

Testing & Validation for Distributed Systems with Apache Spark & BEAM

Melinda
Seckington



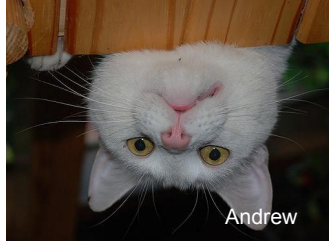
Holden:

- My name is Holden Karau
- Preferred pronouns are she/her
- Developer Advocate at Google
- Apache Spark PMC (as of December!) :)
- previously IBM, Alpine, Databricks, Google, Foursquare & Amazon
- co-author of Learning Spark & High Performance Spark
- [@holdenkarau](https://twitter.com/holdenkarau)
- Slide share <http://www.slideshare.net/hkarau>
- LinkedIn <https://www.linkedin.com/in/holdenkarau>
- Github <https://github.com/holdenk>
- Spark Videos <http://bit.ly/holdenSparkVideos>





What is going to be covered:



- A bit about why you should test & validate your distributed programs
- “Normal” unit testing in Spark (then BEAM)
- Testing at scale(ish) in Spark (then BEAM)
- Validation - how to make simple validation rules & our current limitations
- Cute & scary pictures
 - I promise at least one panda and one cat
 - If you come to my talk tomorrow you may see some duplicate cat photos, I’m sorry.

Who I think you wonderful humans are?

- Nice* people
- Like silly pictures
- Possibly Familiar with one of Scala, Java, or Python
- Possibly Familiar with one of Spark, BEAM, or a similar system (but also ok if not)
- Want to make better software
 - (or models, or w/e)
- Or just want to make software good enough to not have to keep your resume up to date



So why should you test?

- Makes you a better person
- Avoid making your users angry
- Save \$\$
 - AWS (sorry I mean Google Cloud Whatever) is expensive
- Waiting for our jobs to fail is a pretty long dev cycle
- Repeating Holden's mistakes is not fun (see misscategorized items)
- Honestly you came to the testing track so you probably already care



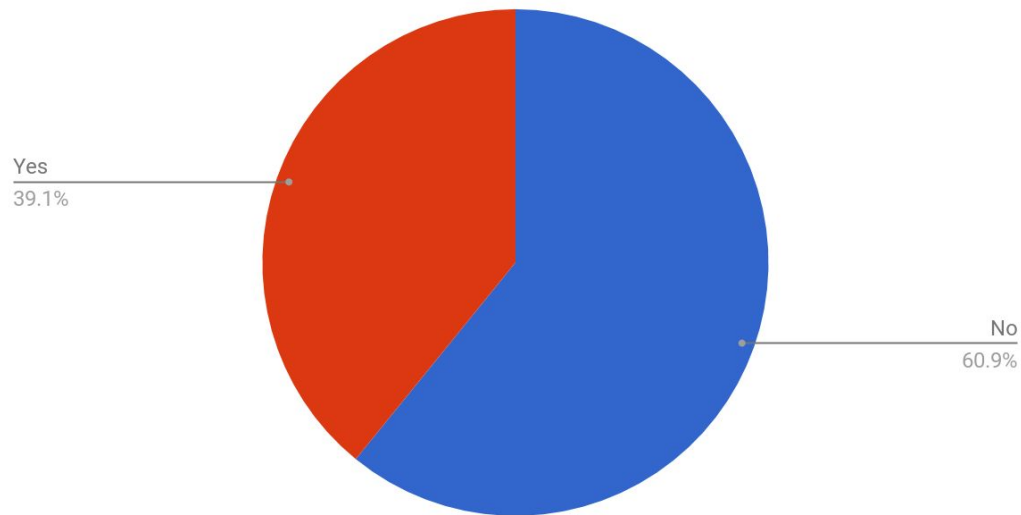
So why should you validate?

- You want to know when you're aboard the failboat
- Halt deployment, roll-back
- Our code will most likely fail
 - Sometimes data sources fail in new & exciting ways (see “Call me Maybe”)
 - That jerk on that other floor changed the meaning of a field :(
 - Our tests won't catch all of the corner cases that the real world finds
- We should try and minimize the impact
 - Avoid making potentially embarrassing recommendations
 - Save having to be woken up at 3am to do a roll-back
 - Specifying a few simple invariants isn't all that hard
 - Repeating Holden's mistakes is still not fun



So why should you test & validate:

Count of Do the results of any of your jobs get automatically deployed to production?



