Newly developed test cases

1. Additional Row Deletion Test Case

Table update is an very important and fundamental function of Cassandra as a distributed software. For its row deletion function, we found out the the current row deletion test suite was pretty brief that does not cover all possible situations that a user will be performing a deletion query. The existing delete test only test if deletion works when a user insert a row of data and then delete that piece of data, it should return null (or an Enum "isExhaustive" in Cassandra). However, it does not cover the situation where user use delete on a piece of data that does not exist in the table. Another different case is when there are multiple pieces of data in a table and we are not sure if this "delete" query will actually delete the one that we want to delete.

1.1. Thinking process

In order to cover the two situations above, we wrote a new test case called `AdditionalDeleteTest`. Initially we created 3 tables: table1, table2, and table3 with id and some text values. Then we have 3 update queries: insert, update and delete. At the end we will test the query result by calling "select" query.

```
session.execute("CREATE TABLE junit.table1 (\n" +
                        "  id int ,\n" +
                        "  id1 int ,\n" +
                        "  text1 text ,\n" +
                        "  PRIMARY KEY ( ( id ), id1 )\n" +
                        ");");
        session.execute("CREATE TABLE junit.table2 (\n" +
                        "  id int ,\n" +
                        "  id2 int ,\n" +
                        "  text2 text ,\n" +
                        "  val text ,\n" +
                        "  PRIMARY KEY ( ( id ), id2 )\n" +
                        ");");
        session.execute("CREATE TABLE junit.table3 (\n" +
                        "  id int ,\n" +
                        "  id3 int ,\n" +
                        "  text3 text ,\n" +
                        "  val text ,\n" +
                        "  PRIMARY KEY ( ( id ), id3 )\n" +
                        ");");

        // insert into table2
        pstmtI = session.prepare("insert into junit.table2 ( id, id2, text2, val) values (?, ?, ?, ?)");
        // update row data from table2 while keeping other 2 tables null
        pstmtU = session.prepare("update junit.table2 set text2=?, val=? where id=? and id2=?");

        pstmtD = session.prepare("delete from junit.table2 where id=? and id2=?");
        pstmt1 = session.prepare("select id, id1, text1 from junit.table1 where id=? and id1=?");
        pstmt2 = session.prepare("select id, id2, text2, val from
junit.table2 where id=? and id2=?");
        pstmt3 = session.prepare("select id, id3, text3, val from
junit.table3 where id=? and id3=?");
    }
```

- Insert 2 pieces of data into table2:

While we keep table1 and table3 unchanged, we make changes to table 2 only. We insert 2 pieces data and test the two queries if they return a result or a null result.

```
session.execute(pstmtI.bind(1, 1, "text2", "valB"));
            session.execute((pstmtI.bind(2, 2, "text2", "valB")));
            ResultSetFuture[] futures = load();


 Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            // two rows inserted
            Assert.assertNotNull(futures[1].getUninterruptibly().one());
            Assert.assertNotNull(futures[2].getUninterruptibly().one());


 Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());
```

- Update a piece of data from table2:

We update the piece of data that has id of 1 with a new text value and check the returning result.

```
// update one row
            session.execute(pstmtU.bind("updatedText2.1", "updatedText2.2",
1, 1));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertNotNull(futures[1].getUninterruptibly().one());
            Assert.assertNotNull(futures[2].getUninterruptibly().one());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());
```

- Delete one piece of data from table2:

We perform a delete query on one of the two data pieces and check the returning result by performing select query on each the two data.

```
//delete a row from table2, there should be 1 row left
            session.execute(pstmtD.bind(1, 1));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[1].getUninterruptibly().isExhausted());
            Assert.assertNotNull(futures[2].getUninterruptibly().one());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());
```

- Delete last row of data from table2:

```java
// delete the last row from table2
            session.execute(pstmtD.bind(2, 2));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[1].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[2].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());
```

- Execute a delete on the empty table2:

Finally, while the table2 contains no data, we perform another delete execution and check if it possibly destroys the whole table, throws some errors or acts however it should be acting.

