cloudera

Top 5 mistakes when writing Spark applications

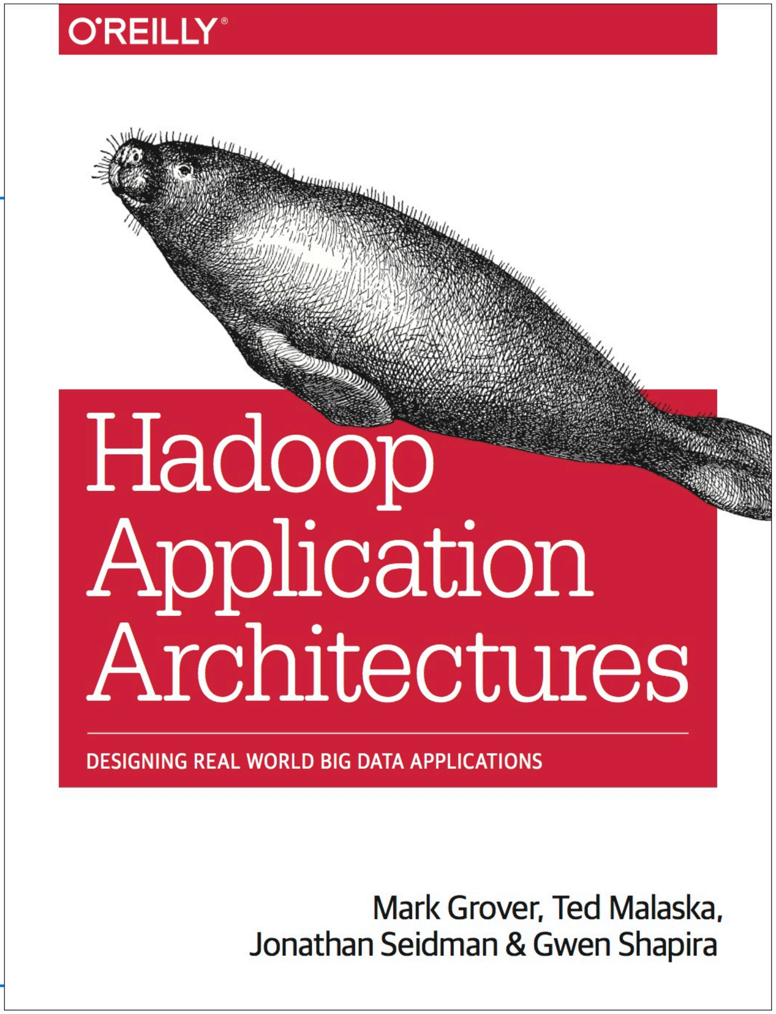
tiny.cloudera.com/spark-mistakes

Mark Grover | Software Engineer, Cloudera | @mark_grover

Ted Malaska | Technical Group Architect, Blizzard| @TedMalaska

About the book

- @hadooparchbook
- hadooparchitecturebook.com
- github.com/hadooparchitecturebook
- slideshare.com/hadooparchbook



Mistakes people make

when using Spark

Mistakes ~~people~~ we've made

when using Spark

Mistakes people make

when using Spark

Mistake # 1

Executors, cores, memory !?!

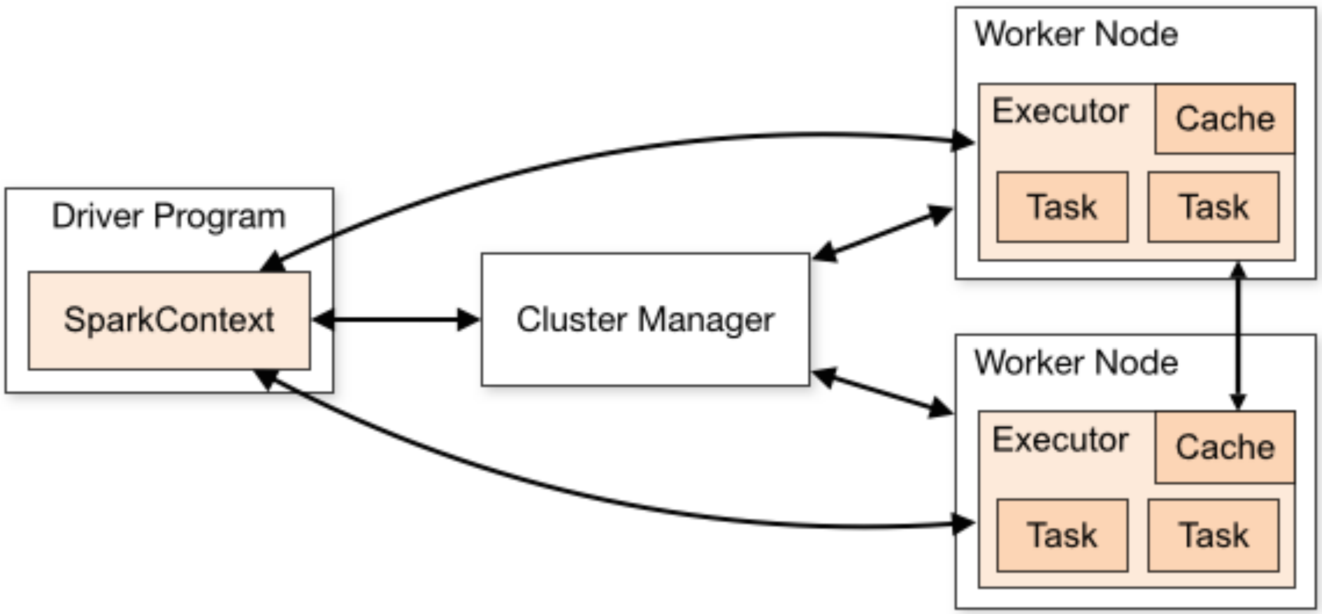
- 6 Nodes
- 16 cores each
- 64 GB of RAM each

Decisions, decisions, decisions

- 6 nodes
- 16 cores each
- 64 GB of RAM

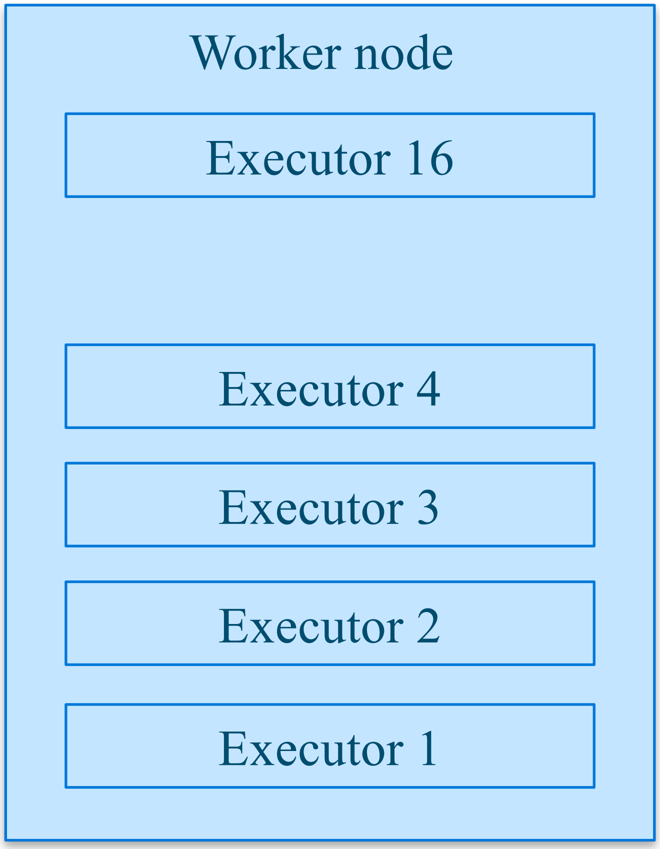
- Number of executors (--num-executors)
- Cores for each executor (--executor-cores)
- Memory for each executor (--executor-memory)

