

## Contents

|   |    |
|---|----|
| 10 Ways  Spark Performance Tuning   Apache Spark Tutorial.....  | 3  |
| <a href="https://www.youtube.com/watch?v=6zg7NTw-kTQ">https://www.youtube.com/watch?v=6zg7NTw-kTQ</a> ..... | 3  |
| TreeReduce Vs Reduce.....   | 3  |
| Use Broadcast Join wherever possible.....   | 4  |
| Use spark 2.x.....  | 4  |
| Use Right File format .....   | 4  |
| Right File compression .....  | 5  |
| Handled Skewed Data.....  | 5  |
| Salting.....  | 5  |
| Right Configurations .....  | 6  |
| Avoid unnecessary calculations .....  | 6  |
| Yarn configuration .....  | 6  |
| Use Vectorization.....  | 6  |
| Why Vectorization helps .....   | 6  |
| Use Bucketing .....   | 6  |
| Why Bucketing helps.....  | 7  |
| Spark Shuffle service.....  | 7  |
| AggreagateByKey Vs CombineByKey .....   | 8  |
| Catalyst Optimizer.....   | 8  |
| Spark Sql Joins.....  | 9  |
| Evolution of Spark.....   | 9  |
| Shuffle Hash Join.....  | 10 |
| Uneven sharing and Limited Parallelism.....   | 11 |
| More Performance considerations .....   | 12 |
| Broadcast Hash Join .....   | 15 |
| Cartesian Join .....  | 16 |
| One to Many Join .....  | 16 |
| Theta Join.....   | 17 |
| Working Skewed Data.....  | 17 |

|   |    |
|---|----|
| Top 5 Mistakes When Writing Spark Applications.....     | 17 |
| #Mistake 1.....   | 18 |
| Architecture recap .....                                | 19 |
| Yarn configuration.....                                 | 21 |
| #1 Memory overhead .....                                | 21 |
| #2 YARM AM needs a core: Client mode .....              | 22 |
| Spark cluster configuration calculation .....           | 24 |
| #2Mistake 2.....  | 26 |
| Define shuffle and Partitions .....                     | 27 |
| Repartition or coalesce .....                           | 28 |
| How many partitions?.....                               | 28 |
| IF Number of partitions > 2000 or < 2000.....           | 29 |
| Few tips to have optimum number of partitions?.....     | 30 |
| Mistake #3.....   | 30 |
| Slow jobs on join and shuffle .....                     | 30 |
| Mistake – Skew: Answers.....                            | 31 |
| Managing parallelism.....                               | 32 |
| Mistake – Skew : Salting: Two stage aggregation .....   | 33 |
| Mistake – Skew : Salting: Isolated Salting.....         | 33 |
| Mistake – Skew: Isolated Map Join.....                  | 34 |
| Managing parallelism: Cartesian join.....               | 34 |
| Repartitioning .....                                    | 35 |
| Mistake #4:.....  | 36 |
| Mistake – DAG Management.....                           | 36 |
| Shuffles are to be avoided .....                        | 36 |
| ReduceByKey over GroupByKey.....                        | 36 |
| TreeReduce over Reduce .....                            | 36 |
| Use Complex/NestedTypes .....                           | 36 |
| Complex Types .....                                     | 38 |
| Mistake #5.....   | 39 |
| Summary: 5 mistakes .....                               | 40 |
| Working with Skewed Data: The Iterative Broadcast ..... | 41 |

|                         |    |
|-------------------------|----|
| Overview .....          | 42 |
| Problem statement ..... | 42 |

# 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>

- AggregateByKey Vs CombineByKey

<https://www.youtube.com/watch?v=iCWAVLkGxdM>

- Catalyst Optimizer

<https://www.youtube.com/watch?v=fp53QhSfQcl&feature=youtu.be>

- Optimizing Apache Spark SQL Joins – Spark Summit

<https://www.youtube.com/watch?v=6zg7NTw-kTQ>

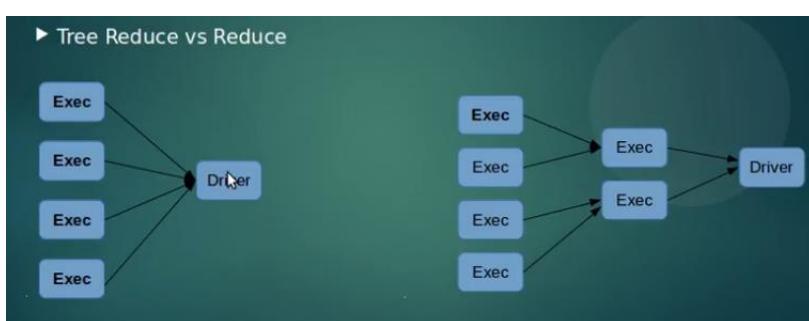
- Working with Skewed Data: The Iterative Broadcast - Rob Keevil & Fokko Driesprong – Spark Summit

## TreeReduce Vs Reduce

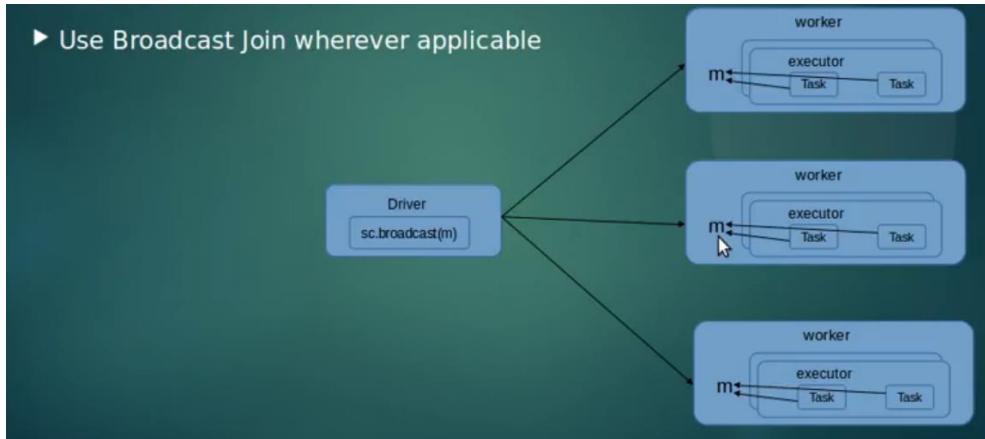
Associative and Commutative nature of data partitions

