Chapter 11.

December 2017

Welcome to the December 2017 edition of DataStax Developer's Notebook (DDN). This month we answer the following question(s);

> As my company is beginning significant development of new applications that target the DataStax Enterprise database server (DSE), we are starting to explore some of the programability that DSE offers us. Specifically, we are very much interested in user defined functions (UDFs). Can you help ?
>
> *Excellent question ! The topic of server side database programming can devolve into one of those information technology religious arguments, something we generally seek to avoid. Still, DataStax Enterprise (DSE) does offer user defined functions (UDFs), user defined aggregates (UDAs), and user defined types (UDTs).*
>
> *Recall that a core value proposition which DSE delivers is time constant lookups, that is; a CQL (Cassandra query language, similar to structured query language, SQL) SELECT, UPDATE or DELETE that uses an equality on at least the partition key columns from the primary key. These are the linearly scalable operations that DSE is distinguished for. If you seek to do aggregates or other set operands on large data sets (online analytics style processing, OLAP), DSE will likely need to perform a scatter gather query (a query that reads from multiple concurrent nodes). Inherently, these types of queries do not perform in low single digit millisecond time; just be advised.*
>
> *In this edition of this document we will detail all that you ask; user defined functions, and also user defined aggregates and user defined types, and we'll write application code for each.*

Software versions

The primary DataStax software component used in this edition of DDN is DataStax Enterprise (DSE), currently release 5.1. 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 CentOS version 7.0, 64 bit.

11.1 Terms and core concepts

As stated above, ultimately the end goal is to detail and/or make use of user defined functions (UDFs), implying also user defined aggregates (UDAs), and since we keep saying "user defined", let us also detail user defined types (UDTs).

We stepped into a bit of a rat hole on the introduction page above related to online analytic processing (OLAP). Just to clear any of this up we offer:

- Online transaction processing (OLTP) versus online analytic processing (OLAP)-

  In our current context we define OLTP to mean;

  > A SELECT, UPDATE or DELETE requiring a very highly performant and linearly scalable service level agreement. As such, these routines are using a *primary key index retrieval*, and in a multi-node database system are processing data from *one node per service invocation*.

  In our current context we define OLAP to mean;

  > A SELECT generally, but possibly also UPDATEs or DELETEs that need to process wide swaths of data, possibly from all nodes and without support from indexed retrieval.

- DataStax Enterprise (DSE) has four primary areas of functionality-
  - DSE Core, time constant lookups, OLTP
  - DSE Analytics, powered by Apache Spark, OLAP and parallel processing
  - DSE Search, powered by Apache Solr/Lucene
  - DSE Graph, powered by Apache TinkerPop

  To achieve the highest performance for OLAP style processing (preferably parallel processing), you will likely want to use the Analytics area of functionality within DSE, powered by Apache Spark. (We ran some Analytics programming in last month's edition of this document using machine learning, predictive analytics.)

Note: And while we have yet to define DataStax Enterprise (DSE) user defined functions and aggregates, we will state now that you may run these via DSE Analytics (parallel processing) or via DSE Core.

User defined types (UDTs)

Just to be clear, user defined types (UDTs) in DataStax Enterprise (DSE) are not generally related to user defined functions (UDFs) or user defined aggregates (UDAs). We include the topic of UDTs merely because they include the phrase

"user defined", and we expect the obvious question; are UDTs related to other things user defined.

DSE has the concept of a *tuple*, detailed here,

```
https://docs.datastax.com/en/cql/3.3/cql/cql_reference/tupleType.htm
l
```

In short, a tuple is like a record or struct(ure) from programming languages. And a tuple is very much equal to an unnamed user defined type (an unnamed UDT). Example 11-1 offers code to create a use a DSE user defined type. A code review follows.