Spark Architecture recap



Answer #1 – Most granular

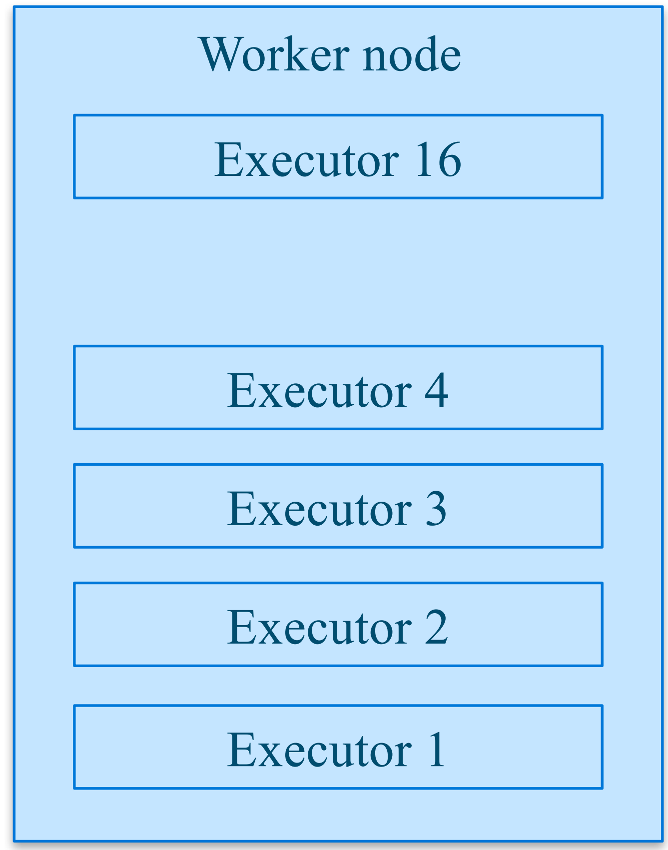
- Have smallest sized executors possible
- 1 core each
- 64GB/node / 16 executors/node = 4 GB/executor
- Total of 16 cores x 6 nodes = 96 cores => 96 executors



cloudera

Answer #1 – Most granular

- Have smallest sized executors possible
- 1 core each
- 64GB/node / 16 executors/node = 4 GB/executor
- Total of 16 cores x 6 nodes = 96 cores => 96 executors



Why?

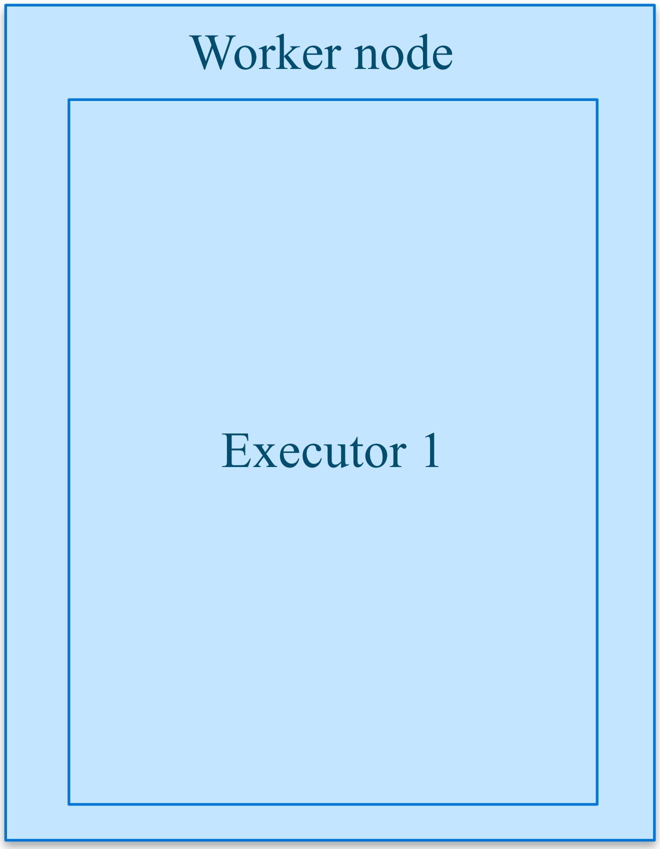
- Not using benefits of running multiple tasks in same executor

Answer #2 – Least granular

- 6 executors in total

=>1 executor per node

- 64 GB memory each
- 16 cores each



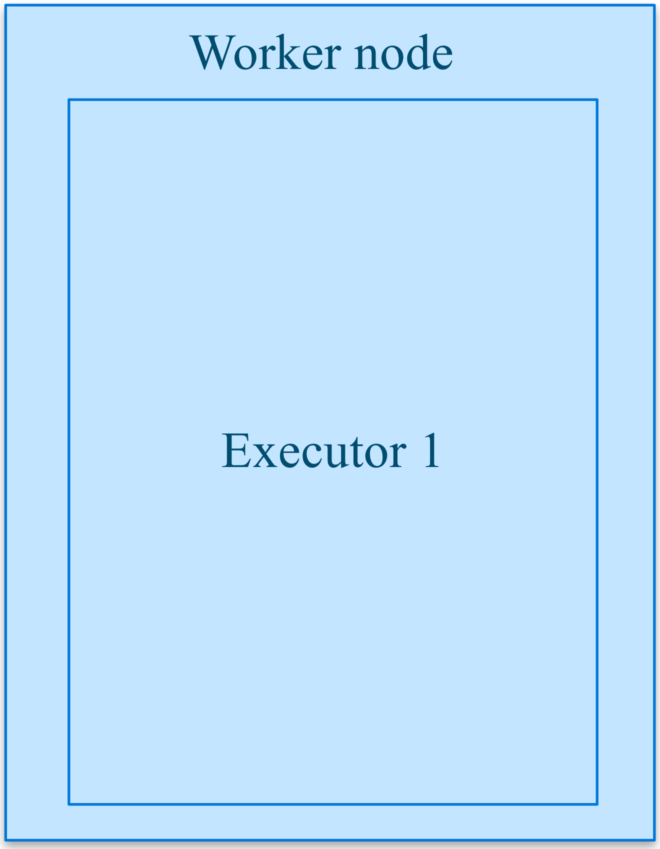
cloudera

Answer #2 – Least granular

- 6 executors in total

=>1 executor per node

- 64 GB memory each
- 16 cores each

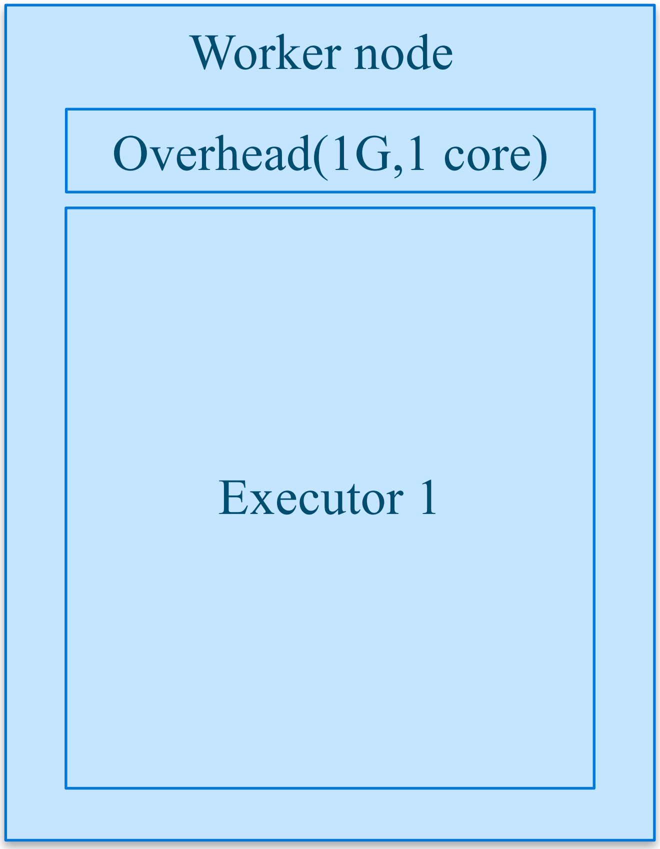


Why?

