Chapter 26.

February 2019

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

> I need to program an inventory management system, and wish to use the time stamp, time to live, and other features found within DSE. Can you help ?
>
> *Excellent question ! The design pattern you implement differs when you are selling a distinct inventory (specifically numbered seats to a concert), or you are selling a true-count, number on hand inventory (all items are the same).*
>
> *Regardless, we will cover all of the relevant topics, and detail how to program same using DSE Core and DSE Analytics (Apache Spark).*

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 Desktop version 18.04, 64 bit.

26.1 Terms and core concepts

As stated above, ultimately the end goal is to program an inventory management system. Specifically the problem statement mentioned time stamp, and time to live, so we will proceed as though delivering an on line electronic commerce site, with the ability to place items in a shopping cart as reserved, and then have the items automatically time out and return to stock, should the end user abandon.

Also as stated above, the design pattern matters whether you are:

- Selling a distinct inventory; I.e., uniquely numbered/identified seats to a specific concert.

  You can still use time to live feature we detail below, however, this design pattern is much simpler than the design pattern to sell from a total amount of inventory, where all items in inventory are the same.
- Selling a common inventory; I.e., I have 100 (count) door-buster flat screen televisions, and wish to sell only 100, and not overcommit and sell 140.
- Below we detail all of the topics relevant to implementation at the database tier, and worry less about details relevant to user interface other.

Note: More detail relevant to the inventory management use case is available at,

`https://www.datastax.com/dev/blog/scalable-inventory`

`https://mortimo.wordpress.com/2017/07/03/distributed-inventory/`

Example table moving forward

Example 26-1details a table and data used in the examples moving forwarded. A code review follows.

Example 26-1 Example table and data

```
DROP KEYSPACE IF EXISTS ks_26;

   CREATE KEYSPACE ks_26
      WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
      // AND graph_engine = 'Native'
      ;

   USE ks_26;

      //

   CREATE TABLE t1
```



```
      (
      col1       TEXT,
      col2       TEXT,
      col3       TEXT,
      col4       TEXT,
      PRIMARY KEY ((col1), col2)
      );

   INSERT INTO t1 (col1, col2, col3, col4)
      VALUES ('aaa', 'aaa', 'aaa', 'aaa');
   INSERT INTO t1 (col1, col2, col3, col4)
      VALUES ('bbb', 'bbb', 'bbb', 'bbb');
```

Relative to Example 26-1, the following is offered:

- The partition key to table 't1' is 'col1'. If col1 was an attribute similar to a product sku, we might be concerned about fat-nodes, or hot spotting if a given single sku was under heavy load.
- Otherwise, table t1 is a standard (Hello World), bland, vanilla table.

Using writetime() with CQL

Example 26-2 details use of writetime() using CQL. A code review follows.

Example 26-2 CQL and writetime()

```
SELECT writetime(col1) FROM t1;
   SELECT writetime(col2) FROM t1;

   //  try to read PK,
   //     InvalidRequest: Error from server: code=2200 [Invalid query]
   //     message="Cannot use selection function writeTime on PRIMARY KEY part
col1"
   //
   // SELECT writetime(col3) FROM t1;

   UPDATE t1
      SET col3 = 'aaa'
      WHERE col1 = 'aaa' and col2  = 'aaa';
   UPDATE t1
      SET col3 = 'ccc'
      WHERE col1 = 'bbb' and col2  = 'bbb';

   SELECT writetime(col3) FROM t1;
```

Relative to Example 26-2, the following is offered:

- writetime() is documented here,

  https://docs.datastax.com/en/dse/6.7/cql/cql/cql_using/useWritetime.html

- writetime() is available for non-primary key columns. (Since the partition key is shared across potentially many data rows, writetime() is not available for the partition key.)

Using writeTime using DSE Analytics, (Spark)

Example 26-3 details using writeTime using DSE Analytics, Apache Spark, RDDs. A code review follows.

