Cassandra Tombstone In Depth

1. Overview

Having an excessive number of tombstones in a Cassandra table is one major reason why Cassandra/DSE based application systems don't perform well. For many times, when this happens, it indicates some potential issues either with the data model or in the application development. In this blog post, I'd like to take a deeper look at different scenarios when tombstones can be generated in the system and how to categorize the tombstones into different groups.

1.1. Testing Schema

In the sections below, I'll go through different tombstone scenarios through examples. The schema of the testing keyspace and table is as below.

```
cqlsh> DESCRIBE KEYSPACE testks ;

CREATE KEYSPACE testks WITH replication = {'class': 'SimpleStrategy', 'replication_factor':
'1'} AND durable_writes = true;

CREATE TABLE testks.testbl (
cola int,
colb int,
colc text,
cold set<text>,
cole list<text>,
colf map<int, text>,
PRIMARY KEY (cola, colb)
)
```

2. Tombstone Source and Category

In Cassandra, tombstone is created when data is deleted. The deletion can be made explicitly from a CQL "delete" statement, can be TTLed, or can be implicitly deleted from Cassandra internal operations (e.g. via materialized view operation). There are also 2 other subtle operations that can cause tombstones: 1) insert/update "null" value; 2) update with collection columns (complex columns).

When a tombstone is generated, it can be marked on different parts of a partition. Based on the location of the tombstone marker location, I categorize tombstones into different groups and each category typically corresponds to one unique type of the data deletion operation. Please note that internally, Cassandra use different markers between explicit deletion and TTLed data. So I put them in separate categories.

- Partition tombstone
- Row tombstone
- Range tombstone
- ComplexColumn tombstone
- Cell tombstone
- TTLed tombstone
    - Row level
    - Cell level

2.1. Partition Tombstone

Partition tombstone happens when an entire partition is deleted explicitly. Using a CQL "delete" statement, the where clause in the statement is simply *an equality condition against the partition key.*

```
cqlsh> delete from testbl where cola = 2;
```

Looking at the "sstabledump" output for this partition, you will see that the tombstone marker, **deletion_info**, is at partition level and is not associated with any rows or cells within the partition.

```
{
"partition" : {
"key" : [ "2" ],
"position" : 191,
"deletion_info" : { "marked_deleted" : "2017-11-14T15:54:50.685395Z", "local_delete_time" :
"2017-11-14T15:54:50Z" }
},
"rows" : [ ]
}
```

2.2. Row Tombstone

When coming to the Row tombstone, it refers to the case where the schema having a composite primary key that includes both partition key and clustering key. Row level tombstone happens when a particular row within a certain partition is delete explicitly. Using a CQL "delete" statement, the where clause in the statement is *an equality condition against both the partition key AND the clustering key*.

```
cqlsh> delete from testbl where cola = 3 and colb = 1;
```

Looking at the "sstabledump" output for this partition, you will see that the tombstone marker, **deletion_info**, is at row level that is identified by a clustering key under the partition. Neither the partition nor row has the tombstone marker.

```
{
"partition" : {
"key" : [ "3" ],
"position" : 210
},
"rows" : [
{
"type" : "row",
"position" : 244,
"clustering" : [ "1" ],
"deletion_info" : { "marked_deleted" : "2017-11-14T15:57:30.375376Z", "local_delete_time" :
"2017-11-14T15:57:30Z" },
"cells" : [ ]
},
{
... ... // other rows in the same partition
}
]
}
```

2.3. Range Tombstone

For Range tombstone, it also refers to the case where the schema having a composite primary key that includes both partition key and clustering key. But range tombstone happens when several rows within a certain partition that can be expressed through a range search are deleted explicitly. Using a CQL "delete" statement, the where clause in the statement is *an equality condition against the partition key and an inequality condition against the clustering key*.

```
cqlsh> delete from testbl where cola = 1 and colb > 3;
```

Looking at the "sstabledump" output for this partition, you will see that the range tombstone marker, **deletion_info**, are at row levels with a special boundary marker, **range_tombstone_bound**, that marks the range scope (identified by clustering key values) of the rows that are deleted.

```
{
"partition" : {
"key" : [ "1" ],
"position" : 0
},
"rows" : [
{
... ... // other rows in the same partition
},
{
"type" : "range_tombstone_bound",
"start" : {
"type" : "exclusive",
"clustering" : [ "3" ],
"deletion_info" : { "marked_deleted" : "2017-11-14T15:59:05.787225Z", "local_delete_time" :
"2017-11-14T15:59:05Z" }
}
},
{
"type" : "range_tombstone_bound",
"end" : {
"type" : "inclusive",
"deletion_info" : { "marked_deleted" : "2017-11-14T15:59:05.787225Z", "local_delete_time" :
"2017-11-14T15:59:05Z" }
}
}
]
}
```