- Need to leave some memory overhead for OS/Hadoop daemons

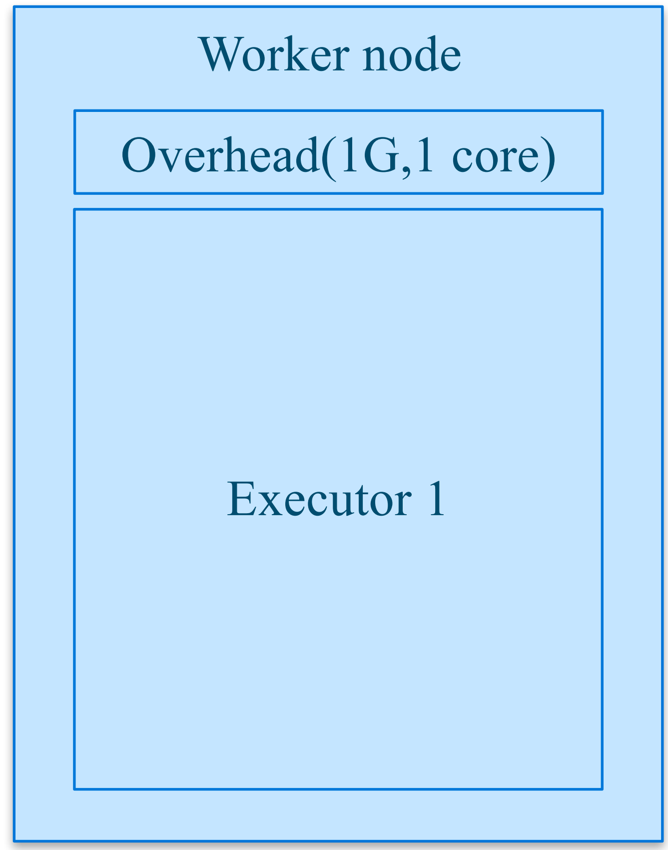
Answer #3 – with overhead

- 6 executors – 1 executor/node
- 63 GB memory each
- 15 cores each



Answer #3 – with overhead

- 6 executors – 1 executor/node
- 63 GB memory each
- 15 cores each



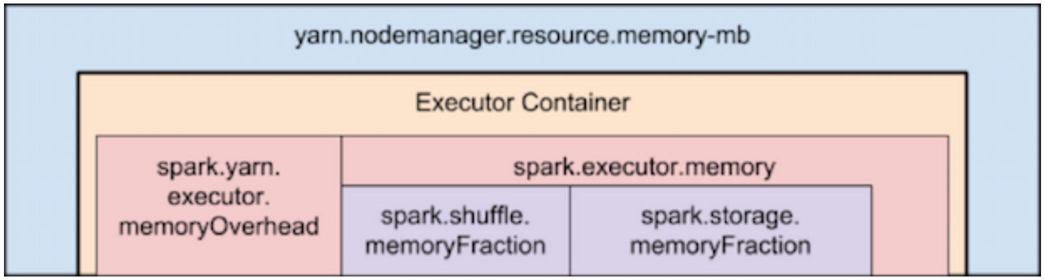
Let’s assume…

- You are running Spark on YARN, from here on…

3 things

- 3 other things to keep in mind

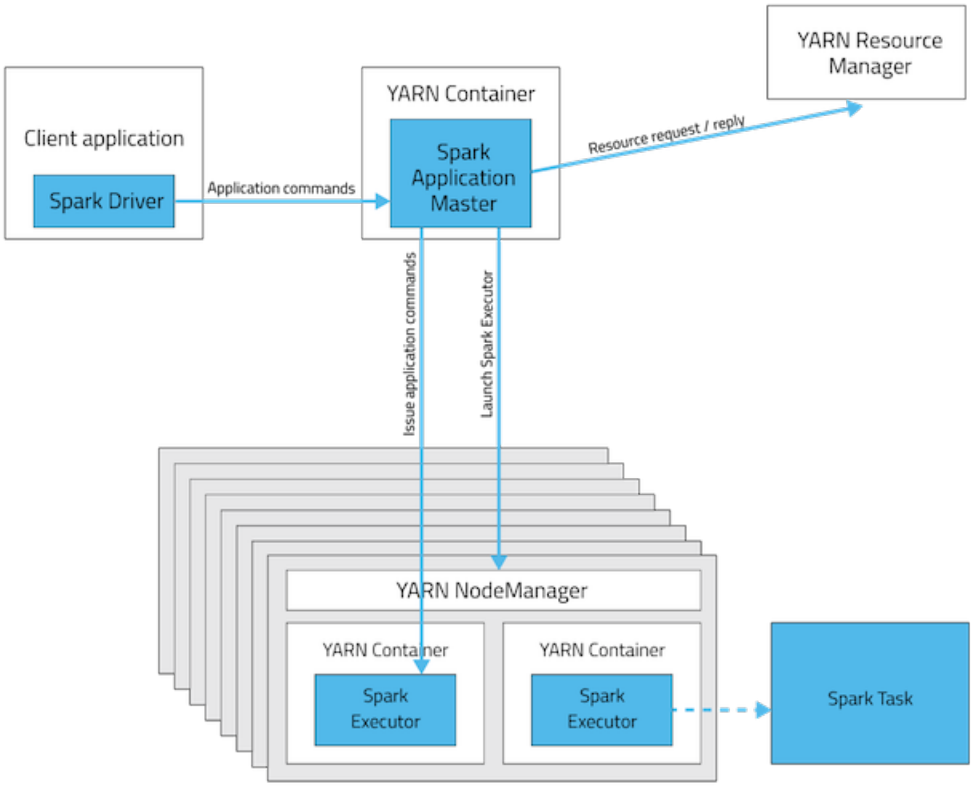
#1 – Memory overhead



- --executor-memory controls the heap size
- Need some overhead (controlled by spark.yarn.executor.memory.overhead) for off heap memory
  - Default is max(384MB, .07 * spark.executor.memory)

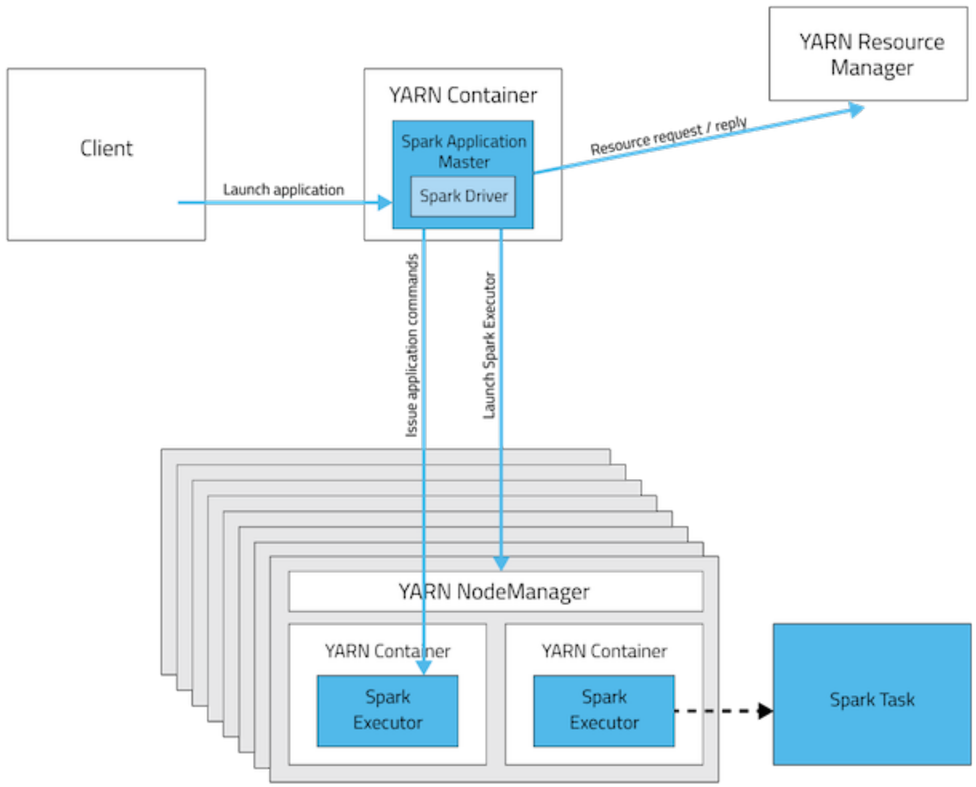
cloudera

#2 - YARN AM needs a core: Client mode



cloudera

#2 YARN AM needs a core: Cluster mode



cloudera

#3 HDFS Throughput

- 15 cores per executor can lead to bad HDFS I/O throughput.
- Best is to keep under 5 cores per executor

