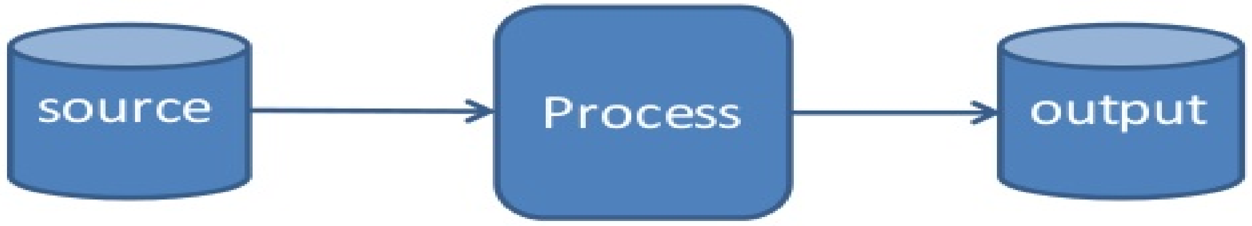
Automatic Checkpointing in Spark

Nimbus Goehausen
Spark Platform Engineer
ngoehausen@bloomberg.net

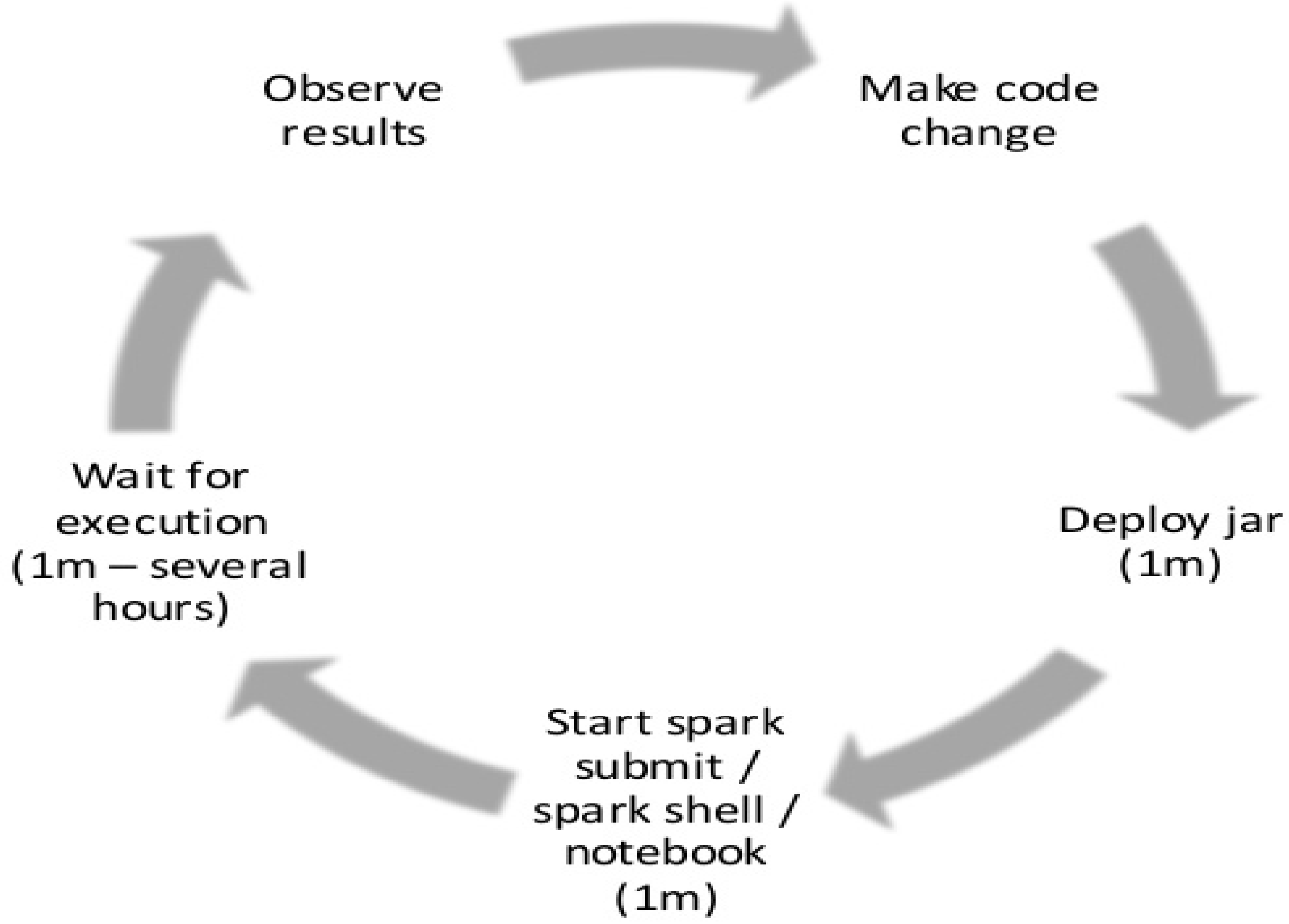
Bloomberg

A Data Pipeline

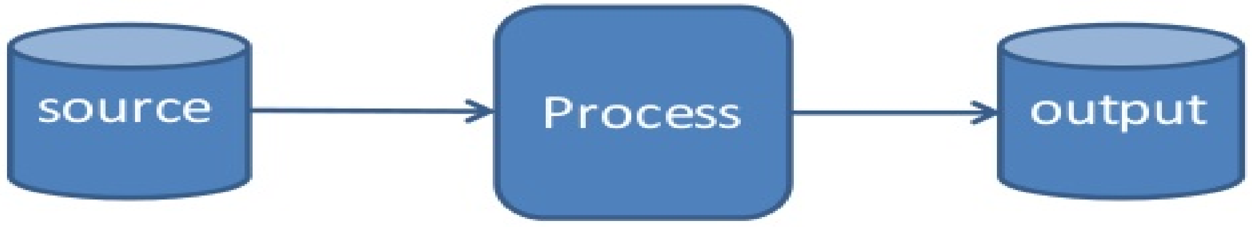


output = Process(source)

Spark development cycle

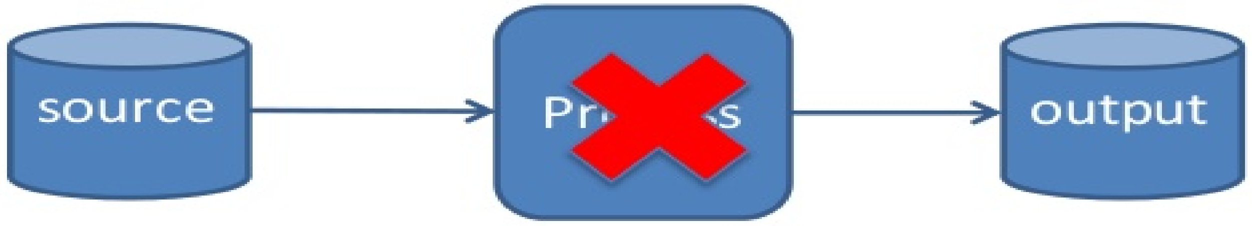

Observe results
Make code change
Deploy jar (1m)
Start spark submit / spark shell / notebook (1m)
Wait for execution (1m – several hours)

This should work, right?