Example 26-3 writetime() and DSE Analytics

```
import com.datastax.spark.connector._
   import org.apache.spark.sql.SparkSession
   import com.datastax.spark.connector.writer._

   case class MyClass
      (
      col1     :  String,
      col2     :  String,
      col3     :  String,
      col3_ts  :  Long,
      col4     :  String,
      col4_ts  :  Long
      )

   val spark = SparkSession.builder.
      appName("my_app").
      enableHiveSupport().
      getOrCreate()

         //

   val recs_01 = spark.sparkContext.cassandraTable[MyClass]("ks_26", "t1").
      select("col1", "col2", "col3", "col3".writeTime as
      "col3_ts", "col4", "col4".writeTime as "col4_ts")

   recs_01.collect().foreach(println)

   // MyClass(aaa,aaa,aaa,1554494889105696,aaa,1554494171990099)
   // MyClass(bbb,bbb,ccc,1554494889108018,bbb,1554494171995595)
   // recs_01: com.datastax.spark.connector.rdd.CassandraTableScanRDD[MyClass]
=
   //    CassandraTableScanRDD[1] at RDD at CassandraRDD.scala:19
```


Relative to Example 26-3, the following is offered:

- Written in Scala, the case class allows us to apply a schema to our RDD.
- recs_01 uses a SELECT that specifies the (col).writeTime property. As above, time is returned in milliseconds.
- Sample results as shown.

Note: This technique, and more, are documented at,

```
https://github.com/datastax/spark-cassandra-connector/blob/master
/doc/3_selection.md#example-using-select-to-retreive-ttl-and-time
stamp
```

With version 6.7 of DSE, this technique is not supported with DataFrames. This issue is reported with this Jira,

```
https://datastax.jira.com/browse/DSP-17044
```

and is currently planned for release 6.8 of DSE.

Setting writeTime using DSE Analytics, (Spark)

At this point in this document we can read writeTime using CQL and (Spark), now we turn to setting writeTime. Example 26-4 begins to detail this technique. A code review follows.

Example 26-4 Setting writeTime using DSE Analytics

```
case class MyClass2
      (
      col1     :  String,
      col2     :  String,
      col3     :  String,
      col4     :  String,
      my_ts    :  Long
      )

   val recs_04 = sc.parallelize(
      Seq(
      MyClass2("fff", "fff", "fff", "fff", 100),
      MyClass2("ggg", "ggg", "ggg", "ggg", 200)
      ) )

   val recs_05 = sc.parallelize(
      Seq(
      MyClass2("fff", "fff", "fff", "fff", 300),
```



```
      MyClass2("ggg", "ggg", "ggg", "ggg", 400)
      ) )
```

Relative to Example 26-4, the following is offered:

- As a single row is written (insert or update), there will be a simple time stamp for all attribute in the row. So the new case class contains a single attribute as such.
- We parallelize two sequences, to be used for an insert, then update scenario.

Example 26-5 details the actual code to set writeTime. A code review follows.

Example 26-5 Actually setting writeTime using DSE Analytics

```
recs_04.saveToCassandra("ks_26", "t1", SomeColumns(
      "col1", "col2", "col3", "col4") )

   val recs_06 = spark.sparkContext.cassandraTable[MyClass]("ks_26", "t1").
      select("col1", "col2", "col3", "col3".writeTime as
      "col3_ts", "col4", "col4".writeTime as "col4_ts")
         //
   recs_06.collect().foreach(println)


   recs_05.saveToCassandra("ks_26", "t1", SomeColumns(
      "col1", "col2", "col3", "col4"),
      writeConf = WriteConf(timestamp = TimestampOption.perRow("my_ts")))

   val recs_07 = spark.sparkContext.cassandraTable[MyClass]("ks_26", "t1").
      select("col1", "col2", "col3", "col3".writeTime as
      "col3_ts", "col4", "col4".writeTime as "col4_ts")
         //
   recs_07.collect().foreach(println)
```

Relative to Example 26-5, the following is offered:

- The first saveToCassandra() writes the recs_04 RDD to DSE, and timestamps are set to the values specified (100 and 200).
- The second saveToCassandra() writes the recos_05 RDD, which effectively updates the time stamps.

Note: Q: If you re-write the first RDD, what time stamps would be recorded ?
A: The latest time stamps (300, 400).

Recall what DSE uses time stamps for; consolidating multi-writes, eventual consistency.

(Helper functions)

Example 26-6 displays a number of helper functions. A code review follows.