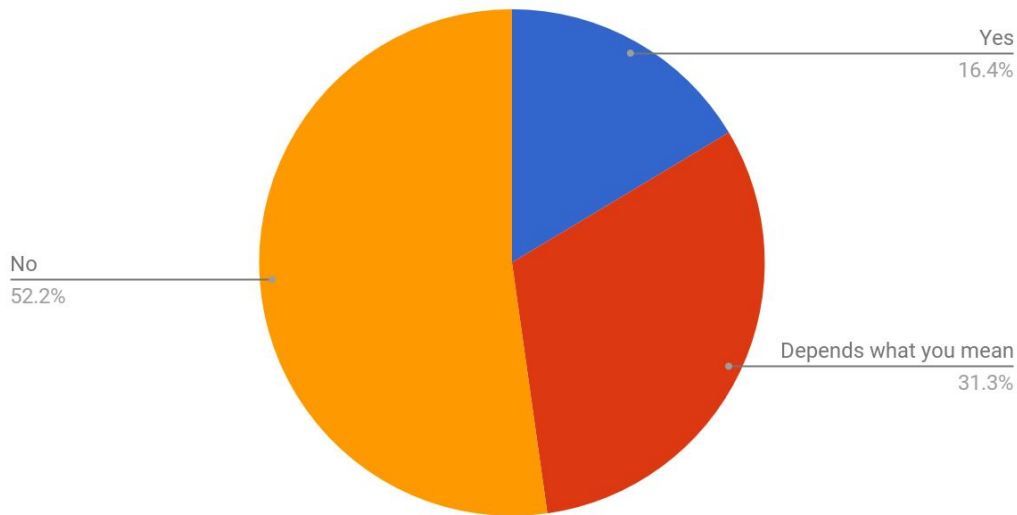
Results from: Testing with Spark survey <http://bit.ly/holdenTestingSpark>



So why should you test & validate - cont



Count of Has the output of your Spark jobs ever caused a "serious" production outage?



Results from: Testing with Spark survey <http://bit.ly/holdenTestingSpark>

Why don't we test?

- It's hard
 - Faking data, setting up integration tests
- Our tests can get too slow
 - Packaging and building scala is already sad
- It takes a lot of time
 - and people always want everything done yesterday
 - or I just want to go home see my partner
 - Etc.
- Distributed systems is particularly hard

Your application is a special snowflake



Expert

Excuses for
Not Writing Unit Tests

○ RLY?

@ThePracticalDev

Why don't we test? (continued)

```
In [17]: sc = SparkContext(master="yarn")
```

```
In [18]: sc
```

```
Out[18]: SparkContext
```

[Spark UI](#)

Version

v2.2.0

Master

yarn

AppName

pyspark-shell

```
In [19]: rdd = sc.parallelize(range(10))
```

```
In [20]: import sys
rdd.map(lambda x: (x, sys.executable)).collect()
```

```
Out[20]: [(0,
  '/hadoop/yarn/nm-local-dir/usercache/root/appcache/application_1516589627335_0012/container_1516589627335_0012_01_000005/tmp/coffee_boat_tmp_BX66Ae/auto059d3708c86a4cabb4b5ebb049e308c1/bin/python'),
  (1,
  '/hadoop/yarn/nm-local-dir/usercache/root/appcache/application_1516589627335_0012/container_1516589627335_0012_01_000005/tmp/coffee_boat_tmp_BX66Ae/auto059d3708c86a4cabb4b5ebb049e308c1/bin/python'),
  (2,
  '/hadoop/yarn/nm-local-dir/usercache/root/appcache/application_1516589627335_0012/container_1516589627335_0012_01_000005/tmp/coffee_boat_tmp_BX66Ae/auto059d3708c86a4cabb4b5ebb049e308c1/bin/python'),
  (3,
  '/hadoop/yarn/nm-local-dir/usercache/root/appcache/application_1516589627335_0012/container_1516589627335_0012_01_000005/tmp/coffee_boat_tmp_BX66Ae/auto059d3708c86a4cabb4b5ebb049e308c1/bin/p
```



Expert

Excuses for
Not Writing Unit Tests

ORLY?

@ThePracticalDev

Why don't we validate?

- We already tested our code
- What could go wrong?

Also extra hard in distributed systems

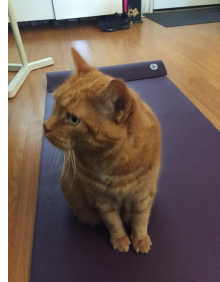
- Distributed metrics are hard
- not much built in (not very consistent)
- not always deterministic
- Complicated production systems





A simple unit test with spark-testing-base

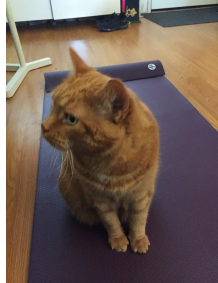
```
class SampleRDDTest extends FunSuite with SharedSparkContext {  
  test("really simple transformation") {  
    val input = List("hi", "hi holden", "bye")  
    val expected = List(List("hi"), List("hi", "holden"), List("bye"))  
    assert(SampleRDD.tokenize(sc.parallelize(input)).collect().toList === expected)  
  }  
}
```



A simple unit test with BEAM (no libs!)

```
PCollection<KV<String, Long>> filteredWords = p.apply(...)
List<KV<String, Long>> expectedResults = Arrays.asList(
    KV.of("Flourish", 3L),
    KV.of("stomach", 1L));
PAssert.that(filteredWords).containsInAnyOrder(expectedResults);

p.run().waitUntilFinish();
```



Where do those run?