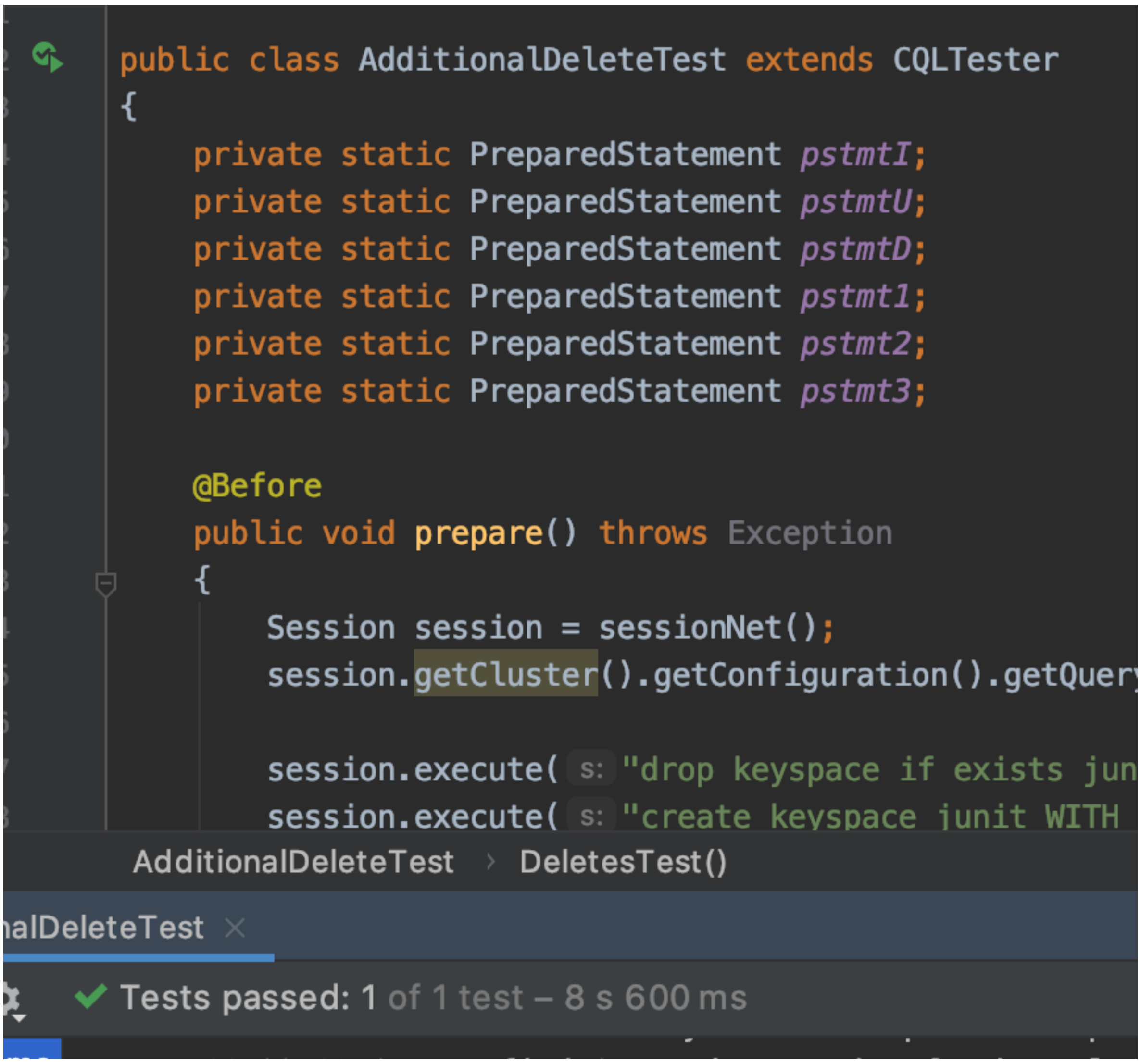
```java
// execute deletion on an empty table
            session.execute(pstmtD.bind(1, 1));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[1].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[2].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());
```

After reviewing related existing test case and writing this new test case, we learned that even Cassandra is a high level, NoSql, distributed database software, the CQL (Cassandra Query Language) also follows the general SQL format such as normal "insert", "delete", and "update" query. Also while the backend test cases of Cassandra cover most feature of the system, the features that are more front-end such as parsing user query into data access to the database has not been fully tested. There is no point of performing exhaustive testing but it is meaningful to choose some boundary cases to test upon.