Example 11-1 CQL code to create a user defined type (UDT)

```
create type orderLineItem
      (
      itemNum             int,
      quantity            int,
      perUnitPrice        float
      );

   create table customerOrders
      (
      custNum             int,
      custName            text static,
      orderNum            int,
      orderDate           date,
      itemList            frozen<list
                    <orderLineItem>>,
      primary key ((custNum), orderNum,
         orderDate)
      ) with clustering order by (
      orderNum asc, orderDate asc);

   insert into customerOrders
      (custNum, custName, orderNum,
      orderDate, itemList)
   values
      (101, 'Ram Smith', 1, '2005-12-12',
         [ ( 1, 5, 10.0 ), (2, 1, 5.0) ] );

   #  no
   select itemlist.quantity from customerOrders;
   #  yes
   select itemlist          from customerOrders;

   #  no
   select * from customerOrders where
```



```
      itemList.itemnum = 1;

   drop table customerOrders;
   drop type  orderLineItem;
```

Relative to Example 11-1, the following is offered:

- In this example we create a user defined type (a named tuple), and then include this type in a table definition. The net effect becomes we can embed a structure (a set of named and typed columns) inside an array.

  DataStax Enterprise (DSE) has 3 array types (officially titled, collections); set, list and map. Each are documented here,

  ```
  https://docs.datastax.com/en/cql/3.3/cql/cql_using/useCollections.html
  ```

  Sets are unordered but do enforce uniqueness, where *lists* do not enforce uniqueness and are ordered. *Maps* support an array of key/value pairs.

- After the single row insert, we execute three CQL SELECT statements-
  - The first select statement fails. Currently you can not make reference to a sub element of a user defined type in a list (there is currently no supported dot notation for these elements). In effect, the list of user defined types is treated as a blob; a payload to be managed and transported by DSE.
  - The second select succeeds. Currently you can reference the whole of the list.
  - And the third select statement fails, for the same reason the first select statement failed.

    But .. you can select and even index non-tupled (non user defined types) inside sets, lists and maps. See Example 11-2, followed by a code review.

Example 11-2 More sets, lists and maps.

```
create table playlists
      (
      album               text primary key,
      tags                set<text>
      );

   insert into playlists (album, tags)
      values ( 'tres hombres',
         { 'blues', '1973' } );
   insert into playlists (album, tags)
      values ( 'say what',
```



```
      { 'blues', 'rock', '1984' } );
insert into playlists (album, tags)
   values ( 'best of the cure',
      { 'rock', '1984' } );

select * from playlists
where tags contains 'rock'
allow filtering;

create index on playlists (tags);
   #
select * from playlists
where tags contains 'rock';

drop table playlists;

   --

create table playlists
   (
   album               text primary key,
   tags                map<text, text>
   );
insert into playlists (album, tags)
   values ( 'tres hombres',
      { '1973' : 'blues' } );
insert into playlists (album, tags)
   values ( 'say what',
      { '1984' : 'blues', '1977' : 'rock' } );
insert into playlists (album, tags)
   values ( 'best of the cure',
      { '1984' : 'rock' } );

select * from playlists
where tags contains 'blues'
allow filtering;

create index on playlists (tags);

select * from playlists
where tags contains 'blues';

select * from playlists
where tags contains keys '1977';

select * from playlists
where tags contains key '1977'
allow filtering;
```



```
create index on playlists (keys (tags));

select * from playlists
where tags contains key '1977';

drop table playlists;
```

Relative to Example 11-2 the following is offered:

- All of the CQL SELECTs in Example 11-2 work.

  "*allow filtering*" is a DataStax Enterprise (DSE) keyword, a query modifier, that allows a query to run which is not supported via an index. The given query may still time out (be shut down by DSE because is does not complete within a reasonable time frame).

- The first CQL SELECT would fail, if not for the presence of the allow filtering modifier. After the index is created, the CQL SELECT runs with index support.

Note: Even with the presence of the (any) index, these queries could be scatter gather. E.g., these queries may still read data from every node.

Only primary key lookups (queries using an equality on the whole of the partition key) are guaranteed to be time constant, to read data from just one node using an index.

- The last two index create statements detail that using a map collection type, you may choose to index the values in a key value array, or the keys, or both values and keys.