Example 26-6 Helper functions

```
// var recs_8 = spark.sql(
   // "truncate table ks_26.t1"
   // )
   // recs_8.collect()
   //
   // org.apache.spark.sql.AnalysisException: Operation not allowed:
   //    TRUNCATE TABLE on external tables: `ks_26`.`t1`;
   //    at org.apache.spark.sql.execution.command.   ...


   import com.datastax.spark.connector.cql.CassandraConnector
   import com.datastax.driver.dse.DseSession

   // val sc = new SparkContext(sparkConf)
   val cc = CassandraConnector(sc.getConf).
      openSession().
      asInstanceOf[DseSession]

   cc.execute("TRUNCATE TABLE ks_26.t1")

   -------------------------------------------------
   -------------------------------------------------

   import java.util._

   //  Don't forget the L
   //
   val dd = new Date(1554497727676888L)
   //
   // dd: java.util.Date = Sat Jan 26 12:21:16 PST 51230
```

Relative to Example 26-6, the following is offered:

- The first code block that is commented out, is to show what is not possible; you do not run "truncate table" command via standard SQL.

  You do run "truncate table" via the CassandrConnector execute() method, detailed above.

- The java.util.Date() method can be used to convert milliseconds to a human-readable data value, as shown.

Using Time to Live (TTL) using CQL and DSE Analytics

Where timeStamp records in milliseconds, time to live (TTL) records in whole units seconds. Example 26-7 details the first TTL example, which INSERTs a row that will (expire) in 20 seconds time. A code review follows.

Example 26-7 TTL, first example

```
INSERT INTO t1 (col1, col2, col3, col4)
      VALUES ('mmm', 'mmm', 'mmm', 'mmm')
      USING TTL 20;

   SELECT TTL(col3), col1, col2, col3, col4 FROM t1;

      //

   UPDATE t1
      USING TTL 20
      SET
         col3 = 'mmm',
         col4 = 'mmm'
      WHERE
         col1 = 'mmm'
      AND
         col2 = 'mmm';

   SELECT TTL(col3), col1, col2, col3, col4 FROM t1;
```

Relative to Example 26-7, the following is offered:

- Recall that all INSERTs or UPDATEs using DSE are actually (UPSERTs). Still however, the row being inserted in this example did not exist, and is added (INSERTed) with a 20 second time to live.
- The SELECT statement will, or will not, return the row based on whether the row time to live has expired or not.

- As the UPDATE statement exists, all column values are set to the same values. In effect then, this UPDATE lengthens/resets to time to live to another 20 seconds duration.

Example 26-8 details some nuances relative to the time to live capability. A code review follows.

Example 26-8 Time to live, nuances

```
//  Below; col4 left unchanged
   //
   UPDATE t1 SET col3 = 'mmm', col4 = 'mmm' WHERE col1 = 'mmm' AND col2 =
'mmm';
   UPDATE t1 SET col3 = 'nnn'               WHERE col1 = 'mmm' AND col2 =
'mmm';
      //
   SELECT TTL(col3), col1, col2, col3, col4 FROM t1;

   TRUNCATE t1;

   //  Below; col4 will go away after 20
   //
   UPDATE t1 USING TTL 20
             SET col3 = 'mmm', col4 = 'mmm' WHERE col1 = 'mmm' AND col2 =
'mmm';
   UPDATE t1 SET col3 = 'nnn'               WHERE col1 = 'mmm' AND col2 =
'mmm';
      //
   SELECT TTL(col3), col1, col2, col3, col4 FROM t1;
```

Relative to Example 26-8, the following is offered:

- The first wo UPDATE statement demonstrate that an UPDATE can change a subset of columns in a DSE row. (You can also DELETE a subset of columns in a row.)

  Similar to timeStamp, you can also query the remaining time to live; example as shown.

- The last two updates demonstrate that different columns in a single row can themselves have different time to live values.

  In this example, col4 will be set to NULL, when its time to live expires.

Setting Time to Live using DSE Analytics

Example 26-9 sets time to live using DSE Analytics, Spark RDDs. A code review follows.

