Chapter 25.

January 2019

Welcome to the January 2019 edition of DataStax Developer's Notebook (DDN). This month we answer the following question(s);

> Graph, graph, graph; what the heck is up with graph- I think (hope ?) there's something graph databases do that standard relational databases do not, but I can't articulate what that function or advantage actually is. Can you help ?
>
> *Excellent question ! Yes, but we're going to take two editions of this document to do so.*
>
> *Sometimes there are nuances when discussing databases; what really is the difference between a data warehouse, data mart, data lake, other ? Why couldn't you recreate some or most non-relational database function using a standard relational database ?*
>
> *In this edition of DataStax Developer's Notebook (DDN), we provide a graph database primer; create a graph, and load it. In a future edition of this same document, we will actually have the chance to provide examples where you might determine that graph databases have an advantage over relational databases for certain use cases.*

Software versions

The primary DataStax software component used in this edition of DDN is DataStax Enterprise (DSE), currently release 6.7. 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 version 18.0, 64 bit.

25.1 Terms and core concepts

The problem statement listed above is to articulate why a graph database is advantageous over a relational database, presumably for given individual use cases. Recall that DataStax Enterprise Server (DSE) is composed of four primary functional areas. These four areas include:

- DSE Core
- DSE Search
- DSE Analytics
- DSE Graph (aka, graph)

DSE Core

DSE Core carries many responsibilities, including that it is the (disk) storage tier to all of DSE. While DSE Search calculates key column values, that data is actually stored in DSE Core. DSE Graph data, is stored in DSE Core.

Compared to common relational databases, DSE Core provides at least two truly unique capabilities. These are:

- Time constant lookups-

  DSE Core is centered on a hash indexed write ahead log. Effectively, this design does scale linearly regardless of the size of data being hosted. (B-Tree indexes and relational databases scale well, but they do not scale linearly.)

- Network partition fault tolerance

  Some customers describe DSE as a "multi-write database", meaning; the same piece of data is writable on two or more nodes simultaneously.

  This is advantageous for global applications. A single piece of data is writable in, for example; Germany and the USA at the same time.

  With this capability, DSE can also accept writes when the network between Germany and the USA is down. With users in both countries, a standard relational database would even prevent reads on one side of the (down network). Most NoSQL databases would at least allow reads.

  DSE allow reads and writes, and creates a single best surviving (most accurate) record once the network is restored.

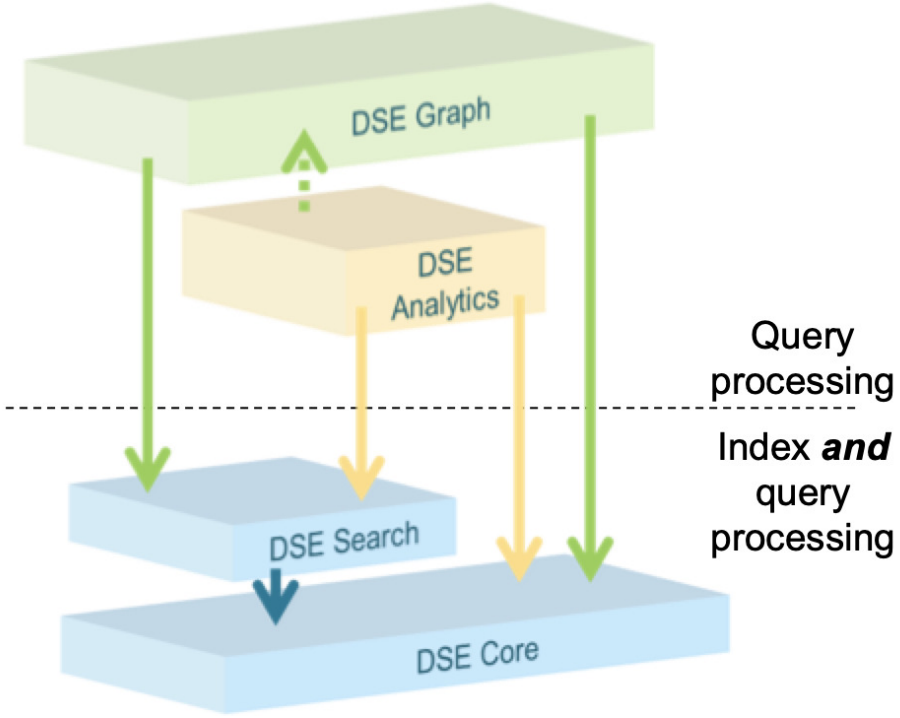

Figure 25-1 DSE, 4 primary functional areas

DSE Search, DSE Analytics, DSE Graph

Dse Search (aka, Apache Solr/Lucene), gives DSE (and thus DSE Graph), advanced indexes for:

- Spatial and geo-spatial queries, distance from, if in polygon, other
- Time series
- Text analytics; sounds like, mis-spellings, stemming, synonyms, other
- And more

DSE Analytics (aka, Apache Spark), gives DSE (and thus DSE Graph), parallel inserts, updates, deletes, access to Spark/SQL, machine learning, streaming, and Spark DataFrames and GraphFrames.

And of course, DSE Graph itself. As a result of sitting atop DSE Core, the DSE Graph component is network partition fault tolerant, and scales linearly across nodes; generally, both features other graphs fail to deliver.

DSE Graph object hierarchy

Figure 25-2 displays the DSE Graph object hierarchy. A code review follows.

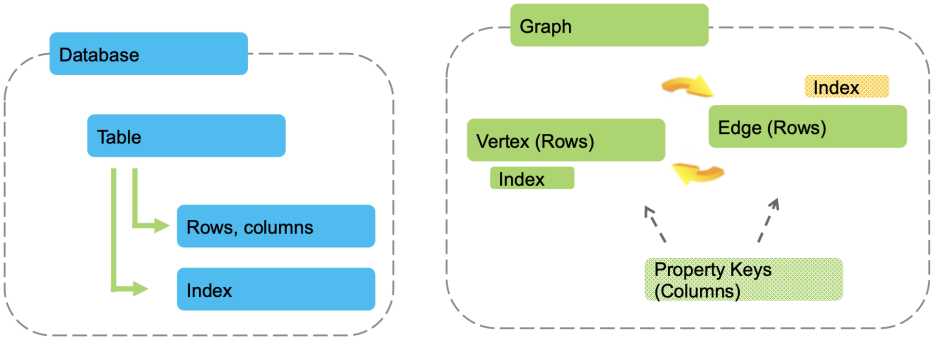

Figure 25-2 DSE Graph object hierarchy

Relative to Figure 25-2, the following is offered:

- Logical terms and physical terms; generally, physical terms exist, they have mass. Logical terms generally group, or identify one of more physical terms.

  Each of the terms in the above diagram are either physical or logical.

- A DSE (graph) is most similar to a relational database (database). A graph contains one or more vertices, and edges, where a relational database contains tables.

- Unlike a relational database, a graph defines property keys first. Property keys most resemble a relational database column, with a data type, null-ability condition, other.

  Property keys form the (columns) inside (tables and indexes), and their properties are global in scope to a graph.

- A graph vertex is most similar to a relational database table; rows and property keys (columns).

- In a relational database, the relationships between tables are *defined* on every SQL SELECT statement. In a graph, the relationships between tables are *recorded* via edges.

  On some level, you could view a graph edge as a relational database materialized view; the edge contains the join pairs between vertices, pre-recorded, pre-calculated. This condition makes traversing a graph very fast.

Note: Relational databases *define* the relationship between rows in every table, on every SQL SELECT, via query predicates in the SQL SELECT WHERE clause. These join relationships can change, and even be ad-hoc (a never before imagined query), on every query.

If the relationships between rows in graph vertices are recorded as join pairs within edges, doesn't that make graphs and graph queries more rigid (less impactful) than relational database queries ?

No.

First, consider a standard relational database Customer to Orders relationship; what is the use case to join a Customer with Orders they did not place ?

While a graph traversal also has (query predicates), these predicates exist to shape the query, not define the query.

E.g., I see a big storm is heading to Houston, TX, and Houston is likely to flood. Show me only high value customers who have any relationship to Houston-

The rows returned might be merchandise that is being sourced from a Houston warehouse, or deliveries that are routed through Houston. It could also be that an inspector for any merchandise has a mother who lives in Houston, TX.

