Contents

10 Ways |Spark Performance Tuning | Apache Spark Tutorial

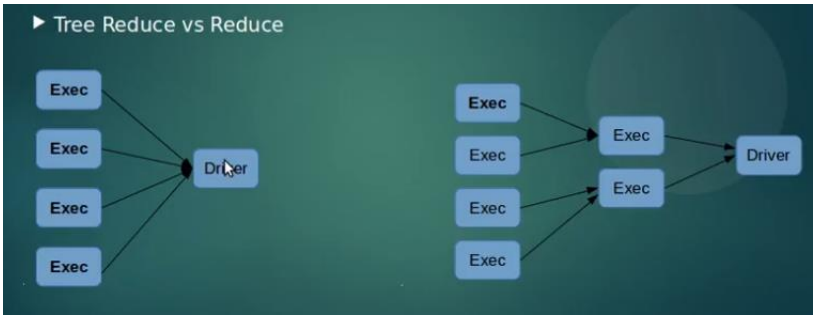
https://www.youtube.com/watch?v=5fpcReZ7UxI&t=126s
https://www.youtube.com/watch?v=vwQhm5LPjvQ
https://www.youtube.com/watch?v=Lqc231nEPsU
https://www.youtube.com/watch?v=ZIEDwEw5SVM
https://www.youtube.com/watch?v=AsVYpQ2Ow4A
https://www.youtube.com/watch?v=BacOO0xwans
- AggreagateByKey Vs CombineByKey
https://www.youtube.com/watch?v=iCWAVLkGxdM
- Catalyst Optimizer
https://www.youtube.com/watch?v=fp53QhSfQcI&feature=youtu.be
- Optimizing Apache Spark SQL Joins – Spark Summit
https://www.youtube.com/watch?v=6zg7NTw-kTQ
- Working with Skewed Data: The Iterative Broadcast - Rob Keevil & Fokko Driesprong – Spark Summit

TreeReduce Vs Reduce

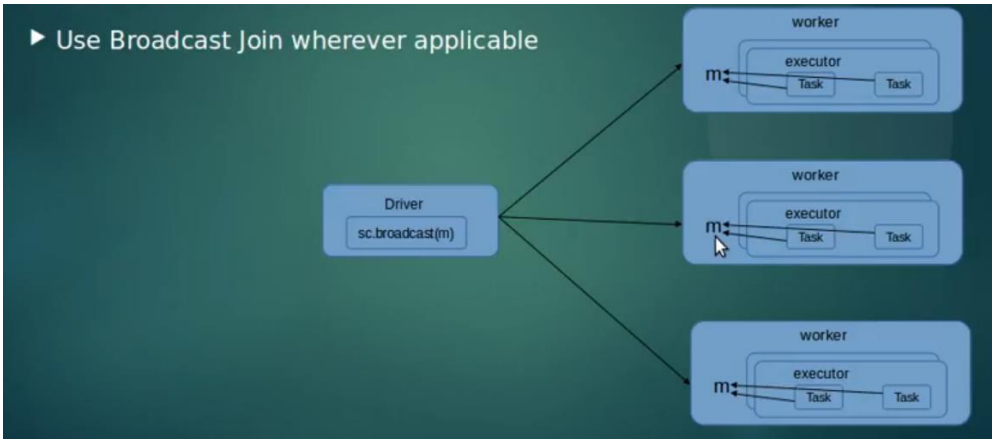
Associative and Commutative nature of data partitions