Calculations

- 5 cores per executor
  - For max HDFS throughput
- Cluster has 6 * 15 = 90 cores in total

after taking out Hadoop/Yarn daemon cores)

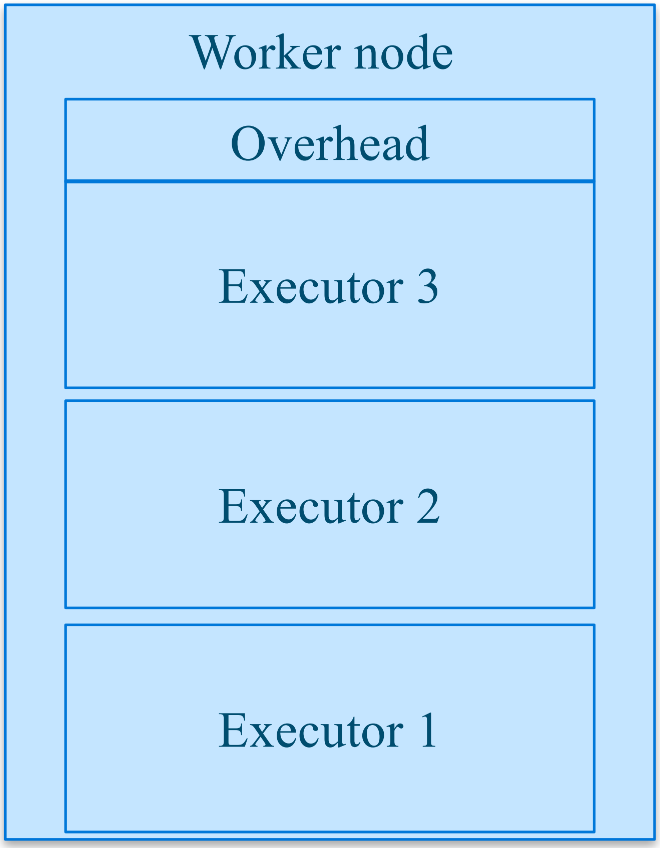
- 90 cores / 5 cores/executor

= 18 executors

- Each node has 3 executors
- 63 GB/3 = 21 GB, 21 x (1-0.07)

~ 19 GB

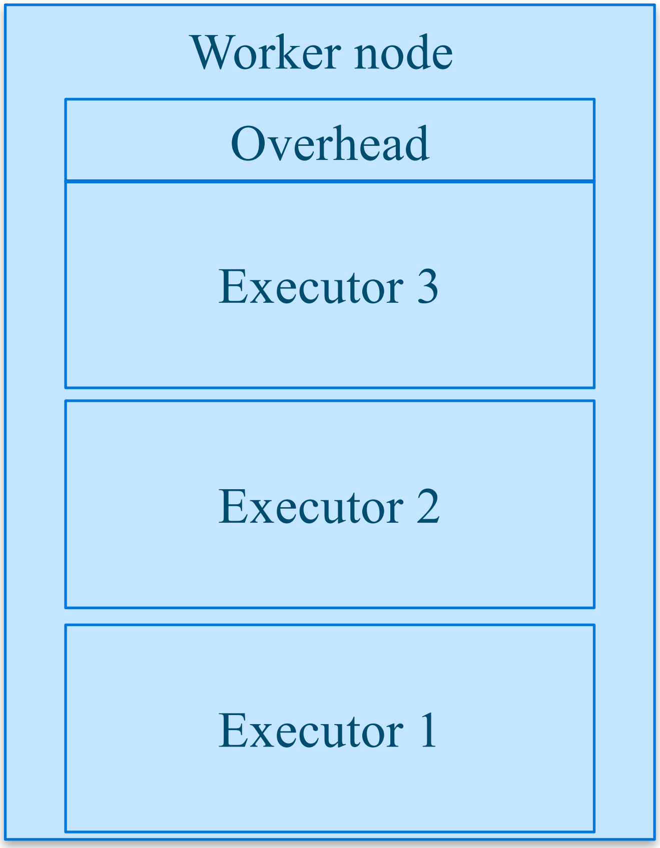
- 1 executor for AM => 17 executors



Correct answer

- 17 executors in total
- 19 GB memory/executor
- 5 cores/executor

** Not etched in stone*



Dynamic allocation helps with though, right?

- *Dynamic allocation allows Spark to dynamically scale the cluster resources allocated to your application based on the workload.*
- Works with Spark-On-Yarn

Decisions with Dynamic Allocation

- 6 nodes
- 16 cores each
- 64 GB of RAM

- Number of executors (--num-executors)
- Cores for each executor (--executor-cores)
- Memory for each executor (--executor-memory)

Read more

- From a great blog post on this topic by Sandy Ryza:

http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

Mistake # 2

Application failure

```
15/04/16 14:13:03 WARN scheduler.TaskSetManager: Lost task 19.0 in stage
6.0 (TID 120, 10.215.149.47): java.lang.IllegalArgumentException: Size
exceeds Integer.MAX_VALUE
at sun.nio.ch.FileChannelImpl.map(FileChannelImpl.java:828) at
org.apache.spark.storage.DiskStore.getBytes(DiskStore.scala:123) at
org.apache.spark.storage.DiskStore.getBytes(DiskStore.scala:132) at
org.apache.spark.storage.BlockManager.doGetLocal(BlockManager.scala:517)
at
org.apache.spark.storage.BlockManager.getLocal(BlockManager.scala:432)
at org.apache.spark.storage.BlockManager.get(BlockManager.scala:618) at
org.apache.spark.CacheManager.putInBlockManager(CacheManager.scala:146)
at org.apache.spark.CacheManager.getOrCompute(CacheManager.scala:70)
```

Why?

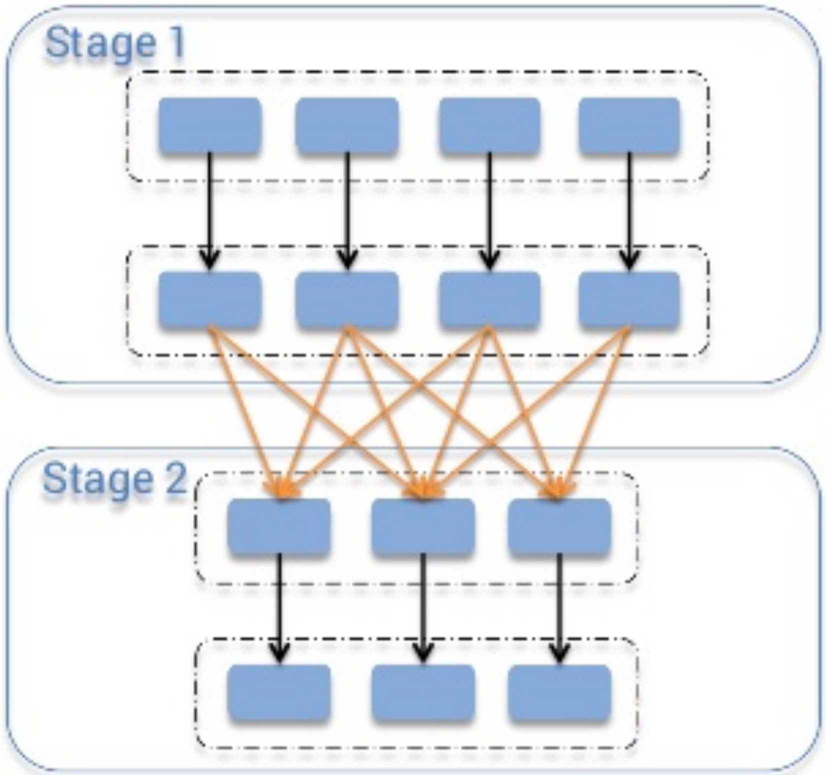
- No Spark shuffle block can be greater than 2 GB

Ok, what's a shuffle block again?

- In MapReduce terminology, a file written from one Mapper for a Reducer
- The Reducer makes a local copy of this file (reducer local copy) and then 'reduces' it

cloudera

Defining `shuffle` and `partition`



Each yellow arrow in this diagram represents a shuffle block.