In a graph query, you effectively ask the database to show you how or if (two rows) are related. In a relational database, you have to know (and specify in the SQL SELECT syntax), exactly how rows are related.

Lastly, graph query predicates are much more (programmable). Consider a recursive join on a Persons table-

On the first level join, you might specify that the person you know has to be a relative, but on the next level join, the person you know has to be a coach or teacher.

Even if you believe you could answer the very same queries using a relational database, you will be writing much more code to answer these types of questions using relational; dense, fragile code, that will likely break when the structure of the database changes.

- Graph edges also resemble relational database tables, in that edges contain property keys; not only the join pair columns between two vertices, but also property keys proper. I.e., How long has Bob known Dave.
- If you chose to view graph edges metaphorically as indexes, we can live with that; edges do contain the join pair columns from the vertices after all.

However, graph vertices can benefit from all of the indexes provided through DSE Core and DSE Search, allowing geo-spatial query predicates, and more.

Example queries, relational and graph

Figure 25-3 displays a standard relational database query. A code review follows.

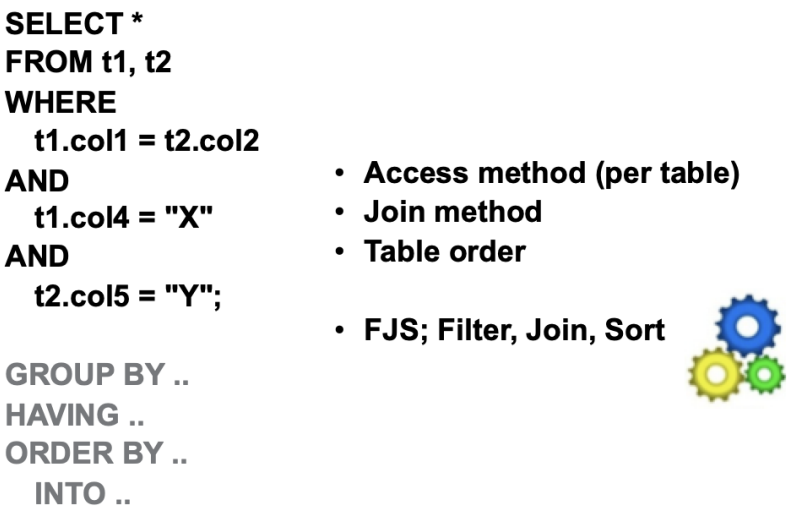

Figure 25-3 Example Relational Query

Relative to Figure 25-3, the following is offered:

- A SQL SELECT has up to seven clauses; two mandatory (select column list, from table list), and five optional clauses (group by, having, ..). The clauses, if present, can appear only once and must appear in order.
- An automatic subsystem to the relational database is the query optimizer. The query optimizer determines how to provide your request for SQL Service; which indexes to use, other.
- In the example above-
  - Two query predicates, col4 and col5, both equalities; which can be supported via an index ?
  - And then table order; whether you read a 100 row table and join into a 1 million row table, or 1 million row table then 100 row table matters. (Tuple arithmetic and resource consumption.)

- Generally, the query optimizer will pick the most efficient (index supported) query predicate first (filter), then determine table order based on join conditions (join).

  Sort most often comes last, as the newly created (table) did not exist until the join and could not have been pre-sorted.

Figure 25-4 displays a Gremlin traversal (similar to a SQL SELECT). A code review follows.