How many hops between the executors and the driver

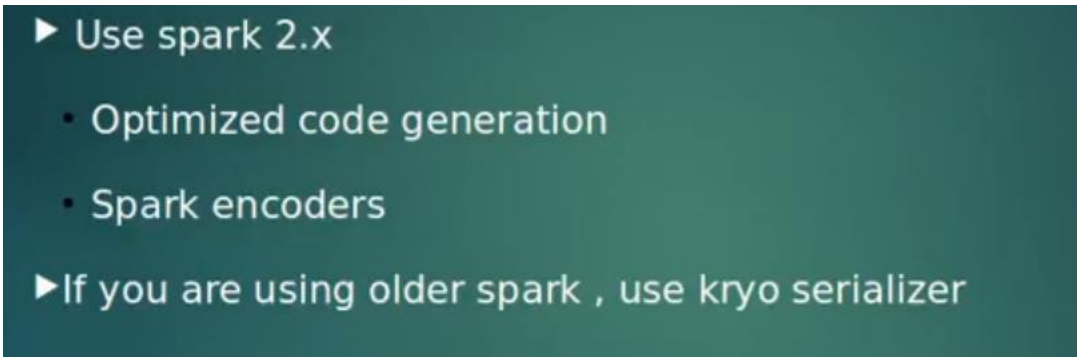
Depth of the tree can be defined



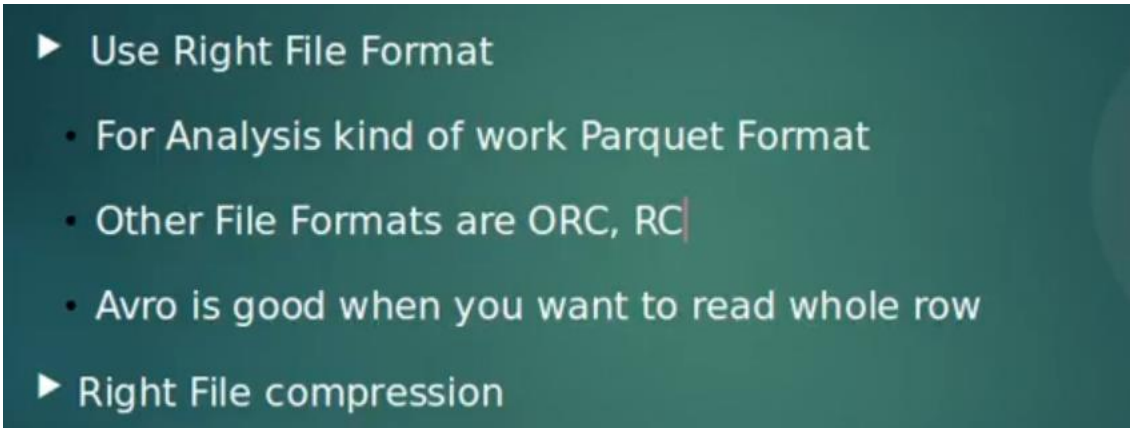
Use Broadcast Join wherever possible



Use spark 2.x



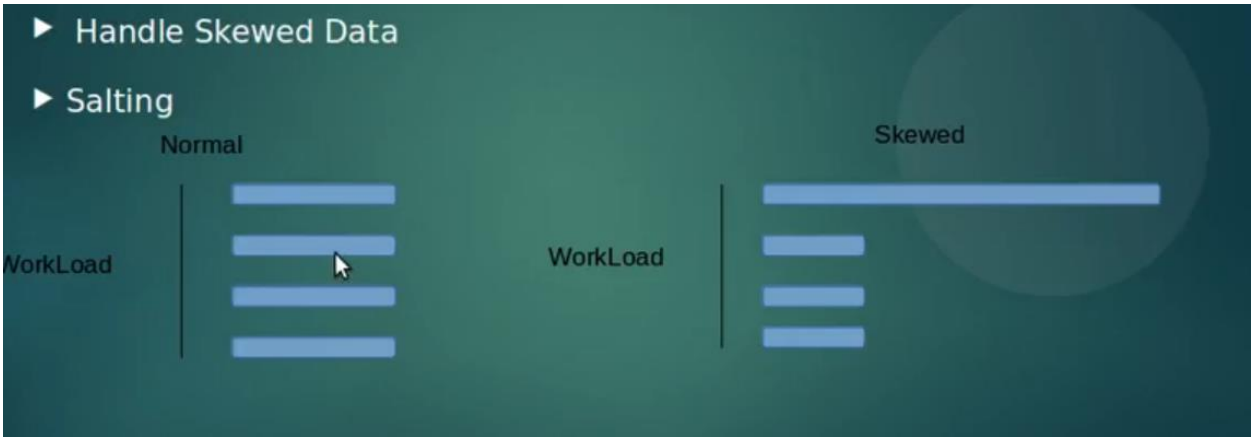
Use Right File format



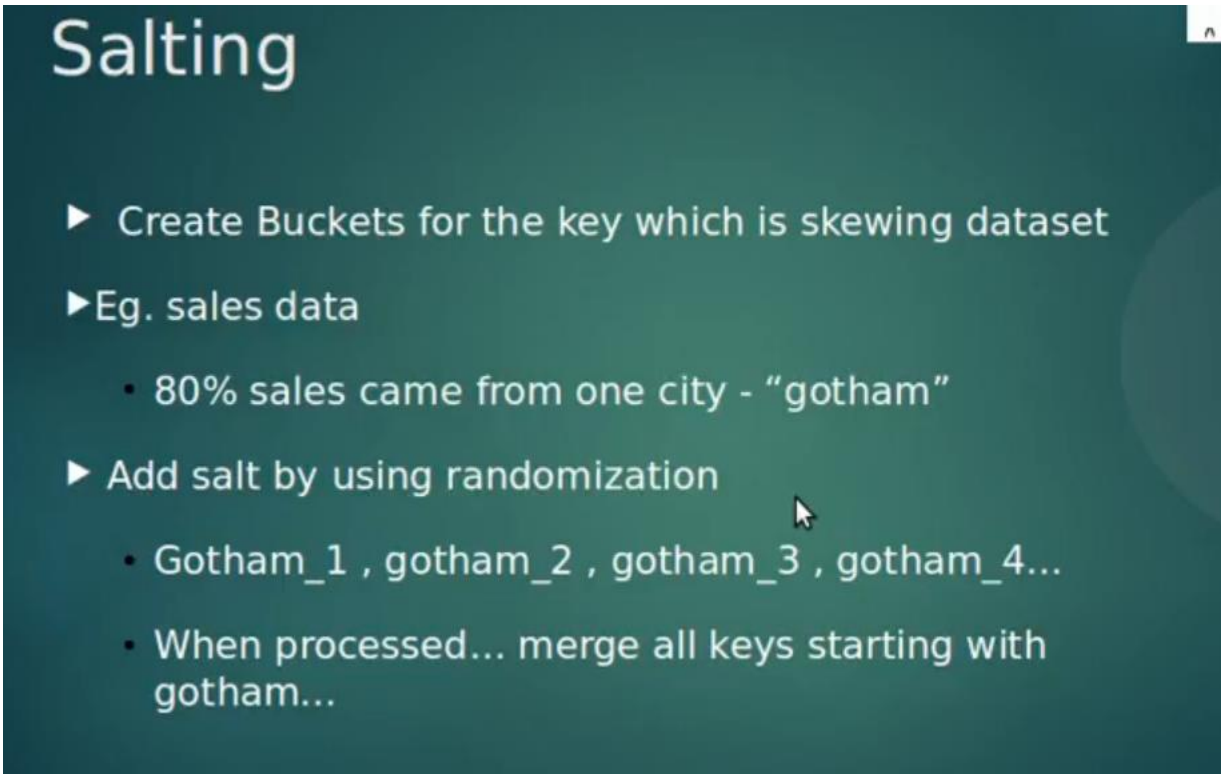
Right File compression

When Lot of shuffling happens, use right file compression

Handled Skewed Data



Salting

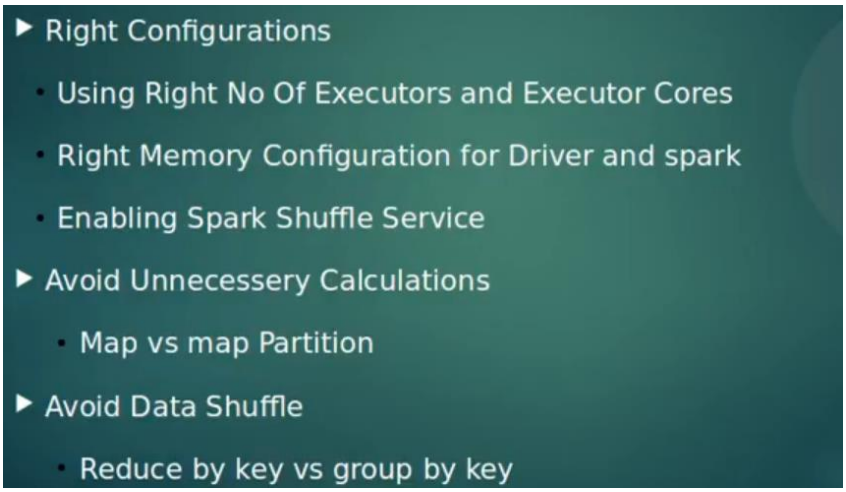


Right Configurations

Using right No. of executors and Executor Cores
Right memory configuration for Driver and Spark
Enabling Spark Shuffle Service

Avoid unnecessary calculations

Map vs map Partition's
Avoid Data shuffle
Reduce by key Vs group by key



Yarn configuration

In case of shared cluster, where different jobs are waiting for the spark cluster resources
Define the queue with priority

Use Vectorization

Process batch of records instead of a row

Why Vectorization helps

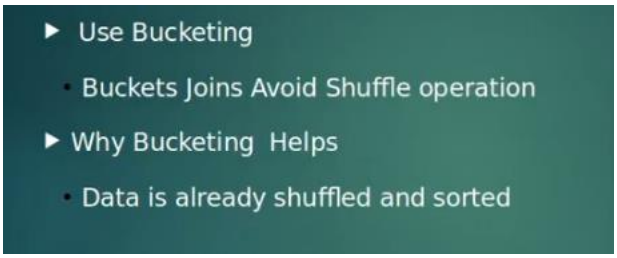
No Object inspections

Use Bucketing

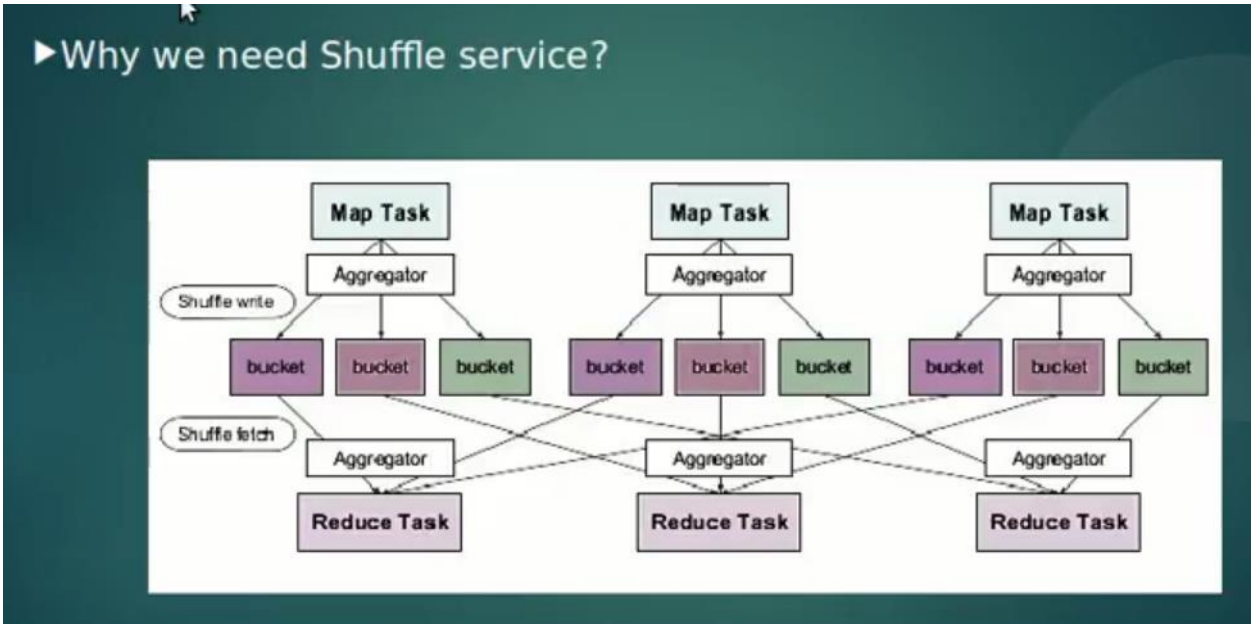
Bucket joins Avoid shuffle operations

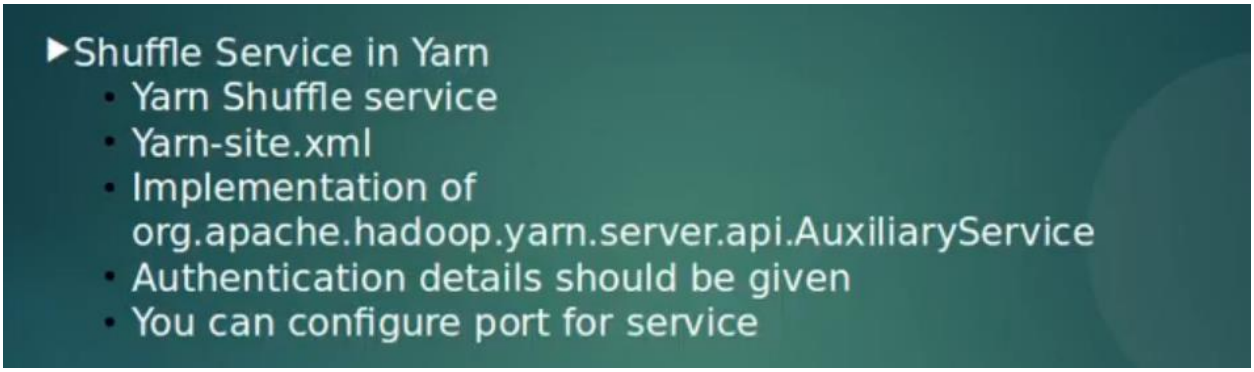
Why Bucketing helps

Data is already shuffled and sorted



Spark Shuffle service





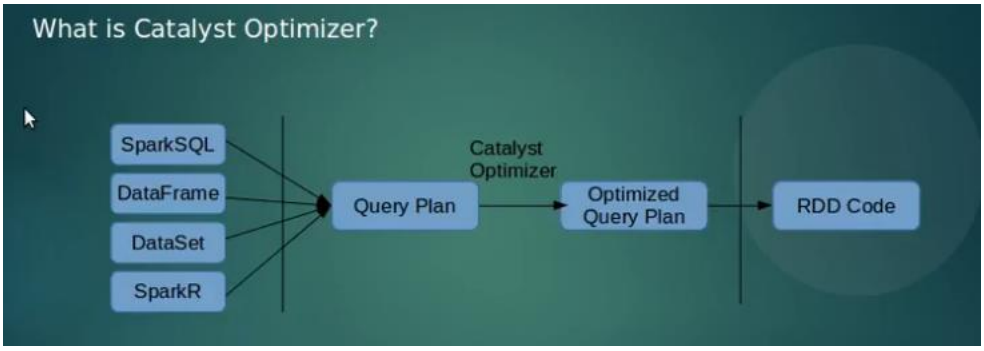
```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>spark_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-
services.spark_shuffle.class</name>

<value>org.apache.spark.network.yarn.YarnShuffleService</value>
  </property>
  <!-- optional -->
  <property>
      <name>spark.shuffle.service.port</name>
      <value>10000</value>
  </property>
  <property>
      <name>spark.authenticate</name>
      <value>true</value>
  </property>
</configuration>
```

- Set Properties
  - spark.dynamicAllocation.enabled
  - spark.shuffle.service.enabled
  - spark.dynamicAllocation.schedulerBacklogTimeout
  - spark.dynamicAllocation.executorIdleTimeout
  - spark.shuffle.service.port

AggreagateByKey Vs CombineByKey

Catalyst Optimizer



Spark Sql Joins

Evolution of Spark

2014:

- Spark 1.x
- RDD based API's.
- Everyday I'm Shufflin'

2017:

- Spark 2.x
- Dataframes & Datasets
- Adv SQL Catalyst
- Optimizing Joins

Topics Covered Today

Basic Joins:

- Shuffle Hash Join
  - Troubleshooting
- Broadcast Hash Join
- Cartesian Join

Special Cases:

- Theta Join
- One to Many Join

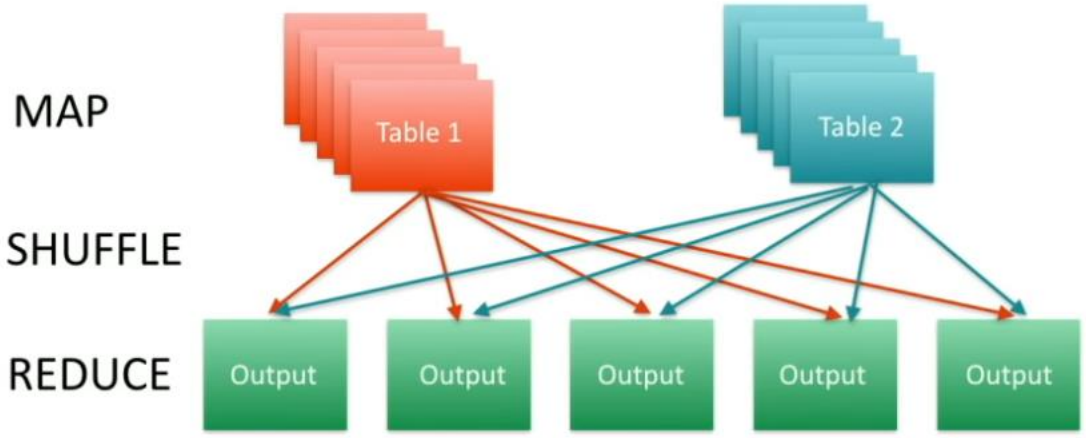
Shuffle Hash Join

Shuffle Hash Join

A Shuffle Hash Join is the most basic type of join, and goes back to Map Reduce Fundamentals.

