

July 2019

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

I'm confused. I saw a presentation at the 2019 DataStax world conference (Accelerate 2019), detailing how to deliver a product recommendation engine using DSE Graph. I've also seen DSE articles detailing how to deliver a product recommendation engine using DSE Analytics.

Can you help ?

Excellent question ! As discussed in previous editions of this document, there are 4 primary functional areas within DataStax Enterprise (DSE). DSE Analytics can deliver a 'content-based' product recommendation (aka, product-product). DSE Graph can deliver a 'collaborative-based' product recommendation engine (aka, user-user). Both DSE Analytics and DSE Graph use DSE Core as their storage engine, and DSE Search as their advanced index engine; a full integration, not just a connector.

In this edition of this document we'll detail all of the code needed to deliver the above, and include data. We'll also use this edition of this document to provide a Graph query primer (Gremlin language primer), and answer the nuanced question of; Why Graph ?

Software versions

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

31.1 Terms and core concepts

As stated above, ultimately the end goal is to detail and differentiate the delivery of product recommendation engines ("users who bought (X) also bought (Y); up-sell, increase revenue per transaction), when using DataStax Enterprise (DSE) Analytics and/or DSE Graph.

In this document, the following are held true;

- Really the only prerequisites are that you have access to a working DSE server instance, and DSE Studio Web client. You also need a working Apache Zeppelin, that points to your working DSE.

Earlier versions of the document in this series detail how to make the above happen. If you have none of the above, and no previous skill to make these prerequisites happen, you're looking at about 1-5 hours worth of work and reading.

- All data and program code used in this article are available at,
`tinyurl.com/ddn3000`

You want the July/2019 dated set of artifacts.

- In this document, we deliver a product recommendation engine using DSE Analytics (Apache Spark).
- Also in this document, we deliver a product recommendation engine using DSE Graph (Apache TinkerPop/Gremlin).

This document series has never provided a Gremlin (the query language to Graph) primer, and we use this example to do so.

In other words; you'll also learn Gremlin in this document.

- Through the work/detail above, we also seek to answer the question; When/Why Graph ?

Machine Learning, is it real

In addition to providing the core Apache Spark data abstraction (DataFrames, Datasets, and RDDs; resilient distributed datasets), Spark streaming, and more, DataStax Enterprise (DSE) also provides 28+ machine learning routines out of the box.

Our favorite definition of machine learning is; "an algorithm whose accuracy improves given more data". (And, we greatly prefer the phrase, machine learning, to its idiot cousin phrase, artificial intelligence (AI)).

As a topic, machine learning first appeared in human history circa 1965 with quotes including:

"Machines will be capable, within 20 years, of doing any work a man can do."

-- Herbert Simon, 1965, Ph.D, IIT

"From three to eight years, we will have a machine with the general intelligence of an average human being."

-- Marvin Minsky, 1970, Ph.D, Princeton, MIT, Turing Award 1969

https://en.wikipedia.org/wiki/Herbert_A._Simon

https://en.wikipedia.org/wiki/Marvin_Minsky

There have been, since 1965, many "AI Winters", meaning; there have been many false starts between now and then. Machine learning is now a reality due to many factors:

- Availability of lots of data
- Fast, parallel, cheap computing power
- Now a focus on weak/narrow (AI), versus General Intelligence (AGI), which was more the focus in 1965.
- Machine learning (ML) is a focus of many companies, mathematicians, universities, and more, aiding the global economy of scale for companies like; Amazon, Google, US DoD/NSA, Cisco, FedEx, and many more.
- And it can not be understated the impact of open source and social coding in the area of advancing ML.

Machine Learning, the process

Out of the box, DataStax Enterprise (DSE) comes with 28 or more machine learning routines. Learning each routine would take 1-6 hours each (6 or more hours for experimentation), and 8-20 or more pages of documentation each.

In this edition of this document (DataStax Developer's Notebook, DDN), we detail one routine titled; frequent pattern mining. Product recommendation (users who bought (X) also bought (Y)), is a seminal use case for frequent pattern mining.



Figure 31-1 Frequency pattern mining, customers who bought (X) bought (Y)

In general, however, the following process is observed towards delivering a given machine learning routine:

- Generally, machine learning routines fall into one of two categories; supervised, or unsupervised.

