Suplemmentary Material for the paper

Property-based testing for Spark Streaming

published in Theory and Practice of Logic Programming

A. RIESCO

Universidad Complutense de Madrid, Spain

J. RODRÍGUEZ-HORTALÁ

(*e-mail:* ariesco@fdi.ucm.es juan.rodriguez.hortala@gmail.com)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Appendix B Introduction to Spark and Spark Streaming

Spark (Zaharia et al., 2012) is a distributed processing engine that was designed as an alternative to Hadoop MapReduce (Marz and Warren, 2015), but with a focus on iterative processing—e.g. to implement distributed machine learning algorithms—and interactive low latency jobs—e.g. for ad hoc SQL queries on massive datasets. The key to achieving these goals is an extended memory hierarchy that allows for an increased performance in many situations, and a data model based on immutable collections inspired in functional programming that is the basis for its fault tolerance mechanism. The core of Spark is a batch computing framework (Zaharia et al., 2012) that is based on manipulating so called Resilient Distributed Datasets (RDDs), which provide a fault tolerant implementation of distributed collections. Computations are defined as transformations on RDDs, that should be deterministic and side-effect free, as the fault tolerance mechanism of Spark is based on its ability to recompute any fragment (partition) of an RDD when needed. Hence Spark programmers are encouraged to define RDD transformations that are pure functions from RDD to RDD, and the set of predefined RDD transformations includes typical higher-order functions like map, filter, etc., as well as aggregations by key and joins for RDDs of key-value pairs. We can also use Spark actions, which allow us to collect results into the *driver program* or store them into an external data store. The driver program is the local process that starts the connection to the Spark cluster, and issues the execution of Spark jobs, acting as a client of the Spark cluster. Spark actions are impure, so idempotent actions are recommended in order to ensure a deterministic behavior even in the presence of recomputations triggered by the fault tolerance or speculative task execution mechanisms (Apache Spark Team, 2016). Spark is written in Scala and offers APIs for Scala, Java, Python, and R; in this work we focus on the Scala API. The example in Figure B 1 uses the Scala Spark shell to implement a variant of the famous word count example that in this case computes the number of occurrences of each character in a sentence. For that we use `parallelize`, a feature of Spark that allows us to create an RDD from

```
scala> val cs: RDD[Char] = sc.parallelize("let's count some letters", numSlices=3)
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()¡
res4: Array[(Char, Int)] = Array((t,4), ( ,3), (l,2), (e,4), (u,1), (m,1), (n,1),
                                 (r,1), ('',1), (s,3), (o,2), (c,1))
```

Fig. B 1. Letter count in Spark

```
object HelloSparkStreaming extends App {
  val conf = new SparkConf().setAppName("HelloSparkStreaming")
                            .setMaster("local[5]")
  val sc = new SparkContext(conf)
  val batchInterval = Duration(100)
  val ssc = new StreamingContext(sc, batchInterval)
  val batches = "let's count some letters, again and again"
                .grouped(4)
  val queue = new Queue[RDD[Char]]
  queue ++= batches.map(sc.parallelize(_, numSlices = 3))
  val css : DStream[Char] = ssc.queueStream(queue,
                                            oneAtATime = true)
  css.map{(_, 1)}.reduceByKey{_+_}.print()
  ssc.start()
  ssc.awaitTerminationOrTimeout(5000)
  ssc.stop(stopSparkContext = true)
}
```

```
-----------------------
Time: 1449638784400 ms
-----------------------
(e,1)
(t,1)
(l,1)
('',1)
...
-----------------------
Time: 1449638785300 ms
-----------------------
(i,1)
(a,2)
(g,1)
-----------------------
Time: 1449638785400 ms
-----------------------
(n,1)
```

Fig. B 2. Letter count in Spark Streaming

a local collection, which is useful for testing. We start with a set of chars distributed among 3 partitions, we pair each char with a 1 by using `map`, and then group by first component in the pair and sum by the second by using `reduceByKey` and the addition function (`_+_`), thus obtaining a set of (char, frequency) pairs. We collect this set into an `Array` in the driver with `collect`.