2.4. ComplexColumn Tombstone

ComplexColumn tombstone happens when dealing with collection types: set, list, and map. In Cassandra, when inserting/updating a collection type column as a whole, it will trigger a tombstone. It is better to describe this with some examples.

```
cqlsh> insert into testbl (cola, colb, colf) values (1, 1, { 0: 'pen', 1: 'pencil' } );
cqlsh> update testbl set cole = ['horse'] where cola = 1 and colb = 1;
```

Looking at the "sstabledump" output for this partition, we can see that although there is no explicit deletion on columns "cole" (list type) and "colf" (map type), there is still tombstone marker, **deletion_info**, associated with them at cell level.

```
{
"partition" : {
"key" : [ "1" ],
"position" : 0
},
"rows" : [
{
"type" : "row",
"position" : 119,
"clustering" : [ "1" ],
"liveness_info" : { "tstamp" : "2017-11-14T15:50:01.933546Z" },
"cells" : [
{ "name" : "cole", "deletion_info" : { "marked_deleted" : "2017-11-14T16:02:43.420015Z",
"local_delete_time" : "2017-11-14T16:02:43Z" } },
{ "name" : "cole", "path" : [ "3f94a9c1-c955-11e7-bea2-f709ea23126f" ], "value" : "horse",
"tstamp" : "2017-11-14T16:02:43.420016Z" },
{ "name" : "colf", "deletion_info" : { "marked_deleted" : "2017-11-14T15:50:01.933545Z",
"local_delete_time" : "2017-11-14T15:50:01Z" } },
{ "name" : "colf", "path" : [ "0" ], "value" : "pen" },
{ "name" : "colf", "path" : [ "1" ], "value" : "pencil" }
]
},
{
... ... // other rows in the same partition
}
]
}
```

2.5. Cell Tombstone

Cell tombstone is easy to understand. It happens when explicitly deleting a value from a cell (aka, a column for a specific row of a partition); or inserting/updating a cell with "null" values.

```
cqlsh> insert into testbl (cola, colb, colc) values (110, 1, null);
```

Looking at the "sstabledump" output for this partition, the tombstone marker, **deletion_info**, is associated with the particular cell.

```
{
"partition" : {
"key" : [ "110" ],
"position" : 0
},
"rows" : [
{
"type" : "row",
"position" : 32,
"clustering" : [ "1" ],
"liveness_info" : { "tstamp" : "2017-11-14T23:47:51.774107Z" },
"cells" : [
{ "name" : "colc", "deletion_info" : { "local_delete_time" : "2017-11-14T23:47:51Z" } }
]
}
]
}
```

2.6. TTLed Tombstone

TTLed data will generate tombstone when TTL window is passed. But internally, Cassandra marks TTLed data differently from regluar tombstoned data (aka, the data that is explicitly deleted). TTL expiration marker can happen either at row or cell level. Even if a partition only has one single row (no clustering key), the TTL mark is still made at row level, not at partition level.

```
cqlsh> insert into testbl (cola, colb, colc) values (130, 1, 'beautiful') using ttl 1;   //
TTL an entire row
cqlsh> update testbl using ttl 1 set colc = 'boat' where cola = 300 and colb = 1;        //
TTL one cell
```

Looking at the "sstabledump" output for these partitions, the first CQL statement above marks the row (partition key: 130, clustering key: 1) with a TTL expiration marker, **"expired" : true,** in the row's liveness_info section. Similarly, the second CQL statement above marks the cell (partition key: 300, clustering key: 1, column name: colc) with a TTL expiration marker, **"expired" : true,** for that specifi cell.

```
{
"partition" : {
"key" : [ "130" ],
"position" : 66
},
"rows" : [
{
"type" : "row",
"position" : 109,
"clustering" : [ "1" ],
"liveness_info" : { "tstamp" : "2017-11-14T23:49:28.083970Z", "ttl" : 1, "expires_at" :
"2017-11-14T23:49:29Z", "expired" : true },
"cells" : [
{ "name" : "colc", "value" : "beautiful" }
]
}
]
},
{
"partition" : {
"key" : [ "300" ],
"position" : 0
},
"rows" : [
{
"type" : "row",
"position" : 54,
"clustering" : [ "1" ],
"liveness_info" : { "tstamp" : "2017-11-15T02:21:20.521498Z" },
"cells" : [
{ "name" : "colc", "value" : "boat", "tstamp" : "2017-11-15T02:22:59.396389Z", "ttl" : 1,
"expires_at" : "2017-11-15T02:23:00Z", "expired" : true }
]
}
]
}
```

3. Tombstone Counter Utility

Currently, there doesn't have a tool to count the number of tombstones in a Cassandra table for different categories. But in my opinion, having a deeper view of tombstones at different categories can be beneficial to pinpoint the root cause of tombstone issues in data model and/or application development. In order to answer this question, I developed a tool called tombstone_counter to get the total number of tombstones in a Cassandra table, as well as the number of tombstones per single category as discussed above. The tool can be found from my Github repository at: https://github.com/yabinmeng/tombstone_counter .