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 spark 2.x
  - Optimized code generation
  - Spark encoders
- ▶ If you are using older spark , use kryo serializer

## Use Right File format

- ▶ Use Right File Format
  - For Analysis kind of work Parquet Format
  - Other File Formats are ORC, RC
  - Avro is good when you want to read whole row
- ▶ Right File compression

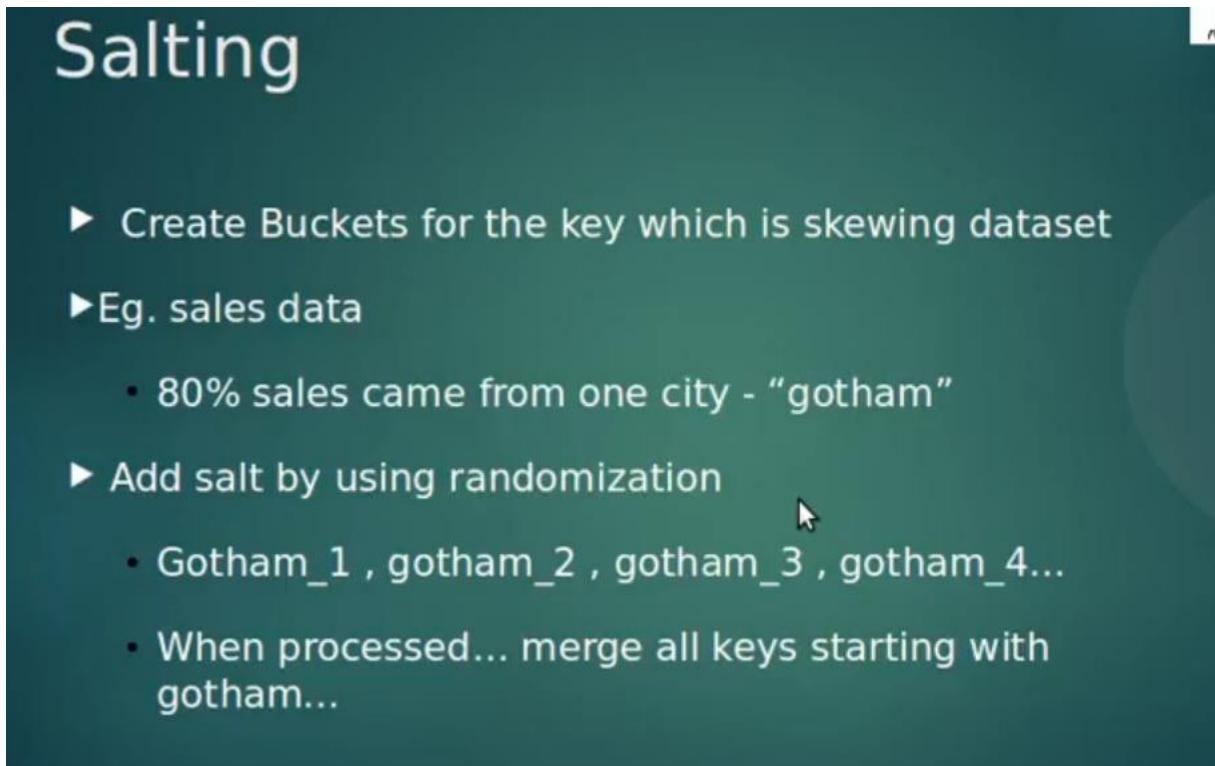
## Right File compression

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

- ▶ 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
- ▶ 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 Vectorization
  - Process Batch Of Records Instead of a row
- ▶ Why Vectorization Helps
  - No Object Inspections

## Use Bucketing

Bucket joins Avoid shuffle operations

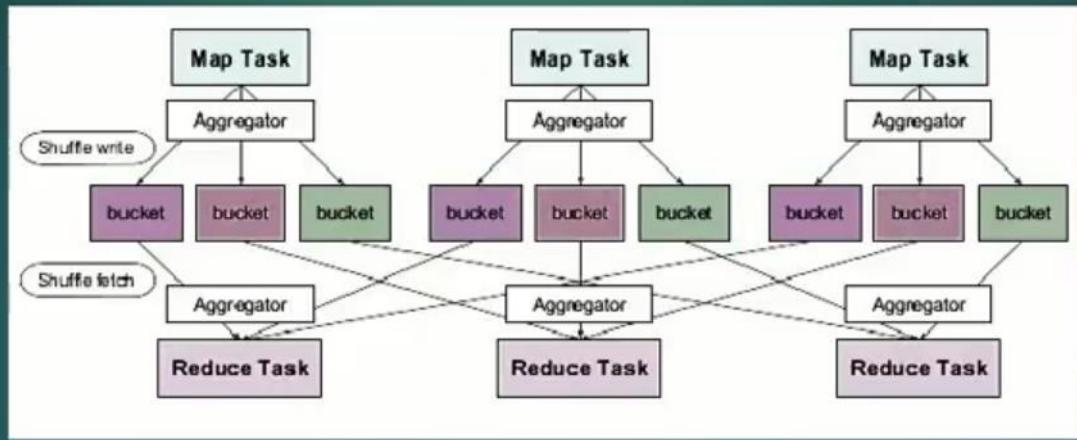
## Why Bucketing helps

Data is already shuffled and sorted

- ▶ Use Bucketing
  - Buckets Joins Avoid Shuffle operation
- ▶ Why Bucketing Helps
  - Data is already shuffled and sorted