Besides the core RDD API, the Spark release contains a set of high level libraries that accelerates the development of Big Data processing applications, and that are also one of the reasons for its growing popularity. This includes libraries for scalable machine learning, graph processing, a SQL engine, and Spark Streaming, which is the focus of this work. In Spark Streaming, the notions of transformations and actions are extended from RDDs to DStreams (Discretized Streams), which are series of RDDs corresponding to splitting an input data stream into fixed time windows, also called micro batches. Micro batches are generated at a fixed rate according to the configured *batch interval*. Spark Streaming is synchronous in the sense that given a collection of input and transformed DStreams, all the batches for each DStream are generated at the same time as the batch interval is met. Actions on DStreams are also periodic and are executed synchronously for each micro batch. The code in Figure B 2 is the streaming version of the code in Figure B 1. In this case we process a DStream of characters, where batches are obtained by splitting a String into pieces by making groups (RDDs) of 4 consecutive characters. We use the testing utility class `QueueInputDStream`, which generates batches by picking RDDs from a queue, to generate the input DStream by parallelizing each substring into an RDD with 3 partitions. The program is executed using the local master mode of Spark, which replaces slave nodes in a distributed cluster by local threads, which is useful for developing and testing.

Appendix C Overview of property-based testing and ScalaCheck

Classical unit testing with xUnit-like frameworks (Meszaros, 2007) is based on specifying input – expected output pairs, and then comparing the expected output with the actual output obtained by applying the test subject to the input. On the other hand, in property-based testing (PBT) a test is expressed as a property, which is a formula in a restricted version of first order logic that relates program input and output. The testing framework checks the property by evaluating it against a bunch of randomly generated inputs. If a counterexample for the property is found then the test fails, otherwise it passes. This allows developers to obtain quite a good test coverage of the production code with a fairly small investment on development time, specially when compared to xUnit frameworks. However xUnit frameworks are still useful for testing corner cases that would be difficult to cover with a PBT property. The following is a "hello world" ScalaCheck property that checks the commutativity of addition:[1]

```
class HelloPBT extends Specification with ScalaCheck {
  def is = s2"""Hello world PBT spec,
                where int addition is commutative $intAdditionCommutative"""

  def intAdditionCommutative =
    Prop.forAll("x" |: arbitrary[Int], "y" |: arbitrary[Int]) { (x, y) =>
      x + y === y + x
  }.set(minTestsOk = 100)
}
```

PBT is based on *generators* (the functions in charge of computing the inputs, which define the domain of discourse for a formula) and *assertions* (the atoms of a formula), which together with a *quantifier* form a *property* (the formula to be checked). In the example above the universal quantifier `Prop.forAll` is used to define a property that checks whether the assertion `x + y === y + x` holds for 100 values for `x` and `y` randomly generated by two instances of the integer generator `arbitrary[Int]`. Each of those pairs of values generated for `x` and `y` is called a *test case*, and a test case that refutes the assertions of a property is called a *counterexample*. Here `arbitrary` is a higher order generator that is able to generate random values for predefined and custom types. Besides universal quantifiers, ScalaCheck supports existential quantifiers—although these are not much used in practice (Nilsson, 2014; Venners, 2015)—, and logical operators to compose properties. PBT is a sound procedure to check the validity of the formulas implied by the properties, in the sense that any counterexample that is found can be used to build a definitive proof that the property is false. However, it is not complete, as there is no guarantee that the whole space of test cases is explored exhaustively, so if no counterexample is found then we cannot conclude that the property holds for all possible test cases that could had been generated: *all failing properties are definitively false, but not all passing properties are definitively true*. PBT is a lightweight approach that does

[1] Here we use the integration of ScalaCheck with the Specs2 (Torreborre, 2014) testing library.

not attempt to perform sophisticated automatic deductions, but it provides a very fast test execution that is suitable for the test-driven development (TDD) cycle, and empirical studies (Claessen and Hughes, 2011; Shamshiri et al., 2015) have shown that in practice random PBT obtains good results, with a quality comparable to more sophisticated techniques. This goes in the line of assuming that in general testing of non trivial systems is often incomplete, as the effort of completely modeling all the possible behaviors of the system under test with test cases is not cost effective in most software development projects, except for critical systems.