- **Map** through two different data frames/tables.
- Use the fields in the join condition as the **output key**.
- **Shuffle** both datasets by the output key.
- In the **reduce** phase, join the two datasets now any rows of both tables with the same keys are on the same machine and are sorted.

Shuffle Hash Join



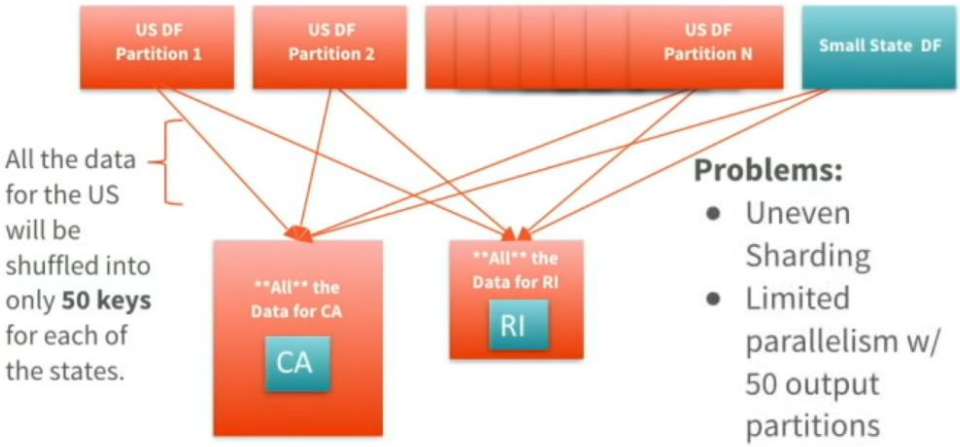
Shuffle Hash Join Performance

Works best when the DF's:

- Distribute evenly with the key you are joining on.
- Have an adequate number of keys for parallelism.

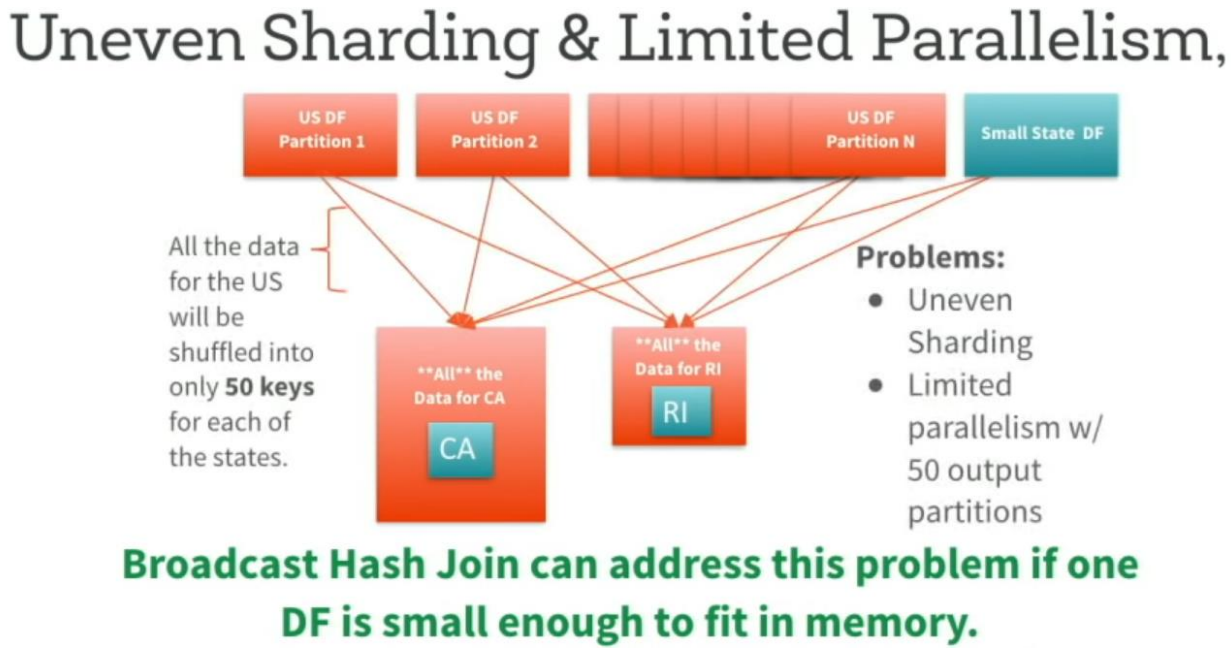
```
join_rdd = sqlContext.sql("select *
  FROM people_in_the_us
  JOIN states
  ON people_in_the_us.state = states.name")
```

Uneven Sharding & Limited Parallelism,

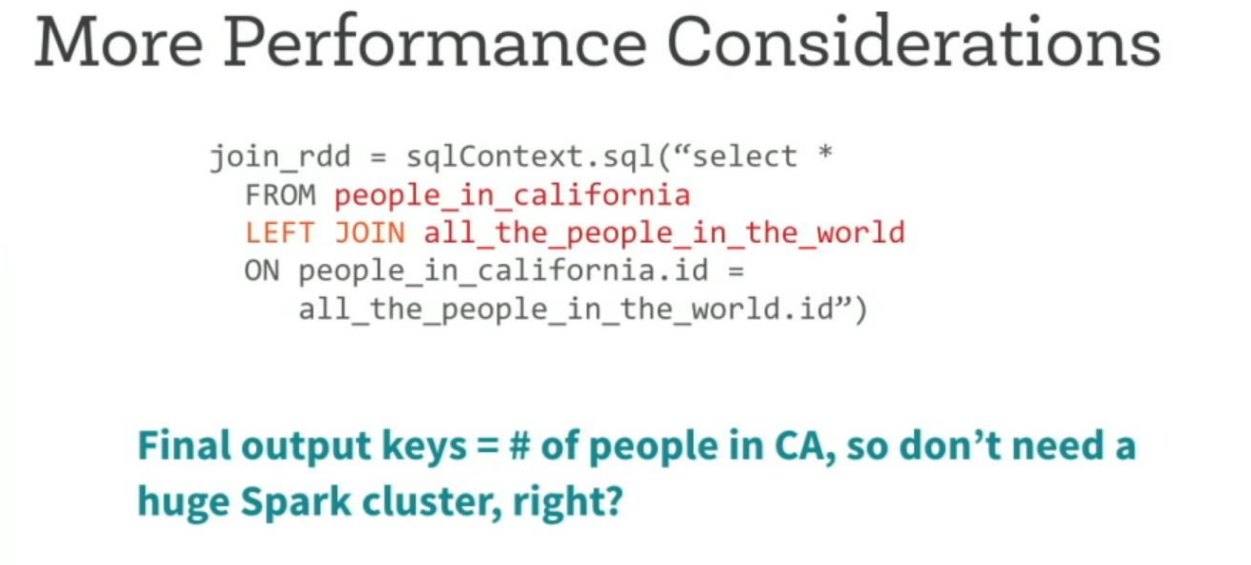


A larger Spark Cluster will not solve these problems!

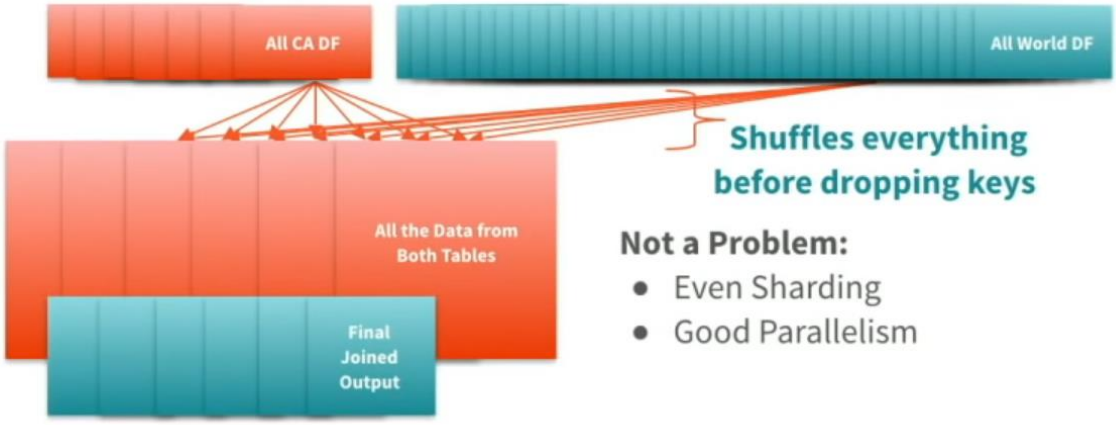
Uneven sharing and Limited Parallelism



More Performance considerations



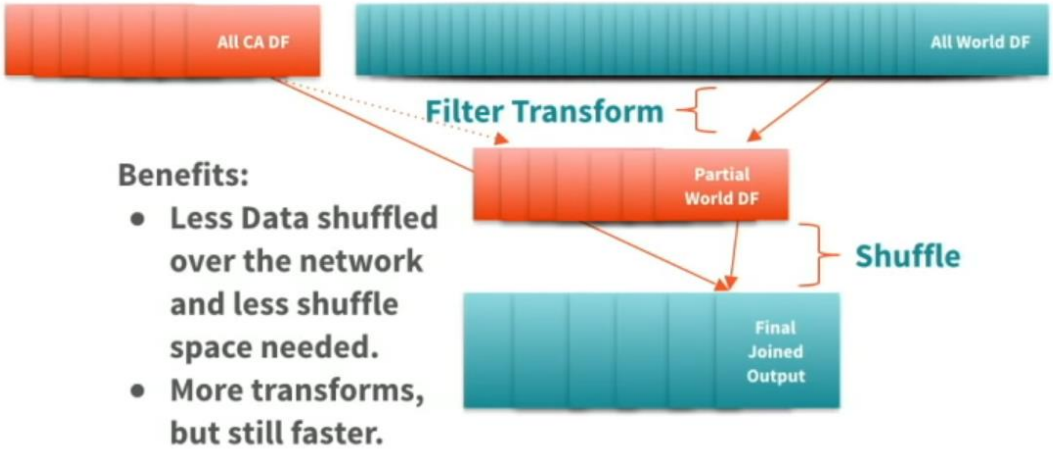
Left Join - Shuffle Step



The Size of the Spark Cluster to run this job is limited by the Large table rather than the Medium Sized Table.