Each blue block is a partition.

Once again

- Overflow exception if shuffle block size > 2 GB

What's going on here?

- Spark uses `ByteBuffer` as abstraction for blocks

```
val buf = ByteBuffer.allocate(length.toInt)
```

- `ByteBuffer` is limited by `Integer.MAX_SIZE` (2 GB)!

Spark SQL

- Especially problematic for Spark SQL
- Default number of partitions to use when doing shuffles is 200
  - This low number of partitions leads to high shuffle block size

Umm, ok, so what can I do?

1. Increase the number of partitions
   – Thereby, reducing the average partition size
2. Get rid of skew in your data
   – More on that later

Umm, how exactly?

- In Spark SQL, increase the value of `spark.sql.shuffle.partitions`
- In regular Spark applications, use `rdd.repartition()` or `rdd.coalesce()` (latter to reduce #partitions, if needed)

But, how many partitions should I have?

- Rule of thumb is around 128 MB per partition

But! There’s more!

- Spark uses a different data structure for bookkeeping during shuffles, when the number of partitions is less than 2000, vs. more than 2000.

Don’t believe me?

- In `MapStatus.scala`

```
def apply(loc: BlockManagerId, uncompressedSizes: Array[Long]):
MapStatus = {
  if (uncompressedSizes.length > 2000) {
    HighlyCompressedMapStatus(loc, uncompressedSizes)
  } else {
    new CompressedMapStatus(loc, uncompressedSizes)
  }
}
```

Ok, so what are you saying?

If number of partitions < 2000, but not by much, bump it to be slightly higher than 2000.

Can you summarize, please?

- Don't have too big partitions
  - Your job will fail due to 2 GB limit
- Don't have too few partitions
  - Your job will be slow, not making using of parallelism
- Rule of thumb: ~128 MB per partition
- If #partitions < 2000, but close, bump to just > 2000
- Track SPARK-6235 for removing various 2 GB limits

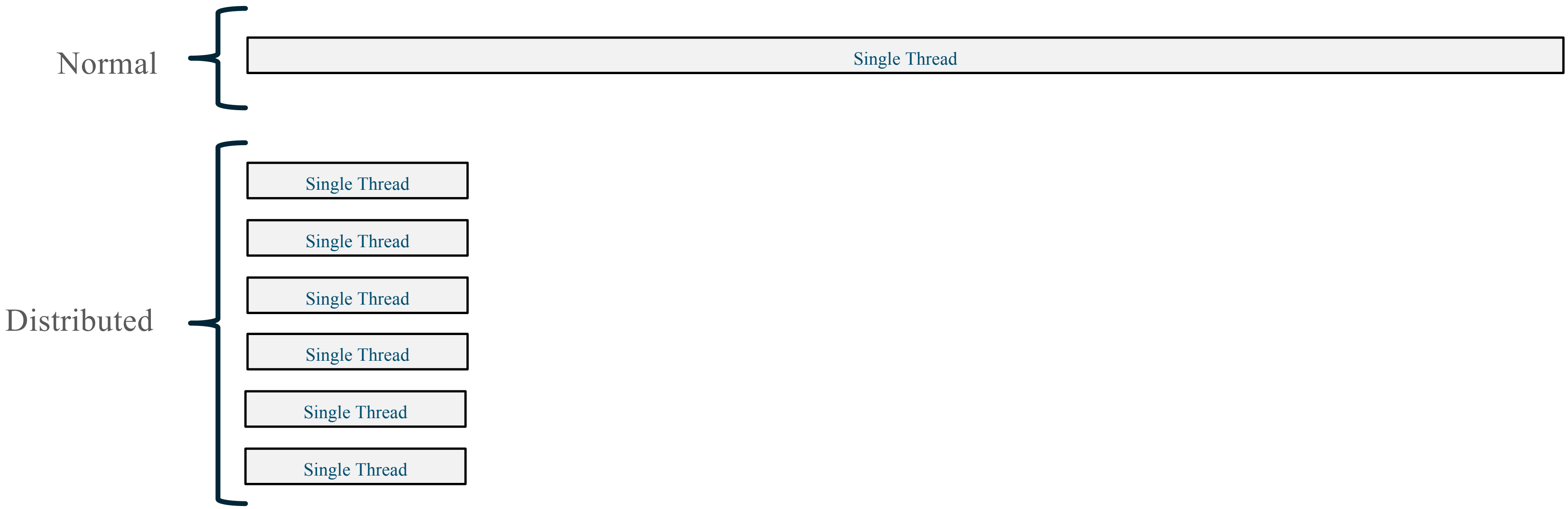
Mistake # 3

Slow jobs on Join/Shuffle

- Your dataset takes 20 seconds to run over with a map job, but take 4 hours when joined or shuffled. What wrong?