Appendix D Code for AMP Camp's Twitter tutorial with sscheck

Now we will present a more complex example, adapted for Berkeley's AMP Camp training on Spark,[2] but adding sscheck properties for the functions implemented in that tutorial. The complete code for these examples is available at `https://github.com/juanrh/sscheck-examples/releases/tag/0.0.4`.

Our test subject will be an object `TweetOps`, which defines a series of operations on a stream of tweets. A tweet is a piece of text of up to 140 characters, together with some meta-information like an identifier for the author or the creation date. Those words in a tweet that start with the `#` character are called "hashtags", and are used by the tweet author to label the tweet, so other users that later search for tweets with a particular hashtag might locate those related tweets easily. The operations below take a stream of tweets and, respectively, generate the stream for the set of hashtags in all the tweets; the stream of pairs (hashtags, number of occurrences) in a sliding time window with the specified size[3]; and the stream that contains a single element for the most popular hashtag, i.e. the hashtag with the highest number of occurrences, again for the specified time window.

```
object TweetOps {
 def getHashtags(tweets: DStream[Status]): DStream[String]
 def countHashtags(batchInterval: Duration, windowSize: Int)
                  (tweets: DStream[Status]): DStream[(String, Int)]
 def getTopHashtag(batchInterval: Duration, windowSize: Int)
                  (tweets: DStream[Status]): DStream[String]
}
```

In this code, the class `twitter4j.Status` from the library Twitter4J (Yamamoto, 2010) is used to represent each particular tweet. In the original AMP Camp training, the class `TwitterUtils`[4] is used to define a `DStream[Status]` by repeatedly calling the Twitter public API to ask for new tweets. Instead, in this example we replace the Twitter API by an input DStream defined by using an sscheck generator, so we can control the shape of the tweets that will be used as the test inputs. To do that

[2] `http://ampcamp.berkeley.edu/3/exercises/realtime-processing-with-spark-streaming.html`

[3] See `https://spark.apache.org/docs/1.6.2/streaming-programming-guide.html#window-operations` for details on Spark Streaming window operators.

[4] `https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/streaming/twitter/TwitterUtils.html`

we employ the mocking (Mackinnon et al., 2001) library Mockito (Kaczanowski, 2012) for stubbing (Fowler, 2007) `Status` objects, i.e. to easily synthetize objects that impersonate a real `Status` object, and that provide predefined answers to some methods, in this case the method that returns the text for a tweet.

```
object TwitterGen {
  /** Generator of Status mocks with a getText method
   *  that returns texts of up to 140 characters
   *
   *  @param noHashtags if true then no hashtags are generated in the
   *  tweet text
   * */
  def tweet(noHashtags: Boolean = true): Gen[Status]
  /** Take a Status mocks generator and return a Status mocks
   *  generator that adds the specified hashtag to getText
   * */
  def addHashtag(hashtagGen: Gen[String])
                (tweetGen: Gen[Status]): Gen[Status]
  def tweetWithHashtags(possibleHashTags: Seq[String]): Gen[Status]
  def hashtag(maxLen: Int): Gen[String]
  def tweetWithHashtagsOfMaxLen(maxHashtagLength: Int): Gen[Status]
}
```

D.1 Extracting hashtags

Now we are ready to write our first property, which checks that `getHashtags` works correctly, that is, it computes the set of *hashtags* (words starting with `#`). In the property we generate tweets that use a predefined set of hashtags, and then we check that all hashtags produced in the output are contained in that set.