## Spark Shuffle service

### ► Why we need Shuffle service?



### ► Shuffle Service in Yarn

- Yarn Shuffle service
- Yarn-site.xml
- Implementation of  
`org.apache.hadoop.yarn.server.api.AuxiliaryService`
- Authentication details should be given
- You can configure port for 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>

```

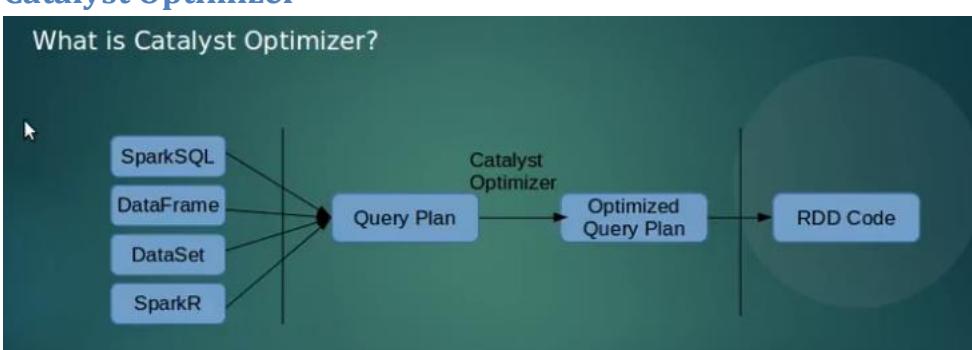
## ► Set Properties

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

## AggreagteByKey Vs CombineByKey

## Catalyst Optimizer

What is 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

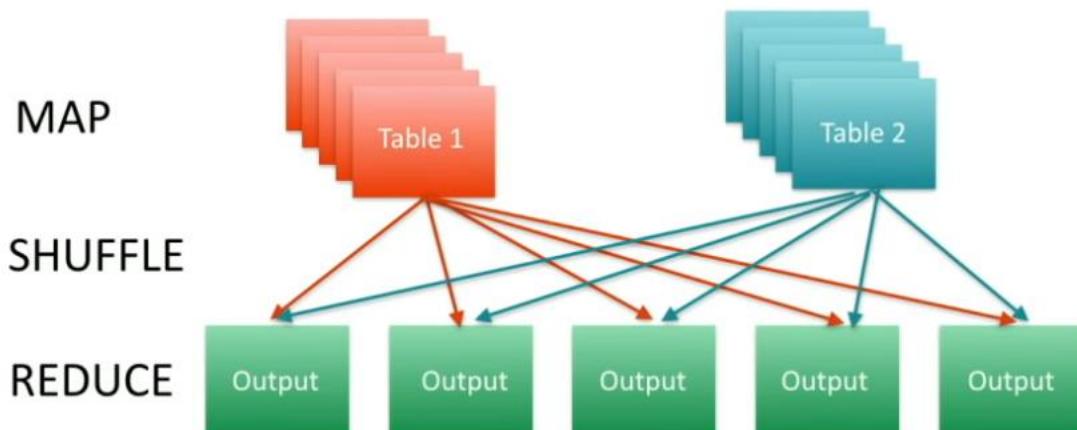
### Shuffle Hash Join

## 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



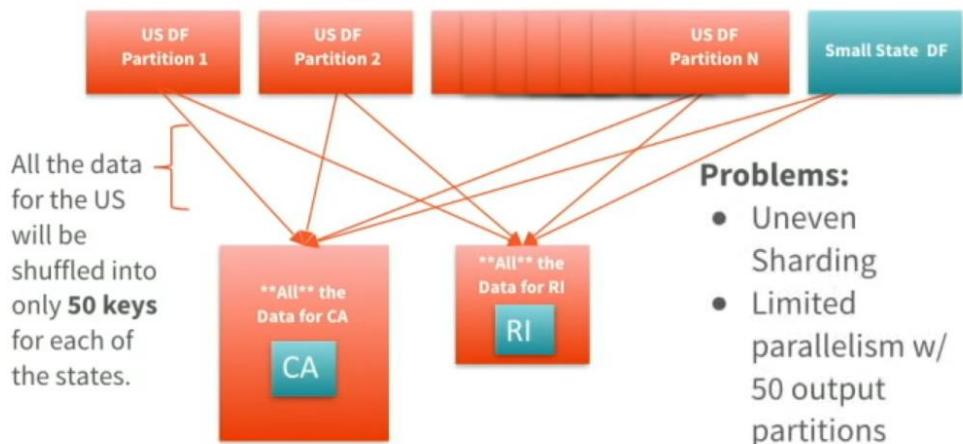
# Shuffle Hash Join Performance

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")
```

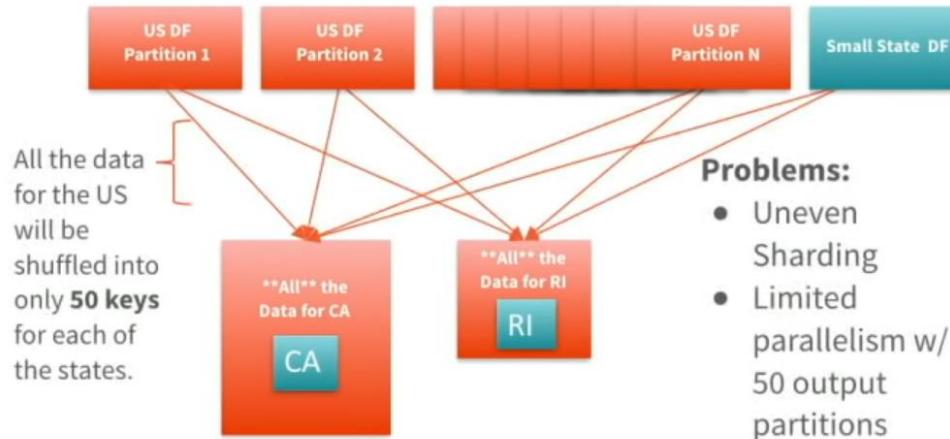
## Uneven Sharding & Limited Parallelism,



**A larger Spark Cluster will not solve these problems!**

Uneven sharing and Limited Parallelism

## Uneven Sharding & Limited Parallelism,



**Broadcast Hash Join can address this problem if one DF is small enough to fit in memory.**

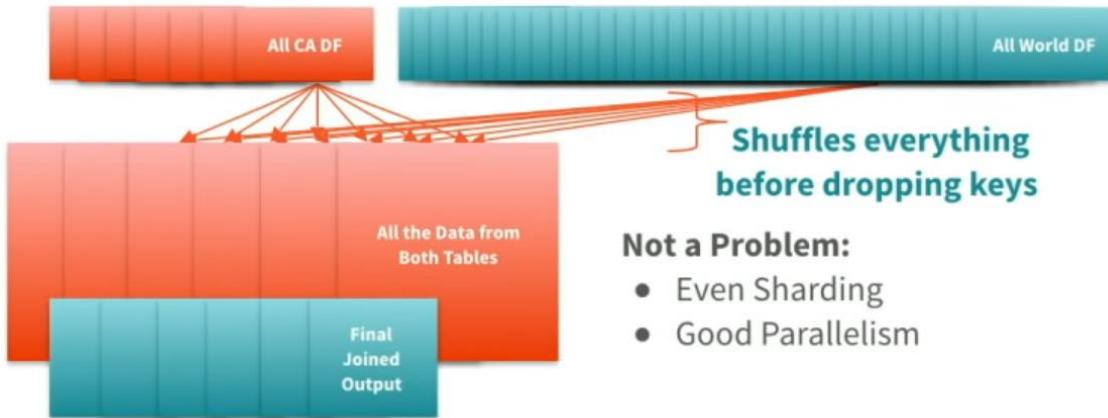
### More Performance considerations

## More Performance Considerations

```
join_rdd = sqlContext.sql("select *
    FROM people_in_california
    LEFT JOIN all_the_people_in_the_world
    ON people_in_california.id =
    all_the_people_in_the_world.id")
```