1.2. New test case code

```java
package org.apache.cassandra.cql3;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import com.datastax.driver.core.ConsistencyLevel;
import com.datastax.driver.core.PreparedStatement;
import com.datastax.driver.core.ResultSetFuture;
import com.datastax.driver.core.Session;

public class AdditionalDeleteTest extends CQLTester
{
    private static PreparedStatement pstmtI;
    private static PreparedStatement pstmtU;
    private static PreparedStatement pstmtD;
    private static PreparedStatement pstmt1;
    private static PreparedStatement pstmt2;
    private static PreparedStatement pstmt3;

    @Before
    public void prepare() throws Exception
    {
        Session session = sessionNet();
        session.getCluster().getConfiguration().getQueryOptions().setConsistencyLevel(ConsistencyLevel.ONE);

        session.execute("drop keyspace if exists junit;");
        session.execute("create keyspace junit WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 2 };");
        // create 3 tables, only table 2 will be changed
        session.execute("CREATE TABLE junit.table1 (\n" +
                        "  id int ,\n" +
                        "  id1 int ,\n" +
                        "  text1 text ,\n" +
                        "  PRIMARY KEY ( ( id ), id1 )\n" +
                        ");");
        session.execute("CREATE TABLE junit.table2 (\n" +
                        "  id int ,\n" +
                        "  id2 int ,\n" +
                        "  text2 text ,\n" +
                        "  val text ,\n" +
                        "  PRIMARY KEY ( ( id ), id2 )\n" +
                        ");");
        session.execute("CREATE TABLE junit.table3 (\n" +
                        "  id int ,\n" +
                        "  id3 int ,\n" +
                        "  text3 text ,\n" +
                        "  val text ,\n" +
                        "  PRIMARY KEY ( ( id ), id3 )\n" +
                        ");");

        // insert into table2
        pstmtI = session.prepare("insert into junit.table2 ( id, id2, text2, val) values (?, ?, ?, ?)");
        // update row data from table2 while keeping other 2 tables null
        pstmtU = session.prepare("update junit.table2 set text2=?, val=? where id=? and id2=?");

        pstmtD = session.prepare("delete from junit.table2 where id=? and id2=?");
        pstmt1 = session.prepare("select id, id1, text1 from junit.table1 where id=? and id1=?");
        pstmt2 = session.prepare("select id, id2, text2, val from junit.table2 where id=? and id2=?");
        pstmt3 = session.prepare("select id, id3, text3, val from junit.table3 where id=? and id3=?");
    }

    @Test
    public void DeletesTest()
    {
        Session session = sessionNet();

        for (int i = 0; i < 2; i++)
        {
            // insert function will be called twice to test deletion from rows with no or multiple lines
            session.execute(pstmtI.bind(1, 1, "text2", "valB"));
            session.execute((pstmtI.bind(2, 2, "text2", "valB")));
            ResultSetFuture[] futures = load();

            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            // two rows inserted
            Assert.assertNotNull(futures[1].getUninterruptibly().one());
            Assert.assertNotNull(futures[2].getUninterruptibly().one());

            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());

            // update one row
            session.execute(pstmtU.bind("updatedText2.1", "updatedText2.2", 1, 1));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertNotNull(futures[1].getUninterruptibly().one());
            Assert.assertNotNull(futures[2].getUninterruptibly().one());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());

            //delete a row from table2, there should be 1 row left
            session.execute(pstmtD.bind(1, 1));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[1].getUninterruptibly().isExhausted());
            Assert.assertNotNull(futures[2].getUninterruptibly().one());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());

            // delete the last row from table2
            session.execute(pstmtD.bind(2, 2));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[1].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[2].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());

            // execute deletion on an empty table
            session.execute(pstmtD.bind(1, 1));
            futures = load();
            Assert.assertTrue(futures[0].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[1].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[2].getUninterruptibly().isExhausted());
            Assert.assertTrue(futures[3].getUninterruptibly().isExhausted());
        }
    }

    private ResultSetFuture[] load() {
        Session session = sessionNet();

        return new ResultSetFuture[]{
        session.executeAsync(pstmt1.bind(1, 1)),
        session.executeAsync(pstmt2.bind(1, 1)),
        session.executeAsync(pstmt2.bind(2, 2)),
        session.executeAsync(pstmt3.bind(1, 1)),
        };
    }
}