Example 1

```
def getHashtagsOk = {
  type U = (RDD[Status], RDD[String])
  val hashtagBatch = (_ : U)._2

  val numBatches = 5
  val possibleHashTags = List("#spark", "#scala", "#scalacheck")
  val tweets = BatchGen.ofNtoM(5, 10,
                               tweetWithHashtags(possibleHashTags)
                               )
  val gen = BatchGen.always(tweets, numBatches)

  val formula = always {
    at(hashtagBatch){ hashtags =>
      hashtags.count > 0 and
      ( hashtags should foreachRecord(possibleHashTags.contains(_)) )
    }
  } during numBatches

  forAllDStream(
    gen)(
    TweetOps.getHashtags)(
```

```
    formula)
}
```

In the next example we use the "reference implementation" PBT technique (Nilsson, 2014) to check the implementation of `TweetOps.getHashtags`, which is based on the Spark transformations `flatMap` and `filter` also using `String.startsWith`, against a regexp-based reference implementation. This gives us a more thorough test, because we use a different randomly generated set of hashtags for each batch of each test case, instead of a predefined set of hashtags for all the test cases.

Example 2

```
private val hashtagRe = """#\S+""".r
private def getExpectedHashtagsForStatuses(statuses: RDD[Status])
: RDD[String] =
  statuses.flatMap { status => hashtagRe.findAllIn(status.getText)}

def getHashtagsReferenceImplementationOk = {
  type U = (RDD[Status], RDD[String])
  val (numBatches, maxHashtagLength) = (5, 8)

  val tweets = BatchGen.ofNtoM(5, 10,
                               tweetWithHashtagsOfMaxLen(maxHashtagLength))
  val gen = BatchGen.always(tweets, numBatches)

  val formula = alwaysR[U] { case (statuses, hashtags) =>
    val expectedHashtags = getExpectedHashtagsForStatuses(statuses).cache()
    hashtags must beEqualAsSetTo(expectedHashtags)
  } during numBatches

  forAllDStream(
    gen)(
    TweetOps.getHashtags)(
    formula)
}
```

D.2 Counting hashtags

In order to check `countHashtags`, in the following property we setup a scenario where the hashtag `#spark` is generated for some period, and then the hashtag `#scala` is generated for another period, and we express the expected counting behaviour with several subformulas: we expect to get the expected count of hashtags for spark for the first period (`laterAlwaysAllSparkCount`); we expect to eventually get the expected count of hastags for scala (`laterScalaCount`); and we expect that after reaching the expected count for spark hashtags, we would then decrease the count as time passes and elements leave the sliding window (`laterSparkCountUntilDownToZero`).

Example 3

```
def countHashtagsOk = {
  type U = (RDD[Status], RDD[(String, Int)])
  val countBatch = (_ : U)._2
```

```
  val windowSize = 3
  val (sparkTimeout, scalaTimeout) = (windowSize * 4, windowSize * 2)
  val sparkTweet = tweetWithHashtags(List("#spark"))
  val scalaTweet = tweetWithHashtags(List("#scala"))
  val (sparkBatchSize, scalaBatchSize) = (2, 1)
  val gen = BatchGen.always(BatchGen.ofN(sparkBatchSize, sparkTweet),
                                         sparkTimeout) ++
            BatchGen.always(BatchGen.ofN(scalaBatchSize, scalaTweet),
                                         scalaTimeout)

  def countNHashtags(hashtag : String)(n : Int)  =
    at(countBatch)(_ should existsRecord(_ == (hashtag, n : Int)))
  val countNSparks = countNHashtags("#spark") _
  val countNScalas = countNHashtags("#scala") _
  val laterAlwaysAllSparkCount =
    later {
        always {
          countNSparks(sparkBatchSize * windowSize)
        } during (sparkTimeout -2)
    } on (windowSize + 1)
  val laterScalaCount =
    later {
      countNScalas(scalaBatchSize * windowSize)
    } on (sparkTimeout + windowSize + 1)
  val laterSparkCountUntilDownToZero =
    later {
      { countNSparks(sparkBatchSize * windowSize) } until {
        countNSparks(sparkBatchSize * (windowSize - 1)) and
          next(countNSparks(sparkBatchSize * (windowSize - 2))) and
          next(next(countNSparks(sparkBatchSize * (windowSize - 3))))
        } on (sparkTimeout -2)
    } on (windowSize + 1)
  val formula =
    laterAlwaysAllSparkCount and
    laterScalaCount and
    laterSparkCountUntilDownToZero

  forAllDStream(
    gen)(
    TweetOps.countHashtags(batchInterval, windowSize)(_))(
    formula)
}
```

Then we check the safety of `countHashtags` by asserting that any arbitrary generated hashtag is never skipped in the count. Here we again exploit the reference implementation technique to extract the expected hashtags, and join this with the output counts, so we can assert that all and only all expected hastags are counted, and that those countings are never zero at the time the hashtag is generated.