A Better Solution

Filter the World DF for only entries that match the CA ID

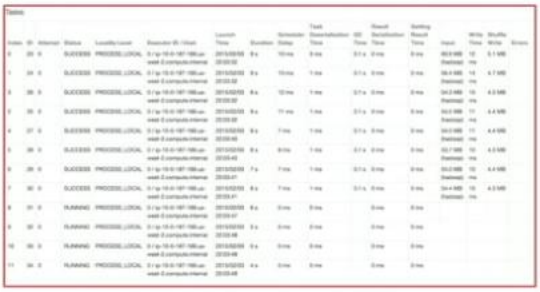


What's the Tipping Point for Huge?

- Can't tell you.
- There aren't always strict rules for optimizing.
- If you were only considering two small columns from the World RDD in Parquet format, the filtering step may not be worth it.

You should understand your data and it's unique properties in order to best optimize your Spark Job.

In Practice: Detecting Shuffle Problems



Check the Spark UI pages for task level detail about your Spark job.

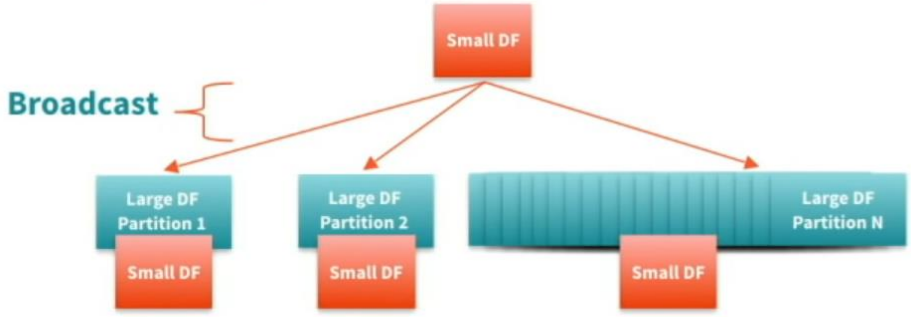
Things to Look for:

- Tasks that take much longer to run than others.
- Speculative tasks that are launching.
- Shards that have a lot more input or shuffle output.

Broadcast Hash Join

Broadcast Hash Join

Optimization: When one of the DF's is small enough to fit in memory on a single machine.



Parallelism of the large DF is maintained (n output partitions), and shuffle is not even needed.

Broadcast Hash Join

- Often optimal over Shuffle Hash Join.
- Use "**explain**" to determine if the Spark SQL catalyst hash chosen Broadcast Hash Join.
- Should be automatic for many Spark SQL tables, may need to provide hints for other types.

Cartesian Join

Cartesian Join

- A cartesian join can easily explode the number of output rows.

  100,000 X 100,000 = 10 Billion

- Alternative to a full blown cartesian join:
  - Create an RDD of UID by UID.
  - Force a Broadcast of the rows of the table .
  - Call a UDF given the UID by UID to look up the table rows and perform your calculation.
- Time your calculation on a sample set to size your cluster.

One to Many Join

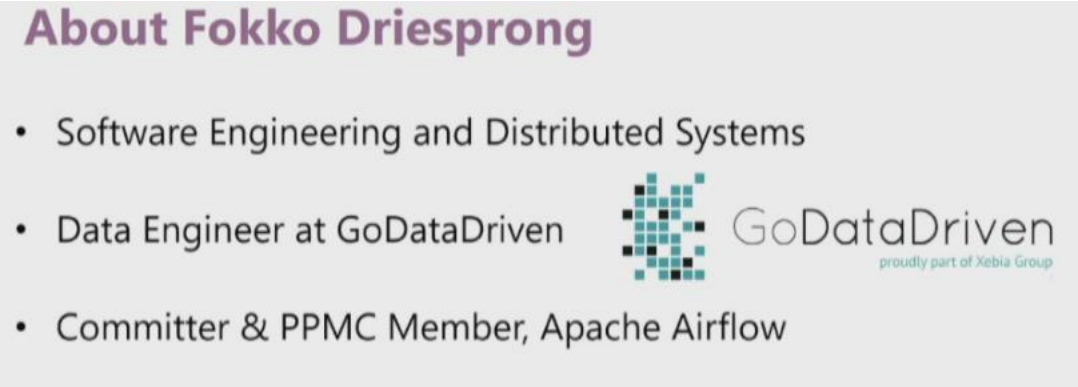
One To Many Join

- A single row on one table can map to many rows on the 2nd table.
- Can explode the number of output rows.
- Not a problem if you use parquet - the size of the output files is not that much since the duplicate data encodes well.

Theta Join

Working Skewed Data

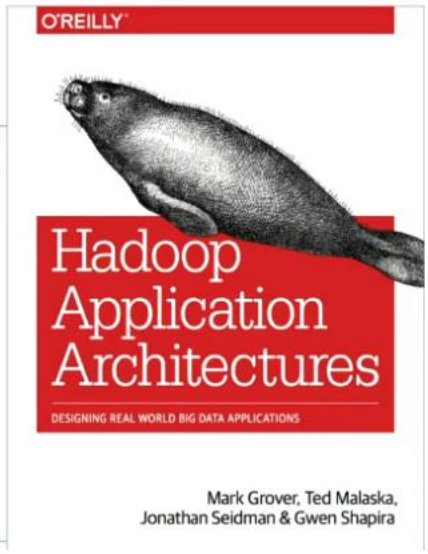
The Iterative Broadcast



Top 5 Mistakes When Writing Spark Applications

About the book

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



#Mistake 1

Executors, cores, memory !?!

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

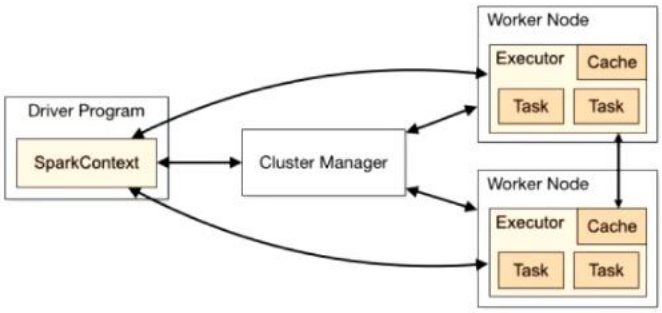
Decisions, decisions, decisions

- 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)

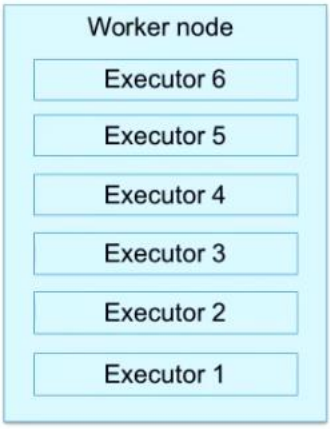
Architecture recap

Spark Architecture recap



Answer #1 – Most granular

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



SPARK SUMMIT 2016

Why?

- Not using benefits of running multiple tasks in same executor

Answer #2 – Least granular

- 6 executors in total

=>1 executor per node

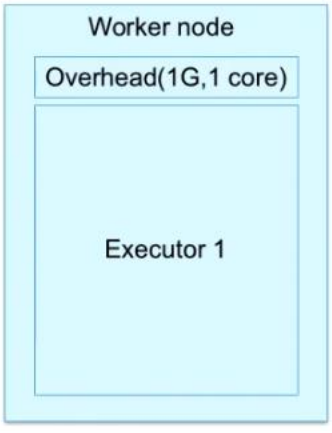
- 64 GB memory each
- 16 cores each

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



Yarn configuration

Let's assume…

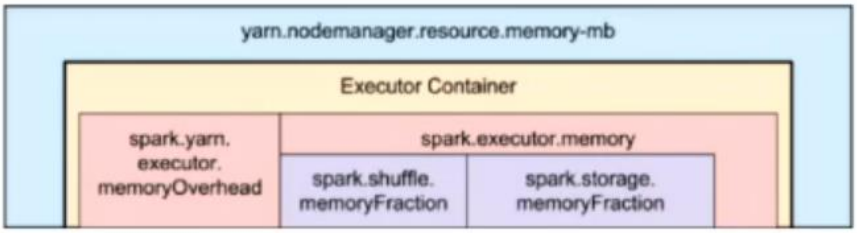
- You are running Spark on YARN, from here on…