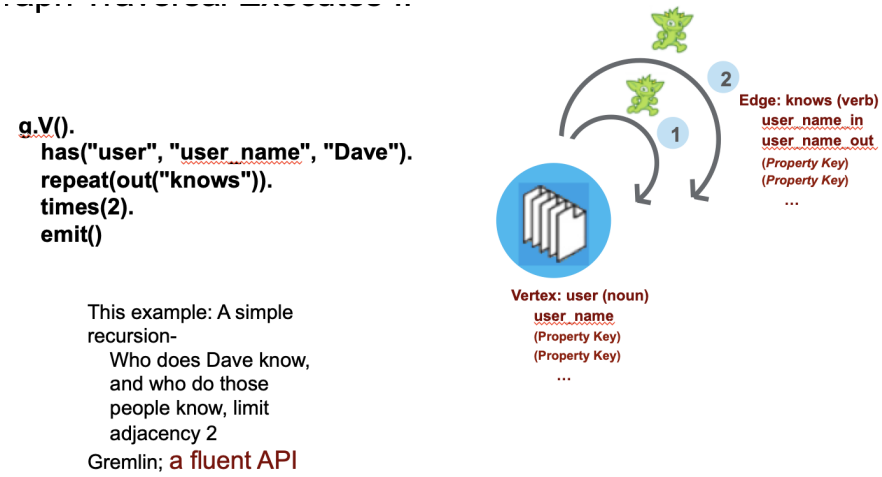

Figure 25-4 Gremlin traversal, happens to be a recursion

Relative to Figure 25-4, the following is offered:

- g.V() is the command similar to SQL SELECT. (There are others, this is one.)
- In this example, we call to start at the "user" table, and leave along the relationship titled, "knows". We filter starting with just the "user_name" valued, "Dave".

  We call to repeat two times, so; who does Dave know, and who do those people know.
- Graph traversals do not have to be recursive, but graph does do recursion really, really well. Each step in the recursion could be based on a different set of query predicates.
- Similar to a relational query, an index on "user_name" would aid performance; as efficiently as possible, reduce the number of "user.user_name" rows that have to be examined.

After this predicate, rows are pre-joined and supplied key values come from the graph edge.

– Graph vertices tend to be named after nouns, and graph edges tend to be named after verbs.

An Example using Customer

Figure 25-5 displays a simple graph we will use on the next set of pages. A code review follows.

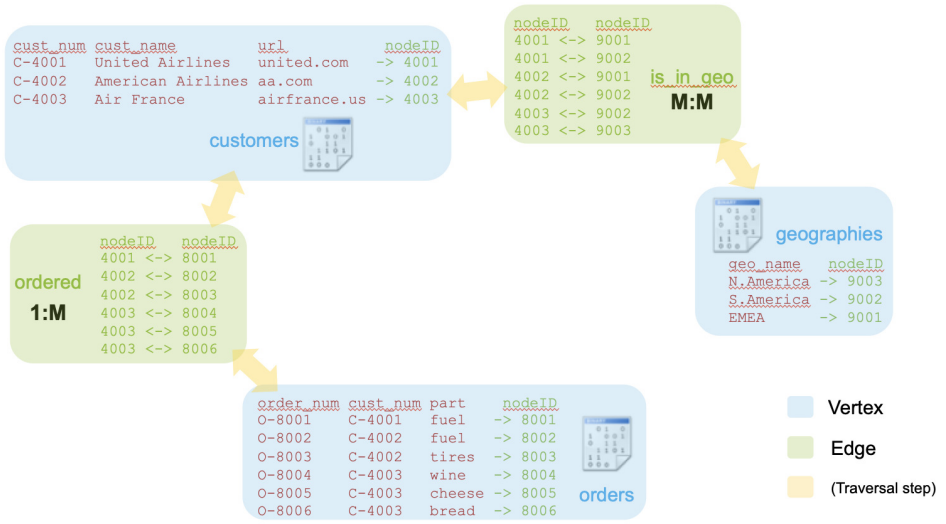

Figure 25-5 A simple "Customer" graph

Relative to Figure 25-5, the following is offered:

– Three vertices are displayed in blue; customer, orders, and geographies. Vertices are normally named after nouns.

– Two edges are displayed in green; ordered, and `is_in_geo`. Edges are normally named after verbs.

– `cust_num` appears in both customer and orders as a natural key, likely a key we inherited from any legacy system we are now replacing with graph.

A system generated key also appears in each vertex with the property key name, `nodeID`.

The data in the edge titled, ordered, could easily be generated via a Spark/SQL SELECT statement similar to that as displayed in Example 25-1.

Example 25-1 Spark/SQL, run-able as Scala in this example

```
val ordered = spark.sql(
   "select " +
   "   t1.nodeID as nodeID_C, " +
   "   t2.nodeID as nodeID_O   " +
   "from " +
   "   customers t1, " +
   "   orders t2 " +
   "where " +
   "   t1.cust_num = t2.cust_num"
   )
```

Note: Example 25-1 is written in Spark/SQL, specifically Scala. This statement would automatically run in parallel across multiple nodes using DSE.