- By default your local host with a “local mode”
- Spark’s local mode attempts to simulate a “real” cluster
 - Attempts but it is not perfect
- BEAM’s local mode is a “DirectRunner”
 - This is super fast
 - But think of it as more like a mock than a test env
- You can point either to a “local” cluster
 - Feeling fancy? Use docker
 - Feeling not-so-fancy? Run worker and master on localhost...
 - Note: with BEAM different runners have different levels of support so choose the one matching production

So: setup.sh (or travis.yml or...)

```
pushd spark-2.2.0-bin-hadoop2.7  
./sbin/start-master.sh -h localhost  
./sbin/start-slave.sh spark://localhost:7077  
export SPARK_MASTER="spark://localhost:7077"
```



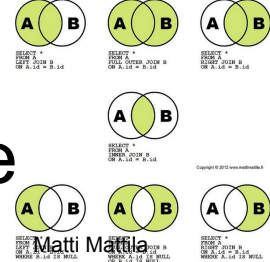
Ok but what about problems @ scale

- Maybe our program works fine on our local sized input
- Our actual workload is probably larger than 3kb
- We need to test partitioning logic, scaling, reading writing, serializing



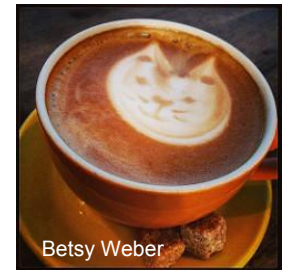
But how do we test workloads too large for a single machine?

(We can't just use parallelize and collect) and we need to run on something more like a cluster



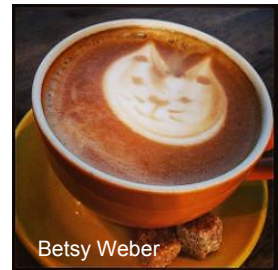
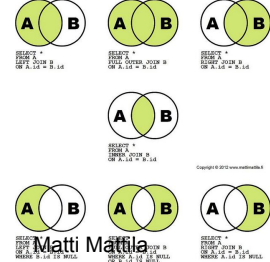
Spark: Distributed “set” operations to the rescue

- Pretty close - already built into Spark
- Doesn't do so well with floating points :(
 - damn floating points keep showing up everywhere :p
- Doesn't really handle duplicates very well
 - {“coffee”, “coffee”, “panda”} **!=** {“panda”, “coffee”} but with set operations...
- Or use RDDComparisions in your favourite** testing library
 - will cogroup / zip as appropriate to compare the results. Asserts when people steal your coffee

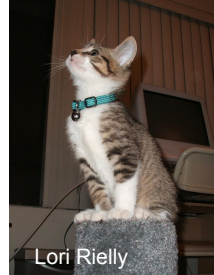


BEAM: CoGroupByKey + DoFn + PAssert

- CoGroup dataset with the dataset for the expected result
- Filter out records where everything matches
- PAssert that the result is empty

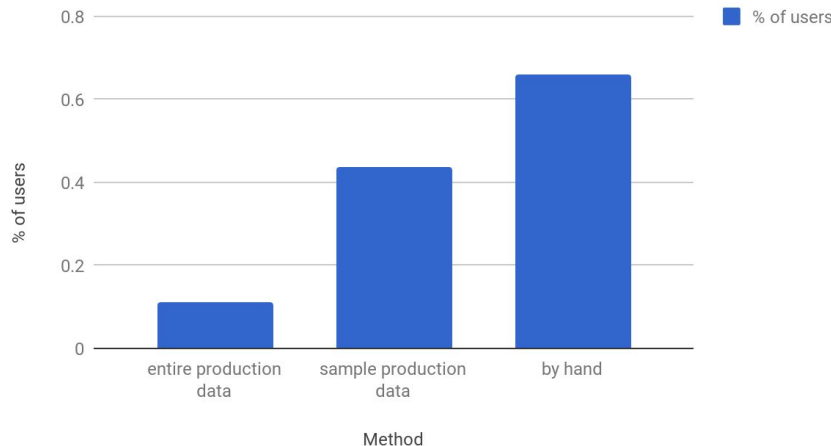


But where do we get the data for those tests?