Note: Coarsely, unsupervised machine learning differs from supervised in that the input columns are not labeled. A seminal unsupervised machine use case is clustering. Given, for example, 20 or so distinct, numeric measures I know about (customers); how are those customers segmented ?

A classic use case for clustering is to discover the potential for a customer to churn; What (cross section of those 20 numeric measures) best identifies customers who wish to discontinue use of my service ?

- Each of the machine learning routines, expects input data in a given format.

For frequency pattern mining, data is expected to be in a single column, comma delimited. Munging the data, getting the data into this format is normally step 1 towards delivering a model.

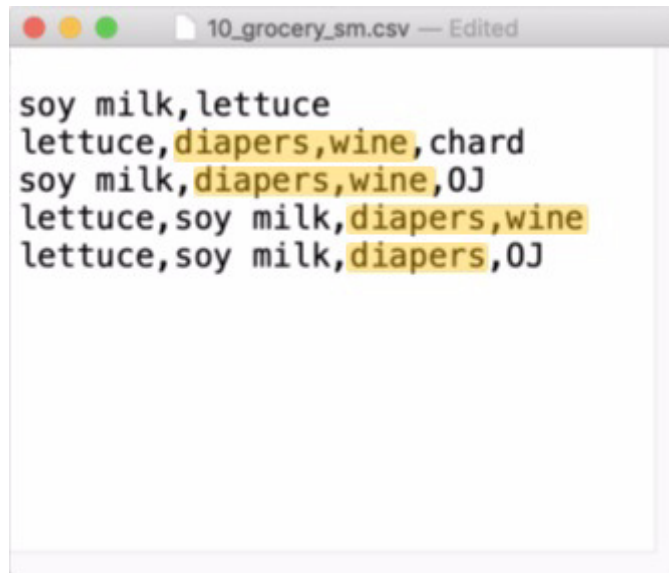


Figure 31-2 Small sized sample data set for frequency pattern mining

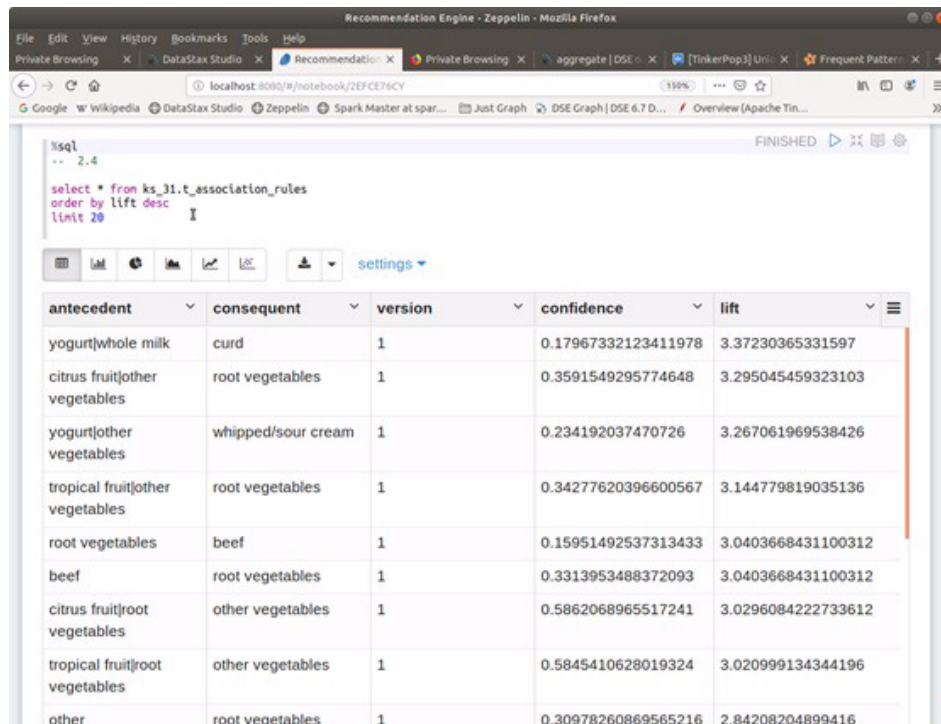
- Then we 'build a model'.

In the case of frequency pattern mining, this is two lines of code; create an (empty) model object with given parameters, then 'fit', or calculate the model.

What is 'the model' ?