Example 4

```
def hashtagsAreAlwasysCounted = {
  type U = (RDD[Status], RDD[(String, Int)])
```

```
  val windowSize = 3
  val (numBatches, maxHashtagLength) = (windowSize * 6, 8)

  val tweets = BatchGen.ofNtoM(5, 10,
                               tweetWithHashtagsOfMaxLen(maxHashtagLength))
  val gen = BatchGen.always(tweets, numBatches)

  val alwaysCounted = alwaysR[U] { case (statuses, counts) =>
    val expectedHashtags = getExpectedHashtagsForStatuses(statuses).cache()
    val expectedHashtagsWithActualCount =
      expectedHashtags
       .map((_, ()))
       .join(counts)
       .map{case (hashtag, (_, count)) => (hashtag, count)}
       .cache()
    val countedHashtags = expectedHashtagsWithActualCount.map{_._1}
    val countings = expectedHashtagsWithActualCount.map{_._2}

    // all hashtags have been counted
    countedHashtags must beEqualAsSetTo(expectedHashtags) and
    // no count is zero
    (countings should foreachRecord { _ > 0 })
  } during numBatches

  forAllDStream(
    gen)(
    TweetOps.countHashtags(batchInterval, windowSize)(_))(
    alwaysCounted)

}
```

D.2.1 Getting the most popular hashtag

Now we check the correctness of `getTopHashtag`, that extracts the most "popular" hashtag, i.e. the hashtag with the highest number of occurrences at each time window. For that we use the following property where we define a scenario in which we start with the hashtag `#spark` as the most popular (generator `sparkPopular`), and after that the hashtag `#scala` becomes the most popular (generator `scalaPopular`), and asserting on the output DStream that `#spark` is the most popular hashtag until `#scala` is the most popular.

Example 5

```
def sparkTopUntilScalaTop = {
  type U = (RDD[Status], RDD[String])

  val windowSize = 1
  val topHashtagBatch = (_ : U)._2
  val scalaTimeout = 6
  val sparkPopular =
    BatchGen.ofN(5, tweetWithHashtags(List("#spark"))) +
    BatchGen.ofN(2, tweetWithHashtags(List("#scalacheck")))
```

```
  val scalaPopular =
    BatchGen.ofN(7, tweetWithHashtags(List("#scala"))) +
    BatchGen.ofN(2, tweetWithHashtags(List("#scalacheck")))
  val gen = BatchGen.until(sparkPopular, scalaPopular, scalaTimeout)

  val formula =
    { at(topHashtagBatch)(_ should foreachRecord(_ == "#spark" )) } until {
      at(topHashtagBatch)(_ should foreachRecord(_ == "#scala" ))
    } on (scalaTimeout)

  forAllDStream(
    gen)(
    TweetOps.getTopHashtag(batchInterval, windowSize)(_))(
    formula)
}
```

Finally, we state the safety of `getTopHastag` by checking that there is always one top hashtag.

Example 6

```
def alwaysOnlyOneTopHashtag = {
  type U = (RDD[Status], RDD[String])
  val topHashtagBatch = (_ : U)._2

  val (numBatches, maxHashtagLength) = (5, 8)
  val tweets =
    BatchGen.ofNtoM(5, 10,
                    tweetWithHashtagsOfMaxLen(maxHashtagLength))

  val gen = BatchGen.always(tweets, numBatches)
  val formula = always {
    at(topHashtagBatch){ hashtags =>
      hashtags.count === 1
    }
  } during numBatches

  forAllDStream(gen)(
    TweetOps.getTopHashtag(batchInterval, 2)(_))(
    formula)
}
```

