WROCŁAW
JAVA USER GROUP

DATA SAMPLING WITH SPARK

An introduction to distributed processing using Spark Core API

Maxime Jeanmart

Wroclaw Java User Group meetup

31 Jan 2018

Who are we?

Iconiq Capital, Battery Ventures, Index Ventures, ...

Spinoff of Vrije Universiteit Brussel

Brussels, Wrocław, New York, London, Paris, everywhere

Belgian enterprise software company

250+ strong

Fast growing

Who are we?

THE SAAS AWARDS WINNER 2017

Forbes 2017 Cloud 100

Recognized by Forrester & Gartner as a Leader

What do we do?

“Collibra helps people find, understand, and trust data”

What do we use?

elastic

spring

APACHE kafka

Java

HIBERNATE

amazon web services

ANSIBLE

docker

PostgreSQL

Spark

OSGi™

TypeScript

iOS

Microsoft .NET

And much more…


In data trust we

WROCŁAW
JAVA USER GROUP

DATA SAMPLING WITH SPARK

An introduction to distributed processing using Spark Core API

Maxime Jeanmart

Wroclaw Java User Group meetup

31 Jan 2018

Goals

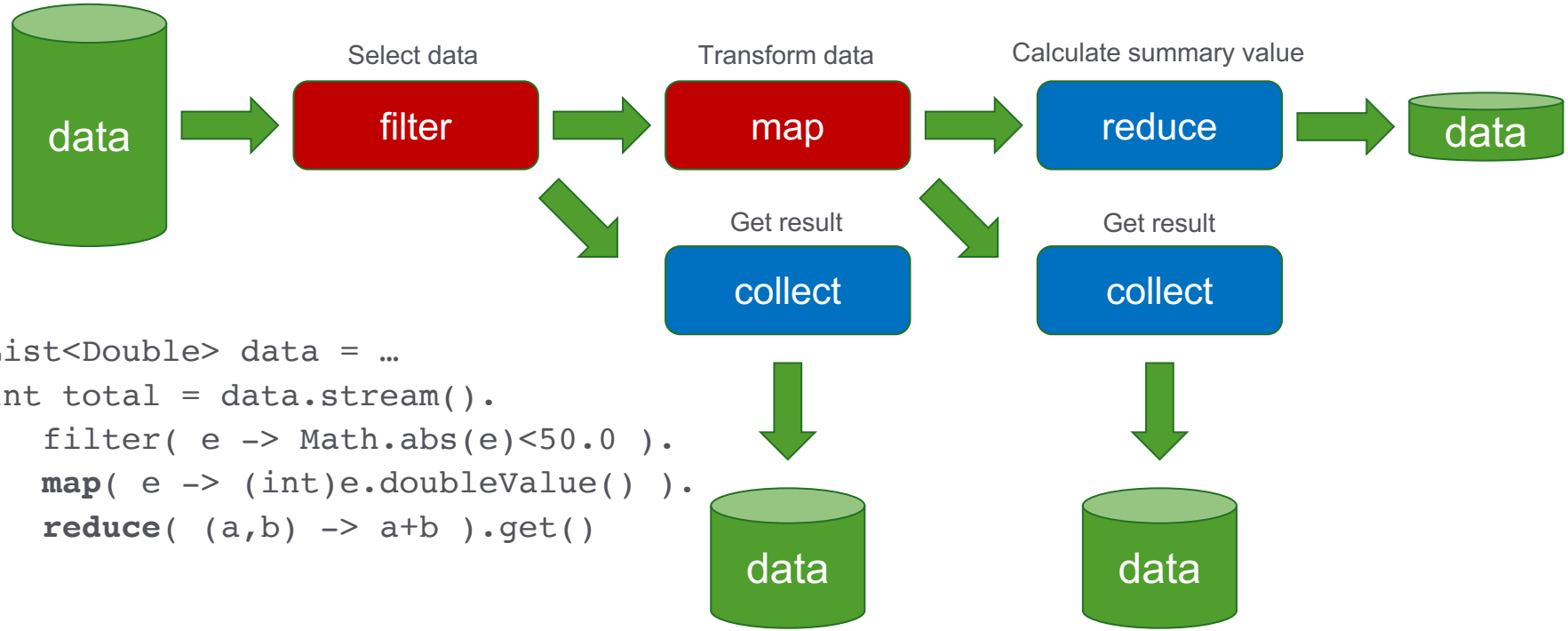
Introduction to Spark Core API

- Understand the difference between for loops, Java stream API and map-reduce with Spark
- Have a grasp on a few challenges rising when parallelizing data processing

Data sampling with Spark

- Compare different data sampling algorithm
- Evaluate which algorithm works best for parallel processing
- Understand how statistics can come handy when working with large data sets

Reminder: Java stream API




```
List<Double> data = …
int total = data.stream().
   filter( e -> Math.abs(e)<50.0 ).
   map( e -> (int)e.doubleValue() ).
   reduce( (a,b) -> a+b ).get()
```


collibra

Reminder: iterating over a loop in java and scala

Java

```
List<String> A = …
int N = A.size();
for (int i = 0; i<N; i++){
  String e = A.get(i);
  …
}
```

- Index available
- No explicit operation
- Encourage side effects (modification of objects outside loop)

```
List<String> A = …
int N = A.size();
for (String e: A){
   …
}
```

- Index not available
- No explicit operation
- Encourage side effects

```
List<String> A = …
int N = A.size();
List<String> B = A.stream().
   filter( e -> … ).
   map( e -> … ).
   reduce( e -> …) OR
   collect(Collectors.toList())
```

- Index not available
- Explicit operation
- **Side effects easily avoided or at least identified as closures**

Scala

```
val A: Seq[String] = …
val N = A.size()
for (i <- 0 until N){
  val e = A(i)
  …
}
```

```
val A: Seq[String] = …
val N = A.size()
for (e <- A){
   …
}
```

```
val A: Seq[String] = …
val N = A.size()
val B = A.
   filter( e => … ).
   map( e => …) AND/OR
   reduce( e => …)
```


collibra

Java stream API vs MapReduce

```
List<Double> data = …
int total = data.stream().
   filter( e -> Math.abs(e)<50.0 ).
   map( e -> (int)e.doubleValue() ).
   reduce( (a,b) -> a+b ).get()
```

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

Reduce is a programming model and an associ-
plementation for processing and generating large
s. Users specify a *map* function that processes a
ıe pair to generate a set of intermediate key/value
ıd a *reduce* function that merges all intermediate
ssociated with the same intermediate key. Many
·ld tasks are expressible in this model, as shown
ıper.

ams written in this functional style are automati-
rallelized and executed on a large cluster of com-
machines. The run-time system takes care of the
of partitioning the input data, scheduling the pro-
execution across a set of machines, handling ma-
ilures, and managing the required inter-machine

given day, etc. Most such computations are co
ally straightforward. However, the input data is
large and the computations have to be distributed
hundreds or thousands of machines in order to f
a reasonable amount of time. The issues of how
allelize the computation, distribute the data, and
failures conspire to obscure the original simple
tation with large amounts of complex code to de
these issues.