**Final output keys = # of people in CA, so don't need a huge Spark cluster, right?**

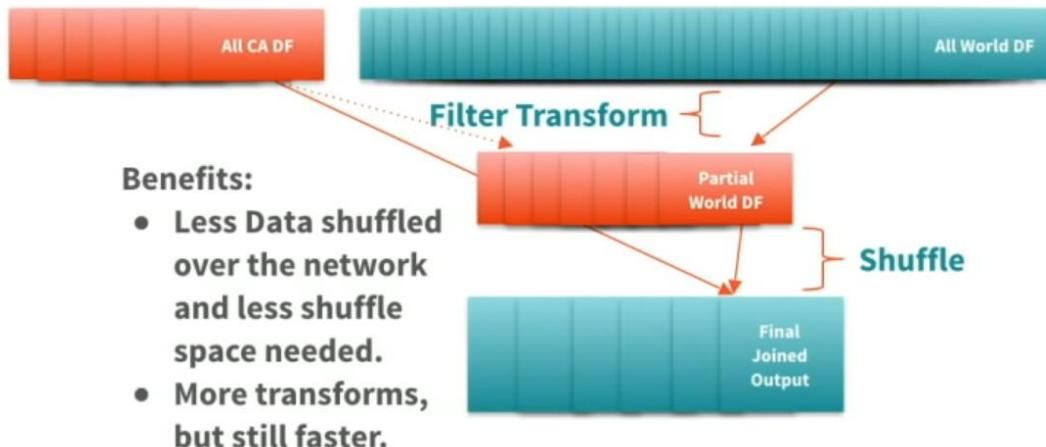
# Left Join - Shuffle Step



**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 its unique properties in order to best optimize your Spark Job.**

## In Practice: Detecting Shuffle Problems

| Index | ID | Attempt | Status    | LocalityLevel | Processor ID | Host                         | Launched   | Start | End  | Duration | Time | Task | Task ID | Task Substitution ID | Block ID | Block Substitution ID | Block Duration | Block Time | Input | Write | Shuffle | Errors |
|-------|----|---------|-----------|---------------|--------------|------------------------------|------------|-------|------|----------|------|------|---------|----------------------|----------|-----------------------|----------------|------------|-------|-------|---------|--------|
| 0     | 25 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.�行 | 2019/03/08 | 9:4   | 10ms | 0ms      | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 1     | 24 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 1ms      | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 2     | 26 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 3     | 25 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 4     | 26 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 5     | 25 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 6     | 27 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 7     | 26 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 8     | 25 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 9     | 26 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 10    | 26 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |
| 11    | 25 | 0       | SUCCEEDED | PROCESS_LOCAL | 3            | ip-10-0-107-104.us-west-1.执行 | 2019/03/08 | 9:4   | 10ms | 10ms     | 0ms  | 0ms  | 0ms     | 0ms                  | 0ms      | 0ms                   | 0ms            | 0ms        | 0ms   | 0ms   | 0ms     |        |

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

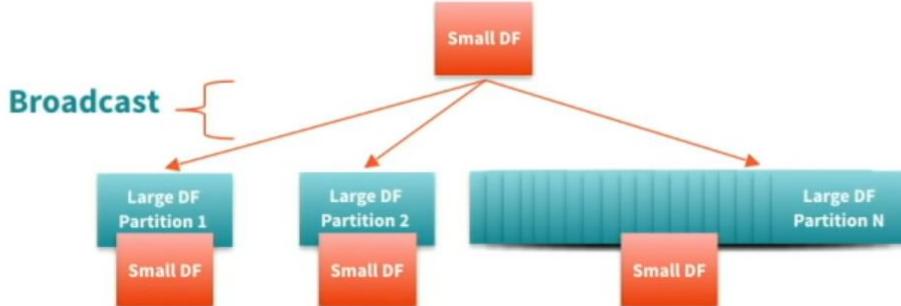
### Things to Look for:

- 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 has 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.  
$$100,000 \times 100,000 = 10 \text{ Billion}$$
- 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

# Theta Join

```
join_rdd = sqlContext.sql("select *
    FROM tableA
    JOIN tableB
    ON (keyA < keyB + 10)")
```

- Spark SQL consider each keyA against each keyB in the example above and loop to see if the theta condition is met.
- Better Solution - create buckets for keyA and KeyB can be matched on.

# Working Skewed Data

The Iterative Broadcast

## About Fokko Driesprong

- Software Engineering and Distributed Systems
- Data Engineer at GoDataDriven
- Committer & PPMC Member, Apache Airflow

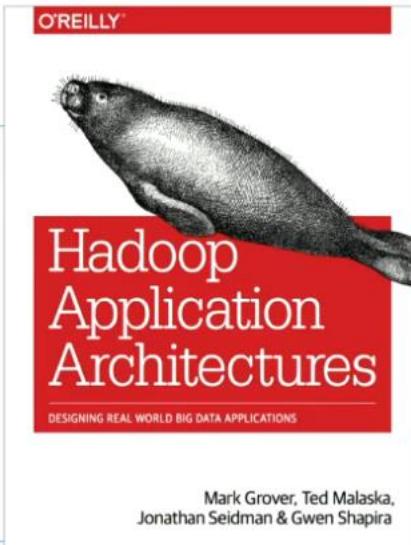


GoDataDriven  
proudly part of Xebia Group

# 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

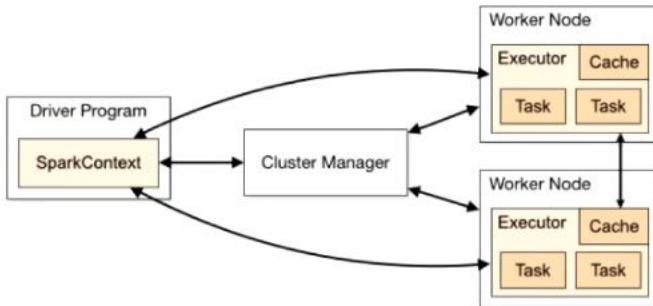
- 6 nodes
- 16 cores each
- 64 GB of RAM