3 things

- 3 other things to keep in mind

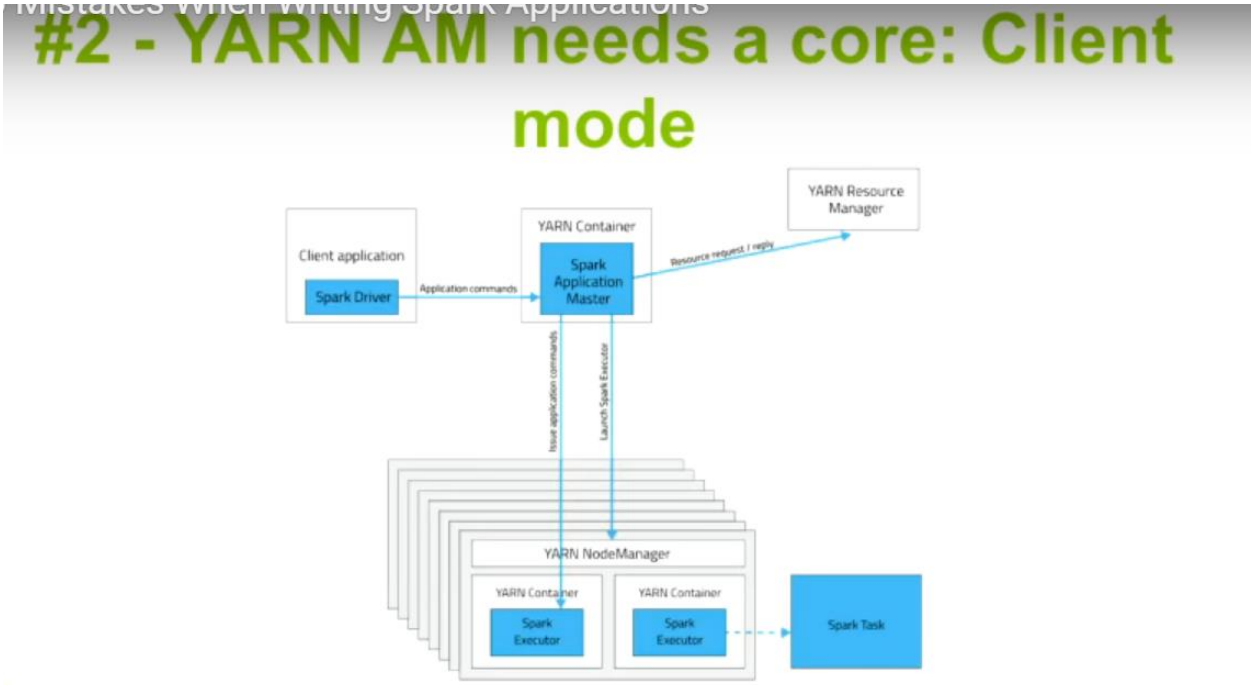
#1 Memory overhead

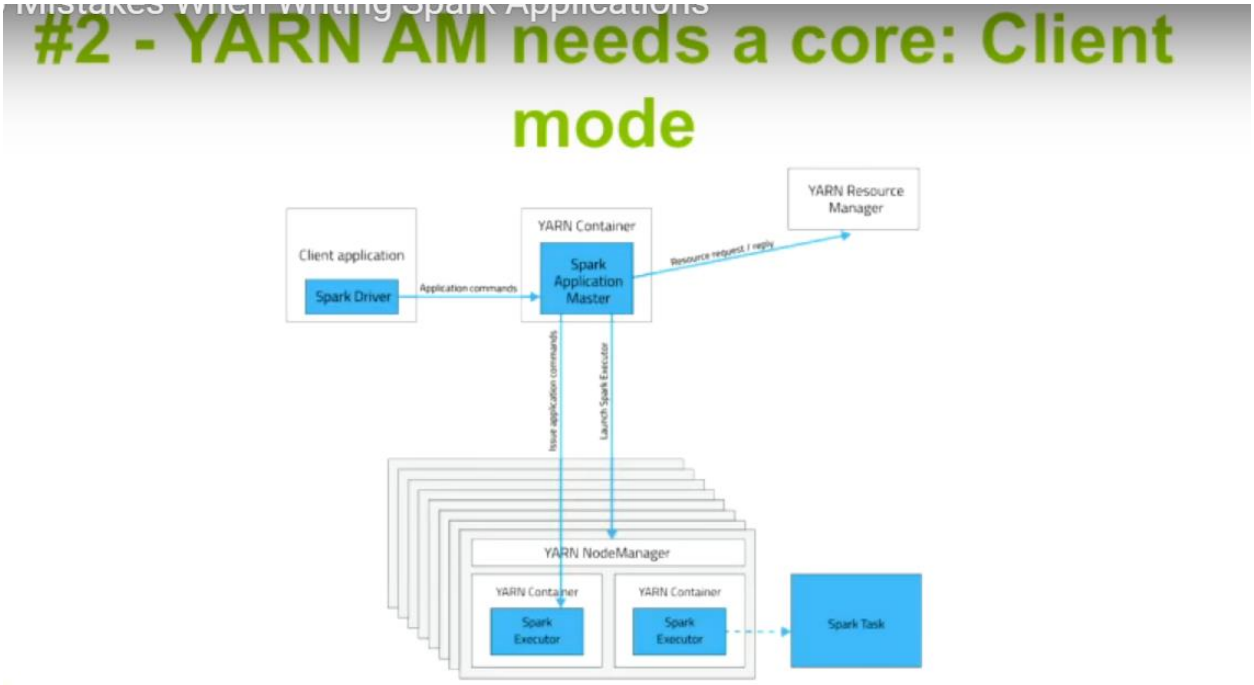
#1 – Memory overhead



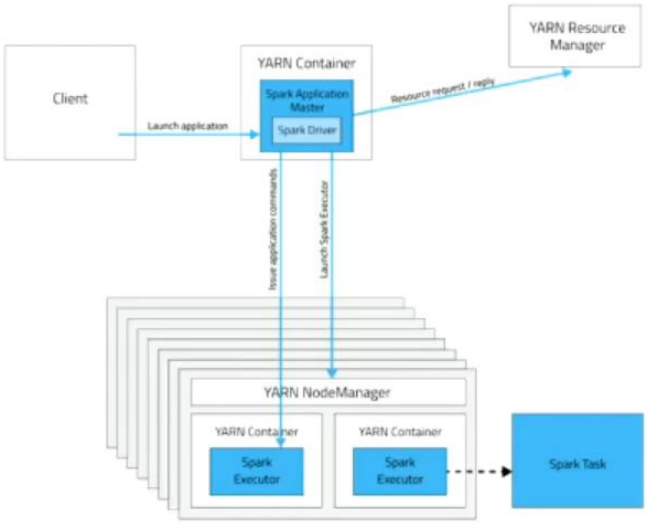
- --executor-memory controls the heap size
- Need some overhead (controlled by spark.yarn.executor.memory.overhead) for off heap memory
  - Default is max(384MB, .07 * spark.executor.memory)

#2 YARM AM needs a core: Client mode





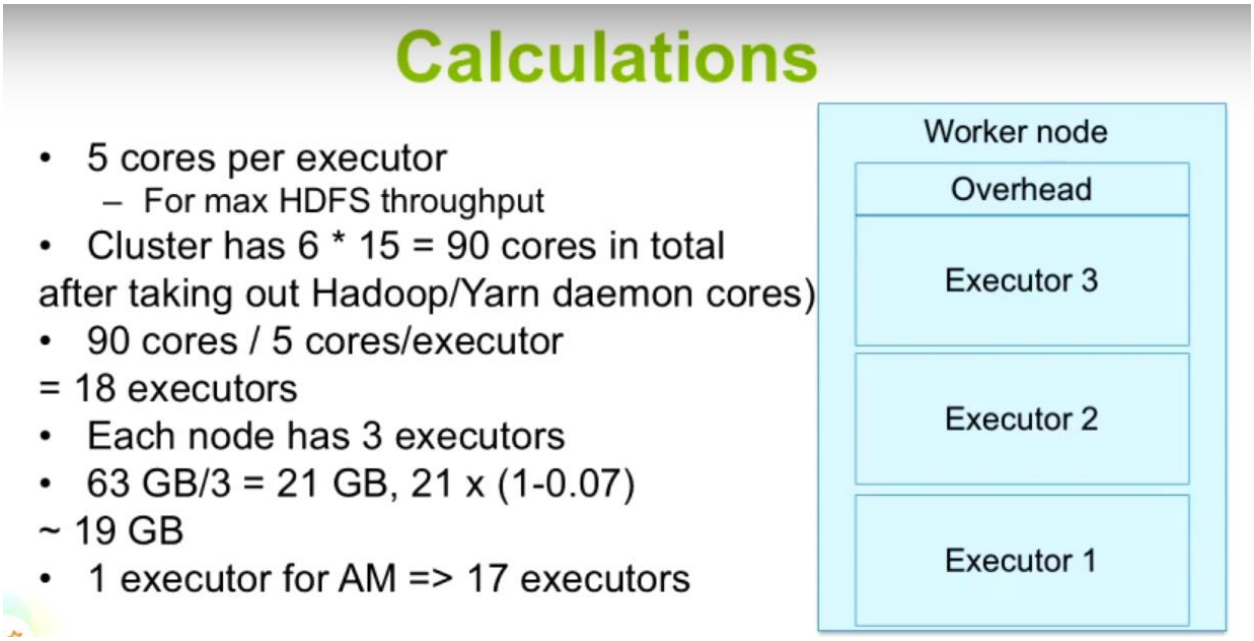
#2 YARN AM needs a core: Cluster mode

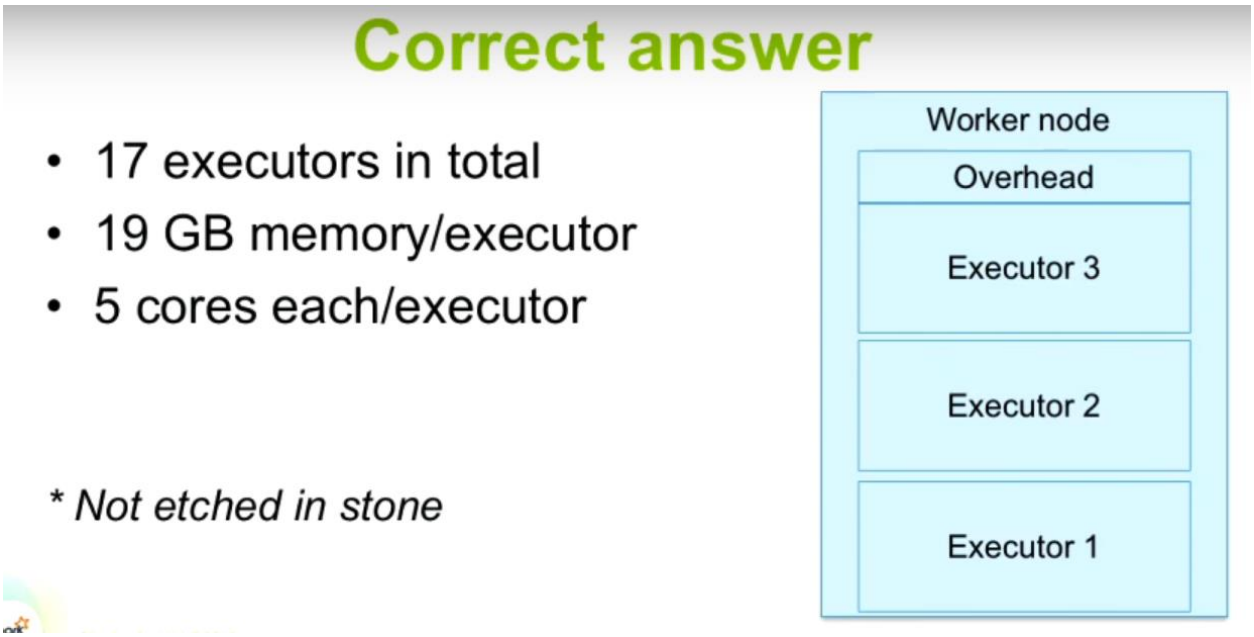


#3 HDFS Throughput

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

Spark cluster configuration calculation





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/

#2Mistake 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:51
7) 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?