Per each of the machine learning routines, a different data structure is created that serves as (the answer); whatever wisdom the given machine learning routine promises to offer.

For frequency pattern mining, the model looks similar to;



The screenshot shows a Zeppelin notebook window titled 'Recommendation Engine - Zeppelin - Mozilla Firefox'. The notebook contains a SQL query that has been executed, resulting in a table of association rules. The query is: `select * from ks_31.t_association_rules order by lift desc limit 20`. The results table has five columns: antecedent, consequent, version, confidence, and lift. The data is sorted by lift in descending order.

antecedent	consequent	version	confidence	lift
yogurt whole milk	curd	1	0.17967332123411978	3.37230365331597
citrus fruit other vegetables	root vegetables	1	0.3591549295774648	3.295045459323103
yogurt other vegetables	whipped/sour cream	1	0.234192037470726	3.267061969538426
tropical fruit other vegetables	root vegetables	1	0.34277620396600567	3.144779819035136
root vegetables	beef	1	0.15951492537313433	3.0403668431100312
beef	root vegetables	1	0.3313953488372093	3.0403668431100312
citrus fruit root vegetables	other vegetables	1	0.5862068965517241	3.0296084222733612
tropical fruit root vegetables	other vegetables	1	0.5845410628019324	3.020999134344196
other	root vegetables	1	0.30978260869565216	2.84208204899416

Figure 31-3 Model created from frequency pattern mining

Relative to Figure 31-3, the following is offered:

- In the context of (customers who bought (X) bought (Y); frequency pattern mining), antecedent is what you already have in your cart. Consequent is the item we would recommend you also purchase. This entire data set, the model produced when frequency pattern mining, is most properly referred to as the, 'association rules'.
- Three additional statistics accompany each 'association rule';
 - 'Support', (not displayed) is used to calculate each of confidence and lift. For example; given 100 total orders, and a given item (or item pair, yadda) appears in 20 of those 100 orders, the support for this item is said to be 20 / 100, or 80%. Items with low support (or low confidence, detailed next) happen infrequently, are can be dropped from the model for efficiency.
 - 'Confidence' is also a calculated using a simple mathematical expression, and we'll use specific data from Figure 31-2 above to detail this-

In Figure 31-2, there are 5 total orders. The item-pair (diapers/wine) appears in 3 orders, and diapers alone appear in 4 orders. Confidence for diapers relative to diapers/wine is defined as,

Support (diapers-wine) / Support (diapers)

$(3/5) / (4/5) == 75\%$

So, 75% of the time we see diapers, we also sell wine.

-- Confidence is useful/interesting, but 'lift' is the most important statistic relative to frequency pattern mining.

In Figure 31-3, we see lift values 2 and above (because we sorted this data set).

Note: Generally a lift greater than 1 indicates a useful association rule; one that should bear results.

A lift equal to 1 means the antecedent and consequent are independent of one another, and no association rule should be drawn.

A lift less than 1 informs of a negative effect between the antecedent and consequent. In fact, it is likely (not conclusive) that the items are substitutes for one another.

Lift is calculated as,

$\text{Support}(A \cup B) / (\text{Support}(A) * \text{Support}(B))$

So for Lettuce-Soy -> Wine the math is,

$0.2 / (0.4 * 0.6) == 0.83$

And we see from Figure 31-2 that wine added to lettuce-soy is infrequent, compared to other lettuce-soy sales.

Note: We added the 'version' column ourselves.

Overtime, you may generate new, more accurate models, and wish to preserve an additional column to identify new models as such.

You could also have models to distinct subsets of users; a model for just Oregon, etcetera.

- 'Scoring' is the process of using, or applying the model.

Figure 31-4 displays the simple SQL/CQL statement we can use to retrieve the recommendation (consequent) given any antecedent. A

primary key lookup using a hash based index, response time should reliably be 0-8 milliseconds.