Note: The following Jira,

`https://issues.apache.org/jira/browse/CASSANDRA-7396`

is one of many that track query completeness in the area of user defined types, sets, lists, and maps.

User defined functions (UDFs)

What is a user defined function (UDF), and what is a user defined aggregate (UDA) ? In short, UDFs are like scalars, or like macros. Further:

- UDFs offer a one to one input/output response, that is; every single or set of (variables) inputted to a UDF produces one return value. You can input or return tuples, however; it is still one input, one output.

Contrast this to UDAs, which may output an equal or fewer number of (rows) per (rows) input.

– Example use cases for UDFs might include:
  - Calculate a check digit for an input column
  - Perform string manipulations; upshift, downshift, etcetera
  - Perform math
  - Other

Note: If you will attempt to run these examples in line (in the order produced in this document), you need to change some defaults. In the `cassandra.yaml` file you must change,

```
enable_user_defined_functions: true
enable_scripted_user_defined_functions: true
```

The first line above enables user defined functions using Java. The second line adds support for user defined functions using JavaScript. (The default values were both, false.)

As parameters in a configuration file, you will need to restart your DataStax Enterprise (DSE) server in order that these changes may take effect.

Example 11-3 offers the first example user defined functions. A code review follows.

Example 11-3 User defined function example

```
create keyspace ks_12
      WITH replication =
      {'class': 'SimpleStrategy',
      'replication_factor': 1};

   use ks_12;

   create table t1
      (
      col1 text primary key,
      col2 text,
      col3 text,
      col4 text
      );

   insert into t1 (col1, col2, col3, col4)
      values ('aaa', 'aaa', 'aaa', 'aaa');
```



```
insert into t1 (col1, col2, col3, col4)
   values ('bbb', 'bbb', 'bbb', 'bbb');
insert into t1 (col1, col2, col3, col4)
   values ('ccc', 'ccc', 'ccc', 'ccc');
insert into t1 (col1, col2, col3, col4)
   values ('ddd', 'ddd', 'ddd', 'ddd');

#// fyi, these work
select max(col1) from t1 allow filtering;

select max(col1)            from t1;
select max(col1), min(col1) from t1;


create or replace function my_upshift( i_arg1 text )
   returns null on null input
   returns text
   language java as '

   return "AAA";

   ' ;

select my_upshift('ooo') from t1;
select my_upshift(col1 ) from t1;


create or replace function my_upshift( i_arg1 text )
   returns null on null input
   returns text
   language java as '

   return i_arg1;

   ' ;


create or replace function my_upshift( i_arg1 text )
   returns null on null input
   returns text
   language java as '

   return i_arg1.toUpperCase();

   ' ;

create or replace function my_upshift( i_arg1 text )
   returns null on null input
   returns text
```



```
      language java as $$

      return i_arg1.toUpperCase();

      $$ ;
```

Relative to Example 11-3, the following is offered:

- First we create a keyspace and table, and add 4 rows of data on which we may operate.
- The first three CQL SELECTs work, using the existing/built in aggregate function titled, max(); return the maximum value of. Note that max() can operate on strings, which is not the case using every database server.
- The next four "create function" statements are progressive, meaning; they create the same function with increasing purpose.
  - The first create function returns a single/constant hard coded string.
  - The two CQL SELECTs that follow detail how to invoke/test this named function.
  - The second create function returns the input parameter unaltered. And the third create function actually folds the input parameter to upper case text, as expected by the function name.
  - The fourth create function uses an alternate syntax; the Java source code body is enclosed in pairs of "$$" characters, versus single left quotation marks per the first three examples.

Note: At this point you can create user defined functions (UDFs) using Java. Go forth, conquer, write Java.

Example 11-4 offers more invocations of the newly create user defined function. A code review follows.