DSE Analytics (Apache Spark) offers a lot of benefit to DSE as a whole; parallel DMLs, SQL syntax, DataFrames and GraphFrames, and more.

- There is no natural key displayed in customer or geo that would populate the edge titled, `is_in_geo`. This relationship would have to be generated via data not currently displayed.

Note: As the original "geo" table arrived, it had a many to many relationship to customer, and was effectively our edge data. It was actually the vertex we derived using a Spark/SQL SELECT DISTINCT.

Figure 25-6 displays the sequence of steps to create and load our example graph. A code review follows.

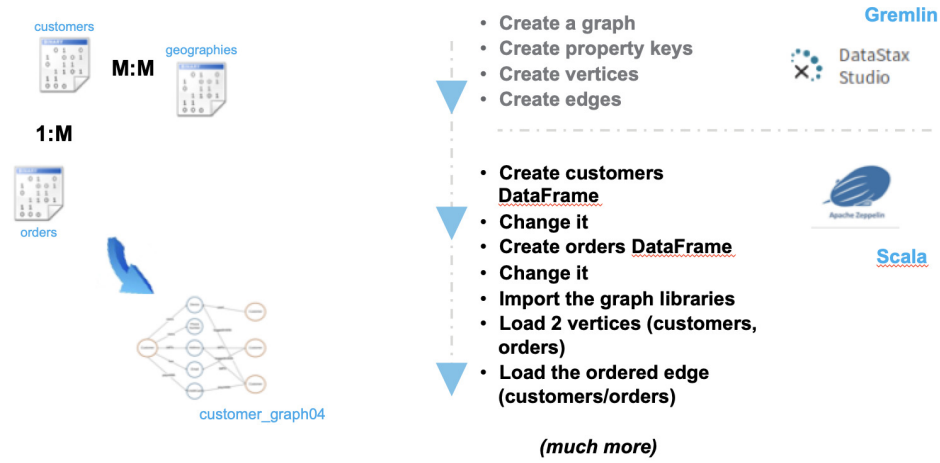

Figure 25-6 Sequence of steps to create and load this graph

Relative to Figure 25-6, the following is offered:

- While there are many paths to complete these steps, we often choose to use DataStax Studio, and the Gremlin interpreter to create; graphs, property keys, vertices, edges, other.
- While DataStax Enterprise (DSE) ships with a Scala REPL (Python REPL, other), we often choose to use the open source, Apache Zeppelin as a graphical (Web based) Spark (Scala) interpreter.

  Using Spark, we can load in parallel with a programmable interface, versus say a utility like a bulk data loader, other, which DSE also offers.

25.2 Complete the following:

Above we overviewed the objects and process to create and load a graph using DSE. In this section, we present the actual code. All of the examples below will run on DSE version 6.7, and likely, earlier versions of DSE.

Using DataStax Studio (Studio), we call to run the Gremlin interpreter to make the property keys, vertices, and edges. Studio will see we want to run Gremlin, and will prompt to make our graph for us via dialog boxes, not shown.

Example 25-2 displays the Gremlin code we use to make the property keys, and more. A code review follows.