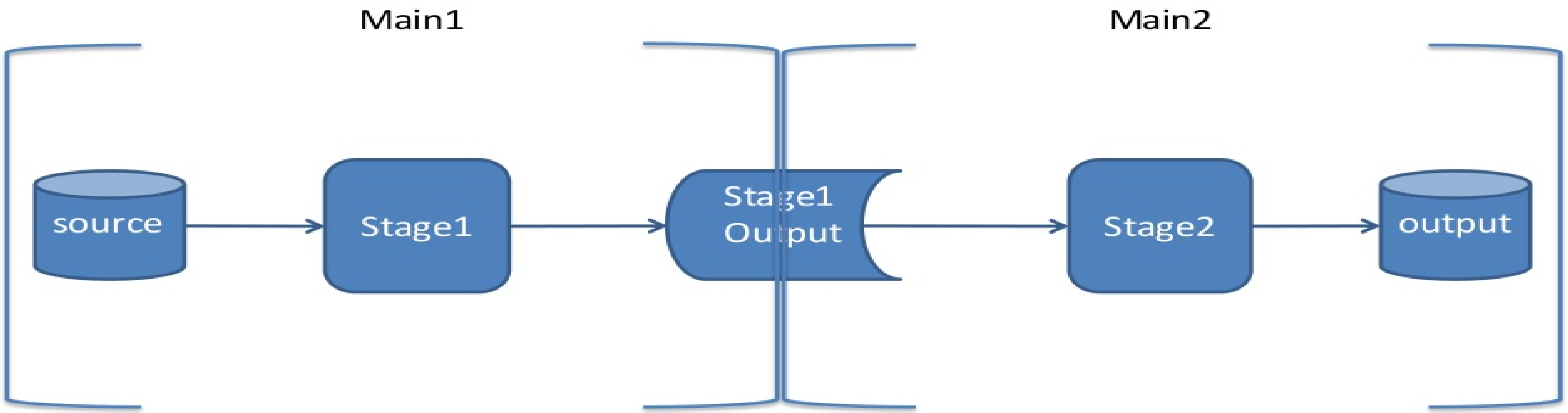
output = Process(source)

What have I done to deserve this?



```
java.lang.HotGarbageException:  You messed  up
        at some.obscure.library.Util$$anonfun$morefun$1.apply(Util.scala:eleventyBillion)
        at another.library.InputStream $$somuchfun$mcV$sp.read(InputStream.scala:42)
        at org.apache.spark.serializer.PotatoSerializer.readPotato(PotatoSerializer.scala:2792)
...
```

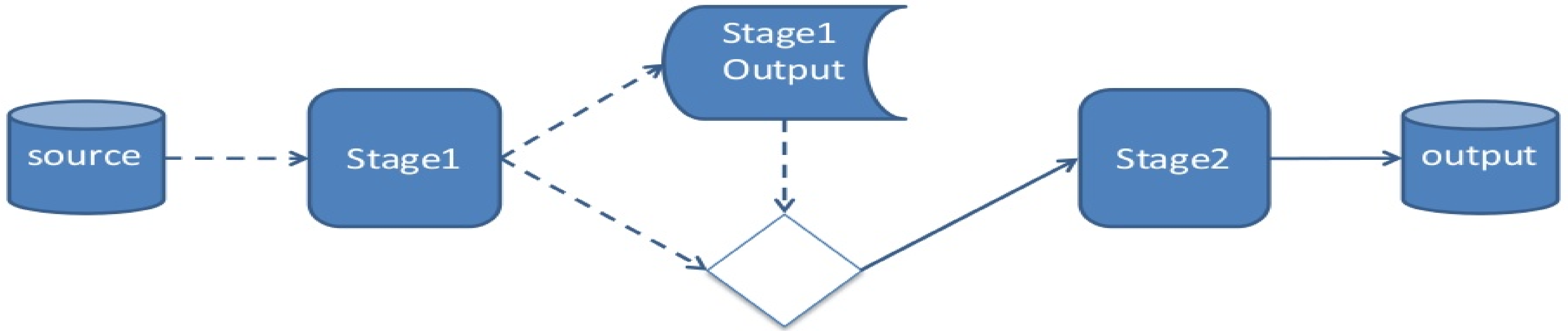
Split it up



```
stage1Out = stage1(source)
stage1Out.save("stage1Path")
```