```

The test result is shown below.



```
public class AdditionalDeleteTest extends CQLTester
{
    private static PreparedStatement pstmtI;
    private static PreparedStatement pstmtU;
    private static PreparedStatement pstmtD;
    private static PreparedStatement pstmt1;
    private static PreparedStatement pstmt2;
    private static PreparedStatement pstmt3;

    @Before
    public void prepare() throws Exception
    {
        Session session = sessionNet();
        session.getCluster().getConfiguration().getQuer

        session.execute( s: "drop keyspace if exists jun
        session.execute( s: "create keyspace junit WITH
```

AdditionalDeleteTest › DeletesTest()

alDeleteTest

✓ Tests passed: 1 of 1 test – 8 s 600 ms

2. Check if NIODataInputStream closes successfully

When we use some kind of data input stream, it is crucial to close the stream properly after we get what we need, or a memory leak may happen and cause a program failure after long time running the application.

In Cassandra, there is an `NIODataInputStream`, wrapping the input stream of a File or a Socket. There are three methods under this class, `rebuffer()`, `available()` and `close()`. The first two methods are tested in the `NIODataInputStreamTest` file. while the `close()` method is not covered by any unit test.

In the method, a `ReadableByteChannel` is closed, an `NIODataInputStream` is closed, and the `ByteBuffer` is set to null.

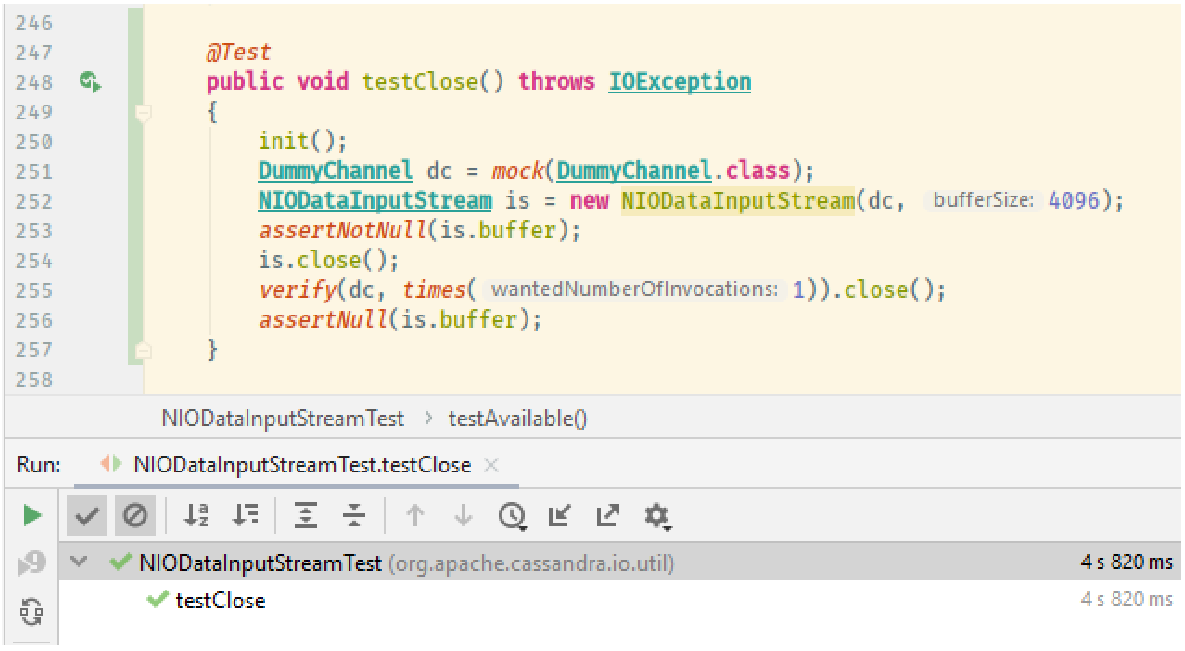
```
    @Override
    public void close() throws IOException
    {
        channel.close();
        super.close();
        FileUtils.clean(buffer);
        buffer = null;
    }
```

As the method does not have a return value, to test if the statements in the method are executed, first of all, we need to add the Mockito framework. Then we mock the channel in the constructor and track if its `close()` method is called successfully. The new test case is put in the existing `NIODataInputStreamTest` file.

```
1       @Test
2       public void testClose() throws IOException
3       {
4           init();
5           DummyChannel dc = mock(DummyChannel.class);
6           NIODataInputStream is = new NIODataInputStream(dc, 4096);
7           assertNotNull(is.buffer);
8           is.close();
9           verify(dc, times(1)).close();
10          assertNull(is.buffer);
11      }
```

Line 5 mocks the `DummyChannel`, which implements `ReadableByteChannel` and is created only for testing purposes. After constructing a new `NIODataInputStream` with the mocking product, we can execute the `close()` method. Line 9 and 10 verifies if the statements in the `close()` method are executed as expected. To further verify the status change of the `buffer` field. It is checked being NotNull after the construction of `is`, and Null after `is` is closed.