Example 25-2 Gremlin code to make property key, and more

```
.  Paragraph 01, Studio, Gremlin
--------------------
//  Paragraph 01

system.graphs()
system.describe()
--------------------


.  Paragraph 02, Studio, Gremlin
--------------------
//  Paragraph 02

schema.drop()
schema.config().option('graph.allow_scan').set('true')
--------------------


.  Paragraph 03, Studio, Gremlin
--------------------
//  Paragraph 03

//  Property keys

schema.propertyKey('nodeID'   ).Int() .single().create()
   //
schema.propertyKey('cust_num' ).Text().single().create()
schema.propertyKey('cust_name').Text().single().create()
schema.propertyKey('url'      ).Text().single().create()
schema.propertyKey('order_num').Text().single().create()
schema.propertyKey('part'     ).Text().single().create()
schema.propertyKey('geo_name' ).Text().single().create()
--------------------


.  Paragraph 04, Studio, Gremlin
--------------------
//  Paragraph 04

//  Vertices

schema.vertexLabel("customers").
   partitionKey("nodeID").
   properties(
      'nodeID'                      ,
      'cust_num'                    ,
      'cust_name'                   ,
```

```
      'url'
   ).ifNotExists().create()

schema.vertexLabel("orders").
   partitionKey("nodeID").
   properties(
      'nodeID'                         ,
      'order_num'                      ,
      'cust_num'                       ,
      'part'
   ).ifNotExists().create()

schema.vertexLabel("geographies").
   partitionKey("nodeID").
   properties(
      'nodeID'                         ,
      'geo_name'
   ).ifNotExists().create()

//  Edges


schema.edgeLabel("ordered").
   single().
   connection("customers", "orders").
   ifNotExists().create()

schema.edgeLabel("is_in_geo").
   single().
   connection("customers", "geographies").
   ifNotExists().create()
--------------------
```

Relative to Example 25-2, the following is offered:

- DataStax Studio (Studio), like Apache Zeppelin, can organize code into paragraphs. Above, paragraphs are labeled, Paragraph 01, etcetera, and delimited via dashed lines. (Do not enter the dashed lines into Studio.)
- Paragraph 01, is read-only, a nervous tick of sorts. These commands tell us we can actually connect to DSE Graph, what graphs exist, other.
- Paragraph 02, has two lines-
  - The first line calls to erase all data and schema objects (property keys, other), from our current graph. We offer this command in the event you cause error, and need to restart this sequence of steps.

- The second command is for development only; in the event we call to query (traverse) our graph for testing, and call for a traversal that should require an index, allow sequential scans instead.

Note: Don't allow sequential scans in production; don't use this command in production.

- Paragraph 03, has several lines-
  - Here we make the property keys, using only two data types; integer and text (string). Obviously DSE graph supports additional data types, and this topic is considered out of scope for our needs today.
  - The opposite of single, is multiple, which is used for DSE collection types (arrays); set, list, and map. This topic is also beyond scope at the moment.
- Paragraph 04 makes our vertices, then edges. Comments-
  - Vertices must exist, before edges can refer to them.
  - Both vertices and edges make use of property keys, although in this example; the edges use property keys for the join keys only. (Edges can have property keys proper, and this condition is not displayed.)
  - The "`partitionKey`" is exactly equal to the DSE Core partition key, part of the primary key to each DSE Core table.

Note: For a further description of `partitionKey(s)`, we point you to the very first edition of the document in this series, which details DSE Core, and primary keys, partition keys, and clustering columns.

- In edges, the single property could be multiple. In this case a person, for example, could rate a movie multiple times, or similar.

In Example 25-3, we move to using Spark to load our graph. We could have continued using Gremlin, but the application framework of Spark is so much more powerful than using (Gremlin commands alone, without a language). A code review follows.