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

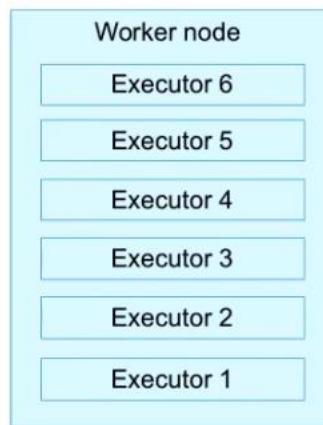
## Architecture recap

# 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

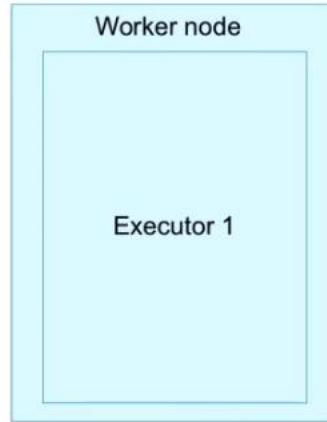


## Why?

- 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

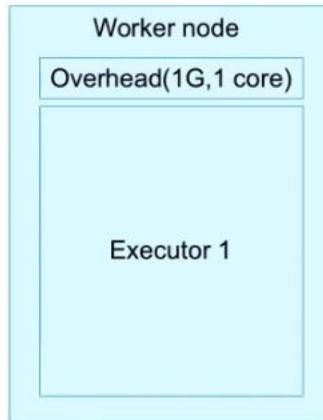


## Why?

- 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

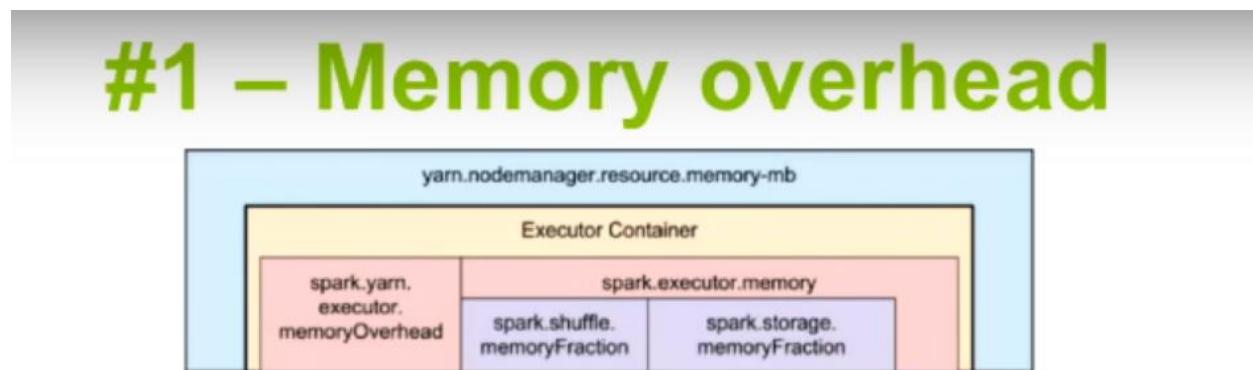
### Let's assume...

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

### 3 things

- 3 other things to keep in mind

#### #1 Memory overhead

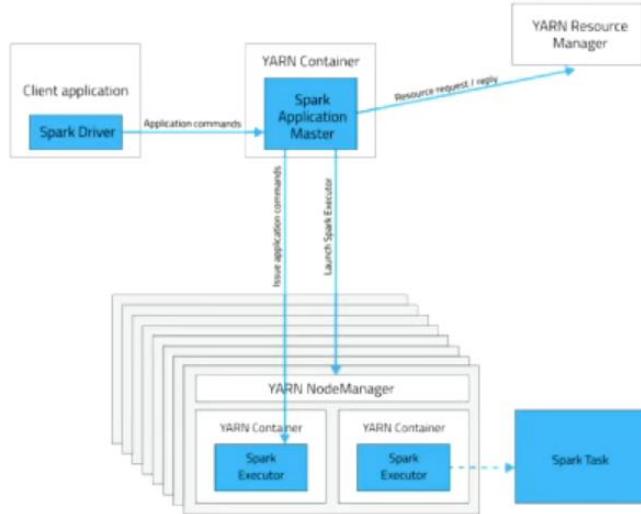


- `--executor-memory` controls the heap size
- Need some overhead (controlled by `spark.yarn.executor.memory.overhead`) for off heap memory
  - Default is  $\max(384\text{MB}, .07 * \text{spark.executor.memory})$

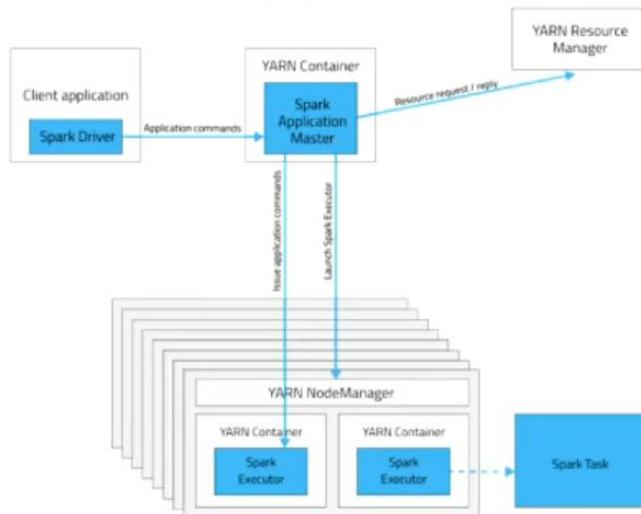
## #2 YARN AM needs a core: Client mode

MISTAKES When Writing Spark Applications

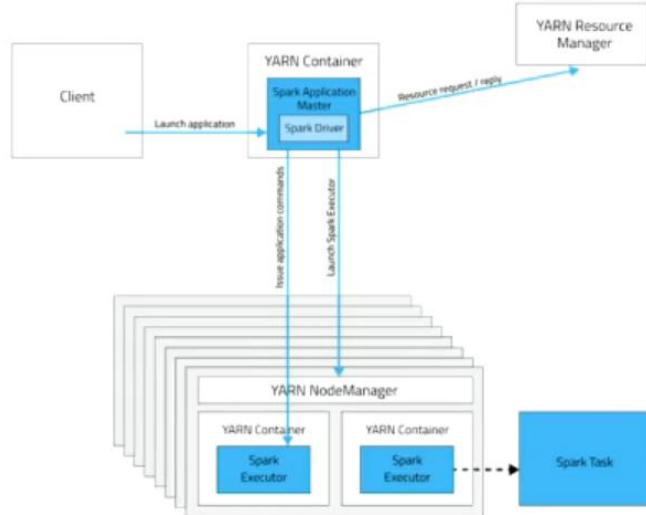
## #2 - YARN AM needs a core: Client mode