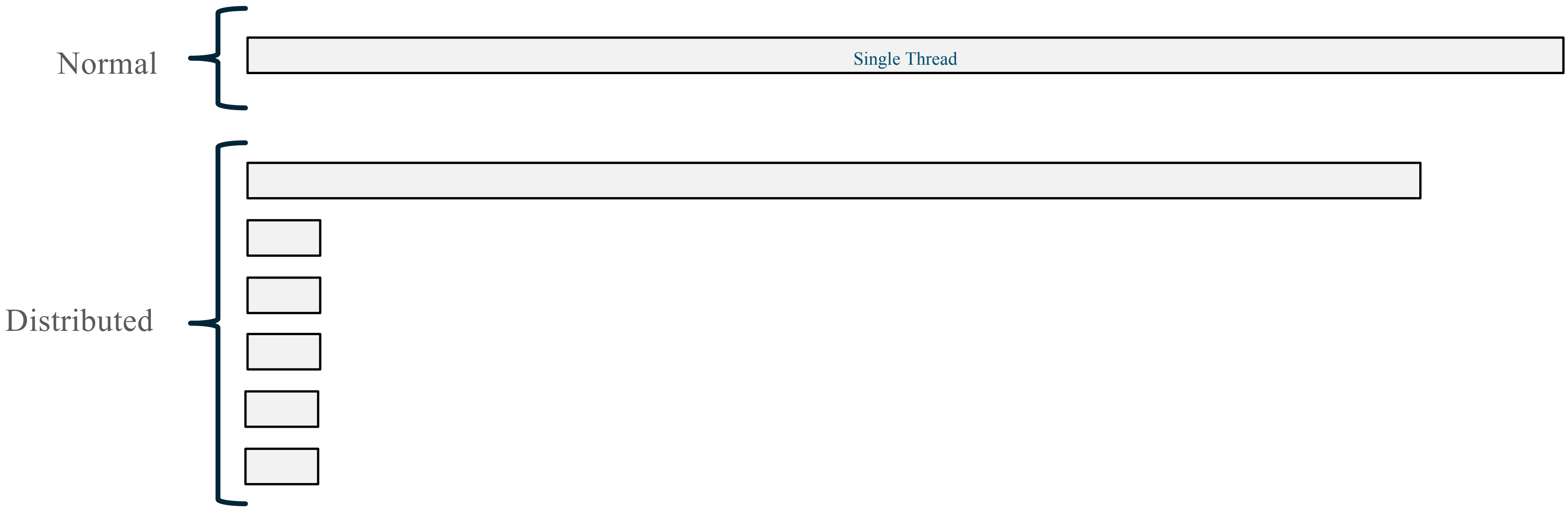
Mistake - Skew

The Holy Grail of Distributed Systems



Mistake - Skew

What about Skew, because that is a thing



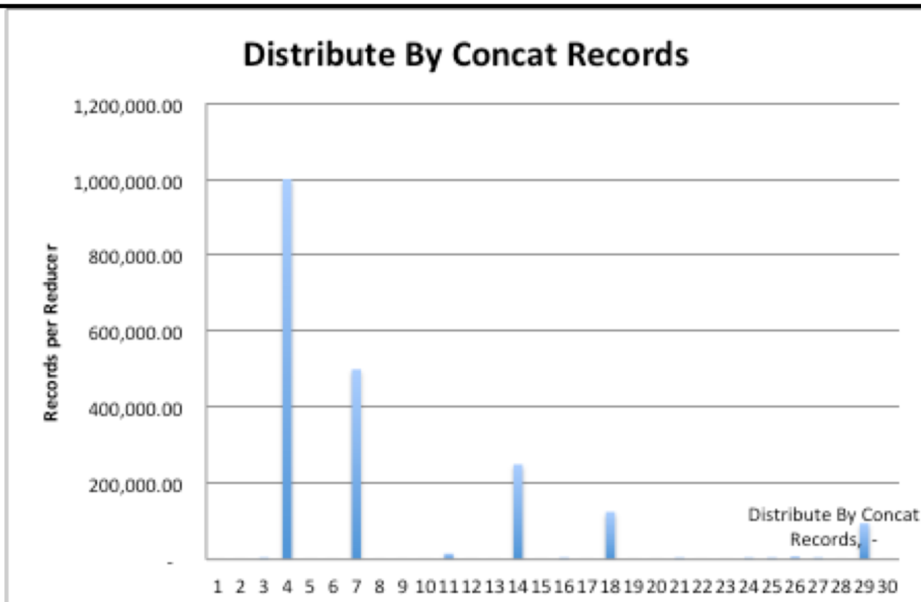
Mistake – Skew : Answers

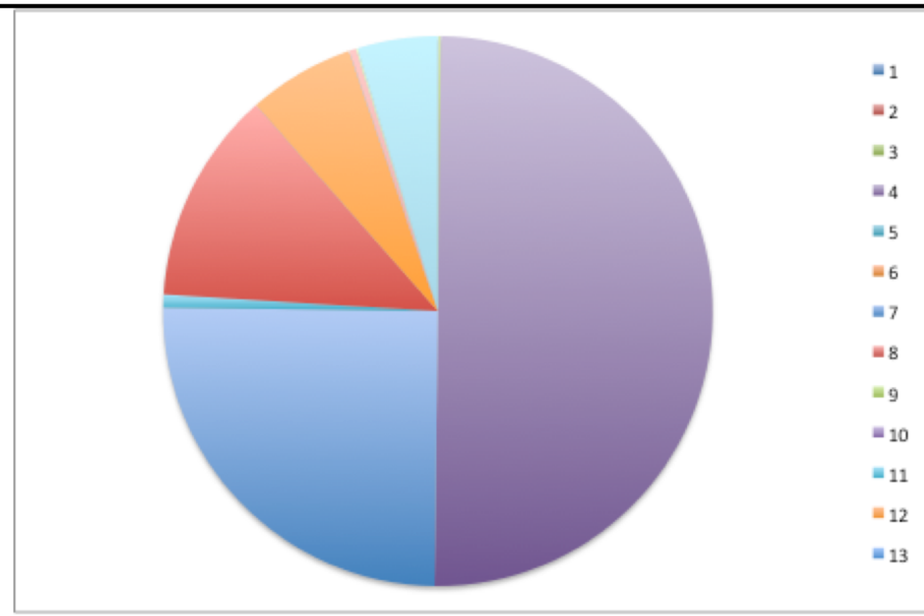
- Salting
- Isolated Salting
- Isolated Map Joins

Mistake – Skew : Salting

- Normal Key: “Foo”
- Salted Key: “Foo” + random.nextInt(saltFactor)