As a reaction to this complexity, we designed
abstraction that allows us to express the simple co
tions we were trying to perform but hides the m
tails of parallelization, fault-tolerance, data dist
and load balancing in a library. Our abstractio
spired by the *map* and *reduce* primitives present
and many other functional languages. We reali

collibra

Java stream API vs MapReduce

```
List<Double> data = …
int total = data.stream().
   parallel().
   filter( e -> Math.abs(e)<50.0 ).
   map( e -> (int)e.doubleValue() ).
   reduce( (a,b) -> a+b ).get()
```

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

Reduce is a programming model and an associ-
plementation for processing and generating large
s. Users specify a *map* function that processes a
ıe pair to generate a set of intermediate key/value
ıd a *reduce* function that merges all intermediate
ssociated with the same intermediate key. Many
·ld tasks are expressible in this model, as shown
ıper.

ams written in this functional style are automati-
rallelized and executed on a large cluster of com-
machines. The run-time system takes care of the
of partitioning the input data, scheduling the pro-
execution across a set of machines, handling ma-
ilures, and managing the required inter-machine

given day, etc. Most such computations are cc
ally straightforward. However, the input data is
large and the computations have to be distributec
hundreds or thousands of machines in order to f
a reasonable amount of time. The issues of how
allelize the computation, distribute the data, and
failures conspire to obscure the original simple
tation with large amounts of complex code to d
these issues.

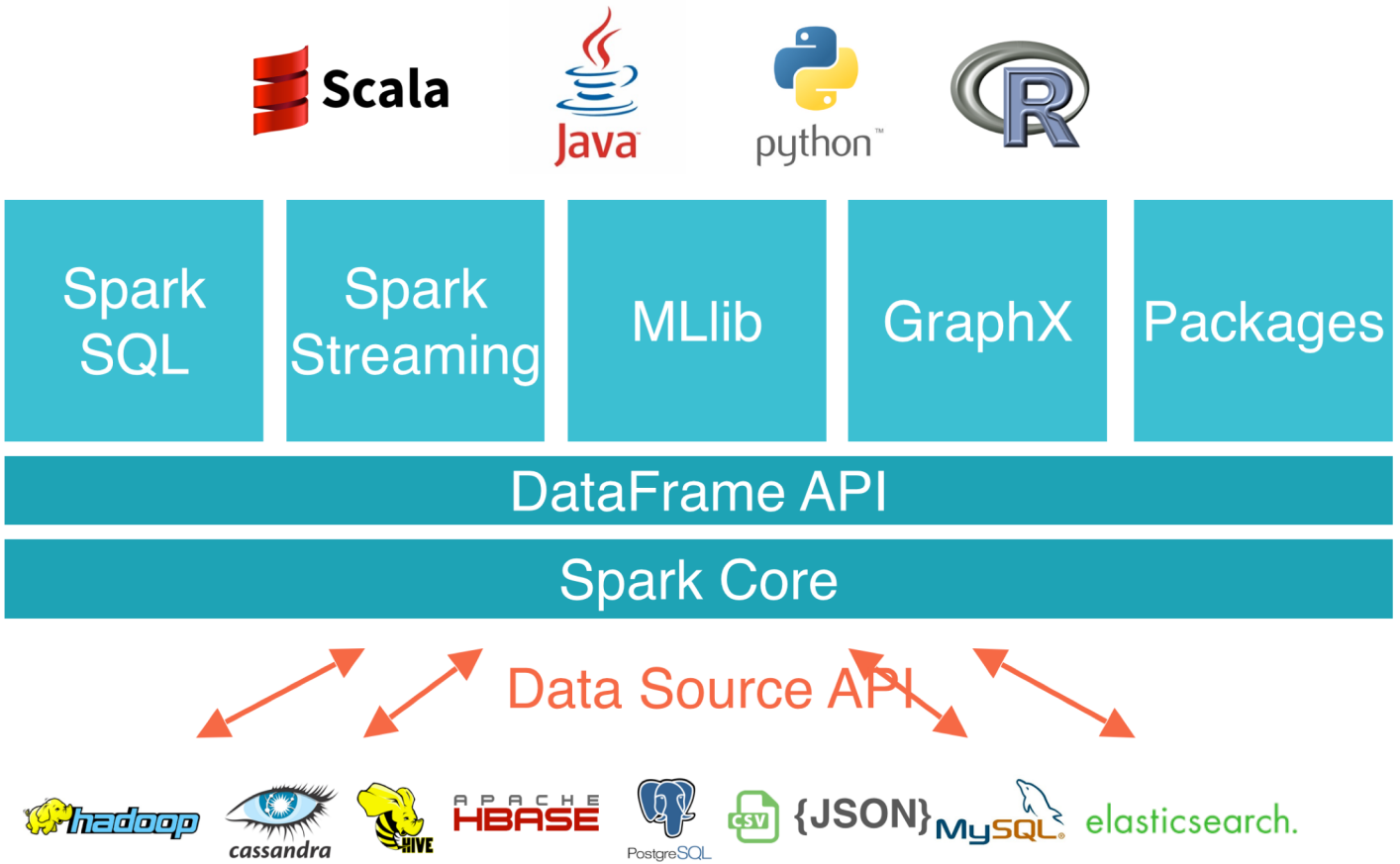
As a reaction to this complexity, we designe
abstraction that allows us to express the simple cc
tions we were trying to perform but hides the m
tails of parallelization, fault-tolerance, data dist
and load balancing in a library. Our abstractic
spired by the *map* and *reduce* primitives present
and many other functional languages. We reali

collibra

Lightning-fast cluster computing

Multi-CPU
Cluster support
Large data sets support
Memory first
Works in Scala, Java, Python, R