Example 25-3 Loading the graph using Spark (Scala)

```
.  Paragraph 01, Zepp
--------------------
%spark
```



```
//  Paragraph 01

import org.apache.spark.sql.functions.{monotonically_increasing_id, col, lit,
concat, max}

val customers = sc.parallelize(Array(
   ("C-4001", "United Airlines"  , "united.com"  ),
   ("C-4002", "American Airlines", "aa.com"      ),
   ("C-4003", "Air France"       , "airfrance.us")
   ) )

val customers_df = customers.toDF ("cust_num", "cust_name", "url").coalesce(1)

val customers_df_nodeID = customers_df.withColumn("nodeID",
monotonically_increasing_id() + 4001)

customers_df_nodeID.getClass()
customers_df_nodeID.printSchema()
customers_df_nodeID.count()
customers_df_nodeID.show()

customers_df_nodeID.registerTempTable("customers")
--------------------


.  Paragraph 02, Zepp
--------------------
%spark

//  Paragraph 02

val orders = sc.parallelize(Array(
   ("O-8001", "C-4001", "fuel"     ),
   ("O-8002", "C-4002", "fuel"     ),
   ("O-8003", "C-4002", "tires"    ),
   ("O-8004", "C-4003", "wine"     ),
   ("O-8005", "C-4003", "cheese"   ),
   ("O-8006", "C-4003", "bread"    )
   ) )
val orders_df = orders.toDF ("order_num", "cust_num", "part").coalesce(1)

val orders_df_nodeID = orders_df.withColumn("nodeID",
monotonically_increasing_id() + 8001)

orders_df_nodeID.getClass()
orders_df_nodeID.printSchema()
orders_df_nodeID.count()
orders_df_nodeID.show()
```



```
orders_df_nodeID.registerTempTable("orders")
--------------------

.  Paragraph 03, Zepp
--------------------
%spark

//  Paragraph 03

val ordered = spark.sql(
   "select " +
   "   t1.nodeID as nodeID_C, " +
   "   t2.nodeID as nodeID_O   " +
   "from " +
   "   customers t1, " +
   "   orders t2 " +
   "where " +
   "   t1.cust_num = t2.cust_num"
   )

ordered.getClass()
ordered.printSchema()
ordered.count()
ordered.show()
--------------------


.  Paragraph 04, Zepp
--------------------
%spark

//  Fourth paragraph

import com.datastax.bdp.graph.spark.graphframe._
import org.apache.spark.ml.recommendation.ALS
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.sql.functions.{monotonically_increasing_id, col, lit,
concat, max}
import java.net.URI
import org.apache.spark.sql.{DataFrame, SaveMode, SparkSession}
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.{FileSystem, Path}
import com.datastax.spark.connector._
import org.apache.spark.sql.cassandra._
import com.datastax.spark.connector.cql.CassandraConnectorConf
import com.datastax.spark.connector.rdd.ReadConf
```



```
import org.apache.spark.sql.expressions.Window

val graphName = "customer_graph04"

val g = spark.dseGraph(graphName)
--------------------

.  Paragraph 05, Zepp
--------------------
%spark

//  Fifth paragraph

val customers_V = customers_df_nodeID.withColumn("~label", lit("customers")).
   withColumn("nodeID"   , col("nodeID")   ).
   withColumn("cust_num" , col("cust_num") ).
   withColumn("cust_name", col("cust_name")).
   withColumn("url"      , col("url")      )
g.updateVertices(customers_V)

val orders_V = orders_df_nodeID.withColumn("~label", lit("orders")).
   withColumn("nodeID"   , col("nodeID")   ).
   withColumn("order_num", col("order_num")).
   withColumn("cust_num" , col("cust_num") ).
   withColumn("part"     , col("part")     )
g.updateVertices(orders_V)

g.V.hasLabel("customers").show()
g.V.hasLabel("orders").show()
--------------------


.  Paragraph 06, Zepp
--------------------
%spark

//  Sixth paragraph

val orders_L = ordered.
   withColumn("srcLabel", lit("customers")).
   withColumn("dstLabel", lit("orders")).
   withColumn("edgeLabel", lit("ordered")
   )
orders_L.show()

val ordered_E = orders_L.select(
   g.idColumn(col("srcLabel"), col("nodeID_C" )) as "src",
   g.idColumn(col("dstLabel"), col("nodeID_O")) as "dst",
   col("edgeLabel") as "~label"
```



```
   )

ordered_E.show()

g.updateEdges(ordered_E)
--------------------


.  Paragraph 07, Zepp
--------------------
%spark

//  Seventh paragraph

g.E.hasLabel("ordered").show()

g.V().hasLabel("customers").has("url",
"aa.com").out("ordered").valueMap("order_num", "cust_num", "part").show()
--------------------

.  Paragraph 08, Zepp
--------------------
%spark

//  Paragraph 08

import org.apache.spark.sql.functions.{monotonically_increasing_id, col, lit,
concat, max}

val geographies = sc.parallelize(Array(
   ("N.America"),
   ("S.America"),
   ("EMEA"     )
   ) )

val geographies_df = geographies.toDF ("geo_name").coalesce(1)

val geographies_df_nodeID = geographies_df.withColumn("nodeID",
monotonically_increasing_id() + 9001)

geographies_df_nodeID.getClass()
geographies_df_nodeID.printSchema()
geographies_df_nodeID.count()
geographies_df_nodeID.show()

geographies_df_nodeID.registerTempTable("geographies")
--------------------
```



```
.  Paragraph 09, Zepp
--------------------
%spark

//  Paragraph 09

val geographies_V = geographies_df_nodeID.withColumn("~label",
lit("geographies")).
   withColumn("nodeID"   , col("nodeID")  ).
   withColumn("geo_name" , col("geo_name"))
g.updateVertices(geographies_V)

g.V.hasLabel("geographies").show()
--------------------

.  Paragraph 10, Zepp
--------------------
%spark

//  Paragraph 10

val is_in_geo = sc.parallelize(Array(
   (4001, 9001),
   (4001, 9002),
   (4002, 9001),
   (4002, 9002),
   (4003, 9002),
   (4003, 9003)
   ) )

val is_in_geo_df = is_in_geo.toDF ("nodeID_C", "nodeID_G").coalesce(1)

val geo_L = is_in_geo_df.
   withColumn("srcLabel", lit("customers")).
   withColumn("dstLabel", lit("geographies")).
   withColumn("edgeLabel", lit("is_in_geo")
   )
geo_L.show()

val geo_E = geo_L.select(
   g.idColumn(col("srcLabel"), col("nodeID_C" )) as "src",
   g.idColumn(col("dstLabel"), col("nodeID_G")) as "dst",
   col("edgeLabel") as "~label"
   )
geo_E.show()

g.updateEdges(geo_E)

g.E.hasLabel("is_in_geo").show()
```



```
g.V().hasLabel("customers").has("url",
"aa.com").out("is_in_geo").valueMap("geo_name").show()
--------------------
```