- Most people generate data by hand
- If you have production data you can sample you are lucky!
 - If possible you can try and save in the same format
- If our data is a bunch of Vectors or Doubles Spark's got tools :)
- Coming up with good test data can take a long time
- Important to test different distributions, input files, empty partitions etc.

How we get our test data

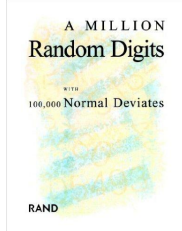


Property generating libs: QuickCheck / ScalaCheck

- QuickCheck (haskell) generates tests data under a set of constraints
- Scala version is ScalaCheck - supported by the two unit testing libraries for Spark
- Sscheck (scala check for spark)
 - Awesome people*, supports generating DStreams too!
- spark-testing-base
 - Also Awesome people*, generates more pathological (e.g. empty partitions etc.) RDDs

*I assume





With spark-testing-base

```
test("map should not change number of elements") {  
  forAll(RDDGenerator.genRDD[String](sc)){  
    rdd => rdd.map(_.length).count() == rdd.count()  
  }  
}
```

With spark-testing-base & a million entries



```
test("map should not change number of elements") {  
  implicit val generatorDrivenConfig =  
    PropertyCheckConfig(minSize = 0, maxSize = 1000000)  
  val property = forAll(RDDGenerator.genRDD[String](sc)){  
    rdd => rdd.map(_.length).count() == rdd.count()  
  }  
  check(property)  
}
```


But that can get a bit slow for all of our tests

- Not all of your tests should need a cluster (or even a cluster simulator)
- If you are ok with not using lambdas everywhere you can factor out that logic and test normally
- Or if you want to keep those lambdas - or verify the transformations logic without the overhead of running a local distributed systems you can try a library like [kontextfrei](#)
 - Don't rely on this alone (but can work well with something like scalacheck)

e.g. Lambdas aren't always your friend



- Lambda's can make finding the error more challenging
- I love `lambda x, y: x / y` as much as the next human but when `y` is zero :(
- A small bit of refactoring for your tests never hurt*
- Difficult to put logs inside of them

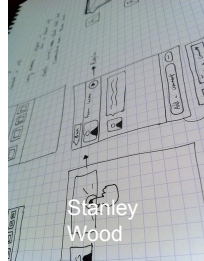
*A blatant lie, but.... it hurts less often than it helps

“Business logic” only w/o - refactor for testing

```
def difficultTokenizerRDD(input: RDD[String]) = {  
    input.flatMap(line => line.split(" "))  
}
```

=>

```
def tokenizeRDD(input: RDD[String]) = {  
    input.flatMap(tokenize)  
}  
  
protected[tokenize] def tokenize(line: String) = {  
    line.split(" ")  
}
```



.... And Spark Datasets

- We can do the same as we did for RDD's (.rdd)
- Inside of Spark validation looks like:

```
def checkAnswer(df: Dataset[T], expectedAnswer: T*)
```

- Sadly it's not in a published package & local only
- instead we expose:

```
def equalDatasets(expected: Dataset[U], result: Dataset[U]) {
```

```
def approxEqualDatasets(e: Dataset[U], r: Dataset[U], tol: Double) {
```



This is what it looks like:

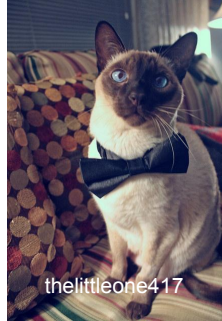
```
test("dataframe should be equal to its self") {  
  val sqlCtx = sqlContext  
  import sqlCtx.implicits._ // Yah I know this is ugly  
  val input = sc.parallelize(inputList).toDF  
  equalDataFrames(input, input)  
}
```



wenliang chen

Or with a generator based on Schema*:

```
test("assert rows' types like schema type") {  
  val schema = StructType(List(StructField("name", StringType)))  
  val rowGen: Gen[Row] =  
    DataFrameGenerator.getRowGenerator(schema)  
  val property =  
    forAll(rowGen) {  
      row => row.get(0).isInstanceOf[String]  
    }  
  check(property)  
}
```



thelittleone417

*For simple schemas, complex types in future versions

Which has “built-in” large support :)



Testing streaming....



Testing streaming the happy panda way

- Creating test data is hard
- Collecting the data locally is ugly
- figuring out when your test is “done”

Let's abstract all that away into **testOperation**



A simple (non-scalable) stream test:

```
test("really simple transformation") {  
    val input = List(List("hi"), List("hi holden"), List("bye"))  
    val expected = List(List("hi"), List("hi", "holden"), List("bye"))  
    testOperation[String, String](input, tokenize _, expected, useSet = true)  
}
```



And putting the two together: Structured Streaming

Structured data, in real time, new
api!