Source:https://ogirardot.files.wordpress.com/2015/05/future-of-spark.png



Data sampling

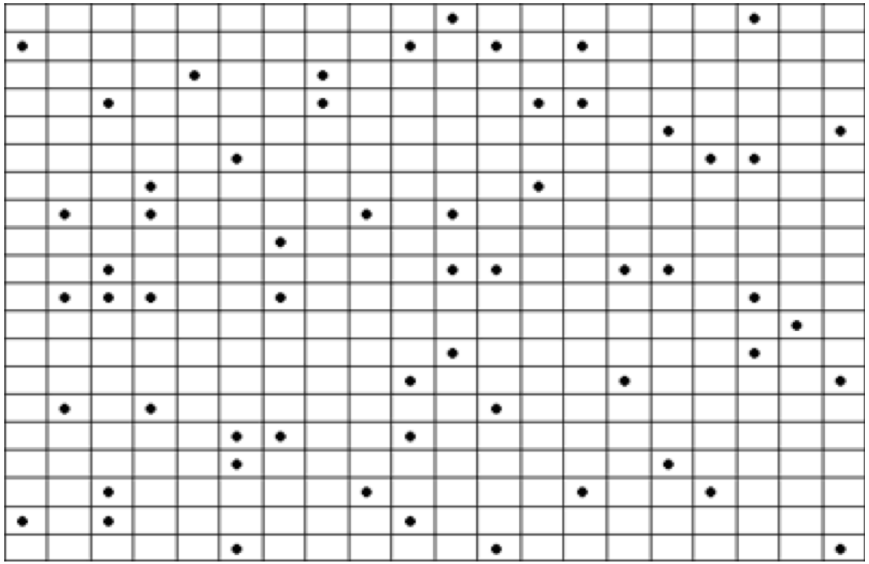
Goal

Use Spark to take random samples from an unknown but possibly large data set

Usage

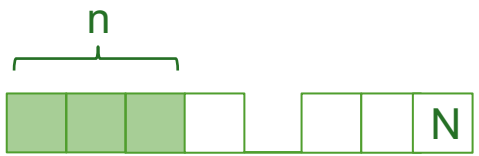
- Show examples of data to user
- Estimate statistics on population samples to speed up calculation
- Partition data (e.g. training / validation / test partitions)

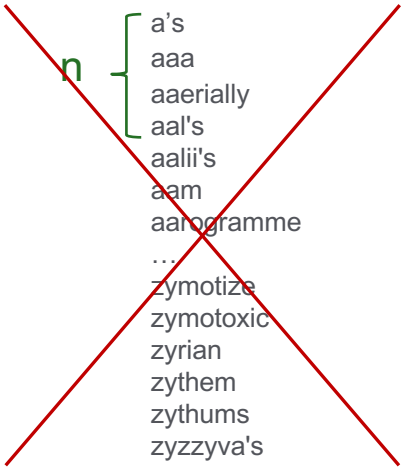
n samples taken among N elements



collibra

Algorithm 1: take n first elements





Not representative samples

Principle

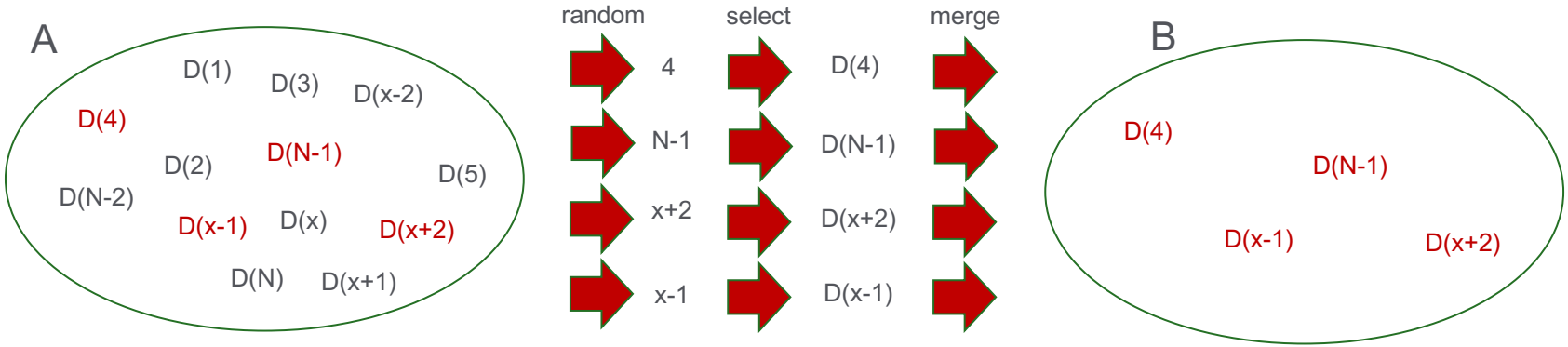
- Source data set has N elements
- Take n first elements
- In java: `list.sublist(0,n)`
- In Spark: `rdd.take(n)`

Analysis

- Simple
- Not random
- Invalid or misleading if data is ordered, clustered or shows patterns

collibra

Algorithm 2 – extraction of n rows using n random numbers




```
Let A be a list of N rows.
Let B be an empty list of rows.
Repeat n times:
   r = random number between 1 and N
   e = take element #r from A
   Add e to B
```


collibra

Algorithm 2 – Java/Scala implementation

```
public List<String> getSamples
      (List<String> A, int n){
   Random random = new Random(42);
   int N = A.size();
   List<String> B = new ArrayList<>();
   for (int i = 0; i < n ; i++){
      int r = random.nextInt(N);
      String e = A.get(r);
      B.add(e);
   }
   return B;
}
```

```
def getSamples(A: Seq[String], n: Int): Seq[String] = {
  val random = new Random(42)
  val N = A.size
  val B = mutable.Buffer[String]()
  for (i <- 0 until n) {
    val r = random.nextInt(N)
    val e = A(r)
    B+=e
  }
  B
}
```

Pros

- Fast execution
- Simple to read / understand

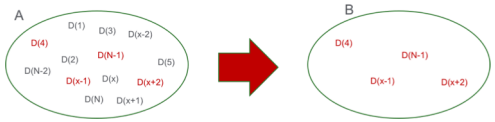
Cons

- ???