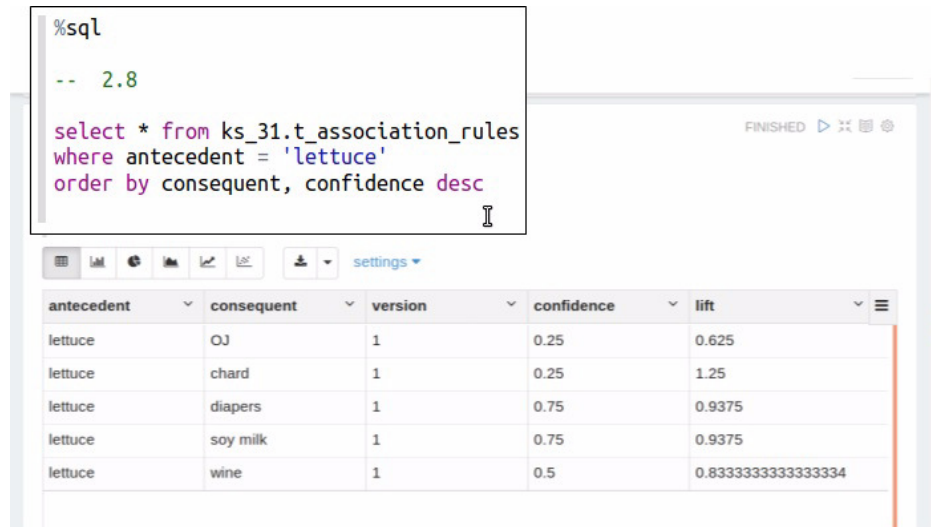


Figure 31-4 Applying (scoring) the model

Product Recommendation using Graph

The previous product recommendation, using frequency pattern mining, is labeled, 'content based', aka, an item-item recommendation. People buy flashlights, and also then buy batteries; makes sense. But what if you don't have this data because it doesn't exist yet; the product hasn't been released yet, so there is no item-item data.

Graph can deliver a collaborative based recommendation. Comments;

- The movie (product) isn't released yet; only reviewers have seen the film.
- You have 100 friends, but 20 who really seem to like the same movies that you like, and this group of friends (with similar movie likes as you), like the same things as this reviewer.
- You haven't seen this film yet, perhaps you should ?

Is Graph Inherently Different than Relational ?

Our answer to this question is; not in the manner you may assume.

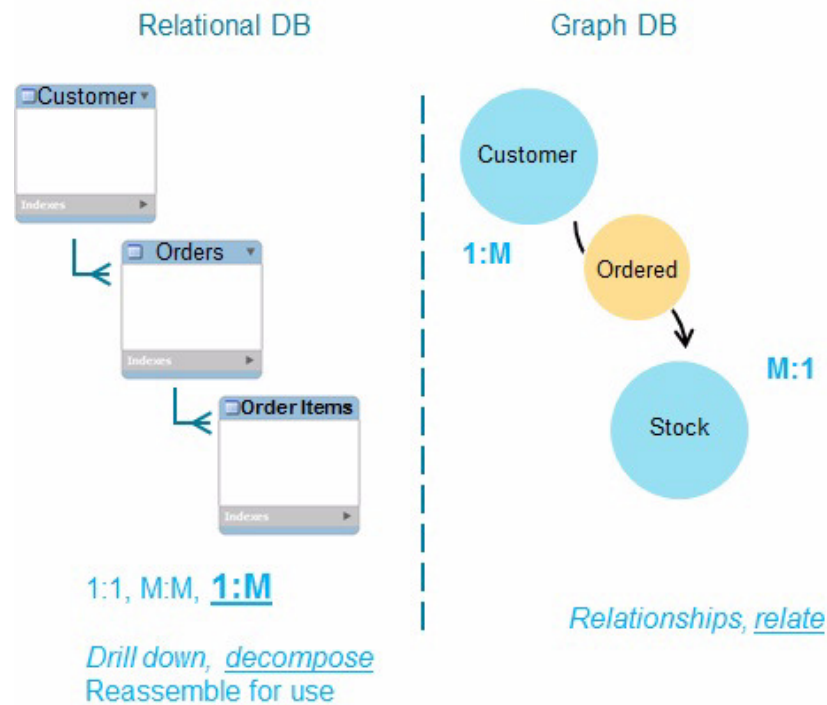


Figure 31-5 Data modeling, relational and graph databases

Generally we state;

- Tables in a relational database have either a one to one relationship, one to many, or many to many.

One to one relationships are rare (another talk for another time), and many to many relationships may indicate an error in modeling. 98% of the time, tables in a relational model have a one to many relationship.

The point is, in general, relational data models pursue a decomposition of a dataset; a customer places one or more orders, each order is composed of one or more items. 400-4000 tables later, you have a database.

Relational databases seek a drill-down of data.