Example 26-9 Setting time to live using DSE Analytics

```
case class MyClass3
      (
      col1     :  String,
      col2     :  String,
      col3     :  String,
      col4     :  String,
      my_ts    :  Long
      )

   val recs_09 = sc.parallelize(
      Seq(
      MyClass3("qqq", "qqq", "qqq", "qqq", 10),
      MyClass3("rrr", "rrr", "rrr", "rrr", 20)
      ) )

         //

   recs_09.saveToCassandra("ks_26", "t1", SomeColumns(
      "col1", "col2", "col3", "col4"),
      writeConf = WriteConf(ttl = TTLOption.perRow("my_ts")))

         //

   val recs_10 = spark.sparkContext.cassandraTable[MyClass]("ks_26", "t1").
      select("col1", "col2", "col3", "col3".writeTime as
      "col3_ts", "col4", "col4".writeTime as "col4_ts")
         //
   recs_10.collect().foreach(println)
```

Relative to Example 26-9, the following is offered:

- "my_ts" is of type long, the number of whole integer seconds, time to live.
- "recs_09" is a Scala sequence with two elements. the first row will live for 10 seconds, the second for 20 seconds.
- And then the SELECT.

A Word on Counters

There is an objects built into DSE titled, counter. A counter is a partition wide (counter) that you can increment, decrement, other. As a topic, counters would seem to be purpose built for a inventory management problem.

We chose not to use counters for this document because we preferred the cart abandonment feature offered by time to live.

Further reading on counters is available here,

https://docs.datastax.com/en/dse/6.7/cql/cql/cql_using/useCounters.html

https://issues.apache.org/jira/browse/CASSANDRA-6506

https://www.slideshare.net/planetcassandra/simplereach-counters-at-scale-a-cautionary-tale

Finally, Inventory Management

Example 26-10 applies each of the topics detailed above to solve the inventory management problem. A code review follows.

Example 26-10 Inventory management problem, complete example

```
//  Stock on hand, complete

   DROP TABLE IF EXISTS stock;

   CREATE TABLE stock
      (
      item          TEXT,
         //
      tx_type       TEXT,
      mk_unique     TEXT,      // TIMEUUID, for uniqueness
         //                    // TEXT for easy demomstration
      qty           INT,
      my_version    INT,
         //
      PRIMARY KEY ((item), tx_type, mk_unique)
      );

   INSERT INTO stock (item, tx_type, mk_unique, qty, my_version)
      VALUES ('bats', 'on_hand', CAST(now() AS TEXT), 100,  0);

   INSERT INTO stock (item, tx_type, mk_unique, qty, my_version)
      VALUES ('hats', 'on_hand', '5000'             ,  50,  0);
   INSERT INTO stock (item, tx_type, mk_unique, qty, my_version)
      VALUES ('hats', 'hold'   , '5001'             ,  -8,  1);

      //

   SELECT SUM(qty)
   FROM stock
   WHERE
      item = 'hats'
   AND
      tx_type IN ('on_hand', 'hold');
```

```
   ------------------------------------------------
   ------------------------------------------------

   // Place on hold; this will fail for wrong my_version
   //
   BEGIN BATCH
UPDATE stock
      SET my_version = 2
      WHERE item = 'hats' AND tx_type = 'on_hand' AND mk_unique = '5000'
      IF my_version = -6;
   INSERT INTO stock (item, tx_type, mk_unique, qty, my_version)
      VALUES ('hats', 'hold', '5002', -12, 2)
      USING TTL 60;
   APPLY BATCH;
   //
   //  [applied] | item | tx_type | mk_unique | my_version | qty
   // -----------+------+---------+-----------+------------+-----
   //      False | hats | on_hand |      5000 |          0 |  50

   ------------------------------------------------
   ------------------------------------------------

   // Place on hold; this will work
   //
   BEGIN BATCH
   UPDATE stock
      SET my_version = 2
      WHERE item = 'hats' AND tx_type = 'on_hand' AND mk_unique = '5000'
      IF my_version =  0;
   INSERT INTO stock (item, tx_type, mk_unique, qty, my_version)
      VALUES ('hats', 'hold', '5002', -12, 2)
      USING TTL 60;
   APPLY BATCH;
   //
   //  [applied]
   // -----------
   //       True


   ------------------------------------------------
   ------------------------------------------------


   SELECT SUM(qty)
   FROM stock
   WHERE
      item = 'hats'
   AND
      tx_type IN ('on_hand', 'hold');
```



```
//  system.sum(qty)
// -----------------
//               30

//  Wait 60

//  system.sum(qty)
// -----------------
//               42

------------------------------------------------
------------------------------------------------

// Moving (hold) to (sold)

BEGIN BATCH
UPDATE stock
   SET my_version = 3, qty = 38
   WHERE item = 'hats' AND tx_type = 'on_hand' AND mk_unique = '5000'
   IF my_version =  2;
      //
INSERT INTO stock (item, tx_type, mk_unique, qty, my_version)
   VALUES ('hats', 'sold', '5002', -12, 3);
DELETE FROM stock
   WHERE item = 'hats' AND tx_type = 'hold' AND mk_unique = '5002'
APPLY BATCH;

//  If 'hold' had timed out
//
//  [applied] | item | tx_type | mk_unique | my_version | qty
// -----------+------+---------+-----------+------------+-----
//      False | hats | on_hand |      5000 |          0 |  50

//  Success
//  [applied]
// -----------
//       True
```