Relative to Example 25-3, the following is offered:

- Paragraph 01,
  - The import allows a SQL expression we use to generate a unique system generated key (`nodeID`).
  - The `parallelize` gives us 3 rows of data to place into our customers vertex.
  - The `toDF` transforms what was a Spark RDD into a Spark DataFrame. Having a DataFrame allows us to run Spark/SQL against this data.

Note: Don't use the coalesce in production, it will force all of our rows to just one node, limiting parallelism.

We do it here, for development, so that our system generated keys are nice and sequential.

  - Starting with `getClass`, these 4 commands are diagnostic/debugging only.
  - The `registerTempTable` makes this DataFrame known to Spark/SQL, so that we may run SQL against this data.
- Paragraph 02,
  - This block of code is identical in function to the paragraph above. Instead of customers, we are now processing orders.
- Paragraph 03, runs Spark SQL-
  - Here we use the natural keys from customers and orders to generate the join pairs between these two vertices.
  - Our intent is to populate the edge between these two vertices, later.
- Paragraph 04, imports and opening our graph
  - There are a lot of common imports we don't actually need for this example, but would use for a larger, more complex example. All of this code could be pasted into a real Scala program.
  - The last two lines give us a connection handle to our graph, which we created with DSE Studio, above.

- Paragraph -5, loads the first two vertices; customers, and orders
  - ordered is our original data DataFrame. We add a new column valued with a literal string constant equal to, "customers".

    Adding this literal constant with the name of the vertex is a requirement to the next method titled, `updateVertices`.
  - And we repeat these steps to load orders.
- Paragraph 06 begins the block to load the edge between customers and orders
  - Similar to the steps to load a vertex, we need to add metadata to the DataFrame containing the (data for the edge proper).
  - `orders_L` adds three columns with string constants; labels required before inserting into the edge.
  - The second/final transform makes use of the `idColumn` method-

    This method is core to loading an edge. Here we reference the system generated keys, and a label, to generate the (addresses) used inside the edge as join pair keys.
  - We end with the method titled, `updateEdges`.
- Paragraph 07 offers a number of read-only, diagnostic traversals; confirming our work.
- Paragraph 08 starts a block equal in function to the work performed above (paragraph 08 through 10.).

  Here we populate the geo vertex, and the `is_in_geo` edge, using the same techniques introduced above.

25.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- A graph primer; essentially a round trip to create and load a graph.
- We load the graph using the most capable and performant means possible; Spark DataFrames and GraphFrames.

Persons who help this month.

Kiyu Gabriel, Jim Hatcher, Alex Ott, and Caleb Rackliffe.

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
```