- Graph databases seek to model many to many relationships. The primary entities in graph databases (the nouns) are titled 'vertices', and are stored as regular SQL/CQL tables.

Between vertices, are 'edges' (also SQL/CQL tables), and edges store the join pairs (between two tables, between two vertices). Where vertices are

named after nouns, edges are normally named after verbs; customers (vertex) place (edge) orders (vertex).

In effect, tables are pre-joined in a graph database; a performance optimization.

Note: If the vertices in a graph database are pre-joined, doesn't that limit the analysis you can perform ?

Not really. (No.)

Why would you ever seek you join a customer with orders they did not place ?!

- (Things) relational databases share with graph databases ?

Under the covers, everything is still a table with columns and rows.

How graph databases do differ from relational databases is in the area of query language. Figure 31-6 introduces the SQL SELECT statement, with a code review to follow.

```
SELECT *  
FROM t1, t2  
WHERE  
    t1.col1 = t2.col2  
AND  
    t1.col4 = "X"  
AND  
    t2.col5 = "Y";
```

```
GROUP BY ..  
HAVING ..  
ORDER BY ..  
INTO ..
```

- Access method (per table)
- Join method
- Table order
- FJS; Filter, Join, Sort



Figure 31-6 SQL SELECT, the crown jewel of relational databases

Relative to Figure 31-6, the following is offered:

- The 'SELECT' statement is core to relational, and specifically SQL, and delivers all of the query and analytics capability.

- As a single command, SELECT has 7 clauses; 2 mandatory and 5 optional.
 - The 2 mandatory clauses include: (SELECT) 'column list', and 'from' table list.
 - The 5 optional clauses (WHERE, GROUP BY, HAVING, ..), if present, may appear only one time and in sequence.
If you need to GROUP BY, and then GROUP BY again, you're out of luck with SELECT.
- There are 'CTEs' (common table expressions) above SELECT. Largely, however, these allow set processing; intersects, other.

Officially SQL SELECT is a declarative programming language; tell the server what you want, and not how to do it.

Gremlin, the Graph Database Query Language

Gremlin is the Apache TinkerPop query language; both read and write. In other words, Gremlin is a replacement for SQL SELECT, when using graph.

Where SELECT has 7 clauses, Gremlin has 40 or more steps, and then also step-modifiers. Why is Gremlin presumably bigger ?

- Where SELECT is declarative, Gremlin as a language, is imperative, functional, and also associative. Gremlin is also fluent.
 - Imperative means Gremlin can tell the server how to process the data; not just what to do with the data.
By example (general programming), you could use Gremlin to set up nested for-loops, take distinct action, based on the data values, on each interaction of the loop, yadda.
SQL can't do this.
 - Scala is perhaps the most famous 'functional' programming language, a topic we will table at this time.
 - And Gremlin has associative capability.
You can, with Gremlin say; "Look, there's 400 tables in this graph. You, the database, should tell me how these two entities are related, without me specifically having to ask-"
SQL can't do this.
 - And then 'fluent'; Gremlin steps are merely appended (in any order, and allowing repeats of steps (verbs)), to form a query.
- Gremlin is just more capable. As such, it takes longer to learn Gremlin, compared to SELECT.

As Gremlin is more programmable (it resembles programming), it is possible that not every SQL SELECT programmer will easily adopt Gremlin; Gremlin is a larger/wider skill set.

A Sample Gremlin traversal (query)

Figure 31-7 displays a simple Gremlin traversal. A code review follows.