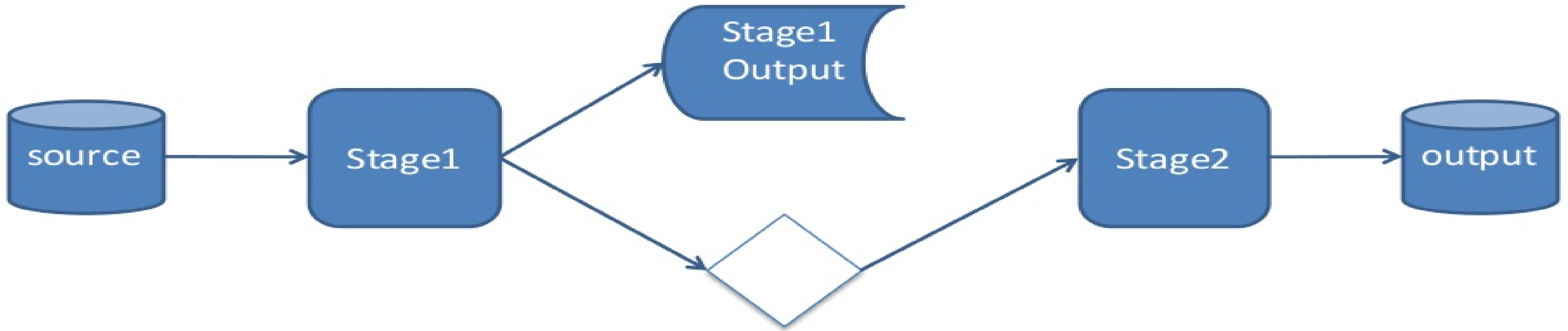
```
stage1Out = load("stage1Path")
output = stage2(stage1Out)
```

Automatic checkpoint



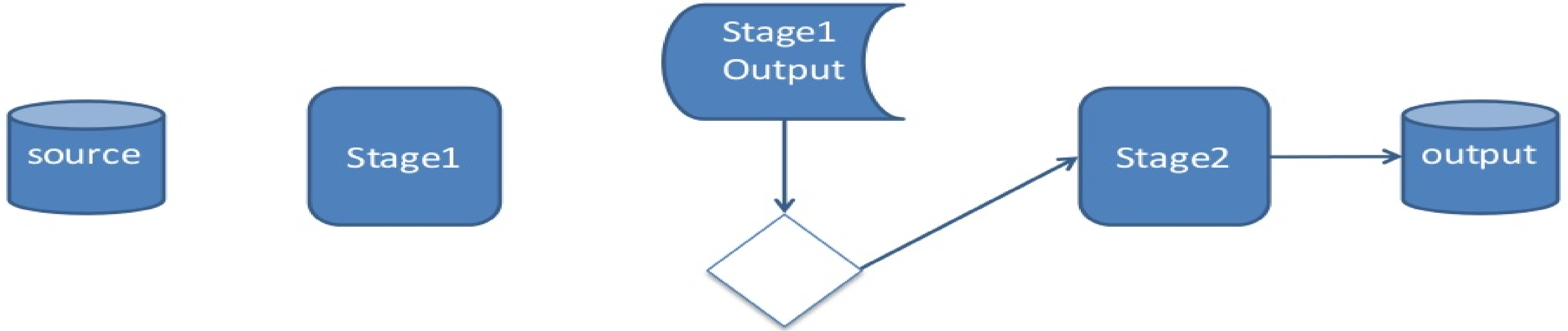
```
stage1Out = stage1(source).checkpoint()
Output = stage2(stage1Out)
```