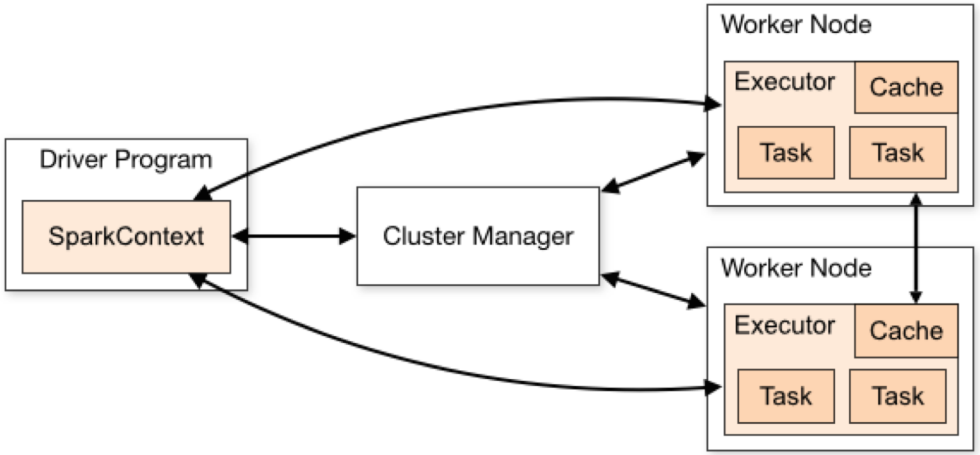
collibra

Image source: https://spark.apache.org/docs/latest/cluster-overview.html

Distributing code



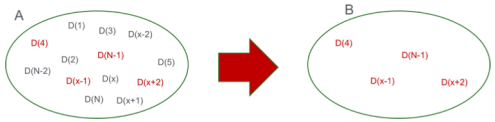

```
val random = new Random(42)
val N = A.size
val B = mutable.Buffer[String]()
for (i <- 0 until n) {
  val r = random.nextInt(N)
  val e = A(r)
  B+=e
}
B
```



collibra

Image source: https://spark.apache.org/docs/latest/cluster-overview.html

Distributing code

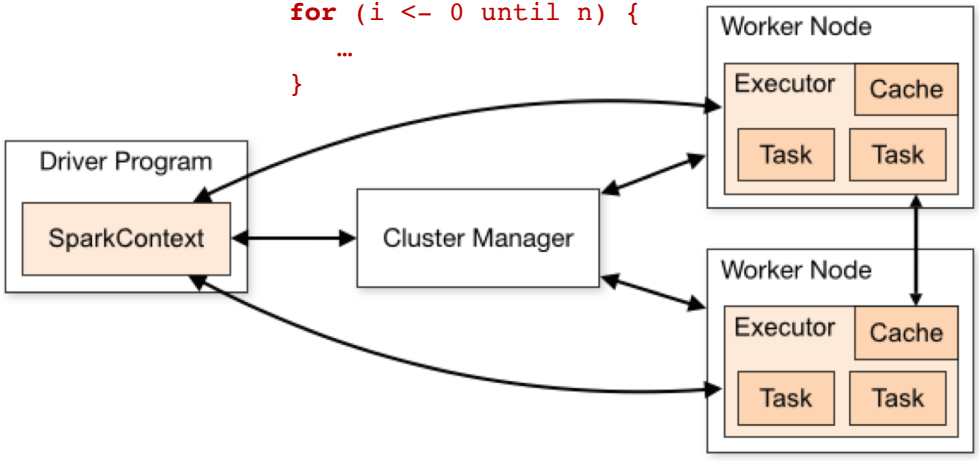



```
val random = new Random(42)
val N = A.size
val B = mutable.Buffer[String]()
for (i <- 0 until n) {
  val r = random.nextInt(N)
  val e = A(r)
  B+=e
}
B
```

P1 P2 P3 … Pp

```
for (i <- 0 until n) {
  …
}
```

```
val random = new
Random(42)
val N = A.size
val B =
mutable.Buffer[String]()
…
B
```

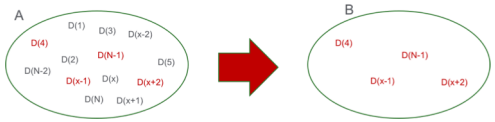



```
val r = random.nextInt(N)
val e = A(r)
B+=e
```

```
val r = random.nextInt(N)
val e = A(r)
B+=e
```


collibra

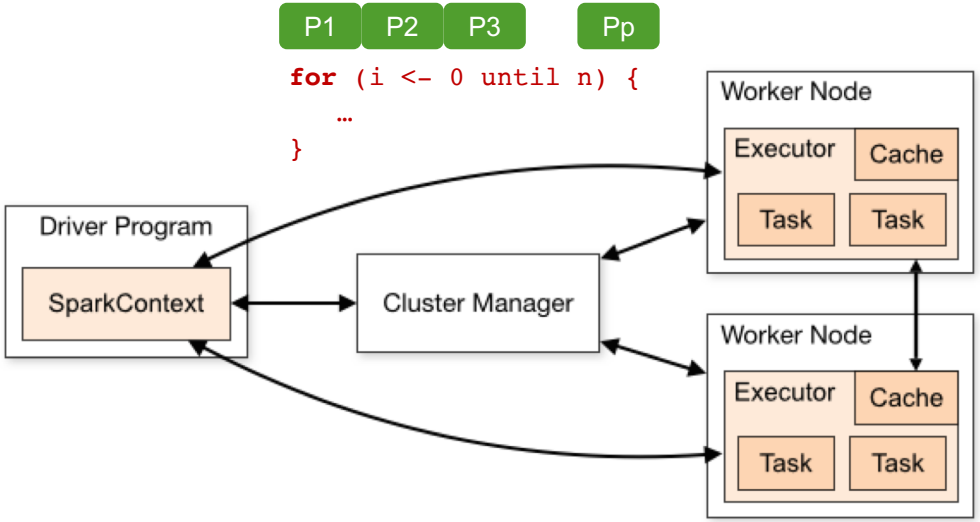
Distributing code



Problems

- The List needs to be transferred and modified in each node => 2 different B Lists
  What is the content of B in the end ?
  There is no code to merge B => B contains the value of last executed worker ?
- Worker code cannot have side-effects, it cannot impact objects outside the loop

```
val random = new
Random(42)
val N = A.size
val B =
mutable.Buffer[String]()
…
B
```




```
val r = random.nextInt(N)
  val e = A(r)
  B+=e
```

```
val r = random.nextInt(N)
  val e = A(r)
  B+=e
```