## #2 - YARN AM needs a core: Client mode



## #2 YARN AM needs a core: Cluster mode



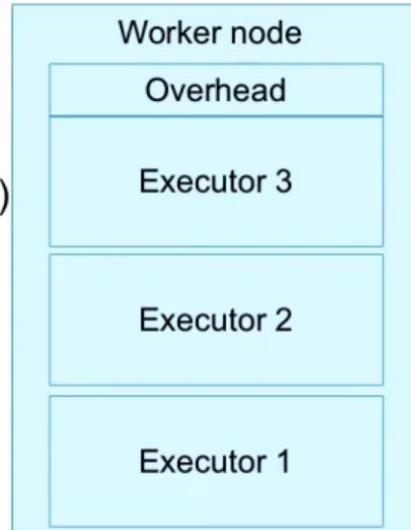
## #3 HDFS Throughput

- 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

# Calculations

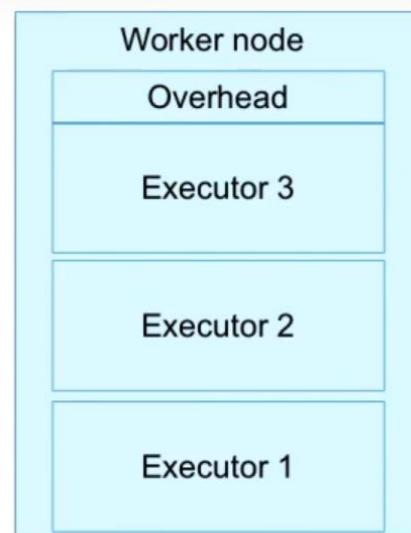
- 5 cores per executor
  - For max HDFS throughput
- Cluster has  $6 * 15 = 90$  cores in total  
after taking out Hadoop/Yarn daemon cores)
- $90 \text{ cores} / 5 \text{ cores/executor}$   
 $= 18 \text{ executors}$
- Each node has 3 executors
- $63 \text{ GB}/3 = 21 \text{ GB}, 21 \times (1-0.07)$   
 $\sim 19 \text{ GB}$
- 1 executor for AM => 17 executors



# Correct answer

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

## #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?

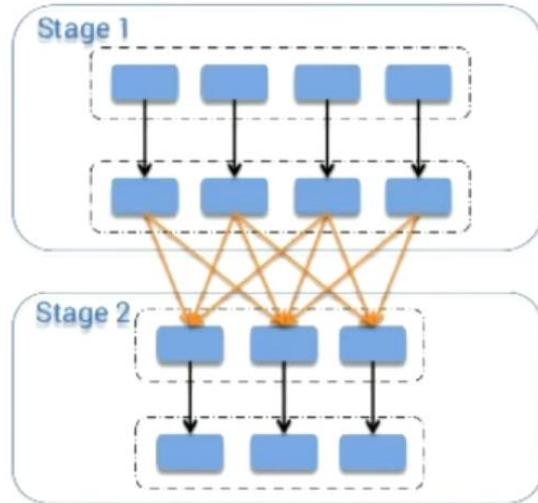
- No Spark shuffle block can be greater than 2 GB

## 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 use 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

The Holy Grail of Distributed Systems



## Mistake - Skew

What about Skew, because that is a thing



Mistake - Skew: Answers

## Mistake – Skew : Answers

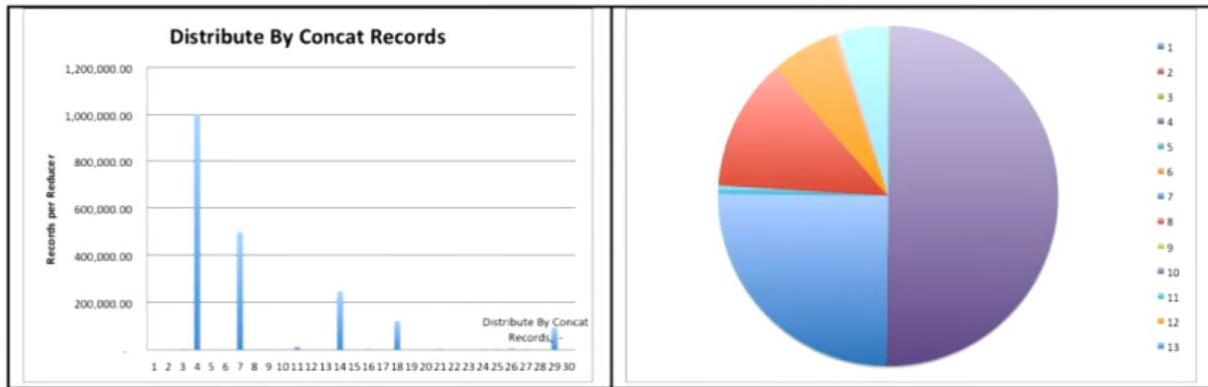
- Salting
- Isolated Salting
- Isolated Map Joins