First run



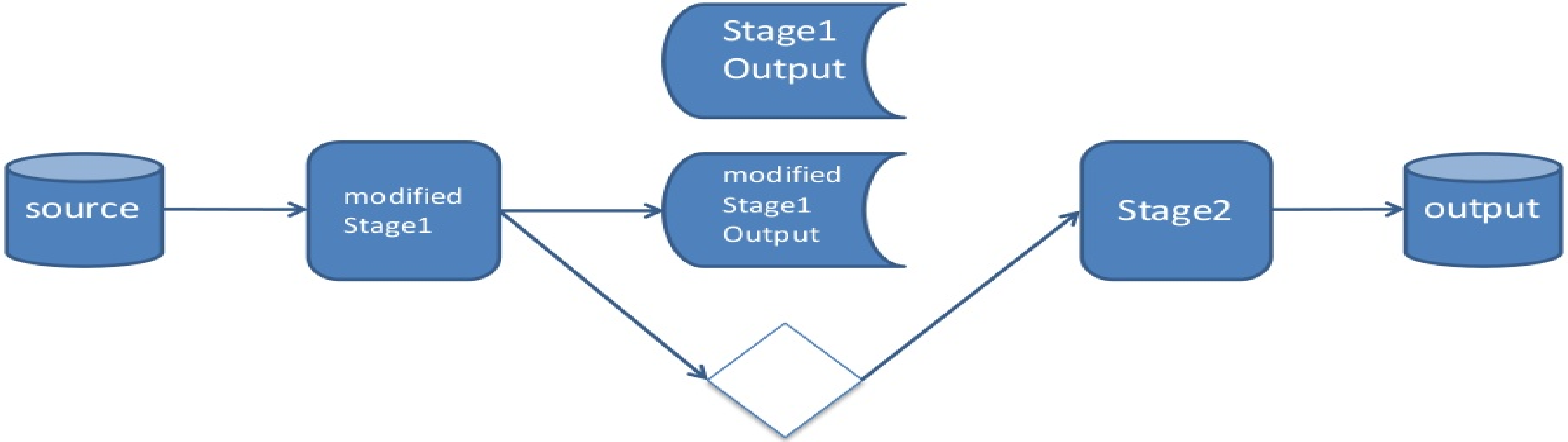
```
stage1Out = stage1(source).checkpoint()
Output = stage2(stage1Out)
```

Second run



```
stage1Out = stage1(source).checkpoint()
Output = stage2(stage1Out)
```

Change logic



```
stage1Out = stage1(source).checkpoint()
Output = stage2(stage1Out)
```

How do we do this?