Example 11-4 More example invocations of a user defined function

```
insert into t1 (col1, col2, col3, col4)
      values ('eee', my_upshift('eee'), 'eee', 'eee');

   select * from t1;
   update t1 set col2 = my_upshift(col2)
      where col1 = 'aaa';    // No
   update t1 set col2 = 'ggg'
      where col1 = 'aaa';    // Yes
   update t1 set col2 = 'aaa'
```

```
   where col1 = 'aaa';    // Yes
select * from t1;

update t1 set col2 = 'aaa'
   where my_upshift(col1) = 'AAA';    // No
update t1 set col2 = 'aaa'
   where col1 = my_upshift('aaa');    // Yes

   //   fyi; upsert above, row did not exist
```

Relative to Example 11-4, the following is offered:

- The first insert works as the user defined function is applied to a literal value, inserted into a table.
- The first and fourth updates fail.

  Recognize from the syntax that the user defined function would be applied to every of what could be hundreds of thousands of input data rows. This becomes less of a "we can not do this" and becomes more of a "do you really want to do this", meaning; the requested operation could be very costly, if we chose to support it.
- The last update does work, as it is applied to a constant/literal.

  fyi: This update is actually an insert (upsert), since this primary key value did not previously exist.

User defined functions written in non-Java

Thus far we have written all of our user defined function using Java. Example 11-5 provides the same upshift user defined function written in JavaScript. A code review follows.

Example 11-5 User defined function written in JavaScript.

```
create or replace function my_upshift( i_arg1 text )
      returns null on null input
      returns text
      language javascript as $$

      i_arg1.toUpperCase();

      $$ ;
```

Relative to Example 11-5, the following is offered:

- While JavaScript does have a return statement, the convention when writing user defined functions using JavaScript is to state the return variable, which calls to evaluate its value.
- And there exists a very strong recommendation to not use languages other than Java for production implementations of user defined functions.

Handling null values when using user defined functions

The concept of null values was new to the domain of relational databases circa 1970 or so. Null is not a value, but the means to display that the value is not known. If you call to average four values, (2, 4, 6, and null), the most correct answer is null; null is unknown, and an average including an unknown is itself unknown.

Or, logically you could treat null as zero. Your business rules decide.

Example 11-6 offers another create function example. A code review follows.

Example 11-6 User defined function example using nulls.

```
create table t2 (col1 int primary key, col2 int, col3 int);

   insert into t2 (col1, col2, col3) values (1, 2, 3);
   insert into t2 (col1, col2, col3) values (2, 6, 4);
   insert into t2 (col1, col2, col3) values (3, 7, 7);

   create or replace function larger_of( i_arg1 int, i_arg2 int )
      called on null input
      returns int
      language java as $$

      if (i_arg1 == null)
         i_arg1 = 0;
      if (i_arg2 == null)
         i_arg2 = 0;

      if (i_arg1 > i_arg2)
         return i_arg1;
      else
         return i_arg2;

      $$ ;

   select larger_of(col2, col3) from t2;
```

Relative to Example 11-6, the following is offered:

- We start with a new table, one with integers just because replacing nulls with zeros is fun.
- A user defined function may be exclusively defined as

  ```
  returns null on null input
  ```

  or

  ```
  called on null input
  ```

  The first statement essentially states that the user defined function is not invoked if any of the input parameters are of the null (unknown) value. The second statement states that the user defined function is invoked in all cases of input value.
- The two "if" blocks change null input values to zero; a business rules decision.
- And then the return.

When to use user defined functions

As stated earlier, the topic of server side functional programming can become a bit of a religious argument. We state the following:

- A defense of server side programming would include the ability to deliver measurable performance gains, which is most likely achieved through a significant reduction in node to node (server to server), or server to client communication overhead (reduce the amount of bytes travelling).

  Since user defined functions offer a largely one input to one output behavior, reduction in communication overhead is not likely here.
- But, not previously defined, user defined functions are required for creating and using user defined aggregates, and user defined aggregates can be used to reduce client server communication volume.

Note: Another argument pro server side programming would be to create a central point to apply lightweight application logic, a point nearest the data.

If it was 30+ years ago, before the arrival of strong software build and versioning systems, we might support this argument more. Today being today, not so much.