collibra

Algorithm 2 – Scala implementation: the cons

```
def getSamples(A: Seq[String], n: Int): Seq[String] = {
  val random = new Random(42)
  val N = A.size
  val B = mutable.Buffer[String]()
  for (i <- 0 until n) {
    val r = random.nextInt(N)
    val e = A(r)
    B+=e
  }
  B
}
```

Pros

- Fast execution

Cons

- Loop code has side effect (mutable Buffer)
- => is not scalable / distributable
- => use functional programming instead

collibra

Algorithm 2 – java implementation reviewed

```
public List<String> getSamplesv2(List<String> A, int n){
   Random random = new Random(42);
   List<String> B = new ArrayList<>();
   int N = A.size();
   return getSamplesv2(A, n, random, N, new ArrayList<>(), 0);
}

private List<String> getSamplesv2(List<String> A, int n,
           Random random, int N, List<String> B, int i){
   if (i==n){
      return B;
   } else {
      int r = random.nextInt(N);
      String e = A.get(r);
      B.add(e);
      return getSamplesv2(A, n, random, N, B, i+1);
   }
}
```

Pros

- Fast execution
- Functional style (no side effect)

Cons

- Stack overflow very fast (1k-10k)

Algorithm 2 – scala implementation reviewed

```
def getSamplesV2(A: Seq[String], n: Int): Seq[String] = {
  val random = new Random(42)
  val N = A.size

  def buildSamples(B: Seq[String], i: Int): Seq[String] =
   if (i==n) B else {
    val r = random.nextInt(N)
    val e = A(r)
    buildSamples(B:+e, i+1)
  }

  buildSamples(Seq[String](), 0)
}
```

Pros

- Fast execution
- Functional (no side effect)

Cons

- Iteration *i+1* cannot be executed before iteration *i* is executed
- => Still not scalable

Algorithm 2 – Spark implementation

```
def getSamplesV2(A: RDD[String], n: Int): Seq[String] = {
  val random = new Random(42)
  val N = A.count()
  val indexed: RDD[(String, Long)] = A.zipWithIndex()

  def buildSamples(B: Seq[String], i: Int): Seq[String] =
   if (i==n) B else {
    val r = (random.nextDouble() * N).toLong
    val e = indexed.filter(_._2 == r).collect()(0)._1
    buildSamples(B:+e, i+1)
  }

  buildSamples(Seq[String](), 0)
}
```

Pros

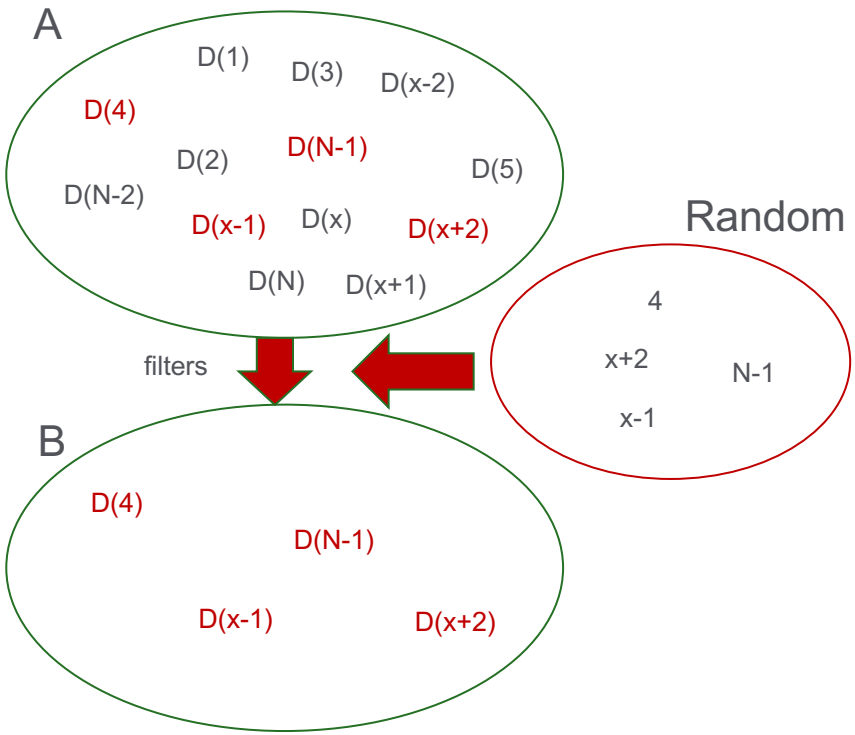
- Spark implementation

Cons

- Count is 1 pass over data
- zipWithIndex is 1 pass
- Each e calculation is 1 pass
- => n+2 passes over the data
  (=> loading/transferring data up to n+2 times)
- Loop over *i* is not distributed

collibra

Algorithm 3 – generation of random numbers list




```
Let A be a list of N rows.
Let B be an empty list of rows.
Let R be an empty set of long values.
Do while size(R) < n:
   r = random number between 1 and N
   Add r to R
For each element e of A:
   If index of e in R, add e to B
```

Algorithm 3 – java code

```
public List<String> getSamples(List<String> A, int n){
   Random random = new Random(42);
   List<String> B = new ArrayList<>();
   Set<Integer> R = new HashSet<>();
   int N = A.size();
   for (int i = 0; i < n ; i++){
      int r = random.nextInt(N);
      R.add(r);
   }
   for (int i = 0; i < N ; i++){
      if (R.contains(i)){
         String e = A.get(i);
         B.add(e);
      }
   }
   return B;
}
```

Pros

- Easy to understand

Cons

- Slower than previous algorithm
- Not scalable

Algorithm 3 – scala code

```
// Scala
def getSamples(A: Seq[String], n: Int): Seq[String] = {
  val random = new Random(42)
  val N = A.size
  val R = (1 to n).map(_ => random.nextInt(N)).toSet
  A.zipWithIndex.
      filter(x => R.contains(x._2)).
      map(_._1)
}

// Spark
def getSamples(A: RDD[String], n: Int): Seq[String] = {
  val random = new Random(42)
  val N = A.count()
  val R = (1 to n).map(_ => (random.nextDouble() * N).toLong).toSet
  A.zipWithIndex.
      filter(x => R.contains(x._2)).
      map(_._1).
      collect()
}
```

Pros

- Easy to understand
- 3 passes over data only
  => faster than previous algo.