# Mistake – Skew : Salting

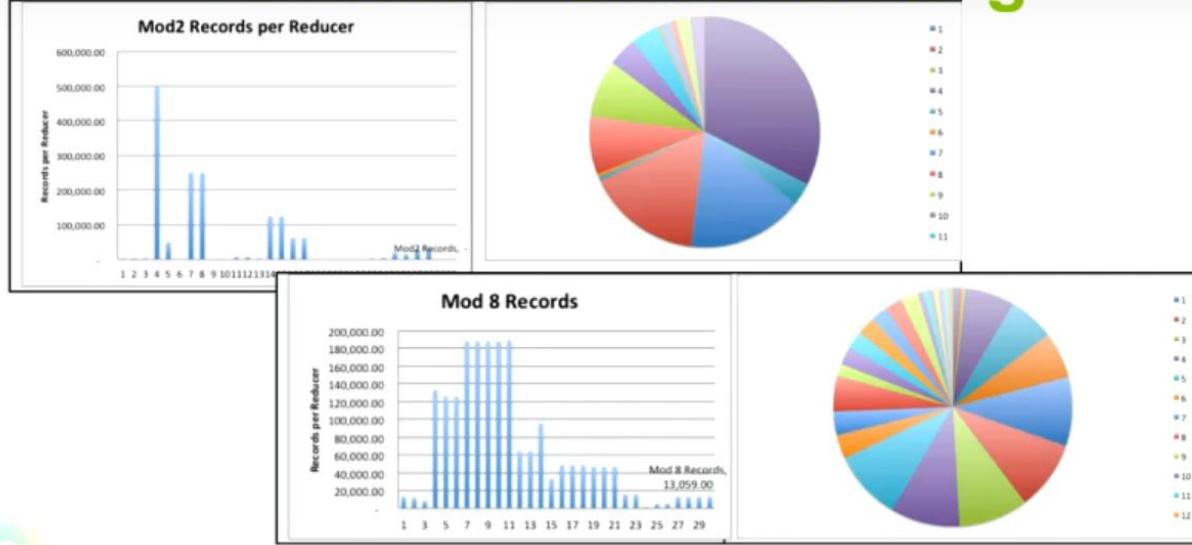
- Normal Key: “Foo”
- Salted Key: “Foo” + random.nextInt(saltFactor)

Managing parallelism

## Managing Parallelism



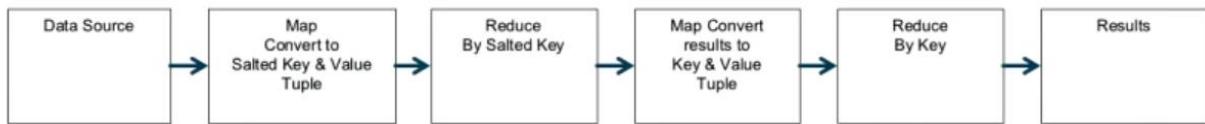
## Mistake – Skew: Salting



Mistake – Skew : Salting: Two stage aggregation

## Mistake – Skew : Salting

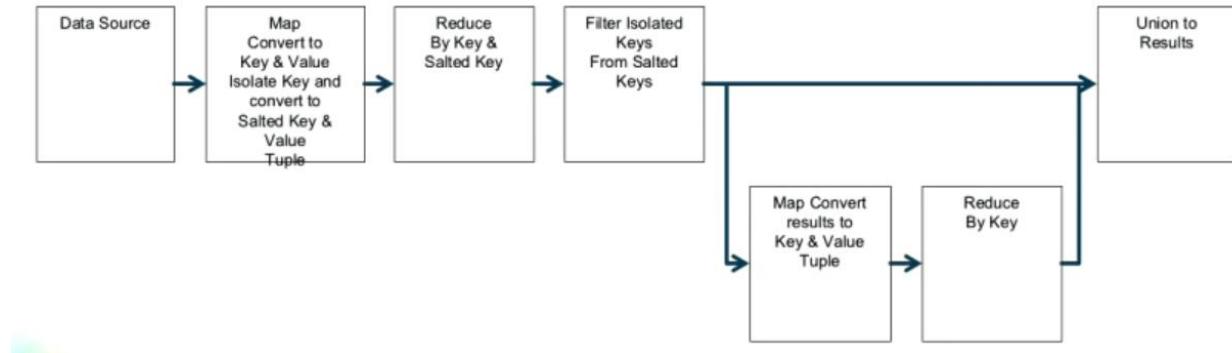
- Two Stage Aggregation
  - Stage one to do operations on the salted keys
  - Stage two to do operation access unsalted key results



Mistake – Skew : Salting: Isolated Salting

## Mistake – Skew : Isolated Salting

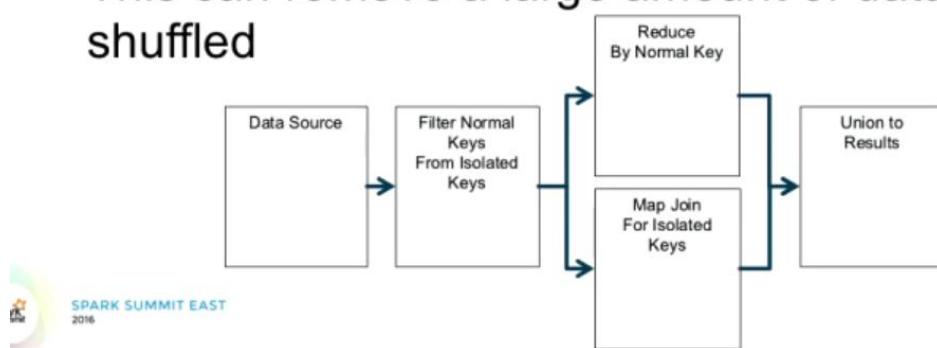
- Second Stage only required for Isolated Keys



### Mistake – Skew: Isolated Map Join

## Mistake – Skew : Isolated Map Join

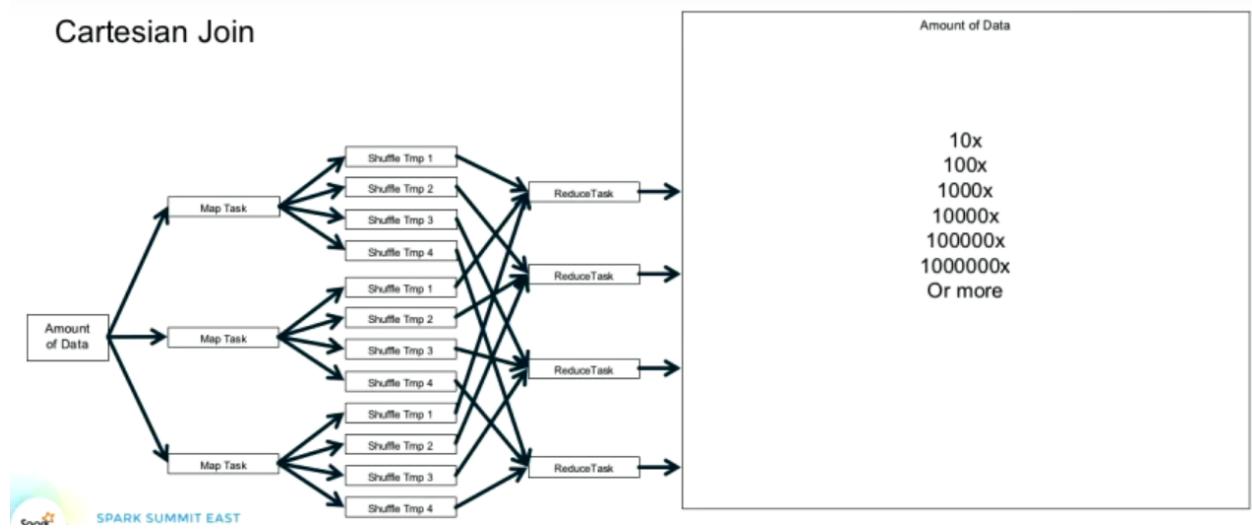
- Filter Out Isolated Keys and use Map Join/Aggregate on those
- And normal reduce on the rest of the data
- This can remove a large amount of data being shuffled