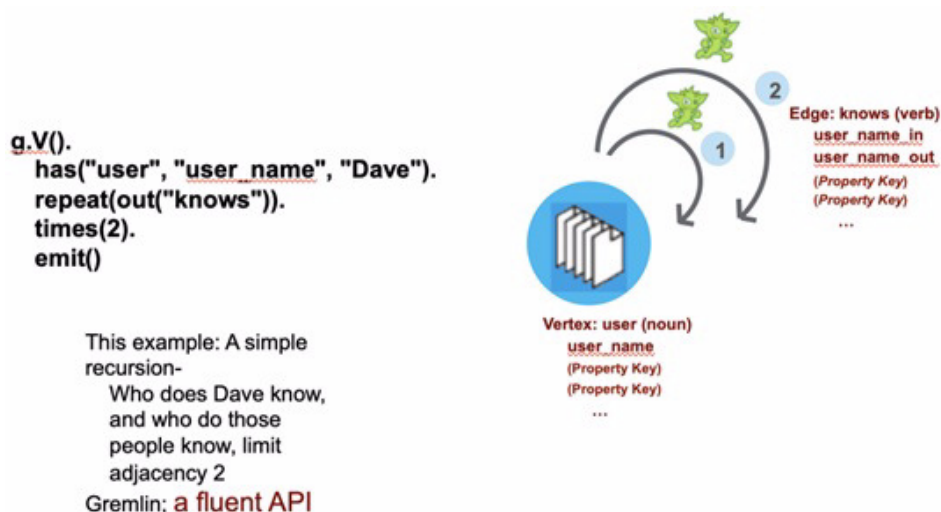


Figure 31-7 A sample Gremlin traversal (query)

Relative to Figure 31-7, the following is offered:

- “g” above is a reference to the database (graph) connection handle; which of several graphs might I be pointing to, which is in scope.
 - “V()” is a reference to all vertices contained in the graph. We could also start on an edge using, “E()”.
- Ending the query after V(), would return all rows from all vertices.
- “has” is a step, and in this use has 3 arguments-
 - The name of the vertex to start a traversal from, “user”.
 - We could list multiple vertex names here.
 - Then “user_name”, a property key (column) to apply a filter (query predicate) to. By default we perform and equality.
 - And then “Dave” the query predicate value.
 - So combined; query the “user” table for all “user_name” equal to “Dave”.

- The next step is actually “out”-
 - Out is a “vertex traversal step”, which calls to, move from one vertex to another (through an edge).

Note: You always move vertex to edge, to vertex, repeat; no exceptions.

However, based on your specific ‘traversal step’, you may stop on an edge, or merely pass through it.

You can go (out) from a vertex, or in, and there is a distinction there; more to come.

- ‘out()’ names the edge to leave the vertex from, in this case, “knows”, (users know other users). As a recursive join, we land back on ‘user’ albeit, in a different context.

Note: “different context” ?!

When you leave the ‘user’ vertex above, on the edge ‘knows’, the specific set of rows, in scope, on the far side of the edge, are only the users you know; not all users.

- “out()” was wrapped in a repeat(), “times(2)”.
- So, in effect; who do you know, and then, who do those people know; your circle of friends, to a depth of two.
- And “emit()”.

The Model we use; KillrVideo

The data model (graph model) we use in most of these examples come from the demonstration database common to DataStax and available at the DataStax Academy, a free training site. Model as pictured in Figure 31-8.

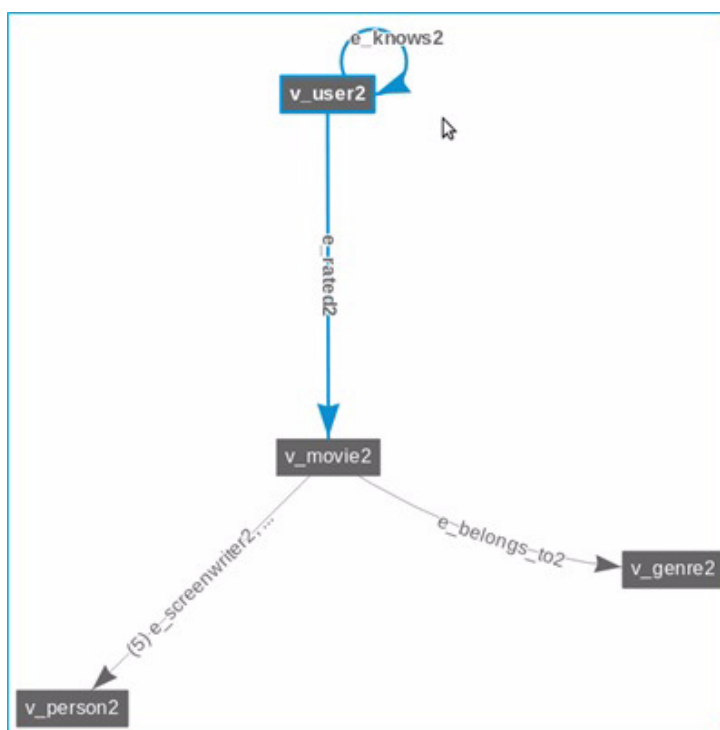


Figure 31-8 KillrVideo graph schema

Relative to Figure 31-8, the following is offered:

- The KillrVideo graph has 4 vertices (user, movie, person, and genre) and many more edges.
Users know other users, and users rate movies. For our purposes today, we won't need/use the other portions of this graph database.
- The 'rated' edge itself has the join pairs (primary keys), between user and movie; users rate movies. As part of a (user having rated a movie), there is also a property on the edge; what rating (number) did the user give the said movie.