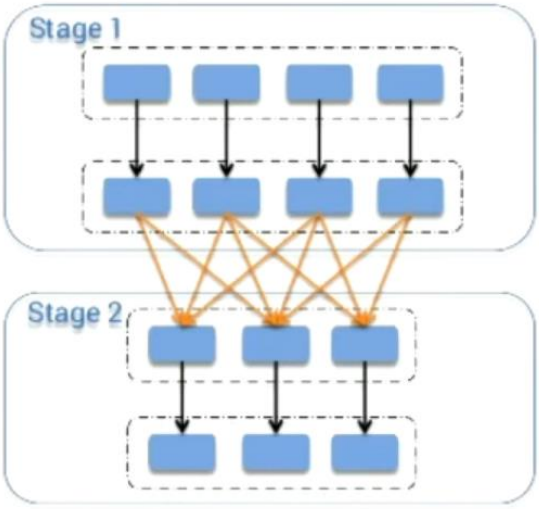
- No Spark shuffle block can be greater than 2 GB

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

Define shuffle and Partitions

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

Repartition or coalesce

Umm, how exactly?

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

How many partitions?

But, how many partitions should I have?

- Rule of thumb is around 128 MB per partition

IF Number of partitions > 2000 or < 2000

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.

Few tips to have optimum number of partitions?

Can you summarize, please?

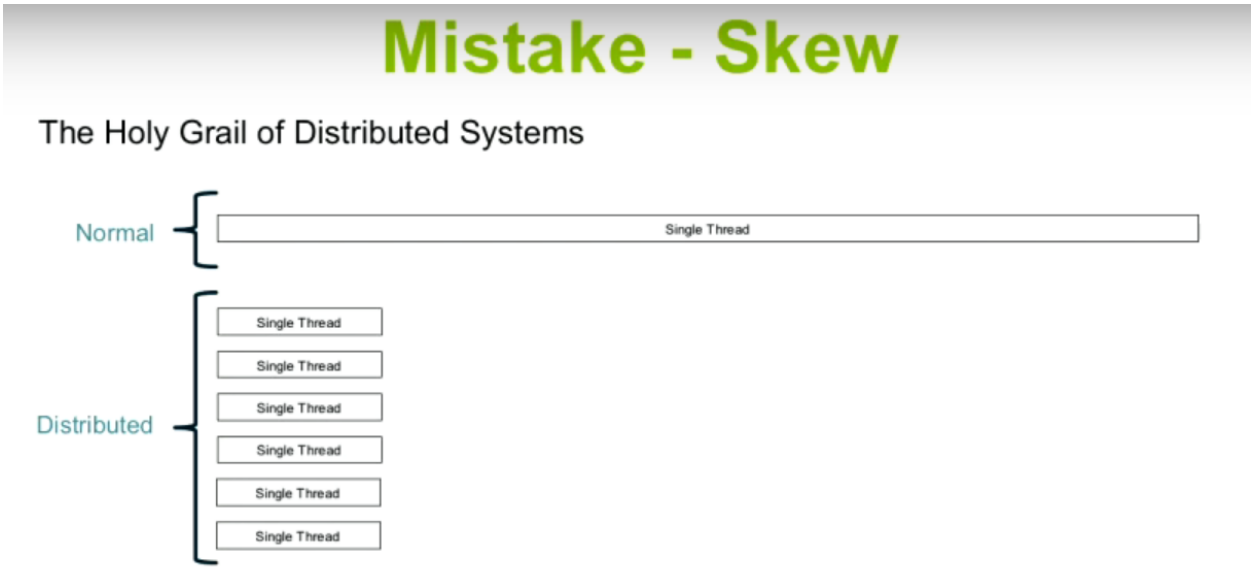
- Don't have too big partitions
  - Your job will fail due to 2 GB limit
- Don't have too few partitions
  - Your job will be slow, not making using of parallelism
- Rule of thumb: ~128 MB per partition
- If #partitions < 2000, but close, bump to just > 2000
- Track SPARK-6235 for removing various 2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

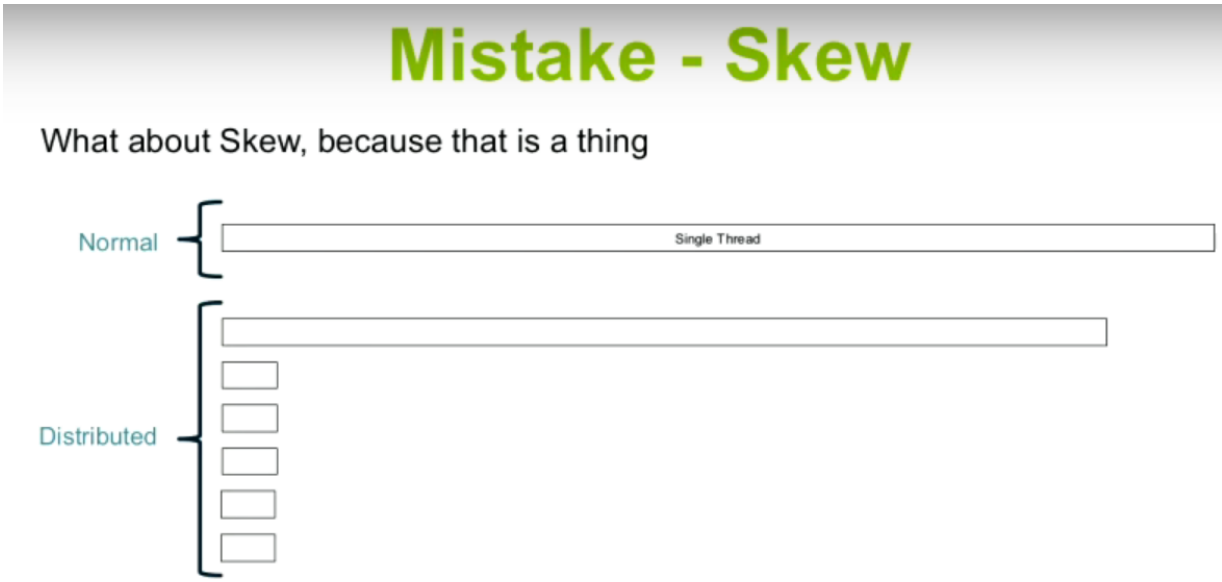
Mistake #3

Slow jobs on join and shuffle

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?





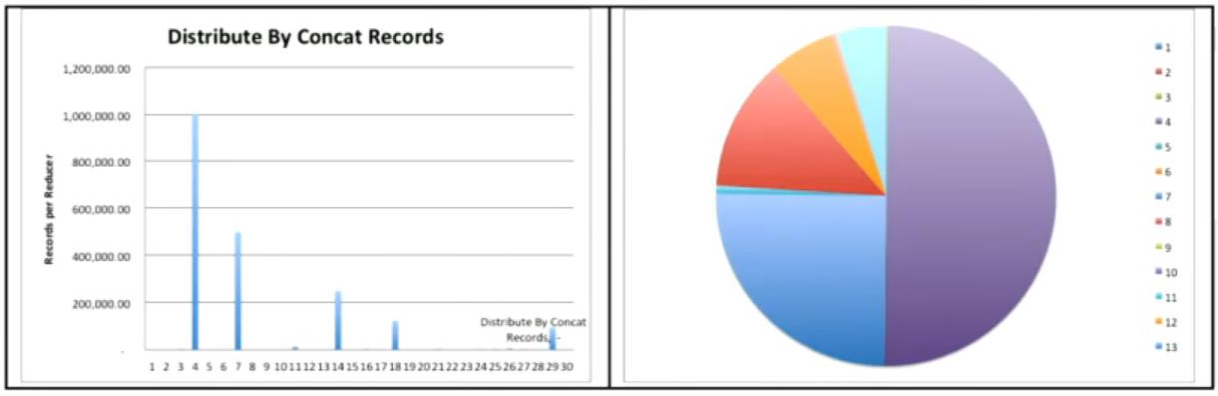
Mistake – Skew: Answers

Mistake – Skew : Salting

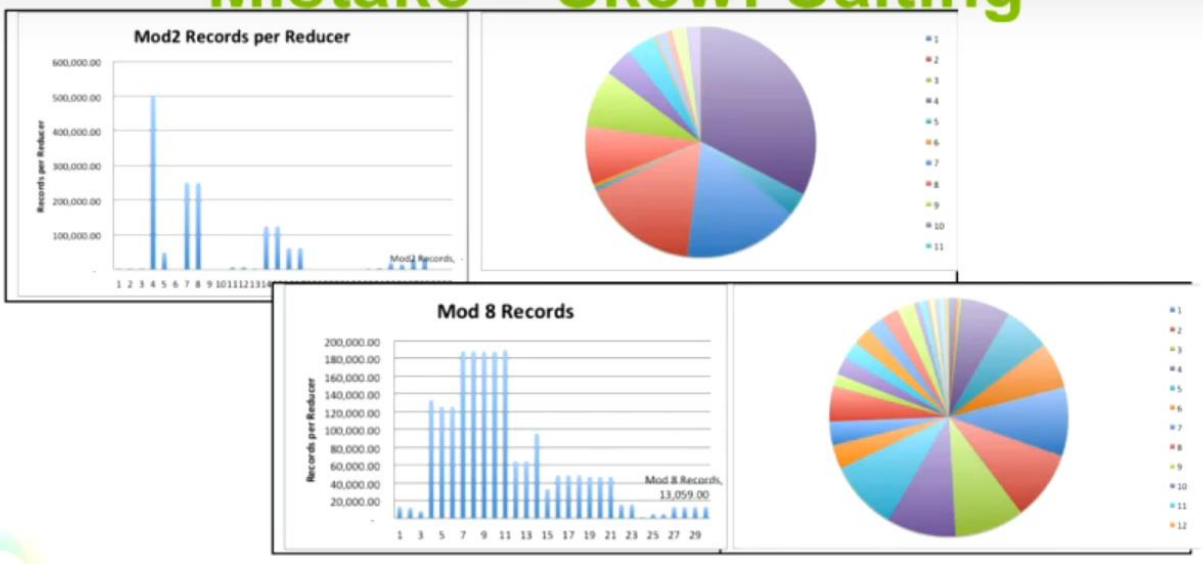
- Normal Key: “Foo”
- Salted Key: “Foo” + random.nextInt(saltFactor)