Cons

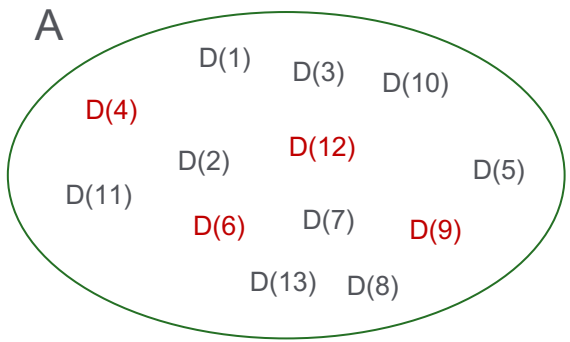
- Transferring R can be slow or lead to OOM if *n* is large

Bernoulli trials

Fair coin	50% head p(true)=0.5	50% tails p(false)=0.5
Unfair coin	60% head p(true)=0.6	40% tails p(false)=0.4

collibra

Sample probability



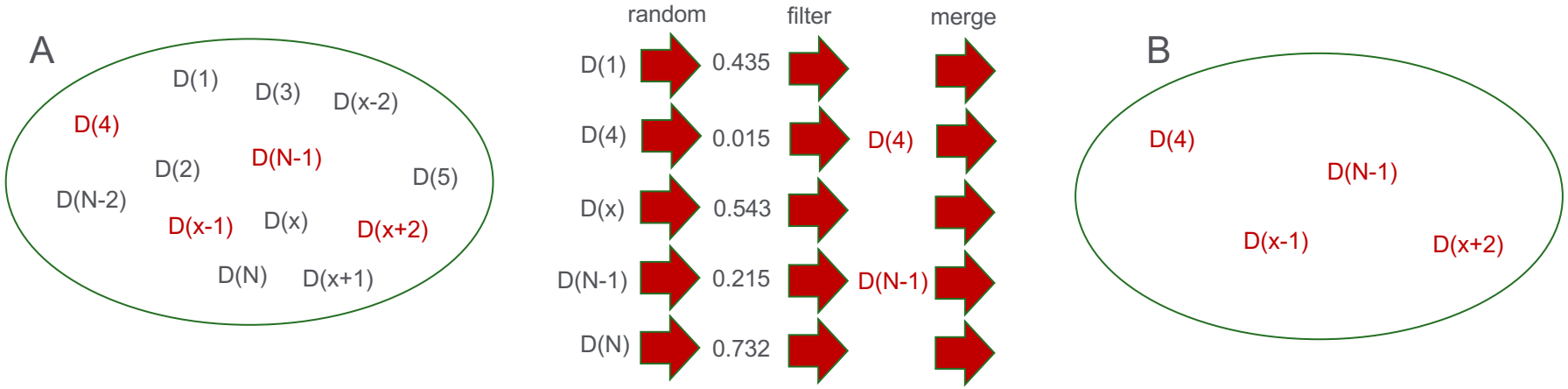
Red = sample (true)
Black = not sample (false)

elements: 13
black: 9
red: 4

p(red) = p(true) = 4/13 = 0.308 = 31%
p(black) = p(false) = 9/13 = 0.692 = 69%

If you pick an element of A, there is 31% chance it is a sample

Algorithm 4 – Bernoulli trials




```
Let A be a list of N rows. Let B be an empty list of rows.
Let R be a uniform distribution of decimal numbers between 0 and 1
p = n / N
For each element e of A:
   r = random number from R
   If r <= p then add e to B
```


collibra

Algorithm 4 – scala code

```
// Scala
def getSamplesV1(A: Seq[String], n: Int): Seq[String] = {
  val R = new Random(42)
  val N = A.size
  val pTrue = n.toDouble / N
  A.filter(x => R.nextDouble()<=pTrue)
}
```

```
// Spark
def getSamplesV1(A: RDD[String], n: Int): Seq[String] = {
  val R = new Random(42)
  val N = A.count()
  val pTrue = n.toDouble / N
  A.filter(x => R.nextDouble()<=pTrue).collect()
}
```

Pros

- Only 2 passes over data if N is unknown
- Only 1 pass if N known or ratio of samples is the input
- Only R and pTrue are distributed