User Defined Aggregates (UDAs)

Above we discussed when to use user defined functions (UDFs). Minimally, UDFs allow (are required by) user defined aggregates (UDAs), and there is a strong justification for using UDAs that includes:

- UDAs can be used to greatly reduce the client server communication overhead; reduce the number of bytes sent over the network.
- UDAs can be used to deliver aggregate calculations beyond those provided for by the current CQL GROUP BY capability.
- Other

You could view the CQL create aggregate command as a *wrapper*, that identifies and configures the use of one of more user defined functions. Let us use an example as the quickest means to explain. Example 11-7 offers our first user defined aggregate. A code review follows.

Example 11-7 First user defined aggregate.

```
create table t3
      (
      my_pk           int primary key,
      st_abbr         text,
      st_pop          int
      );

   insert into t3 (my_pk, st_abbr, st_pop)
      values (1, 'AK',   300);
   insert into t3 (my_pk, st_abbr, st_pop)
      values (2, 'CO',   300);
   insert into t3 (my_pk, st_abbr, st_pop)
      values (3, 'CO',   600);
   insert into t3 (my_pk, st_abbr, st_pop)
      values (4, 'WI',   100);
   insert into t3 (my_pk, st_abbr, st_pop)
      values (5, 'WI',   200);
   insert into t3 (my_pk, st_abbr, st_pop)
      values (6, 'UT',    50);

   works
      #
   select sum(st_pop) from t3;
   select sum(st_pop) from t3 group by my_pk;

   fails
      #
   select sum(st_pop) from t3 group by st_abbr;
      #
   InvalidRequest: Error from server: code=2200
      [Invalid query]
   message="Group by is currently only supported
      on the columns of the PRIMARY KEY, got st_abbr"

   fails
```



```
   #
select sum(st_pop) from t3 group by my_pk, st_abbr;


create table t3
   (
   my_pk           int,
   st_abbr         text,
   st_pop          int,
   primary key (st_abbr, my_pk)
   );

works  (with warning)
   #
select sum(st_pop) from t3 group by st_abbr;

fails
   #
select sum(st_pop) from t3 group by st_abbr;

   --

insert into t3 (my_pk, st_abbr, st_pop)
   values (1, 'AK',   300);
insert into t3 (my_pk, st_abbr, st_pop)
   values (2, 'CO',   300);
insert into t3 (my_pk, st_abbr, st_pop)
   values (3, 'CO',   600);
insert into t3 (my_pk, st_abbr, st_pop)
   values (4, 'WI',   100);
insert into t3 (my_pk, st_abbr, st_pop)
   values (5, 'WI',   200);
insert into t3 (my_pk, st_abbr, st_pop)
   values (6, 'UT',    50);

create or replace function my_agg_state(
   state map<text, int>, st_abbr text, pop int)
returns null on null input
returns map<text, int>
language java as $$

   state.put(st_abbr, pop);

return state;

$$ ;

To run the above now
   #
```



```
select my_agg_state({'fff' : 6 },
   st_abbr, st_pop) from t3;


create or replace aggregate my_agg(text, int)
sfunc my_agg_state
stype map<text, int>
initcond {};

select my_agg(st_abbr, st_pop) from t3;

Outputs
   #
{'AK': 300, 'CO': 600, 'UT': 50, 'WI': 200}


create or replace function my_agg_state(
   state map<text, int>, st_abbr text, pop int)
returns null on null input
returns map<text, int>
language java as $$

   if(state.containsKey(st_abbr))
      state.put(st_abbr,
        (int)state.get(st_abbr) + pop);
   else
      state.put(st_abbr, pop);

return state;

$$ ;

select my_agg(st_abbr, st_pop) from t3;

{'AK': 300, 'CO': 900, 'UT': 50, 'WI': 300}


   --
   --


create or replace function my_agg_state(
   state map<text, int>, st_abbr text, pop int)
returns null on null input
returns map<text, int>
language java as $$

   if(state.containsKey(st_abbr))
      state.put(st_abbr,
```



```
         (int)state.get(st_abbr) + pop);
   else
      state.put(st_abbr, pop);

return state;

$$ ;

create or replace function my_agg_final(
   state map<text, int>)
returns null on null input
returns map<text, int>
language java as $$

int    l_pop =  0 ;
String l_key = "0";
   //
for (Map.Entry<String, Integer> entry :
      state.entrySet()) {
   Integer l_value = entry.getValue();
      //
   if (l_value > l_pop) {
      l_pop = l_value;
      l_key = entry.getKey();
      }
   }

Map<String, Integer> r_state = new HashMap();
   //
r_state.put(l_key, l_pop);

return r_state;

$$ ;

create or replace aggregate my_agg(text, int)
sfunc my_agg_state
stype map<text, int>
finalfunc my_agg_final
initcond {};

select my_agg(st_abbr, st_pop) from t3;
```