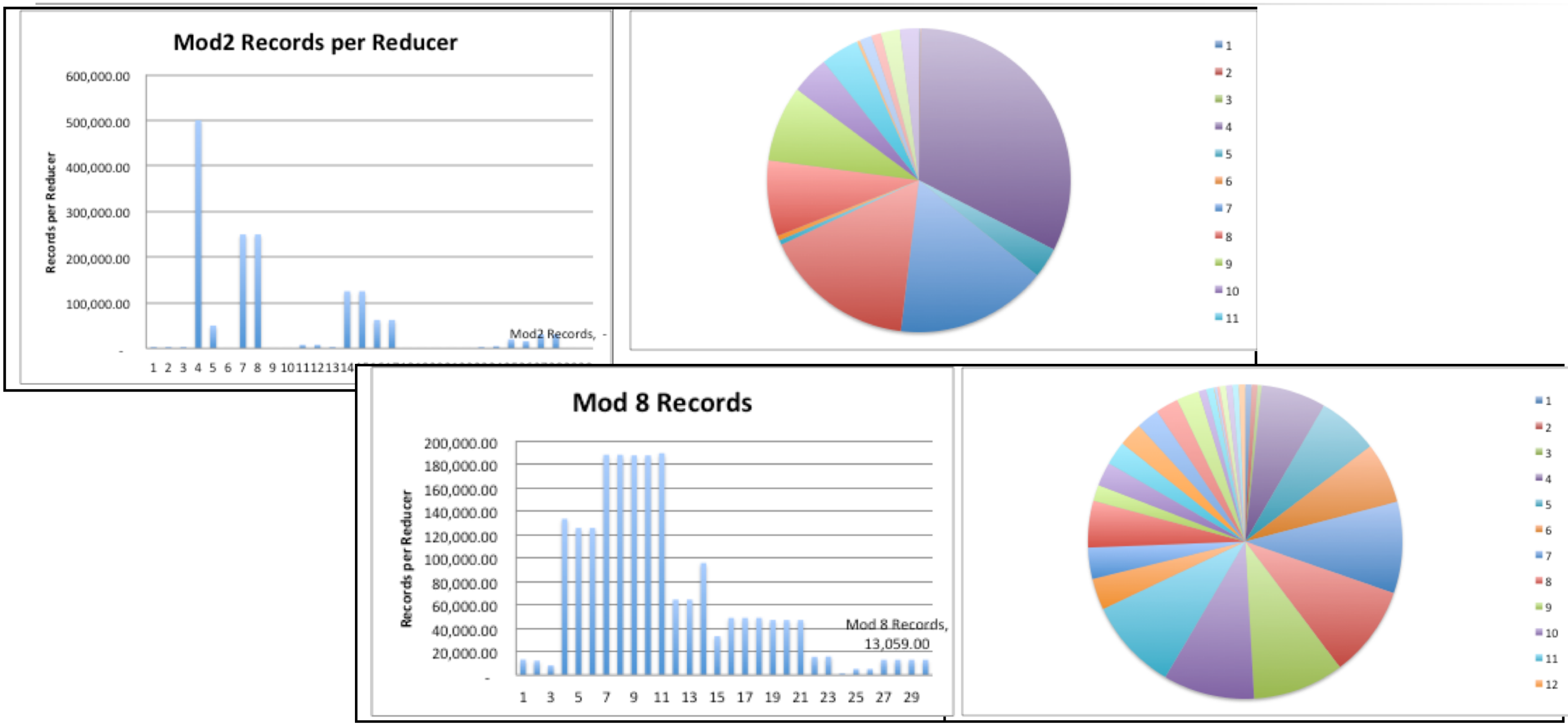
Managing Parallelism





cloudera

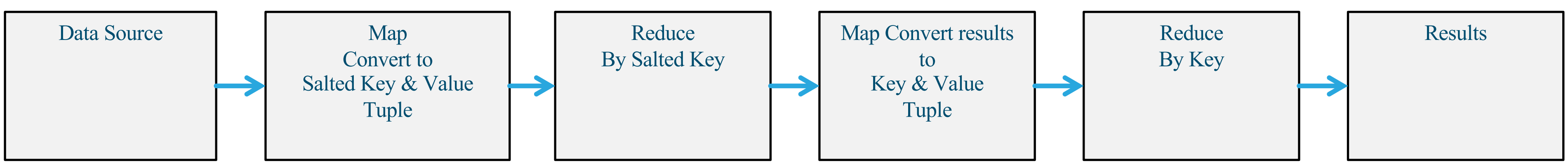
Mistake – Skew: Salting



Add Example Slide

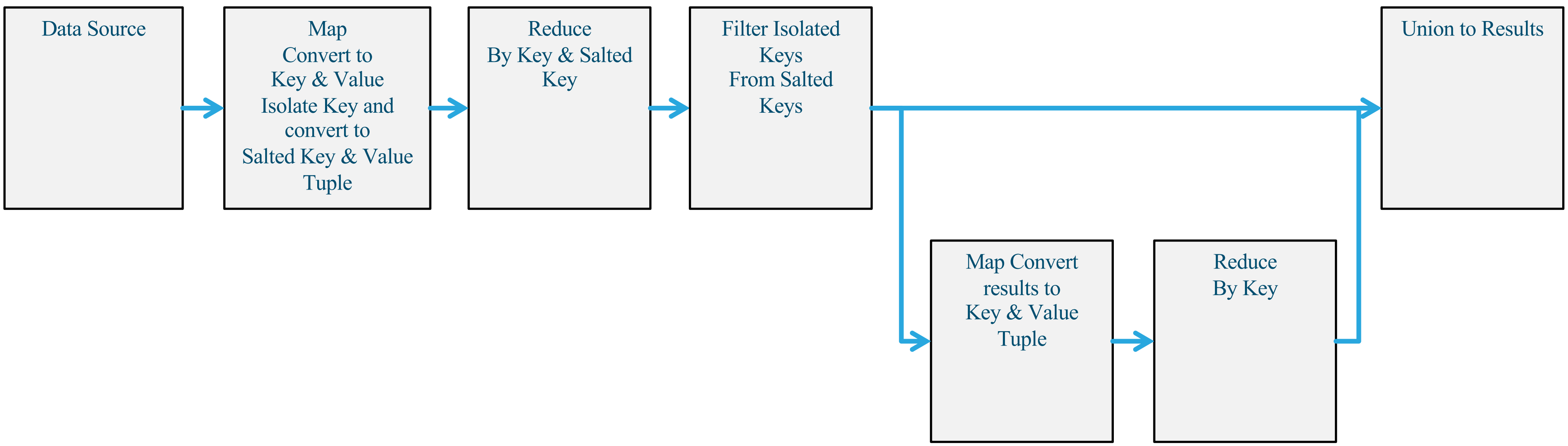
Mistake – Skew : Salting

- Two Stage Aggregation
  - Stage one to do operations on the salted keys
  - Stage two to do operation access unsalted key results



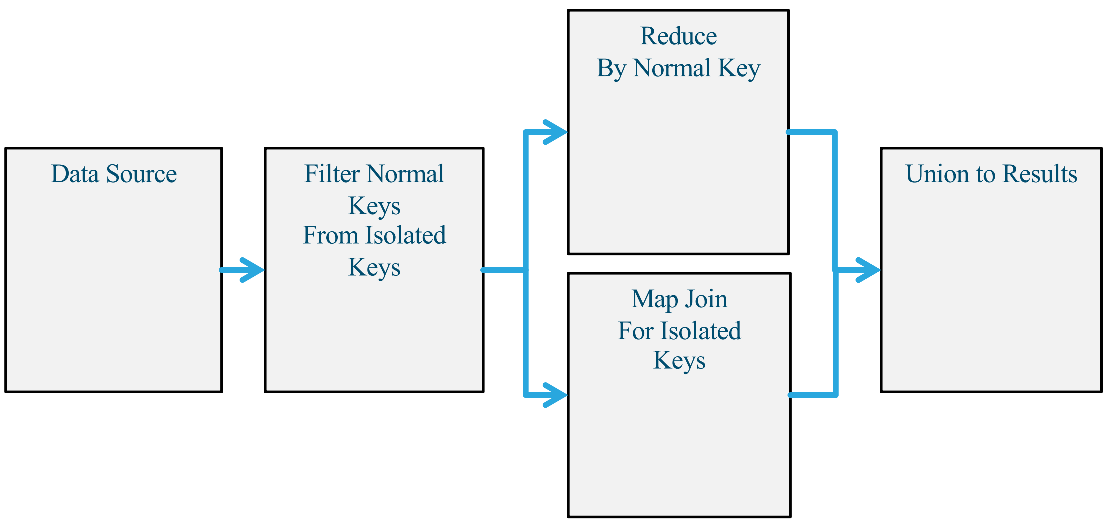
Mistake – Skew : Isolated Salting

- Second Stage only required for Isolated Keys



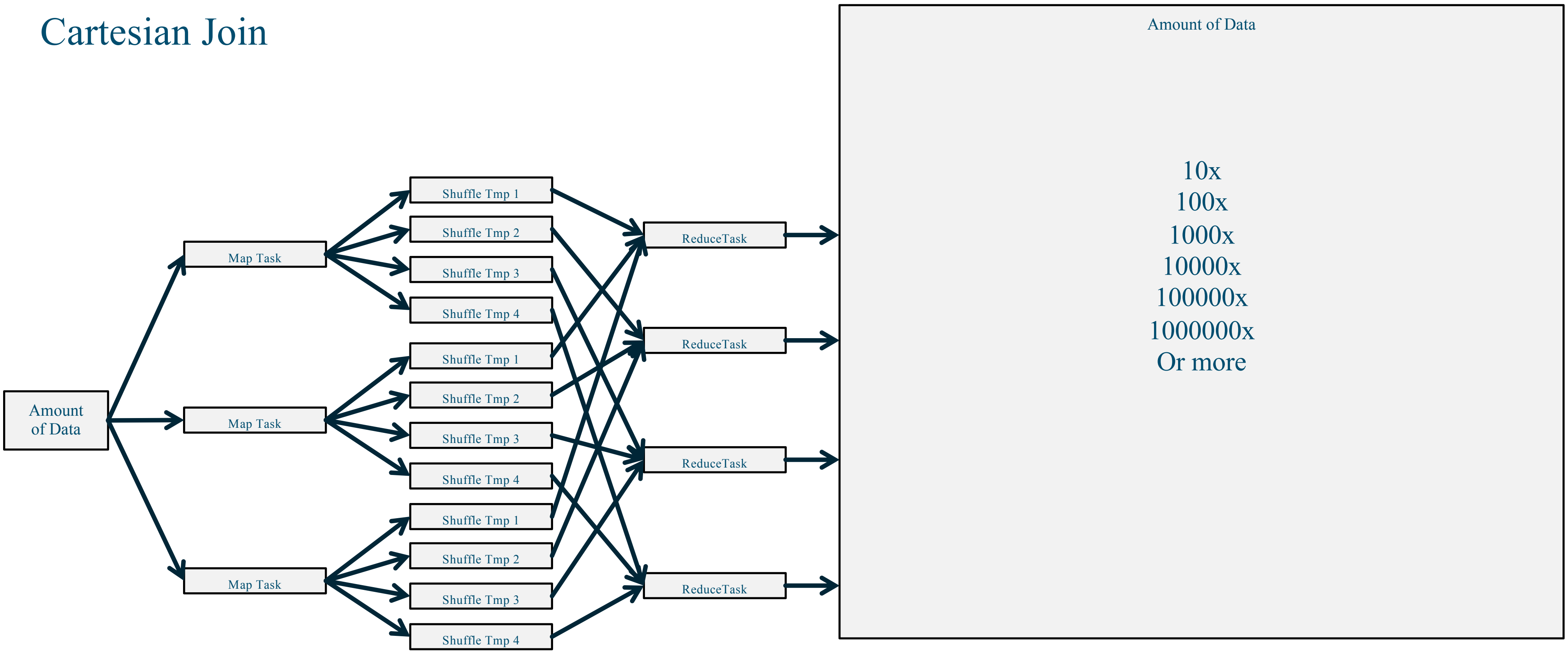
Mistake – Skew : Isolated Map Join

- Filter Out Isolated Keys and use Map Join/Aggregate on those
- And normal reduce on the rest of the data
- This can remove a large amount of data being shuffled