Managing parallelism

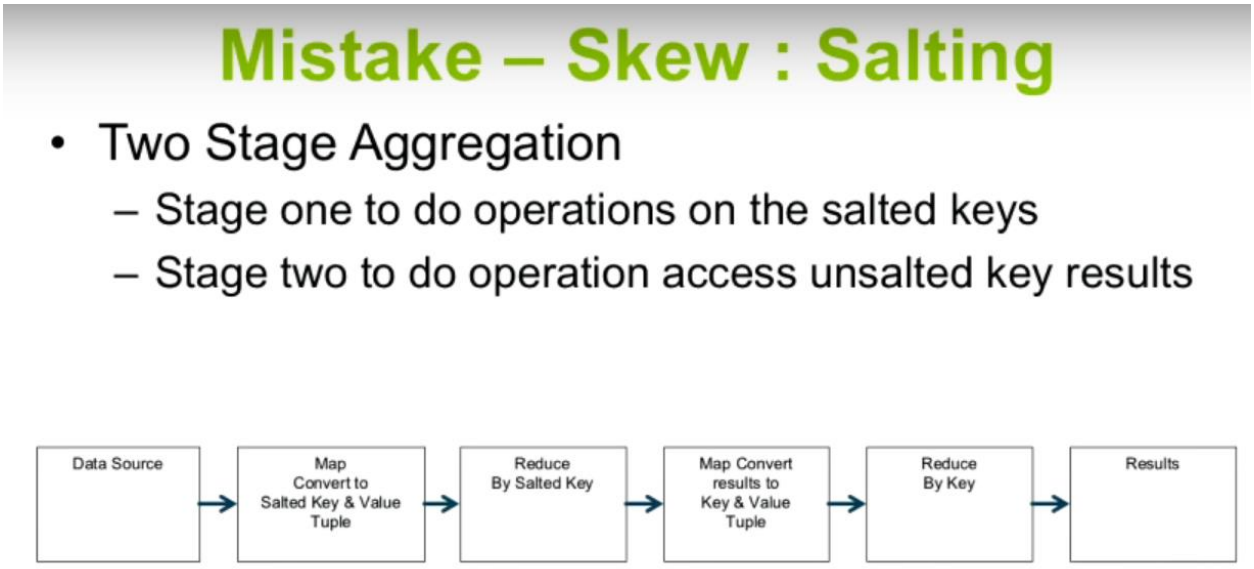
Managing Parallelism



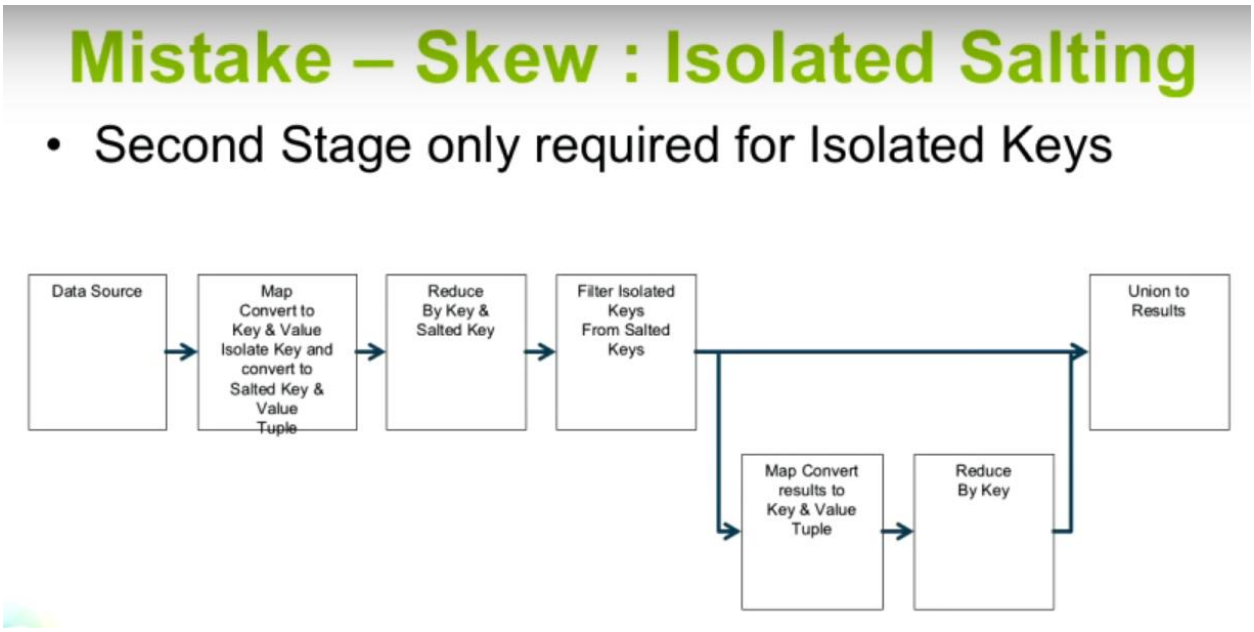
Mistake – Skew: Salting



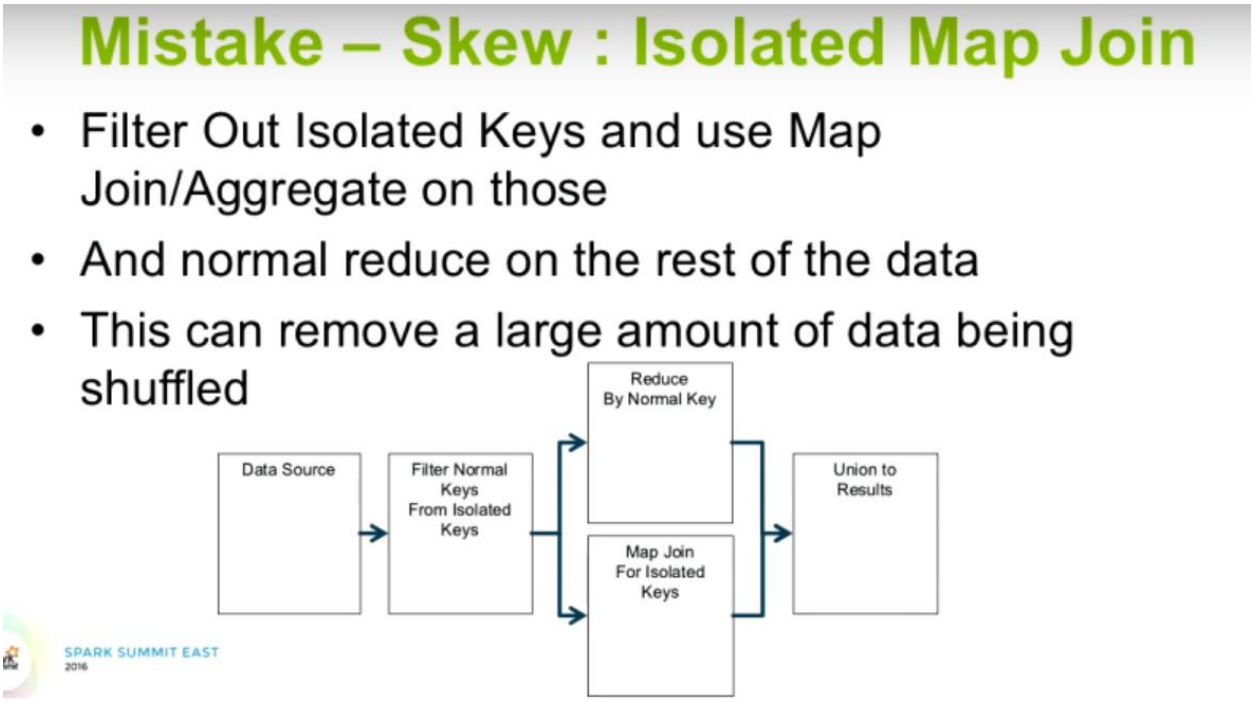
Mistake – Skew : Salting: Two stage aggregation



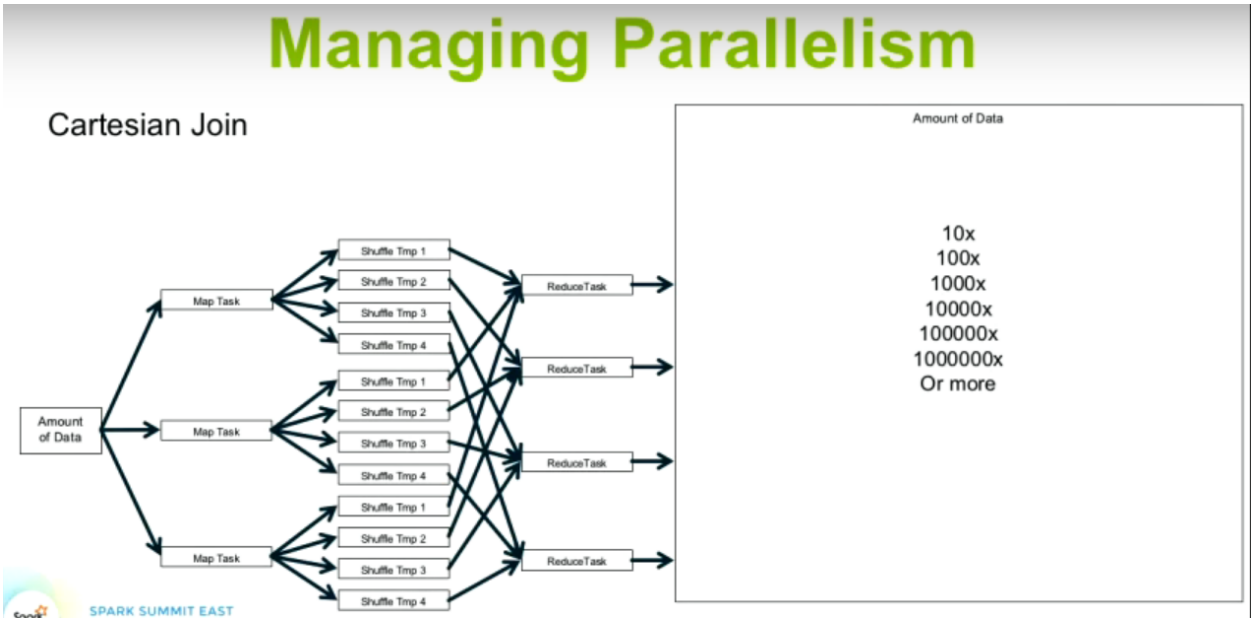
Mistake – Skew : Salting: Isolated Salting