Relative to Example 11-7, the following is offered:

- First we create a new table, presumably states (provinces) in the United States with population numbers.

Note that we have multiple population counts (additive) per identifier, per state abbreviation.

– The first select with a built in aggregate expression works, even without reference to a primary key value. Functionally this select delivers a grand total summation of all rows.

Note: The select above demonstrates reduction of server to client side communication. Instead of returning possibly hundreds of thousands of rows, the select above returns one row with a grand total.

The second select also works, grouping by the primary key value.

Note: This second query is kind of nonsense since we have a single column primary key; we are grouping only one row per primary key by definition. There is no observed savings and what are we actually achieving via the call to sum(). (Nothing.)

– The next two selects fail (selects three and four overall). Comments:

• The third select fails because of the group by clause.

CQL GROUP BY SELECTS must be on the whole of the partitioning key. This table is not partitioned by st_abbr, hence the fail.

Note: To overcome this issue, you could model your table to be partitioned on that single or set of columns you wish to GROUP BY on, or, use a user defined aggregate.

Keep in mind, these are two completely different approaches-

Partitioning data by the group by key affects data storage, and will perform optimally. Use this approach if you can; if the frequency of satisfying this query demands it.

Using a user defined aggregate on a non-anchored key (not using the partition key) will like execute a scatter gather query.

• The fourth select also fails because of the GROUP BY clause.

Here we do include the primary key (the partitioning key), but we also add a key which is not a member of the primary key, st_abbr.

The solution is the same as above; either model the table to support the requisite GROUP BY, or use a user defined aggregate.

- The second create table changes our model, and our primary key to a primary key which supports the GROUP BY clauses we require to deliver our business rules; our intended query.
- And then our first create user defined function, for a user defined function to expectedly be called via a user defined aggregate.
  - Three input arguments are sent to this user defined function-

    The second and third arguments are titled, st_abbr and pop (population); expectedly the state identifier and the number of persons residing in that state.

    These two input parameters should make sense. These are the key and the value we wish to GROUP BY on.

    The first input parameter titled, state, may not make sense at this point.

    User defined functions are stateless, meaning; they receive input parameters, do work (make calculations), and then they exit forgetting all about what they just did. This is all fine if you doing something like upshift of a string value. This is not fine if you wish to calculate a running sum or average across rows.

Note: So user defined functions are stateless. Period

How we maintain state from invocation of user defined function invocation to function invocation is we pass it; we pass the state as a single or set of variables from call to call. Since state, as a variable, likely has many parts (st_abbr and pop), we pass state as a map; an array of many values.

Since this variable passes state, we title it "state".

  - The select statement that follows this user defined function creation displays how you could call this function manually; outside of using this function with a user defined aggregate. (This is the select with the "fff" value.)

    Since this user defined function passes state (when used with a user defined aggregate), and we have no state (we are manually calling this function one time), we pass a dummy first argument for state; a map with the value "{'fff', 6"}". We could have passed a 2 attribute map with any values here, the values are just dummy place holders.
- And then we have the first create aggregate statement-

  Notice that the create aggregate has no body of its own; no procedural or Java program code.

A user defined aggregate is a wrapper of sorts, defining how its single or set of member user defined functions work together to deliver the aggregate calculation. More-

- stype,

  When you see the letter "s" in reference to a user defined aggregate, "s" stands for "state".

  stype defines the map (the count and data type) of columns passed between each invocation of the (calculating) user defined function.