What could go wrong?



Simplest test, it sort of works! (1/2)



```
// Input data source
```

```
val input = MemoryStream[Int]
```

```
// Your business logic goes here. Much business very logic
```

```
val transformed = bussinesLogic(input.toDF())
```

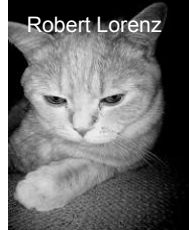
```
val query = transformed.writeStream
```

```
    .format("memory")
```

```
    .outputMode("append")
```

```
    .queryName("memStream")
```

```
    .start()
```



Simplest test, it sort of works! (2/2)

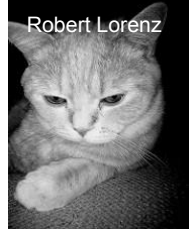
// Add some data

```
input.addData(1, 2, 3)
```

```
query.processAllAvailable()
```

```
val resultRows = spark.table("memStream").as[T]
```

// Now make it into a data frame and check as previously.



ooor...

```
def businessLogic(input: Dataset[Int]): Dataset[String] = {  
    // Business logic goes here  
}
```

```
testSimpleStreamEndState(spark, input, expected, "append",  
    businessLogic)
```

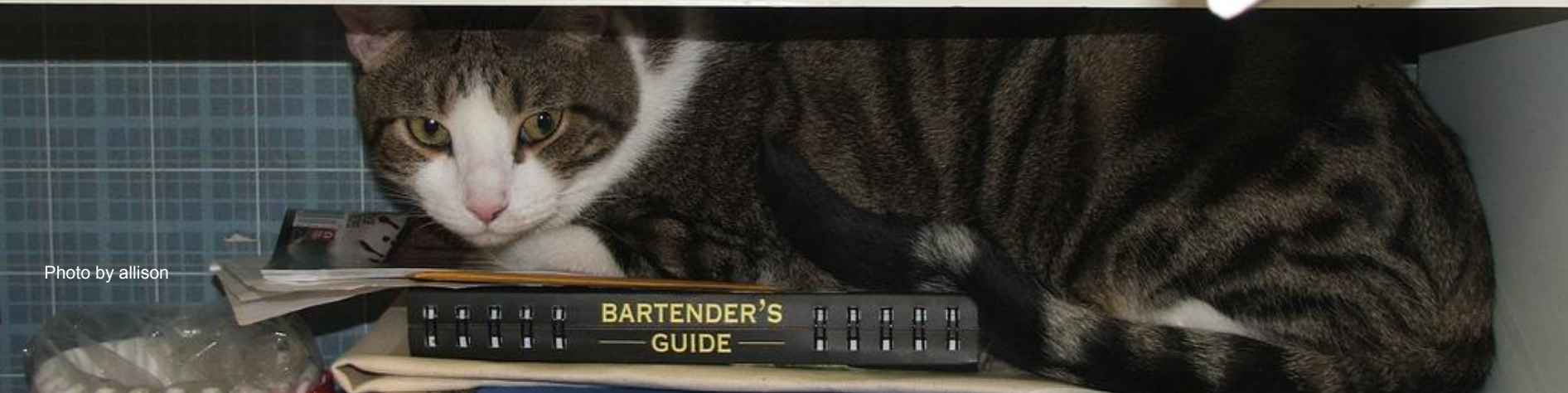
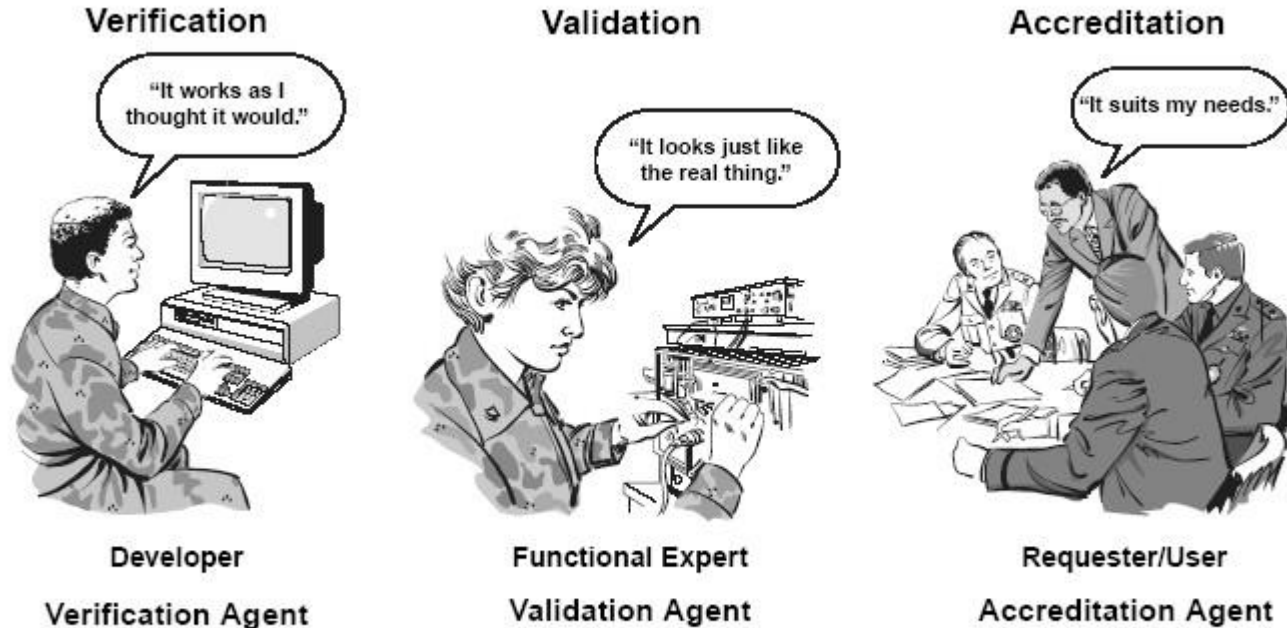