### Managing parallelism: Cartesian join

## 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>>
```

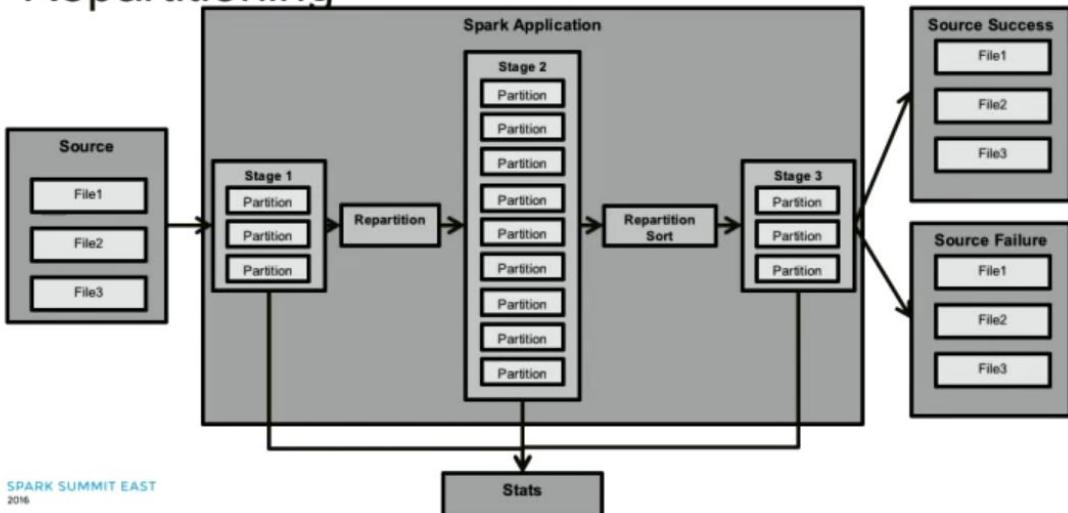
=

```
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

SPARK SUMMIT EAST  
2016

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/NestedTypes

## Mistake – DAG Management

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

### Mistake – DAG Management: Shuffles

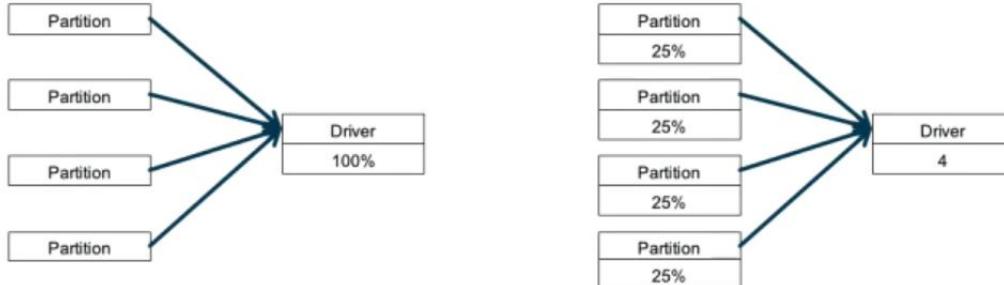
- Map Side reduction, where possible
- Think about partitioning/bucketing ahead of time
- Do as much as possible with a single shuffle
- Only send what you have to send
- Avoid Skew and Cartesians

## 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

# 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



**Mistake #5**

## Ever seen this?

```
Exception in thread "main" java.lang.NoSuchMethodError:  
com.google.common.hash.HashFunction.hashInt()Lcom/google/common/hash/HashCode;  
    at org.apache.spark.util.collection.OpenHashSet.org  
$apache$spark$util$collection$OpenHashSet$$hashcode(OpenHashSet.scala:261)  
    at  
org.apache.spark.util.collection.OpenHashSet$mcl$sp.getPos$mcl$sp(OpenHashSet.scala:165)  
    at  
org.apache.spark.util.collection.OpenHashSet$mcl$sp.contains$mcl$sp(OpenHashSet.scala:102)  
    at  
org.apache.spark.util.SizeEstimator$$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

# THANK YOU.

[tiny.cloudera.com/spark-mistakes](http://tiny.cloudera.com/spark-mistakes)

Mark Grover | @mark\_grover

Ted Malaska | @TedMalaska

## Working with Skewed Data: The Iterative Broadcast

### Working with Skewed Data: The Iterative Broadcast



Rob Keevil - KnowIT

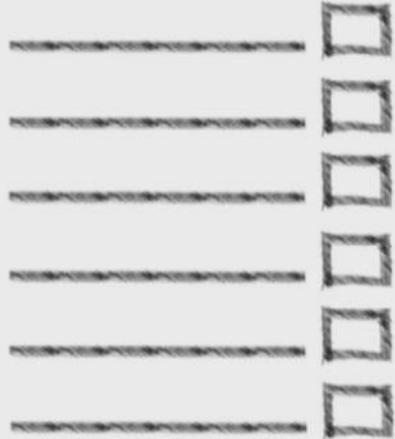
Fokko Driesprong - GoDataDriven

#EUde11

## Overview

### Overview

- The Skew Problem at ING
- Join Strategies in Spark
- Our Solution
- Performance Analysis
- Considerations
- Future Work



## Problem statement

### The Skew Problem at



- Billions of transactions...
- Billions of accounts...
- Millions of companies...
- ...with a very uneven distribution!