- sfunc,

  is the name of the actual user defined function called between each iteration; between each row from the CQL select called for by this user defined aggregate.

- There are more clauses to the user defined aggregate, not defined or used in this example. (We do cover these below in larger and final examples.)

Note: Notice the sample output-

The source table has 6 rows, but the user defined aggregate (which calls a nearly empty user defined function, this function is basically a placeholder) outputs only 4 rows. Why ?

Its because of the key to the map.

The key to the map is the (US) state identifier. We have two records for CO/Colorado, and two for WI/Wisconsin, and any second or subsequent row with this key overwrites the (first).

We will fix this condition below as we complete the user defined function; actually call for this function to do real work.

Note: If I'm going to be writing real Java code at some point, what is the actual Java data type being passed for state ?

It is java.util.HashMap, and its inclusion include all Java methods associated with this object. See this Url for more on HashMap as a Java topic,

```
https://www.tutorialspoint.com/java/util/java_util_hashmap.htm
```

– The next user defined function create statement (the first with an "if" statements), follows.

This user defined function properly checks for the presence of a key in the state map, increments it if found, and just puts into it if not found.

Here we have a fully working "sum the state total per state identifier" user defined function and aggregate pair.

Notice the output, which now correctly (groups) states and adds their collective population values.

Note: At this point in the document, we have a complete and fully functioning user defined function and user defined aggregate pair. We provide the GROUP BY state and sum total population function, a usable routine.

From this point forward we just build on this example to further flesh out the complete function of user defined aggregates.

Note: At this point in this document we have stated nothing about the run time environment for user defined functions, or user defined aggregates-

> *User defined aggregates run as a single instance on one coordinator node, possibly co-located with the source data (when reading from a single node), or most likely not (when reading from multiple scatter gather nodes).*

Why ?

Well, look at the output. If we ran multiple and concurrent (parallel) instances of this (user defined function), we would need a data exchange layer and more. (We'd basically need to replicate a good portion the Apache Spark runtime.)

If you need/want parallelism, run this user defined function, user defined aggregate using DSE Analytics (using Apache Spark).

– We enter the final section of Example 11-7, which is really designed to accomplish one goal; display how to use the additional clause of a user defined aggregate titled, finalfunc.

  - The first user defined function titled, my_agg_state, is unchanged from its earlier listing.
  - The user defined function titled, my_agg_final, is new.

    This user defined function uses Java to produce only the single, and largest value from the state map, in effect; output only one state, the one with the largest population total.

Java HashMaps have no sort function, so we do this work manually and simply using Java.

- The user defined aggregate clause has a new line, specifying which user defined function to call after its source data set is exhausted; E.g., ON LAST ROW.

Thus completes Example 11-7, an end to end user defined aggregate that calculates the sum of population per state, including a final example to output the single and greatest value from this list.

11.2 Complete the following

At this point in this document we have completed a reasonable length primer on DataStax Enterprise (DSE) user defined types, user defined functions, and user defined aggregates. Really the ask was to detail user defined functions, however, user defined functions are largely a means to an end. The real and unique value with user defined functions is to deliver user defined aggregates.

Go forth and conquer.

Write your own and useful user defined aggregate using the details and techniques offered above.

- Use a real data model and data to write a user defined aggregate that delivers value to your business.
- Test your example on a small but real data set, then run it against production sized data.

  If performance is not adequate (because you are running this example serially, and on large data sets), then move your user defined aggregate to DSE Analytics.

  We detailed running a DSE CQL query using Analytics in the prior edition of this document, when we detailed running predictive analytics, A.k.a., machine learning.

11.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- DataStax Enterprise (DSE) user defined types, user defined functions, and user defined aggregates, using Java and JavaScript.
- We delivered a custom US state population sum aggregate, including a final version that outputs only the highest value.

- And we detailed changes necessary to the cassandra.yaml file to support this activity.

Persons who help this month.

Kiyu Gabriel, Paul Mouradian, Jim Hatcher and Thom Valley.

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
```