D.2.2 Defining liveness properties with the consume operator

So far we have basically defined two types of properties: properties where we simulate a particular scenario, and safety properties where we assert that we will never reach a particular "bad" state. It would be also nice to be able to write liveness properties in sscheck, which is another class of properties typically used with temporal logic, where we express that something good keeps happening with a formula of the shape of $\square_{t_1}(\varphi_1 \rightarrow \lozenge_{t_2}\varphi_2)$. In this kind of formulas it would be useful to define the conclusion formula φ_2 that should happen later, based on the value of the word that happened when the premise formula φ_1 was evaluated. This was our

motivation for adding to the LTL_{ss} logic the consume operator $\lambda^o_x.\varphi$, that can be used in liveness formulas of the shape $\Box_{t_1}(\lambda^o_x.\Diamond_{t_2}\varphi_2)$ or $\Box_{t_1}(\lambda^o_x.\varphi_1 \rightarrow \Diamond_{t_2}\varphi_2)$. One example of the former is the following liveness property for `countHashtags`, that checks that always each hashtag eventually gets a count of 0, if we generate empty batches at the end of the test case so all hashtags end up getting out of the counting window.

Example 7

```
def alwaysEventuallyZeroCount = {
  type U = (RDD[Status], RDD[(String, Int)])
  val windowSize = 4
  val (numBatches, maxHashtagLength) = (windowSize * 4, 8)

  // repeat hashtags a bit so counts are bigger than 1
  val tweets = for {
    hashtags <- Gen.listOfN(6, hashtag(maxHashtagLength))
    tweets <- BatchGen.ofNtoM(5, 10,
                addHashtag(Gen.oneOf(hashtags))(tweet(noHashtags=true)))
  } yield tweets
  val emptyTweetBatch = Batch.empty[Status]
  val gen = BatchGen.always(tweets, numBatches) ++
            BatchGen.always(emptyTweetBatch, windowSize*2)

  val alwaysEventuallyZeroCount = alwaysF[U] { case (statuses, _) =>
    val hashtags = getExpectedHashtagsForStatuses(statuses)
    laterR[U] { case (_, counts) =>
      val countsForStatuses =
        hashtags
          .map((_, ()))
          .join(counts)
          .map{case (hashtag, (_, count)) => count}
      countsForStatuses should foreachRecord { _ == 0}
    } on windowSize*3
  } during numBatches

  forAllDStream(gen)(
    TweetOps.countHashtags(batchInterval, windowSize)(_))(
    alwaysEventuallyZeroCount)
}
```

One example of the second kind of liveness properties, that use an implication in the body of an always, is the following property for `getTopHashtag`, that checks that if we superpose two generators, one for a random noise of hashtags that have a small number of occurrences (generator `tweets`), and another for a periodic peak of a random hashtag that suddenly has a big number of occurrences (generator `tweetsSpike`), then each time a peak happens then the corresponding hashtag eventually becomes the top hashtag.

Example 8

```
def alwaysPeakImpliesEventuallyTop = {
```

```
  type U = (RDD[Status], RDD[String])
  val windowSize = 2
  val sidesLen = windowSize * 2
  val numBatches = sidesLen + 1 + sidesLen
  val maxHashtagLength = 8
  val peakSize = 20

  val emptyTweetBatch = Batch.empty[Status]
  val tweets =
    BatchGen.always(
      BatchGen.ofNtoM(5, 10,
                    tweetWithHashtagsOfMaxLen(maxHashtagLength)),
      numBatches)
  val popularTweetBatch = for {
    hashtag <- hashtag(maxHashtagLength)
    batch <-  BatchGen.ofN(peakSize, tweetWithHashtags(List(hashtag)))
  } yield batch
  val tweetsSpike = BatchGen.always(emptyTweetBatch, sidesLen) ++
                        BatchGen.always(popularTweetBatch, 1) ++
                        BatchGen.always(emptyTweetBatch, sidesLen)
  // repeat 6 times the superposition of random tweets
  // with a sudden spike for a random hastag
  val gen = Gen.listOfN(6, tweets + tweetsSpike).map{_.reduce(_++_)}

  val alwaysAPeakImpliesEventuallyTop = alwaysF[U] { case (statuses, _) =>
    val hashtags = getExpectedHashtagsForStatuses(statuses)
    val peakHashtags = hashtags.map{(_,1)}.reduceByKey{_+_}
          .filter{_._2 >= peakSize}.keys.cache()
    val isPeak = Solved[U] { ! peakHashtags.isEmpty }
    val eventuallyTop = laterR[U] { case (_, topHashtag) =>
      topHashtag must beEqualAsSetTo(peakHashtags)
    } on numBatches

    isPeak ==> eventuallyTop
  } during numBatches * 3

  forAllDStream(
    gen)(
    TweetOps.getTopHashtag(batchInterval, windowSize)(_))(
    alwaysAPeakImpliesEventuallyTop)
}
```