Photo by allison

On to validation*



As design matures, re-examine basic assumptions.

*Can be used during integration tests to further validate integration results

So how do we validate our jobs?



Photo by:
Paul Schadler

- Both BEAM & Spark has it own counters
 - Per-stage bytes r/w, shuffle r/w, record r/w. execution time, etc.
 - In UI can also register a listener from spark validator project
- We can add counters for things we care about
 - invalid records, users with no recommendations, etc.
 - Accumulators have some challenges (see [SPARK-12469](#) for progress) but are an interesting option
- We can write rules for if the values are expected
 - Simple rules ($X > J$)
 - The number of records should be greater than 0
 - Historic rules ($X > \text{Avg}(\text{last}(10, J))$)
 - We need to keep track of our previous values - but this can be great for debugging & performance investigation too.

Validation



Photo by:
Paul Schadler

- For now checking file sizes & execution time seem like the most common best practice (from survey)
- [spark-validator](#) is still in early stages and not ready for production use but interesting proof of concept
- Doesn't need to be done in your Spark job (can be done in your scripting language of choice with whatever job control system you are using)
- Sometimes your rules will miss-fire and you'll need to manually approve a job - that is ok!
- Remember those property tests? Could be great Validation rules!

So using names & logging & accs could be:

```
data = sc.parallelize(range(10))
rejectedCount = sc.accumulator(0)
def loggedDivZero(x):
    import logging
    try:
        return [x / 0]
    except Exception as e:
        rejectedCount.add(1)
        logging.warning("Error found " + repr(e))
        return []
transform1 = data.flatMap(loggedDivZero)
transform2 = transform1.map(add1)
transform2.count()
print("Reject " + str(rejectedCount.value))
```

Using a Spark accumulator for validation:

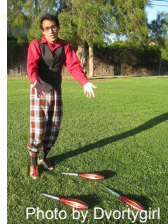


```
val (ok, bad) = (sc.accumulator(0), sc.accumulator(0))
val records = input.map{ x => if (isValid(x)) ok += 1 else bad += 1
  // Actual parse logic here
}
// An action (e.g. count, save, etc.)
if (bad.value > 0.1 * ok.value) {
  throw Exception("bad data - do not use results")
  // Optional cleanup
}
// Mark as safe
```

P.S: If you are interested in this check out [spark-validator](#) (still early stages).

Validating records read matches our expectations:

```
val vc = new ValidationConf(tempPath, "1", true,
  List[ValidationRule](
    new AbsoluteSparkCounterValidationRule("recordsRead", Some(30),
Some(1000)))
)
val sqlCtx = new SQLContext(sc)
val v = Validation(sc, sqlCtx, vc)
//Business logic goes here
assert(v.validate(5) === true)
}
```



Counters in BEAM: (1 of 2)



```
private final Counter matchedWords =  
Metrics.counter(FilterTextFn.class, "matchedWords");  
private final Counter unmatchedWords =  
Metrics.counter(FilterTextFn.class, "unmatchedWords");
```

*// Your special business logic goes here (aka shell out to Fortran
or Cobol)*

Counters in BEAM: (2 of 2)