31.2 Complete the following

At this point in this document we have completed a given length primer on product recommendation engines using DataStax Enterprise (DSE) Analytics (Apache Spark), and then DSE Graph (Apache TinkerPop/Gremlin).

31.2.1 Running Frequency Pattern Mining, writing to a table

Time now for the code; actually delivering this, actually running it. Example 31-1 lists the code, and a code review follows.

Example 31-1 Scala language, delivering frequency pattern mining

```
// 3.1

import org.apache.spark.ml.fpm.FPGrowth
//
import scala.collection.mutable.WrappedArray;

val recs_1 = sc.textFile("file:///mnt/hgfs/My.20/MyShare_1/44 Topics_2019/41
Graph Reco Eng/02 Files/20_grocery_lg.csv")
val recs_1s = recs_1.map(i_str => i_str.split(",")).toDF("items")

val my_fpgrowth1 = new
FPGrowth().setItemsCol("items").setMinSupport(0.01).setMinConfidence(0.01)
val my_model1 = my_fpgrowth1.fit(recs_1s)

val recs_1_ar = my_model1.associationRules.collect()

val recs_1_ar1 = recs_1_ar.map( row => (
    row.get(0).asInstanceOf[WrappedArray[WrappedArray[String]]].mkString("|"),
    row.get(1).asInstanceOf[WrappedArray[WrappedArray[String]]].mkString("|"),
    row.getDouble(2),
    row.getDouble(3))
).toList

val recs_1_ar1df = recs_1_ar1.toDF("antecedent", "consequent", "confidence",
"lift")

val recs_1_ar1df2 = recs_1_ar1df.withColumn("version", lit(1.0) )

recs_1_ar1df2.write.
    format("org.apache.spark.sql.cassandra").
    options(Map( "keyspace" -> "ks_31", "table" -> "t_association_rules" )).
    mode("append").
    save

recs_1_ar1df2.show(20)
```

Relative to Example 31-1, the following is offered:

- The first two lines call to import two libraries we require for execution.
 - FPGrowth, is the machine learning library proper.
 - WrappedArray will allow us to take arrays and cast them into strings.
- The 'val recs_1' line calls to read the source (item/grocery) data from an ASCII text, command separated value list file.
- The 'val my_fpgrowth1' line instantiates a new model object with the given (requested) confidence and support levels.
- The 'val my_model1' line actually calculates the model.

We have nearly 10,000 grocery line items in the large file, and on a laptop, calculating the model is easily sub-second. This 10,000 line file generates 56 association rules.
- The remainder of these lines are pure Scala, used to mung the model into exactly what we want to store in a database table.
 - The line with the 'collect()' method pulls just the association rules into a new object for our use.
 - The line with 'map()' transforms the 4 return columns from collect(), and cast them into strings and floating point numbers; this is how we prefer to store this data in our table.
 - The 'toDF()' line restores the schema to our data set, which map destroyed.
 - The 'withColumn()' line adds our (version number) to this data set.
 - The 'write()' line actually writes to the database table.

31.2.2 Graph Product Recommendation

This document, the program code, and data, are available at,
trinyurl.com/ddn3000

And specifically you want to look at the July-2019 set of artifacts.

Why do we mention this ?

At that location is a 2.5 hour video that fully details how to program the single Gremlin traversal that delivers a collaborative product recommendation. And there are the slides also, in PowerPoint format.

Continuing here; we'll detail this single Gremlin traversal, albeit with perhaps a bit less detail. (For full detail, see the video.)

Example 31-2 delivers the collaborative product recommendation using Graph/Gremlin. A code review follows.