Cons

- *???*

collibra

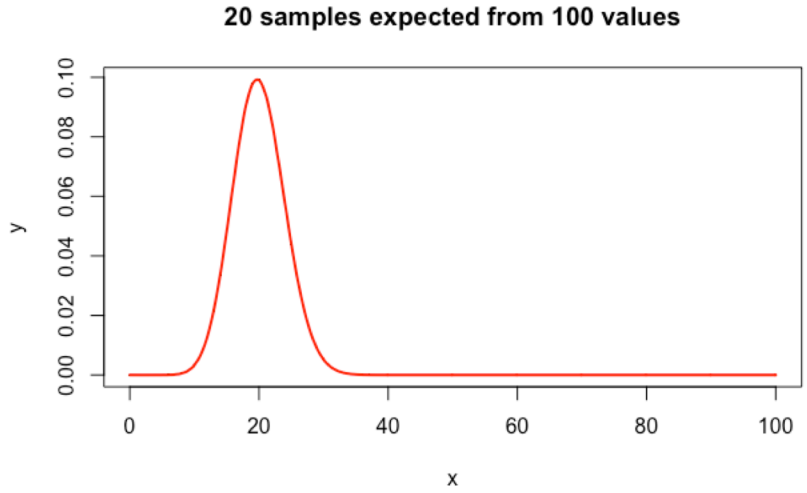
Algorithm 4 execution

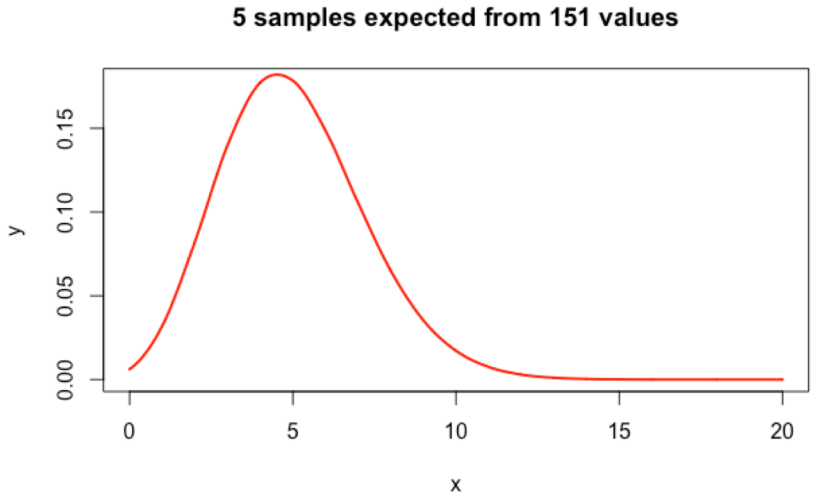
N	n	p(true)	Observed n (10 executions)	Mean n
151	5	3.3%	8, 5, 5, 4, 6, 10, 6, 9, 2, 2	5.7
151	20	13.2%	21, 20, 15, 18, 18, 17, 22, 24, 13, 28	19.6
151	100	66.2%	99, 105, 99, 98, 104, 90, 89, 98, 101, 102	98.5
1000	5	0.5%	5, 3, 4, 10, 5, 8, 6, 7, 7, 6	6.1
1000	20	2%	14, 23, 22, 12, 21, 14, 16, 21, 16, 20	17.9
1000	100	10%	91, 110, 108, 88, 91, 98, 92, 105, 92, 90	96.5

Result is correct on the average

R is a random variable => B based on random variable

Binomial distribution





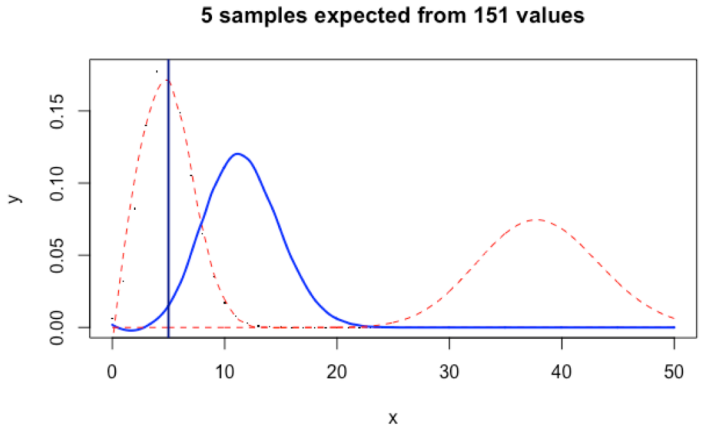
Gives the probability of having a number of ‘true’ values knowing the probability of a single value to be ‘true’

Example: p(5, B(151, 5/151)) = 17.8%
Example: p(3, B(151, 5/151)) = 14.0%

collibra

Use case reviewed

Extract **at least** n samples from a data set of N elements



Idea: raise p(e is a sample) so that

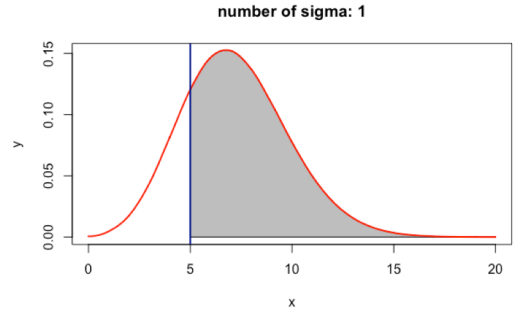
- We have enough samples
- We limit the number of extra samples

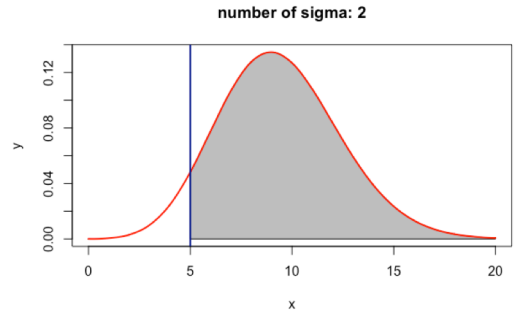
collibra

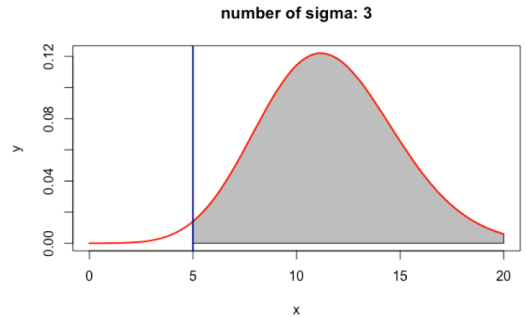
Binomial probability tuning

P transformation	P value (n=5, N=151)	Tail cumulative prob at n	Average number of samples
P = n / N	P=0.033	0.384	5
P1 = (n+σ) / N	P1=0.048	0.730	7
P2 = (n+2*σ) / N	P2=0.062	0.913	9
P3 = (n+3*σ) / N	P3=0.077	0.978	12

$$\sigma = \sqrt{5(1-\frac{5}{151})} = 2.199$$







Use case reviewed again

Extract **exactly** n samples from a data set of N elements

Solution:

1. Oversample using raised sampling probability
2. Resample the result using algorithm #2 or #3

Remarks:

- Exact samples are usually required for small number of samples
  (e.g.: display samples to user)
- Large sample requests are usually expressed as data set percentage and do not require precise numbers
  (e.g.: data partitioning for machine learning)
- Don’t use algorithm #1 for resampling pass, even though it’s tempting to do so… ☺

Algorithm 4 – one last thing…

```
// Scala
def getSamplesV1(A: Seq[String], n: Int): Seq[String] = {
  val R = new Random(42)
  val N = A.size
  val pTrue = n.toDouble / N
  A.filter(x => R.nextDouble()<=pTrue)
}
```

```
// Spark
def getSamplesV1(A: RDD[String], n: Int): Seq[String] = {
  val R = new Random(42)
  val N = A.count()
  val pTrue = n.toDouble / N
  A.filter(x => R.nextDouble()<=pTrue).collect()
}
```

Running Scala code many times

- Expected mean
- Expected variance

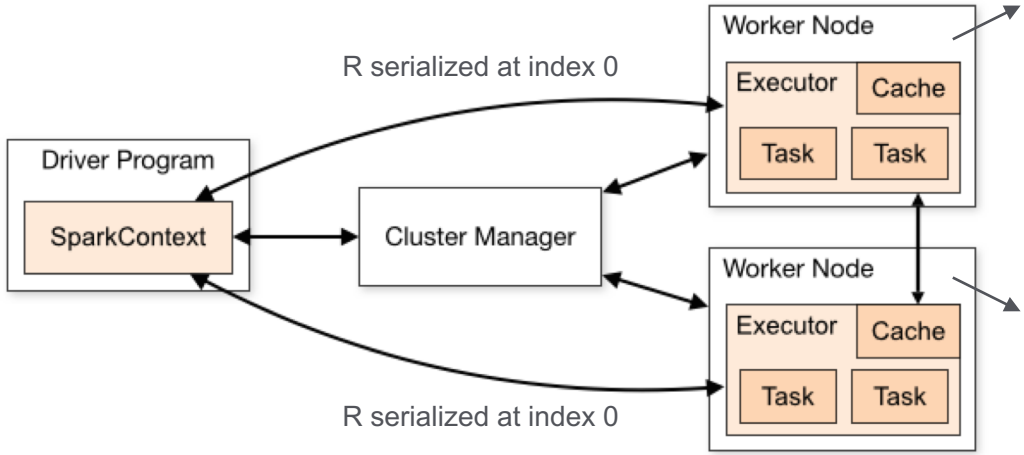
Running Spark code many times

- Expected mean
- **Variance is too large**

collibra

Algorithm 4: distributing random numbers

```
val R = new Random(42)
```




```
R = Random(42)
0.7275636800328681
0.6832234717598454
0.30871945533265976
0.27707849007413665
0.6655489517945736
0.9033722646721782
0.36878291341130565
0.2757480694417024
0.46365357580915334
0.7829017787900358
```

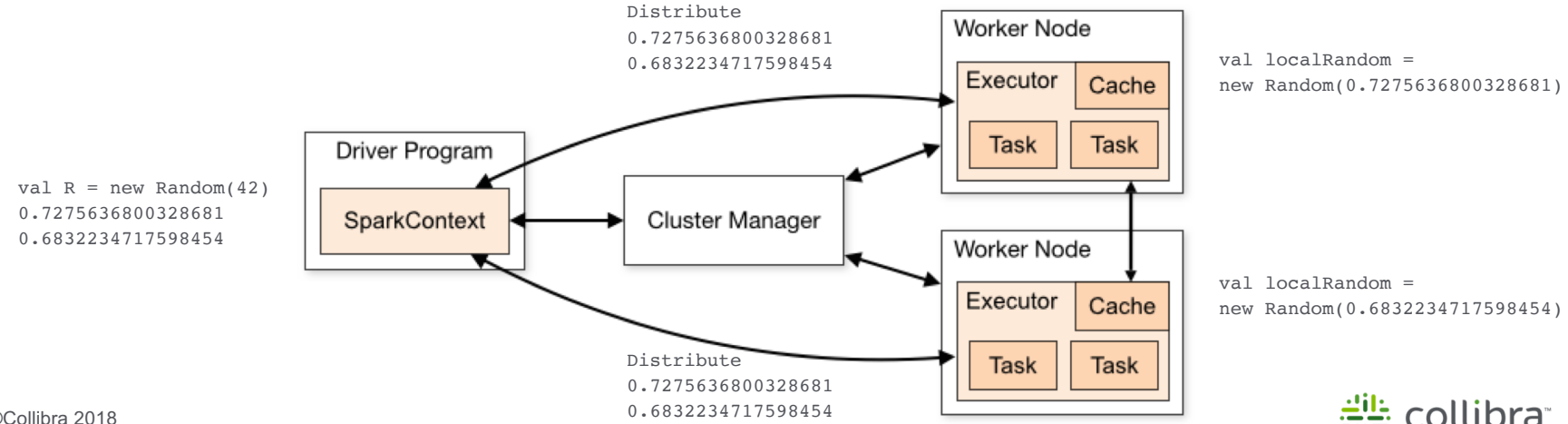
```
R = Random(42)
0.7275636800328681
0.6832234717598454
0.30871945533265976
0.27707849007413665
0.6655489517945736
0.9033722646721782
0.36878291341130565
0.2757480694417024
0.46365357580915334
0.7829017787900358
```

Standard deviation is increased proportionally to the number of partitions

collibra

Algorithm 4 – spark code corrected

```
// Spark
def getSamplesV2(A: RDD[String], n: Int): Seq[String] = {
  val R = new Random()
  val seeds = (1 to A.partitions.size).map(_=>R.nextInt()).toArray
  val N = A.count()
  val pTrue = n.toDouble / N
  A.mapPartitionsWithIndex((partitionIndex, iterator)=>{
    val localRandom = new Random(seeds(partitionIndex))
    iterator.toSeq.
       filter(x => localRandom.nextDouble()<=pTrue).
       iterator
  }).collect()
}
```



Conclusions

When working with large data sets:

- Map & reduce API isolates code to distribute
  - Scala defines the right API for that purpose
  - No side effect
- Identify driver data to distribute
  - Must be small
  - Must be serializable
  - Must be independent of how the data is partioned
- Minimizing the number of passes is more important than code size
- Statistics can pop up any time when dealing with big data

collibra