Managing Parallelism

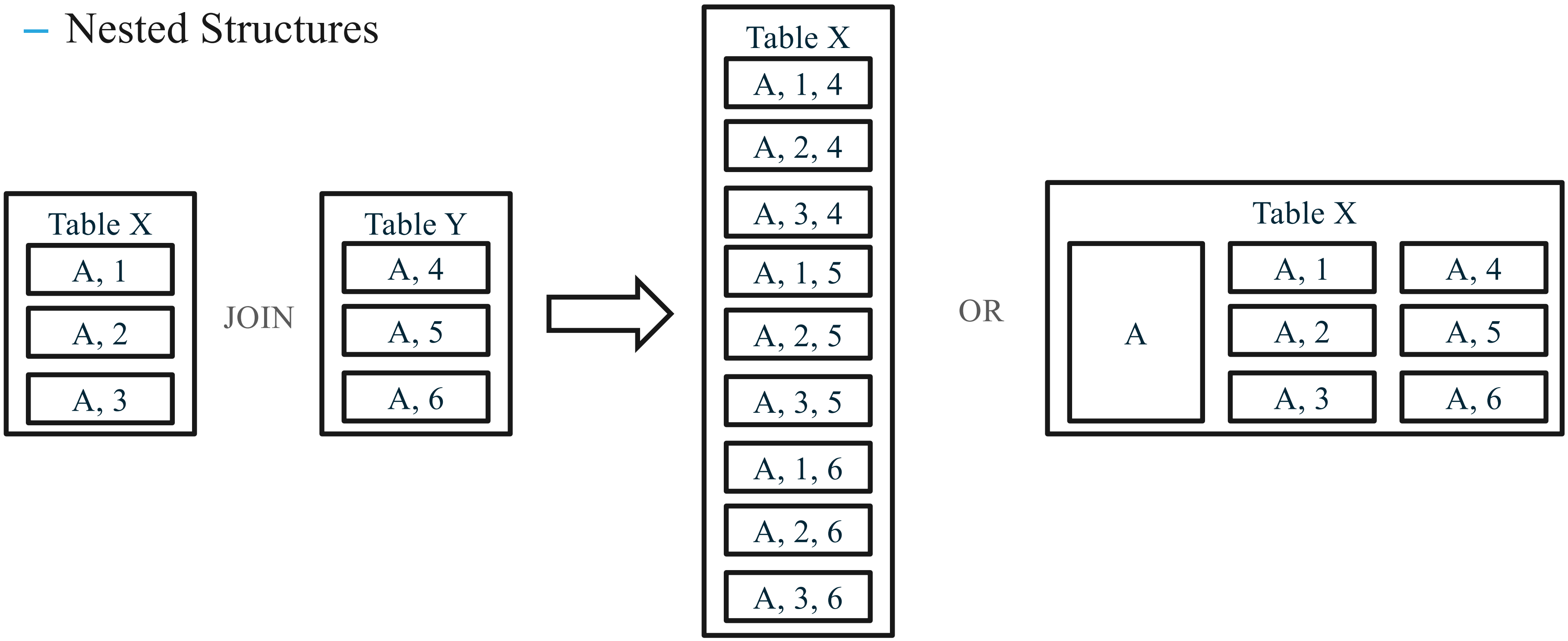
Cartesian Join



cloudera

Managing Parallelism

- How To fight Cartesian Join
  - Nested Structures



Managing Parallelism

- How To fight Cartesian Join
  - Nested Structures

```
create table nestedTable (
  col1 string,
  col2 string,
  col3 array< struct<
   col3_1: string,
   col3_2: string>>
```

=

```
val rddNested = sc.parallelize(Array(
     Row("a1", "b1", Seq(Row("c1_1", "c2_1"),
                         Row("c1_2", "c2_2"),
                         Row("c1_3", "c2_3"))),
     Row("a2", "b2", Seq(Row("c1_2", "c2_2"),
                         Row("c1_3", "c2_3"),
                         Row("c1_4", "c2_4")))), 2)
```

Mistake # 4

Out of luck?

- Do you every run out of memory?
- Do you every have more then 20 stages?
- Is your driver doing a lot of work?

Mistake – DAG Management

- Shuffles are to be avoided
- `ReduceByKey` over `GroupByKey`
- `TreeReduce` over `Reduce`
- Use Complex/Nested Types

Mistake – DAG Management: Shuffles

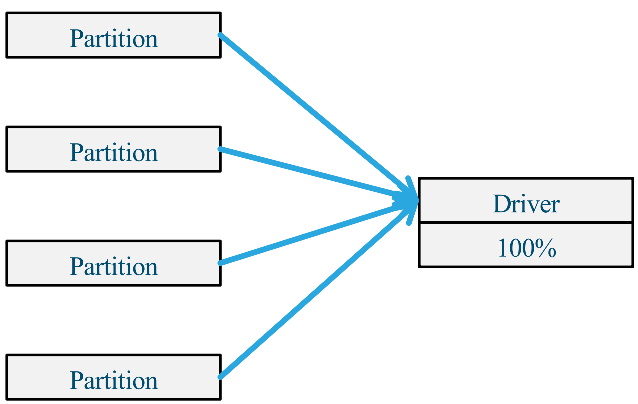
- Map Side reduction, where possible
- Think about partitioning/bucketing ahead of time
- Do as much as possible with a single shuffle
- Only send what you have to send
- Avoid Skew and Cartesians

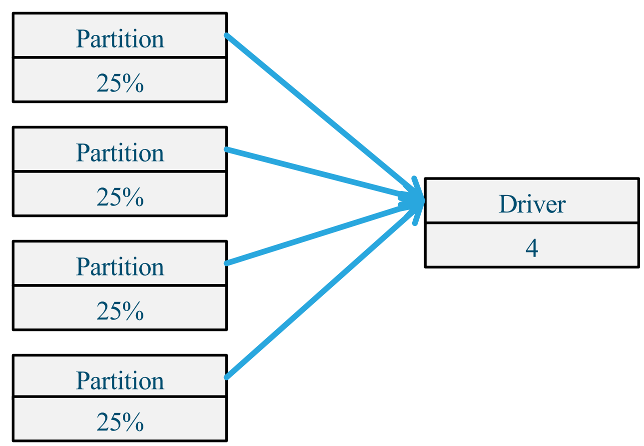
`ReduceByKey` over `GroupByKey`

- `ReduceByKey` can do almost anything that `GroupByKey` can do
  - Aggregations
  - Windowing
  - Use memory
  - But you have more control
- `ReduceByKey` has a fixed limit of Memory requirements
- `GroupByKey` is unbound and dependent on data

TreeReduce over Reduce

- `TreeReduce` & `Reduce` return some result to driver
- `TreeReduce` does more work on the executors
- While `Reduce` bring everything back to the driver





Complex Types

- Top N List
- Multiple types of Aggregations
- Windowing operations

- All in one pass

Complex Types

- Think outside of the box use objects to reduce by
- (Make something simple)

How-to: Do Data Quality Checks using Apache Spark DataFrames

July 9, 2015 | By Ted Malaska | 3 Comments

Categories: How-to Spark

Apache Spark's ability to support data quality checks via DataFrames is progressing rapidly. This post explains the state of the art and future possibilities.

Apache Hadoop and Apache Spark make Big Data accessible and usable so we can easily find value, but that data has to be correct, first. This post will focus on this problem and

cloudera

Mistake # 5

Ever seen this?

```
Exception in thread "main" java.lang.NoSuchMethodError:
com.google.common.hash.HashFunction.hashInt(I)Lcom/google/common/hash/HashCode;
   at org.apache.spark.util.collection.OpenHashSet.org
$apache$spark$util$collection$OpenHashSet$$hashcode(OpenHashSet.scala:261)
   at
org.apache.spark.util.collection.OpenHashSet$mcI$sp.getPos$mcI$sp(OpenHashSet.scala:165)
   at
org.apache.spark.util.collection.OpenHashSet$mcI$sp.contains$mcI$sp(OpenHashSet.scala:102)
   at
org.apache.spark.util.SizeEstimator$$anonfun$visitArray$2.apply$mcVI$sp(SizeEstimator.scala:214)
   at scala.collection.immutable.Range.foreach$mVc$sp(Range.scala:141)
   at
org.apache.spark.util.SizeEstimator$.visitArray(SizeEstimator.scala:210)
   at…....
```

But!

- I already included protobuf in my app's maven dependencies?

Ah!

- My protobuf version doesn't match with Spark's protobuf version!

cloudera

Shading

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.2</version>
...
  <relocations>
    <relocation>
    <pattern>com.google.protobuf</pattern>
    <shadedPattern>com.company.my.protobuf</shadedPattern>
  </relocation>
</relocations>
```

cloudera

Future of shading

- Spark 2.0 has some libraries shaded
  - Gauva is fully shaded

Summary

5 Mistakes

- Size up your executors right
- 2 GB limit on Spark shuffle blocks
- Evil thing about skew and cartesians
- Learn to manage your DAG, yo!
- Do shady stuff, don't let classpath leaks mess you up

THANK YOU.

tiny.cloudera.com/spark-mistakes

Mark Grover | @mark_grover

Ted Malaska | @TedMalaska