Example 31-2 Collaborative product recommendation using Gremlin/Graph

```
// 11.1

def l_user = "u1"
def l_conn = 2
def l_rate = 8

g.withSack(1,sum).
  V().
  has("v_user2", "user_id", l_user).
  sideEffect(
    out("e Rated2").
    aggregate("se_movieSeenByUser")
  ).
  both("e_knows2").
  filter(
    outE("e Rated2").
    has("rating",gte(l_rate)).
    inV().
    inE("e Rated2").
    has("rating",gte(l_rate)).
    outV().
    filter(
      has("user_id", l_user).
      and().
      sack().is(gte(l_conn)))
  ).
  outE("e Rated2").
  has("rating", gte(l_rate)).
  inV().
  dedup().
  where(
    without("se_movieSeenByUser")
  ).
  order().
  by(
    inE("e Rated2").
    values("rating").
    mean(),
    decr).
  limit(5).
  valueMap("movie_id", "title", "year")

// { "movie_id": [ "m278" ], "title": [ "The Simpsons (TV Series)" ], "year": [
"1989" ] },
// { "movie_id": [ "m274" ], "title": [ "One Flew Over the Cuckoos Nest" ],
"year": [ "1975" ] },
// { "movie_id": [ "m13" ], "title": [ "The Pianist" ], "year": [ "2002" ] },
```

```
// { "movie_id": [ "m163" ], "title": [ "The Good", ' the Bad and the Ugly' ],  
"year": [ "1966" ] },  
// { "movie_id": [ "m351" ], "title": [ "Forrest Gump" ], "year": [ "1994" ] }
```

Relative to Example 31-2, the following is offered:

- V().has()

We'll start explaining this traversal from the first line that starts, "V().has()".

Here we start the traversal from a vertex (V()), with the label, "user".

We filter for (the single) user_id equal to whatever value is contained in the variable titled, l_user.

Effect; one row from user. And we're sitting on the user vertex.

- sideEffect()

sideEffect() allows you to store a set of rows, without affecting the current traverser, meaning; from wherever you are, anything within the sideEffect() is stored, like a temporary table, but in memory.

The traversal, continue on as before, as though the sideEffect() did not exist.

As an "out(rated)".aggregate(id), our side effect stores all movie ids we have ever seen, by the identifier titled, id. (We used 'se_movieSeenByUser').

- We continue with a "both()".

We were on user, and both() calls to leave a vertex through the named edge titled, "knows"; which users do you know.

As a both(), we got both users we know, and users who know us (which is actually two different conditions).

As the result of the both(), we are still on the vertex titled, users.

- We continue with a filter().

So that's actually a lot of people, some we may not know well. (Or may not even know, they only know us.)

- We are on users, on a list of people we know, and who also know us.
- We go out on the edge (outE(), which takes us to rating), and remove ratings that are less than 8. (E.g., only give me ratings, from people I know, that are 8 or higher.)
- We're on rating.
- inV()

From rating, the inV() returns to users, in this case, people known, by people I know (friends of friends).

And again, we filter out only highly rated movies, which restricts the user list; only persons who gave high ratings.

- The outV returns to user, and the filter here refers to a sack()

In this exact case, sack() contains a counter, per user, of the people I know, of the number of people we know in common.

Why is sack() containing this counter ?

The very first line of the traversal said this sack() contains, "1, sum", which is a counter.

Sacks, in this case a counter, is local to a traverser, which in this case is a count of the persons a single person knows, who I also know.

- If this sack() (count) is greater than or equal to l_conn keep this pairing, else discard.
- The last inV (to users) is movies; we get a movie list, from people I know well (many friends in common), and from their highly rated movies.
- dedup() we hope is obvious; single movie titles could appear multiple times in this list; remove duplicates.
- The where(without()) refers to our earlier sideEffect(); remove, from the recommendation list, movies I've already seen.
- order(), and limit().

Note: Movies, friends, likes; seems trivial, right ?!

Not at all.

this same pattern, the affinity for two pieces of data is used throughout graph, including but not limited to; entity resolution, fraud. Are these two records the same person ?

We just explained that traversal tersely. If you wish further detail (a picking apart of each step); may we again recommend the adjacent video and PowerPoint deck-

31.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- How to deliver a content-based product recommendation using DSE Analytics.
- How to deliver a collaborative-based product recommendation using DSE Graph.
- How the two above differ; when and why to use.
- And a brief Gremlin graph traversal language primer, (although the video and associated PowerPoint deck offer more detail).

Persons who help this month.

Kiyu Gabriel, Dave Bechberger, Artem Chebotko, 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>