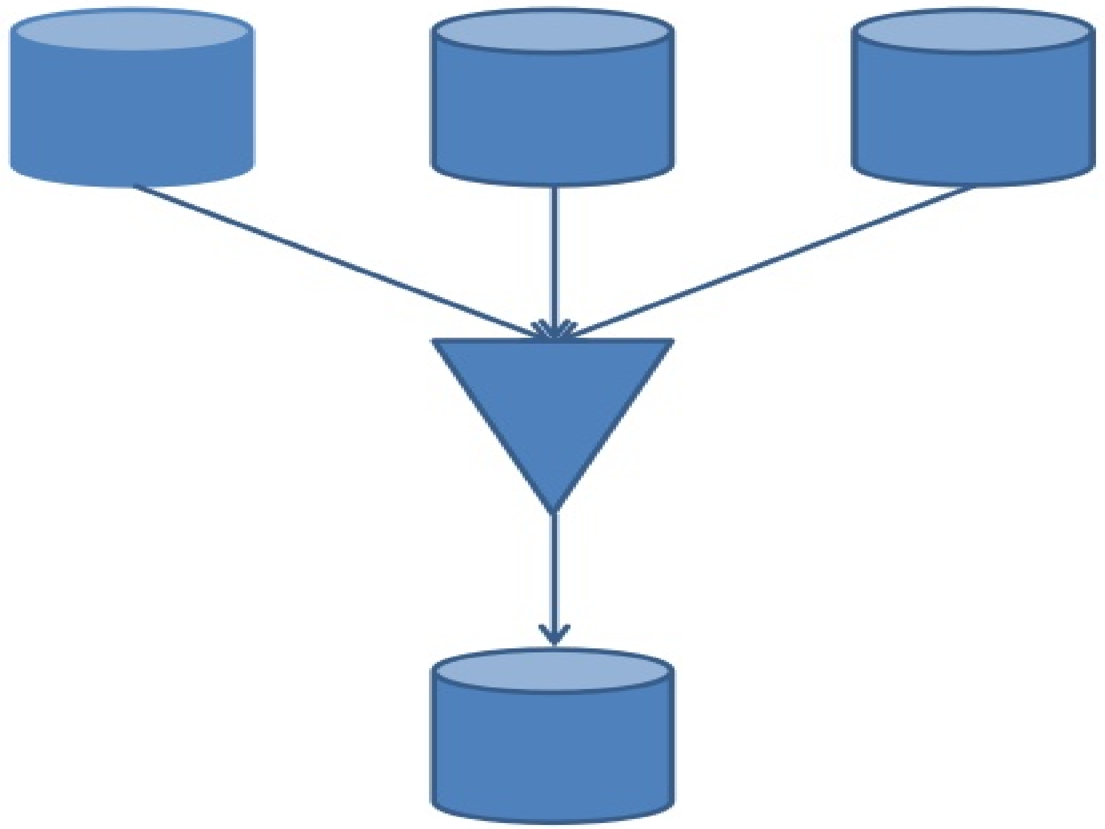
- Need some way of automatically generating a path to save to or load from
- Should stay the same when we want to reload the same data
- Should change when we want to generate new data

Refresher: what defines an RDD

Dependencies

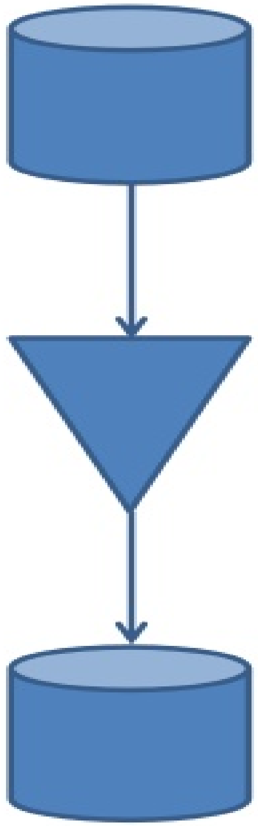
Logic

Result



Bloomberg

Refresher: what defines an RDD



Dependencies	[1,2,3,4,5...]
Logic	map(x => x*2)
Result	[2,4,6,8,10...]

A logical signature

result = logic(dependencies)

signature(result) = hash(signature(dependencies) + hash(logic))

A logical signature

```
val rdd2 = rdd1.map(f)
signature(rdd2) == hash(signature(rdd1)  + hash(f))

val sourceRDD = sc.textFile(uri)
signature(sourceRDD) == hash("textFile:" + uri)
```