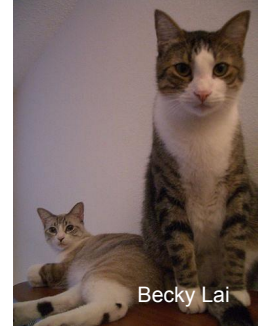
```
Long matchedWordsValue = metrics.metrics().queryMetrics(  
    new MetricsFilter.Builder()  
    .addNameFilter("matchedWords").counters().next().committed();
```

```
Long unmatchedWordsValue = metrics.metrics().queryMetrics(  
    new MetricsFilter.Builder()  
    .addNameFilter("unmatchedWords").counters().next().committed());
```

```
assertThat("unmatchWords less than matched words",  
    unmatchedWordsValue,  
    lessThan(matchedWordsValue));
```

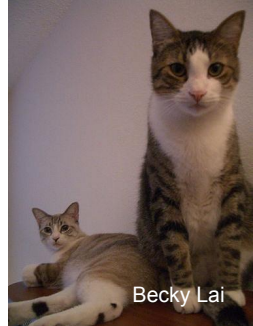
Related talks & blog posts

- [Testing Spark Best Practices \(Spark Summit 2014\)](#)
- [Every Day I'm Shuffling \(Strata 2015\)](#) & [slides](#)
- [Spark and Spark Streaming Unit Testing](#)
- [Making Spark Unit Testing With Spark Testing Base](#)
- [Testing strategy for Apache Spark jobs](#)
- [The BEAM programming guide](#)



Related packages

- spark-testing-base: <https://github.com/holdenk/spark-testing-base>
- sscheck: <https://github.com/juanrh/sscheck>
- spark-validator: <https://github.com/holdenk/spark-validator> *Proof of concept, but updated*
- spark-perf - <https://github.com/databricks/spark-perf>
- spark-integration-tests - <https://github.com/databricks/spark-integration-tests>
- scalacheck - <https://www.scalacheck.org/>



And including spark-testing-base up to spark 2.2

sbt:

```
"com.holdenkarau" %% "spark-testing-base" % "2.2.0_0.8.0" % "test"
```

Maven:

```
<dependency>
```

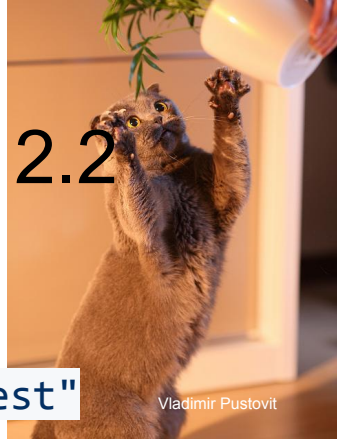
```
  <groupId>com.holdenkarau</groupId>
```

```
  <artifactId>spark-testing-base_2.11</artifactId>
```

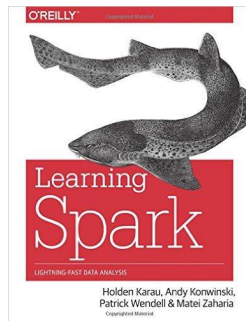
```
  <version>${spark.version}_0.8.0</version>
```

```
  <scope>test</scope>
```

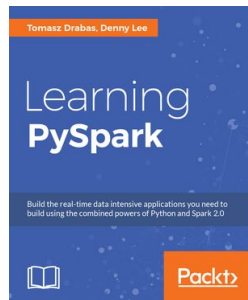
```
</dependency>
```



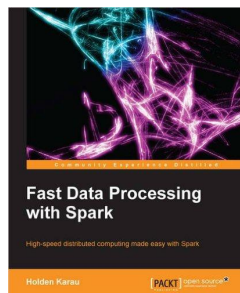
Vladimir Pustovit



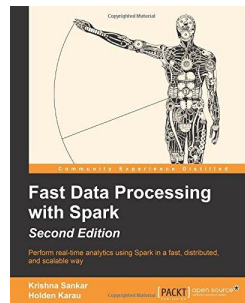
Learning Spark



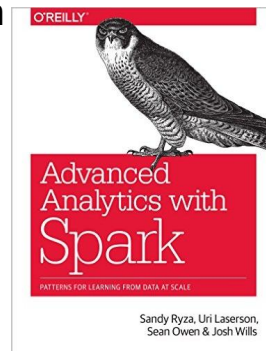
Learning PySpark



Fast Data
Processing with
Spark
(Out of Date)



Fast Data
Processing with
Spark
(2nd edition)



Advanced
Analytics with
Spark



Spark in Action



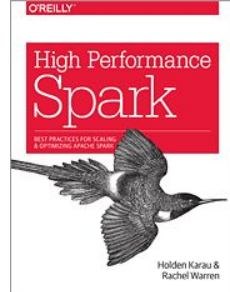
High Performance Spark

High Performance Spark!

Available today!

