# Existing Test Cases

## 1. Rows Test

The *RowsTest* test case is a fairly large class (681 lines) that implements about ten different tests. This test case had a hefty set of prerequisites and class instantiations. Most of these methods were beyond the scope of our testing, however the *StatsCollector* class on line 172 seemed very interesting and worth investigating.

```
1  public static class StatsCollector implements PartitionStatisticsCollector
2  {
3      List<Cell> cells = new LinkedList<>();
4      public void update(Cell cell)
5      {
6          cells.add(cell);
7      }
8
9      List<LivenessInfo> liveness = new LinkedList<>();
10     public void update(LivenessInfo info)
11     {
12         liveness.add(info);
13     }
14
15     List<DeletionTime> deletions = new LinkedList<>();
16     public void update(DeletionTime deletion)
17     {
18         deletions.add(deletion);
19     }
20
21     long columnCount = -1;
22     public void updateColumnSetPerRow(long columnSetInRow)
23     {
24         assert columnCount < 0;
25         this.columnCount = columnSetInRow;
26     }
27
28     boolean hasLegacyCounterShards = false;
29     public void updateHasLegacyCounterShards(boolean hasLegacyCounterShards)
30     {
31         this.hasLegacyCounterShards |= hasLegacyCounterShards;
32     }
33 }
```

The *StatsCollector* class attempts to hold information, regardless of what that information is. In this test case it is used to collect information on several properties of the row, including “deletion time” and “liveness”. The tests themselves seem to somewhat match Behavior Driven Development, but are a little lacking in the descriptiveness of their names. The tests *copy* and *collectStats* give us a general idea of their nature, but at the same time they are a little vague

```
1  public void collectStats()
2  {
```

```
3     int now = FBUtilities.nowInSeconds();
4     long ts = secondToTs(now);
5     Row.Builder builder = BTreeRow.unsortedBuilder();
6     builder.newRow(c1);
7     LivenessInfo liveness = LivenessInfo.create(ts, now);
8     builder.addPrimaryKeyLivenessInfo(liveness);
9     DeletionTime complexDeletion = new DeletionTime(ts-1, now);
10    builder.addComplexDeletion(m, complexDeletion);
11    List<Cell> expectedCells = Lists.newArrayList(BufferCell.live(v, ts,
BB1),
12                                                  BufferCell.live(m, ts,
BB1, CellPath.create(BB1)),
13                                                  BufferCell.live(m, ts,
BB2, CellPath.create(BB2)));
14    expectedCells.forEach(builder::addCell);
15    // We need to use ts-1 so the deletion doesn't shadow what we've created
16    Row.Deletion rowDeletion = new Row.Deletion(new DeletionTime(ts-1, now),
false);
17    builder.addRowDeletion(rowDeletion);
18
19    StatsCollector collector = new StatsCollector();
20    Rows.collectStats(builder.build(), collector);
21
22    Assert.assertEquals(Lists.newArrayList(liveness), collector.liveness);
23    Assert.assertEquals(Sets.newHashSet(rowDeletion.time(),
complexDeletion), Sets.newHashSet(collector.deletions));
24    Assert.assertEquals(Sets.newHashSet(expectedCells),
Sets.newHashSet(collector.cells));
25    Assert.assertEquals(2, collector.columnCount);
26    Assert.assertFalse(collector.hasLegacyCounterShards);
27 }
```

## 2. Simple Query Test

This test case implements several tests that all focus on Cassandra's own query language and its nuances. For example there are a number of test cases which all test a table with, without, and with several clusters of primary keys. The *collectionDeletionTest* on line 427 is particularly interesting.

```
1  @Test
2  public void collectionDeletionTest() throws Throwable
3  {
4     createTable("CREATE TABLE %s (k int PRIMARY KEY, s set<int>);");
5
6     execute("INSERT INTO %s (k, s) VALUES (?, ?)", 1, set(1));
7
8     flush();
9
10    execute("INSERT INTO %s (k, s) VALUES (?, ?)", 1, set(2));
11
12    assertRows(execute("SELECT s FROM %s WHERE k = ?", 1),
13        row(set(2))
14    );
15 }
```

It inserts two sets or lists into the same table with the same key. Then it checks which set is returned after querying the key. The expected value is the second set, which overwrote the first. This unveiled a small, but important, feature about the Cassandra system which was not clear to us before. Interestingly, there was no test to check the same functionality on a single value rather than a set. This raised the question of whether the overwrite feature was unique to sets or could be applied to single objects. Several of the other tests revealed simpler things about the Cassandra query language and system overall. Distinct queries, and simple row deletion are tested. We learned a lot about the differences between CQL and other SQL languages we are more familiar with. Primarily the clustering of Primary keys as well as the syntax of the query when making a table. The test names are more descriptive and better adhere to Behavior Driven testing.

## 3. Role Options Test

This test checks the properties of a *IRoleManager* interface. At the bottom of the class we see an instantiation of *IRoleManager* implementing only its "options" properties. "Options" are a separate class that is passed to the role manager along with a role. The role manager then assigns these options to that role. The options tested throughout this test case are primarily the login property and the password property. The first test checks the object types that can be passed to different properties in an options class. However, the partitions done on each property are poor and can be improved upon. For example, the test checks that the password property is a string by testing against only one integer. This test case revealed a lot about the nature of roles in Cassandra and how the *IRoleManager* interface connects the *options* class to a role. Roles were the first topic my team explored in the Cassandra system. At that time we had no notion of the *options* class and were not aware of its existence. Through reading this test case we gained significant insight on the system.

```
1  @Test
2      public void validateValueTypes()
3      {
4          setupRoleManager(getRoleManager(IRoleManager.Option.values()));
5
6          RoleOptions opts = new RoleOptions();
7          opts.setOption(IRoleManager.Option.LOGIN, "test");
8          assertInvalidOptions(opts, "Invalid value for property 'LOGIN'. It
   must be a boolean");
9
10         opts = new RoleOptions();
11         opts.setOption(IRoleManager.Option.PASSWORD, 99);
12         assertInvalidOptions(opts, "Invalid value for property 'PASSWORD'.
   It must be a string");
13
14         opts = new RoleOptions();
15         opts.setOption(IRoleManager.Option.SUPERUSER, new HashSet<>());
16         assertInvalidOptions(opts, "Invalid value for property 'SUPERUSER'.
   It must be a boolean");
17
18         opts = new RoleOptions();
19         opts.setOption(IRoleManager.Option.OPTIONS, false);
20         assertInvalidOptions(opts, "Invalid value for property 'OPTIONS'.
   It must be a map");
21
22         opts = new RoleOptions();
23         opts.setOption(IRoleManager.Option.LOGIN, true);
```

```
24          opts.setOption(IRoleManager.Option.SUPERUSER, false);
25          opts.setOption(IRoleManager.Option.PASSWORD, "test");
26          opts.setOption(IRoleManager.Option.OPTIONS,
Collections.singletonMap("key", "value"));
27          opts.validate();
28      }
```

In the code above, different member variables that a role can have are tested with invalid input. For example, a LOGIN should be a boolean instead of a string, and a PASSWORD should be a string. Before reading this test, we thought LOGIN would be a string as well, however, now we understand that before Cassandra allows some users to login the database, it will first check the LOGIN name existed in the system already. Also, for each option that a user can do to the system, it is using a map to pair each role and its corresponding role options.