The test result is shown below.



```java
    @Test
    public void testClose() throws IOException
    {
        init();
        DummyChannel dc = mock(DummyChannel.class);
        NIODataInputStream is = new NIODataInputStream(dc,  bufferSize: 4096);
        assertNotNull(is.buffer);
        is.close();
        verify(dc, times( wantedNumberOfInvocations: 1)).close();
        assertNull(is.buffer);
    }
```

```
NIODataInputStreamTest › testAvailable()
Run: NIODataInputStreamTest.testClose
✓ NIODataInputStreamTest (org.apache.cassandra.io.util)    4 s 820 ms
    ✓ testClose                                            4 s 820 ms
```

To improve this test, we first checked more about the potential result if not properly closing an input stream and learned about the memory leak. Then, we are happy to learn about how the Cassandra team deal with the test cases when one object A relies heavily on another object B, but B is an interface. To make sure the test simulates all the situation when object C,D,E... all implements B, a new `DummyChannel` is created, with all the behaviors needed for the tests. This is a smart idea. This `DummyChannel` lives in the test file and will never interrupt the original code.

3. RoleOptionsTest extension

While exploring test cases in Cassandra our team came across the *RoleOptionsTest*. The first test validated that the passed value into a “options” class matched what was expected for the specific options property. Our team was dissatisfied with the partitioning and boundaries of the test. Each property, such as password or login, must be of an appropriate type. However, the test case only tests each property once with an incompatible type and once with the correct type. In our new test case we expand upon this. A framework is set so that any property in the options class can be checked multiplicatively against any number of types. Below is our implementation.

```java
public class RoleOptionsTest_ValidateOptions
{
    private static ArrayList<Object> objectList = new ArrayList<>();

    @Test
    public void validateValueTypes_Login()
    {
        setup_objectlist();
        setupRoleManager(getRoleManager(IRoleManager.Option.values()));

        RoleOptions opts = new RoleOptions();
        for(Object value : objectList){
            opts.setOption(IRoleManager.Option.LOGIN, value);
            assertInvalidOptions(opts, "Invalid value for property 'LOGIN'.
It must be a boolean");
        }
    }

    @Test
    public void validateValueTypes_Password()
    {
        setup_objectlist();
        setupRoleManager(getRoleManager(IRoleManager.Option.values()));

        RoleOptions opts = new RoleOptions();
        for(Object value : objectList){
            opts.setOption(IRoleManager.Option.PASSWORD, value);
            assertInvalidOptions(opts, "Invalid value for property 'PASSWORD'. It must be a string");
        }
    }

    @Test
    public void validateValueTypes_Options()
    {
        setup_objectlist();
        setupRoleManager(getRoleManager(IRoleManager.Option.values()));

        RoleOptions opts = new RoleOptions();
        for(Object value : objectList){
            opts.setOption(IRoleManager.Option.OPTIONS, value);
            assertInvalidOptions(opts, "Invalid value for property 'OPTIONS'. It must be a map");
        }
    }


    private void assertInvalidOptions(RoleOptions opts, String message){
        try
        {
            opts.validate();
            assertTrue("Valid value for property!", true);
        }
        catch (InvalidRequestException e)
        {
            assertTrue(e.getMessage().equals(message));
        }
    }

    private static void setup_objectlist(){
        String aString = "testing a string";
        Integer anInteger = 99;
        Boolean aBoolean = true;
        HashSet<Integer> hashSet = new HashSet<>();

        objectList.add(aString);
        objectList.add(anInteger);
        objectList.add(aBoolean);
        objectList.add(hashSet);
    }
}
```

The `objectList` can be modified with any parameters so that a user can conduct a more thorough test of each property. Furthermore, to better model Behavior Driven Development each options property is separated into a distinct, specifically named, test. With this framework we expect ‘N - 1’ instances of “Invalid Property” and exactly one instance of “Valid value for property!” where N is the number of distinct objects in the `objectList` array.