You can buy it from that scrappy Seattle bookstore, Jeff Bezos needs another newspaper and I want a cup of coffee.

<http://bit.ly/hkHighPerfSpark>



And some upcoming talks:



- FOSDEM HPC room tomorrow @ 4pm
 - Maybe office hours after that if there is interest?
- JFokus
- Strata San Jose
- Strata London
- QCon Brasil
- QCon AI SF
- Know of interesting conferences/webinar things that should be on my radar? Let me know!

A close-up photograph of a black and white cat with striking yellow eyes. The cat is being held by a person's hands, with its front paws visible. The cat has a white patch on its face and chest, and black fur on its head and ears. The background is blurred, showing a person's arm and a grey shirt.

k thnx bye!

If you want to fill out survey:
<http://bit.ly/holdenTestingSpark>

I will use update results in &
give the talk again the next
time Spark adds a major
feature.

Give feedback on this presentation
<http://bit.ly/holdenTalkFeedback>

Other options for generating data:

- mapPartitions + Random + custom code
- RandomRDDs in mllib
 - Uniform, Normal, Poisson, Exponential, Gamma, logNormal & Vector versions
 - Different type: implement the RandomDataGenerator interface
- Random



RandomRDDs

```
val zipRDD = RandomRDDs.exponentialRDD(sc, mean = 1000, size = rows).map(_.toInt.toString)
val valuesRDD = RandomRDDs.normalVectorRDD(sc, numRows = rows, numCols = numCols).repartition(zipRDD.partitions.size)
val keyRDD = sc.parallelize(1L.to(rows), zipRDD.getNumPartitions)
keyRDD.zipPartitions(zipRDD, valuesRDD){
  (i1, i2, i3) =>
    new Iterator[(Long, String, Vector)] {
      ...
    }
}
```



Testing libraries:

- Spark unit testing
 - spark-testing-base - <https://github.com/holdenk/spark-testing-base>
 - sscheck - <https://github.com/juanrh/sscheck>
- Simplified unit testing (“business logic only”)
 - kontextfrei - <https://github.com/dwestheide/kontextfrei> *
- Integration testing
 - spark-integration-tests (Spark internals) - <https://github.com/databricks/spark-integration-tests>
- Performance
 - spark-perf (also for Spark internals) - <https://github.com/databricks/spark-perf>
- Spark job validation
 - spark-validator - <https://github.com/holdenk/spark-validator> *

*Early stage or work-in progress, or proof of concept



Let's talk about local mode



- It's *way* better than you would expect*
- It does its best to try and catch serialization errors
- It's still not the same as running on a “real” cluster
- Especially since if we were just local mode, parallelize and collect might be fine

Options beyond local mode:

- Just point at your existing cluster (set master)
- Start one with your shell scripts & change the master
 - Really easy way to plug into existing integration testing
- [spark-docker](#) - hack in our own tests
- YarnMiniCluster
 - <https://github.com/apache/spark/blob/master/yarn/src/test/scala/org/apache/spark/deploy/yarn/BaseYarnClusterSuite.scala>
 - In Spark Testing Base extend **SharedMiniCluster**
 - Not recommended until after [SPARK-10812](#) (e.g. 1.5.2+ or 1.6+)



Integration testing - docker is awesome

- Spark-docker, kafka-docker, etc.
 - Not always super up to date sadly - if you are last stable release A-OK, if you build from master - sad pandas
- Or checkout JuJu Charms (from Canonical) - <https://jujucharms.com/>
 - Makes it easy to deploy a bunch of docker containers together & configured in a reasonable way.



Setting up integration on Yarn/Mesos

- So lucky!
- You can write your tests in the same way as before - just read from your test data sources
- Missing a data source?
 - Can you sample it or fake it using the techniques from before?
 - If so - do that and save the result to your integration environment
 - If not... well good luck
- Need streaming integration?
 - You will probably need a second Spark (or other) job to generate the test data



“Business logic” only test w/[kontextfrei](#)

```
import com.danielwestheide.kontextfrei.DCollectionOps
trait UsersByPopularityProperties[DColl[_]] extends
BaseSpec[DColl] {
  import DCollectionOps.Imports._
  property("Each user appears only once") {
    forAll { starredEvents: List[RepoStarred] =>
      val result =
logic.usersByPopularity(unit(starredEvents)).collect().toList
      result.distinct mustEqual result
    }
  }
}
... (continued in example/src/test/scala/com/danielwestheide/kontextfrei/example/)
```

Generating Data with Spark

```
import org.apache.spark.mllib.random.RandomRDDs
...
RandomRDDs.exponentialRDD(sc, mean = 1000, size = rows)
RandomRDDs.normalVectorRDD(sc, numRows = rows, numCols = numCols)
```