The consume operator is also useful to define other types of properties like the following, that only uses consume and next as temporal operators, but that is able to express the basic condition for counting correctly and on time. It states that for any number of repetitions n less or equal to the counting window size, and for any random word prefix, if we repeat the word prefix n times then after the $n-1$ instants we will have a count of at least (to account for hashtags randomly generated twice) n for all the hashtags in the first batch. Here we use `def next[T](times: Int)(phi: Formula[T])` that returns the result of applying next `times` times on the given formula.

Example 9

```
def forallNumRepetitionsLaterCountNumRepetitions = {
  type U = (RDD[Status], RDD[(String, Int)])
  val windowSize = 5
  val (numBatches, maxHashtagLength) = (windowSize * 6, 8)

  // numRepetitions should be <= windowSize, as in the worst case each
  // hashtag is generated once per batch before being repeated using
  // Prop.forAllNoShrink because sscheck currently does not support shrinking
  Prop.forAllNoShrink(Gen.choose(1, windowSize)) { numRepetitions =>
    val tweets = BatchGen.ofNtoM(5, 10,
                                 tweetWithHashtagsOfMaxLen(maxHashtagLength))
    val gen = for {
      tweets <- BatchGen.always(tweets, numBatches)
      // using tweets as a constant generator, to repeat each generated
      // stream numRepetitions times
      delayedTweets <- PDStreamGen.always(tweets, numRepetitions)
    } yield delayedTweets

    val laterCountNumRepetitions = nextF[U] { case (statuses, _) =>
      val hashtagsInFirstBatch = getExpectedHashtagsForStatuses(statuses)
      // -2 because we have already consumed 1 batch in the outer nextF, and
      // we will consume 1 batch in the internal now
      next(max(numRepetitions-2, 0))(now { case (_, counts) =>
        val countsForHashtagsInFirstBatch =
          hashtagsInFirstBatch
            .map((_, ()))
            .join(counts)
            .map{case (hashtag, (_, count)) => count}
        countsForHashtagsInFirstBatch should foreachRecord { _ >= numRepetitions }
      })
    }
    forAllDStream(
      gen)(
      TweetOps.countHashtags(batchInterval, windowSize)(_))(
      laterCountNumRepetitions)
  }
}
```

References

Apache Spark Team (2016). Spark programming guide. `https://spark.apache.org/docs/latest/programming-guide.html`.

Claessen, K. and Hughes, J. (2011). QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64.

Fowler, M. (2007). Mocks aren't stubs. `https://martinfowler.com/articles/mocksArentStubs.html`.

Kaczanowski, T. (2012). *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski.

Mackinnon, T., Freeman, S., and Craig, P. (2001). Endo-testing: unit testing with mock objects. *Extreme programming examined*, 1:287–302.

Marz, N. and Warren, J. (2015). *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.

Nilsson, R. (2014). *ScalaCheck: The Definitive Guide*. IT Pro. Artima Incorporated.

Shamshiri, S., Rojas, J. M., Fraser, G., and McMinn, P. (2015). Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1367–1374. ACM.

Torreborre, E. (2014). Specs2 user guide. `https://etorreborre.github.io/specs2/guide/SPECS2-3.6.2/org.specs2.guide.UserGuide.html`.

Venners, B. (2015). Re: Prop.exists and scalatest matchers. `https://groups.google.com/forum/#!msg/scalacheck/Ped7joQLhnY/gNHOSSWkKUgJ`.

Yamamoto, Y. (2010). Twitter4J-A Java library for the twitter API.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.