Design Patterns Resubmission

This is the resubmission of HW5. As the five design patterns are not related to each other, and we lose points for the Builder pattern, so we resubmit only this part.

Builder Pattern

Instead of construction same objects from the class constructor, builder pattern allows clients to create different products by calling building functions of individual parts as needed for that product. A series of parts creation methods will be called to finalized the building process.

In Cassandra, there's a abstract class called `SimpleBuilders` that declares a bunch of concrete builder as inner static class, each of them play as a different role for building something. Some concrete builders include `MutationBuilder`, `PartitionUpdateBuilder`, and `RowBuilder`.

```
1  public abstract class SimpleBuilders
2  {
3      ...
4      public static class MutationBuilder extends
   AbstractBuilder<Mutation.SimpleBuilder> implements Mutation.SimpleBuilder {}
5      public static class PartitionUpdateBuilder extends
   AbstractBuilder<PartitionUpdate.SimpleBuilder> implements
   PartitionUpdate.SimpleBuilder {}
6      public static class RowBuilder extends AbstractBuilder<Row.SimpleBuilder>
   implements Row.SimpleBuilder {}
7      ...
8  }
```

To further discover what product will be built, we chose `RowBuilder` as an example. As expected, this function will create a row instance. Below are some functions that allow whoever is creating this builder to call and add a single row or multiple rows.

```
1  public Row.SimpleBuilder add(String columnName, Object value)
2          {
3              return add(columnName, value, true);
4          }
5
6          public Row.SimpleBuilder appendAll(String columnName, Object value)
7          {
8              return add(columnName, value, false);
9          }
10
11         private Row.SimpleBuilder add(String columnName, Object value,
   boolean overwriteForCollection)
```

Similarly, instead of being able to create something, it is also able to destroy something that is already built. It has 2 delete method that overloads each other, which can perform the deletion of specified row data. (basically it add a column with a null value)

```
public Row.SimpleBuilder delete()
public Row.SimpleBuilder delete(String columnName)
        {
            return add(columnName, null);
        }
```

Other methods also gives users more flexibility of determining how the final product should be built such as returning the current column data:

```
private ColumnMetadata getColumn(String columnName)
```

This row builder class is initiated at an inner public static method of an abstract class `Row`, whose main function is dealing with almost all the row operations.

```
public abstract class Rows
{
    public static Row.SimpleBuilder simpleBuilder(TableMetadata metadata,
Object... clusteringValues)
    {
        return new SimpleBuilders.RowBuilder(metadata, clusteringValues);
    }
```

The final product is created at in a static method called `updateSizeEstimates` in `SystemKeyspace` class. This method first created a "partitionUpdate" builder with some concrete parameters. Then our "row builder" is initialized as a parameter of `add` method, which mean instead of normal builders that build parallel products at the same time, "PartitionUpdateBuilder" rely on products that is built by "Row Builder".

The row product is specified by make parts such as `timestamp`, and `add` methods.

```
    public static void updateSizeEstimates(String keyspace, String table,
Map<Range<Token>, Pair<Long, Long>> estimates)
    {
        long timestamp = FBUtilities.timestampMicros();
        PartitionUpdate.Builder update = new
PartitionUpdate.Builder(SizeEstimates,
UTF8Type.instance.decompose(keyspace),
SizeEstimates.regularAndStaticColumns(), estimates.size());
        // delete all previous values with a single range tombstone.
        int nowInSec = FBUtilities.nowInSeconds();
        update.add(new RangeTombstone(Slice.make(SizeEstimates.comparator,
table), new DeletionTime(timestamp - 1, nowInSec)));

        // add a CQL row for each primary token range.
        for (Map.Entry<Range<Token>, Pair<Long, Long>> entry :
estimates.entrySet())
        {
            Range<Token> range = entry.getKey();
            Pair<Long, Long> values = entry.getValue();
            update.add(Rows.simpleBuilder(SizeEstimates, table,
range.left.toString(), range.right.toString())
                          .timestamp(timestamp)
                          .add("partitions_count", values.left)
```

```
                                .add("mean_partition_size", values.right)
                                .build());
        }
        new Mutation(update.build()).apply();
    }
```