Mistake – Skew: Isolated Map Join



Managing parallelism: Cartesian join



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>>
```

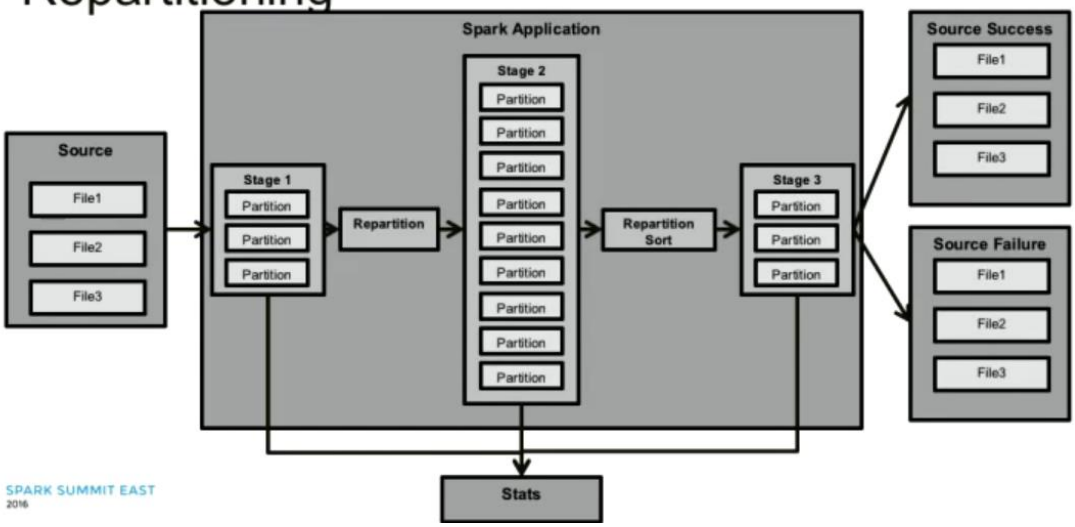
=

```
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)
```

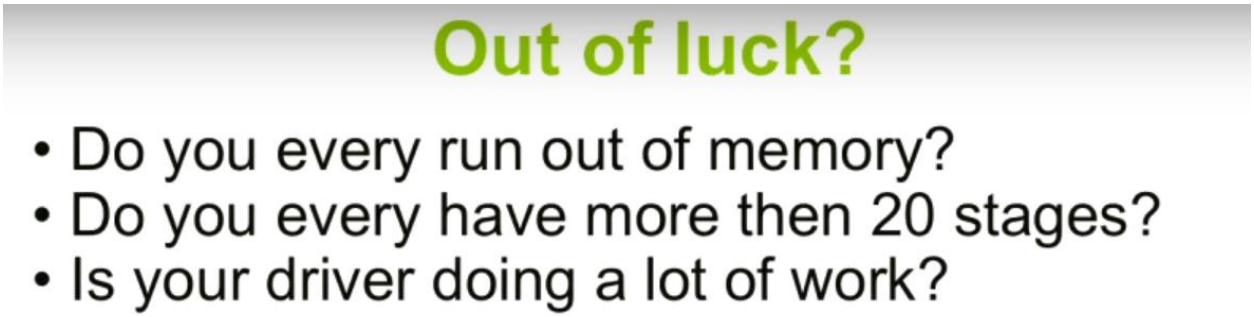
Repartitioning

Managing Parallelism

- Repartitioning



Mistake #4:



Mistake – DAG Management

Shuffles are to be avoided

ReduceByKey over GroupByKey

TreeReduce over Reduce

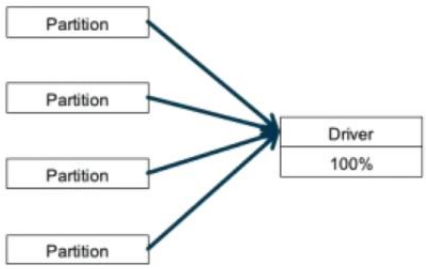
Use Complex/NestedTypes

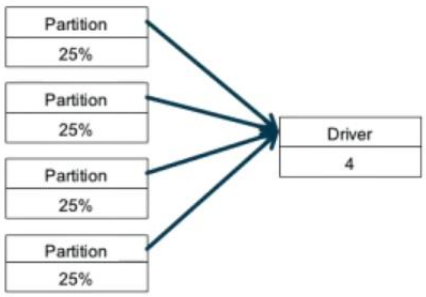
ReduceByKey over GroupByKey

- `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

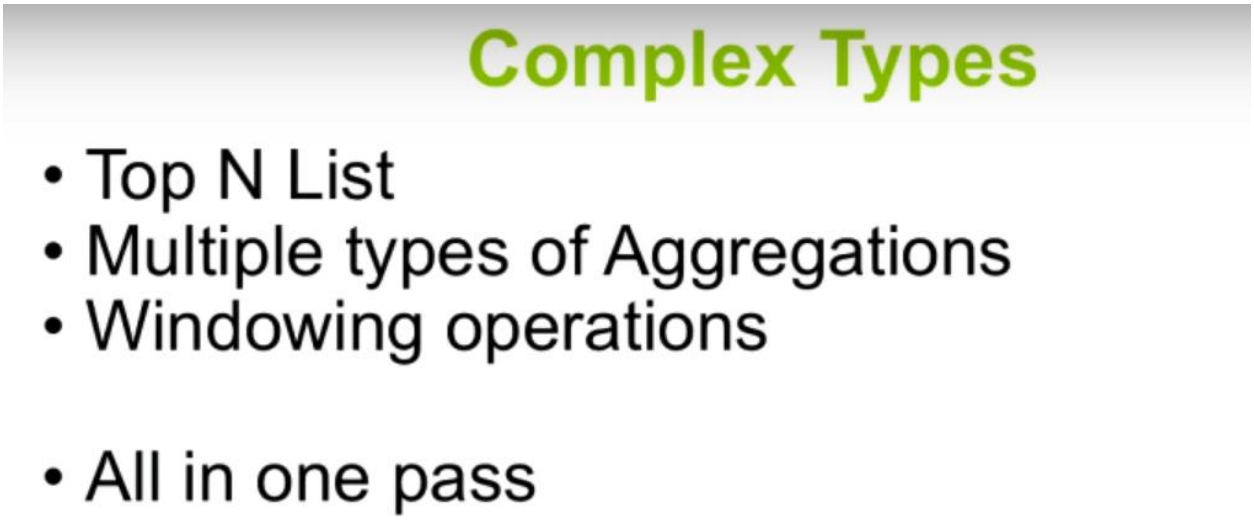
TreeReduce over Reduce

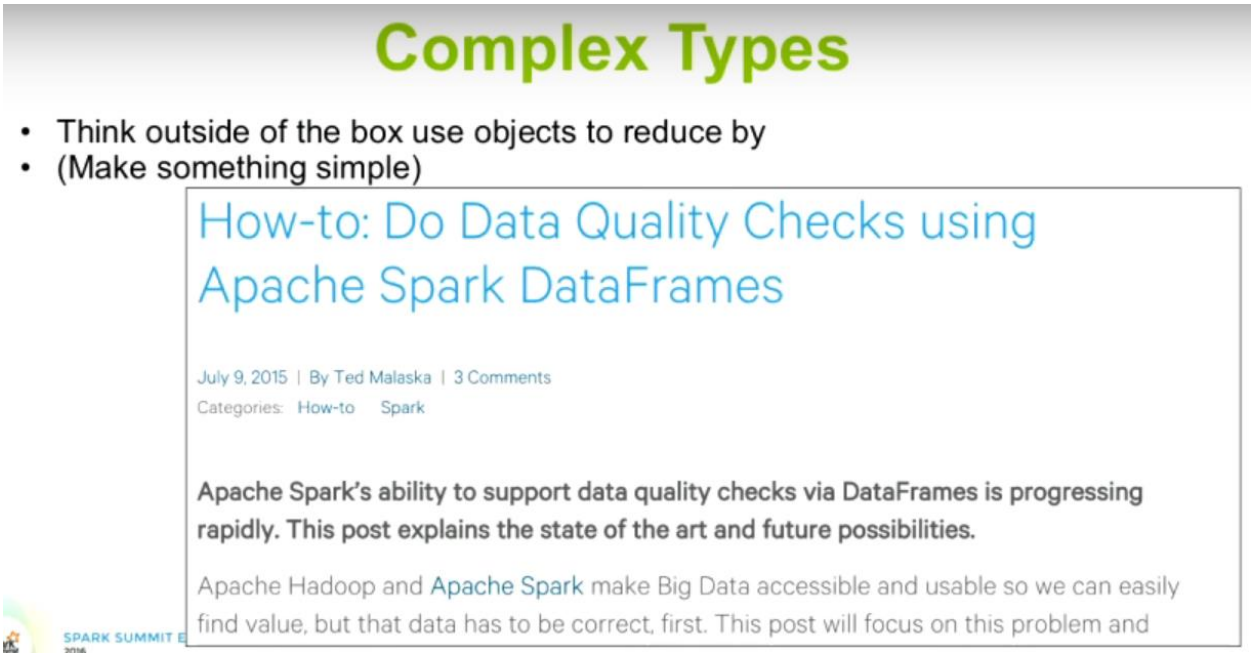
- `TreeReduce` & `Reduce` return some result to driver
- `TreeReduce` does more work on the executors
- While `Reduce` bring everything back to the driver





Complex Types





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!

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>
```

https://stackoverflow.com/questions/13620281/what-is-the-maven-shade-plugin-used-for-and-why-would-you-want-to-relocate-java

Future of shading

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

Summary: 5 mistakes

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

Working with Skewed Data: The Iterative Broadcast

Overview

Problem statement