table with,
  - A 'col1' with a value of '222' in the Cassandra table column, p0.
  - 'colx' contains a nested element, a sub JSON document. This nested document has sub-elements, 'col3' and 'col4'.

Notice how 'p0' has the anchored key column name, 'colx', and then drills down into p1 for the key column names, 'col3' and 'col4'.

This is the storage pattern.

– With the second insert, we call to insert a second, net new document; although the data as displayed show only one document. (The first and second insert share key and column values, but there would be two rows.)

Notice how 'colx', and 'coly' record nested values.

Note: If there isn't a clear industry standard name for this type of model, we could call it, 'document shredding'.

Disadvantage(s):

– This form of storage modeling is likely more CPU intensive to disassemble then re-assemble data as it is stored.

Advantage(s):

– No schema has to be created and stored for each row stored in the table.

– Not displayed; indexing. In effect, minus the partition key, you only need to index the value columns, which are less than 5 in count.

A non-shredded document data store could have dozens of indexes per table, and indexes are not free; they have to be maintained.

56.2 Complete the following

At this point in this document we have the basics. Time now to implement some of these routines. As a starting point, the following is assumed:

– We have a working DataStax K8ssandra cluster, per the instructions of the previous document in this series, including the port forwarding instructions.

– We have a given Cassandra keyspace. In the examples throughout, ours is titled, 'ks1'.

– We can create/receive an authorization token, and run curl(C). (Again, for a how-to, reference the prior document in this series.)

Two examples in the prior insert rows/documents into a table. Now we present SELECT.

```
curl --location \
```

```
      --request GET
"localhost:8082/v2/namespaces/${l_ks}/collections/${l_table}?page-si
ze=20" \
      --header "X-Cassandra-Token: ${l_token}" \
      --header 'Content-Type: application/json'


   #  The parameter `page-size` is limited to 20.


   {"data":
      {

"e85e46fd-f568-445b-8035-4e3f672af839":{"col1":"222","colx":{"col3":
"Bob","col4":"Tom"},"coly":{"col3":"Sal","col4":"Rob"},"colz":{"col5
":["eee","fff","ggg"]}},

"caaa4612-c529-41a2-ab89-248de8b00f99":{"id":"some-stuff","other":"T
his is nonsensical stuff."},

"7cfc9e51-f637-4ae0-8c4a-cc5b2f7dfbaf":{"col1":"222","colx":{"col3":
"Bob","col4":"Tom"}},

"0486a953-ab9f-4249-bde3-28fe585b6630":{"col1":"222","colx":{"col3":
"Bob","col4":"Tom"},"coly":{"col3":"Sal","col4":"Rob"}}
      }
   }
```

Relative to the above, the following is offered:

- This is a sequential scan of the table; all rows. Paging is supported.
- Reading from the partition key or secondary keys, (and all other data manipulations statements), are documented at,

  `https://stargate.io/docs/stargate/1.0/quickstart/quickstart.html`

56.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- A reasonably sized introduction to the Document API of the StarGate component that arrives with DataStax K8ssandra.
- We detailed INSERTs, and SELECTs.
- And we detailed the underlying Apache Cassandra data model.

Persons who help this month.

Kiyu Gabriel, Jim Hatcher, Joshua Norrid, and Yusuf Abediyeh.

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
```