Relative to Example 26-10, the following is offered:

- First we create a table titled, stock. Columns include:
  - item, presumably the item sku. We partition on item only, which could cause a hot node; when a single given item/sku is getting pummeled.

But, solving this problem is not related to inventory management; add a second column to the partition key to spread items/skus across more nodes, share the load across nodes.

- tx_type will contains the values;

  on_hand, how much of this item/sku do I possess, ready to sell

  hold, a quantity of a given item/sku inserted with a time to live, reserved (on hold) but will disappear/release on time out.

  sold, we move hold to sold when the user commits to purchase. At this same time we decrement the amount on hand. Thus, sold is not needed to calculate on hand, and merely serves as a history of items sold.

- mk_unique, so that we may have multiple item/hold records, we add this portion to the primary key only to generate uniqueness of records.

  We can use a UUID, or TIMEUUID to auto generate this value.

  (For ease of testing, the examples below uses a hard coded value in this field. You would generate the UUID/TIMEUUID in your client program, record same, and act as those is were the same hard coded value we detail.))

- qty, an integer count of the items ready for sale.
- my_version, an incrementing (version) number we maintain. E.g., has the number of on_hand changed (has the number of items available to sell) changed without my knowing.

  We tag this (version) in both the on_hand and the hold (and sold) records.

– The INSERT for bats demonstrates how to generate the TIMEUUID using now().

– The two INSERTs for hats set an on_hand, and a hold. The hold is created without a time to live, merely to demonstrate that multiple hold records can exist and maintain (safe to sell) arithmetic.

– The first SELECT demonstrates how we calculate/read the accurate number of items available for sale.

– The first BEGIN BATCH block-

- The BEGIN/APPLY BATCH block allow a multi-statement transaction using DSE.

  Minus I-in-ACID/Isolation, this in an acid compliant routine, available in DataStax.

- This first update will fail, per design, and displays how a transaction will happen entirely, or not at all.

Note: These "IF" blocks on INSERT or UPDATE are titled, LWT: lightweight transactions, and use the Paxos protocol.

This capability is also available using Scala/Spark/RDDs.

- The second BEGIN/APPLY BATCH block-
  - Demonstrates how to put a count of items on hold, with a time to live. Here we place 12 items on hold.
  - We could use a second timestamp on version, but use an incrementing integer of our design and management.
- The SELECT shows an accurate on_hand amount; how to read and calculate that.
- The last BEGIN/APPLY BATCH block-
  - Demonstrates how to move a hold quantity to sold.
  - Again, this block will happen entirely, or not at all.
  - As designed/delivered, the sold item count really serves as history only, and you could add fields for whatever reason; order_num, other.

26.2 Complete the following

At this point in this document we have completed a lengthy primer on DataStax Enterprise (DSE) time stamp, time to live, transaction management (including LWT: lightweight transactions), and more.

And we detailed one version of an inventory management routines, complete with Scala/Spark/RDD programming.

Should you need to, take the above and apply it to your specific inventory management design.

26.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- The DSE time stamp and time to live functions.
- DSE transaction and lightweight transactions.
- How to deliver an inventory management system with reserve, and reclaim of stock count on cart abandonment.

Persons who help this month.

Kiyu Gabriel, Jim Hatcher, and Steven Hubbard.

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
```