Wait, how do you hash a function?

```
val f = (x: Int) => x * 2
```

f compiles to a class file of jvm bytecode

Hash those bytes

Change f to

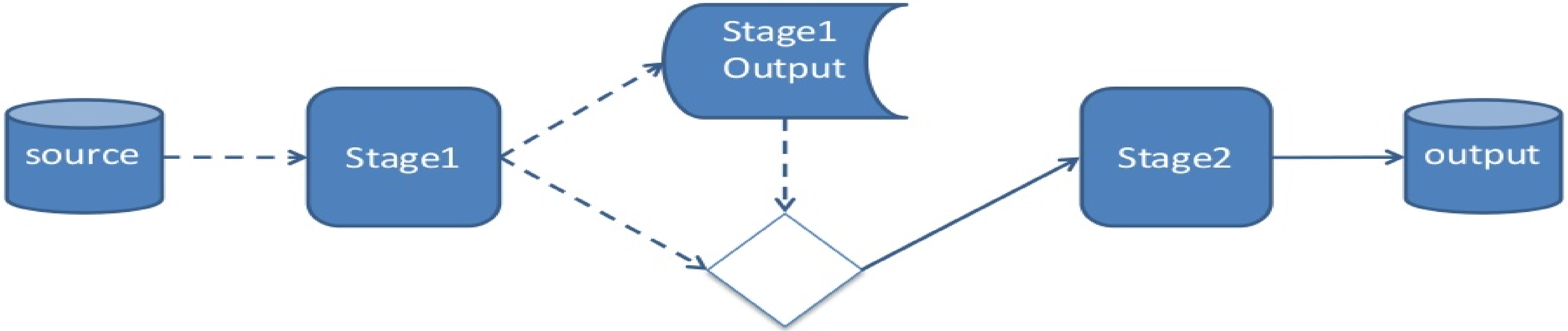
```
val f = (x: Int) => x * 3
```

And the class file will change

What if the function references other functions or static methods?

- f = (x: Int) => Values.v + staticMethod(x) + g(x)
- This is the hard part
- Must follow dependencies recursively, identify all classfiles that the function depends on
- Make a combined hash of the classfiles and any runtime values

Putting this together



```
stage1Out = stage1(source).checkpoint()
Output = stage2(stage1Out)
```

How to switch in checkpoints

- One option: change spark.
- Other option: build on top of spark.

Distributed Collection (DC)

- Mostly same api as RDD, maps, flatmaps and so on
- Can be instantiated without a spark context
- Pass in a spark context to get the corresponding RDD
- Computes logical signature upon definition, before materialization

RDD vs DC

```
val numbers: RDD[Int] = sc.parallelize(1 to 10)
val filtered: RDD[Int] = numbers.filter(_ < 5)
val doubled: RDD[Int] = filtered.map(_ * 2)
```

```
val numbers: DC[Int] = parallelize(1 to 10)
val filtered: DC[Int] = numbers.filter(_ < 5)
val doubled: DC[Int] = filtered.map(_ * 2)

val doubledRDD: RDD[Int] = doubled.getRDD(sc)
```

Easy checkpoint

```
val numbers: DC[Int] = parallelize(1 to 10)
val filtered: DC[Int] = numbers.filter(_ < 5).checkpoint()
val doubled: DC[Int] = filtered.map(_ * 2)
```

Problem: Non lazy definition

```
val numbers: RDD[Int] = sc.parallelize(1 to 10)
val doubles: RDD[Double] = numbers.map(x => x.toDouble)
val sum: Double = doubles.sum()
val normalized: RDD[Double] = doubles.map(x => x / sum)
```

We can’t get a signature of `normalized` without computing the sum!

Solution: Deferred Result

- Defined by a function on an RDD
- Like a DC but contains only one element
- Pass in a spark context to compute the result
- Also has a logical signature
- We can now express entirely lazy pipelines

Deferred Result example

```
val numbers: DC[Int] = parallelize(1 to 10)
val doubles: DC[Double] = numbers.map(x => x.toDouble)
val sum: DR[Double] = doubles.sum()
val normalized: DC[Double] = doubles.withResult(sum).map{case (x, s) => x / s}
```

We can compute a signature for `normalized` without computing a sum

Improving the workflow

- Avoid adding complexity in order to speed things up.
- Get the benefits of avoiding redundant computation, but avoid the costs of managing intermediate results.
- Further automatic behaviors to explore, such as automatic failure traps, and generation of common statistics.

Can I use this?

Yes!

check it out:

https://github.com/bloomberg/spark-flow

- requires spark 2.0
- still early, must build from source currently
- hopefully available on spark packages soon

Questions?

Nimbus Goehausen

ngoehausen@bloomberg.net

@nimbusgo on Twitter / Github

https://github.com/bloomberg/spark-flow

techatbloomberg.com

Bloomberg