Distributed Computing Mastering Parallelism

By Ted Malaska

High Level Section Overview

- Section 1: Background and Fundamentals
- Section 2: Debugging & Partitioning
- Section 3: Advanced

Overview: Section 1

- Section 1
  - Brief History of distributed computing engines
  - Components of a modern execution engine
  - DAG Management
  - Functional programming and distributed computing
  - Transformations And Actions
  - Break and Q/A

Overview: Section 2

- Section 2
  - Collections
  - Partitioning
  - Joins & Shuffles
  - Examples of running the code locally for debugging
  - Break/Q&A

Overview: Section 3

- Section 3
  - Skew
  - Windowing
  - Big and Small
  - Think Big
  - Break Q/A

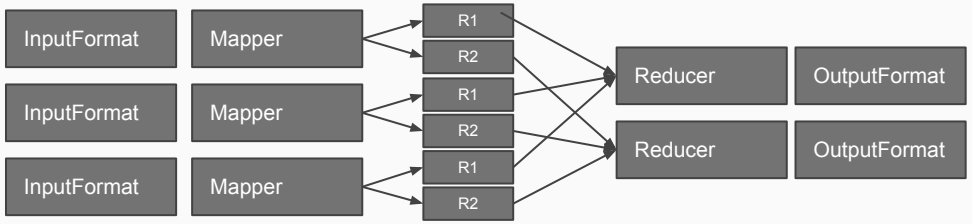
Brief History of Distributed Computing Engines

In the beginning there was MapReduce

- Google File System Paper 2003
  - By Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
- MapReduce - Paper 2004
  - By Jeffrey Dean and Sanjay Ghemawat

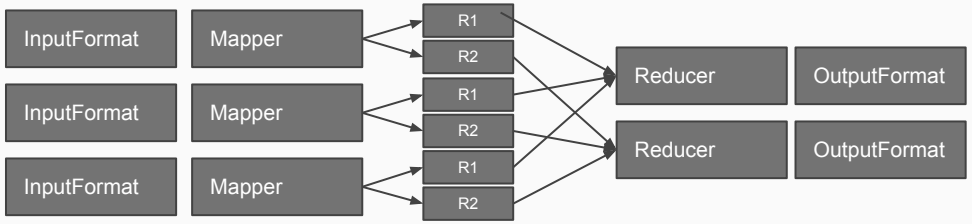
In the beginning there was MapReduce

- Major parts of MapReduce
  - InputFormat
    - Splits/Record Reader
  - Mapper
  - Combinner
  - Shuffle
  - Reducer
  - OutputFormat



In the beginning there was MapReduce

- Pros of MapReduce
  - Process Huge amounts of data
  - Read and write to anything
- Cons of MapReduce
  - Hard to code
  - Hard to chain
  - Hard to debug
  - Long startup times
  - Very IO heavy



Many Efforts to Improve MapReduce

- SQL
  - Hive
- Coding Styles
  - Cascading
  - Pig
  - Crunch

Then Came Spark

- Spark: Cluster Computing with Working Sets Paper 2010
  - By Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
- Addresses
  - Coding style
  - Chaining
  - Startup times
  - Debugging
  - IO

Beyond Spark

- Flink
- Impala
- Presto
- ...

We will focus on Spark

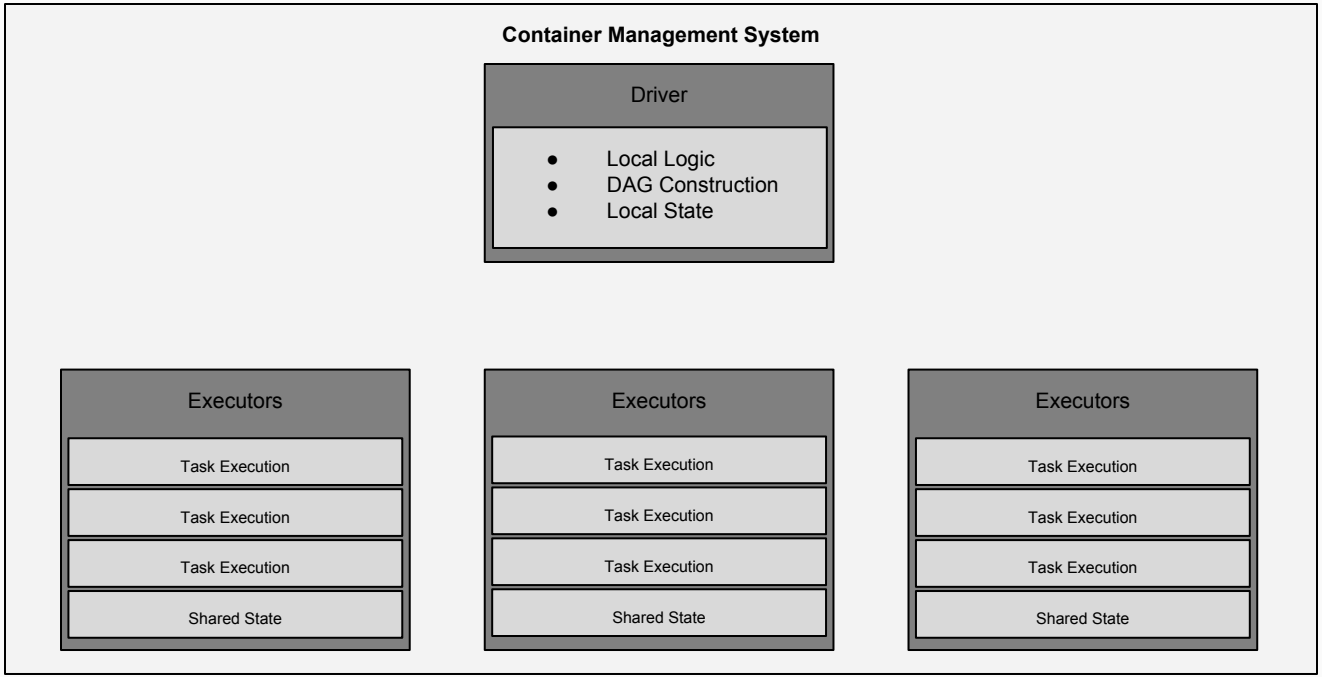
- Mainly because
  - It has SQL but it is more
  - It is a great teaching platform
  - It is the most popular distributed execution system by committers

Components of a Modern Execution Engine

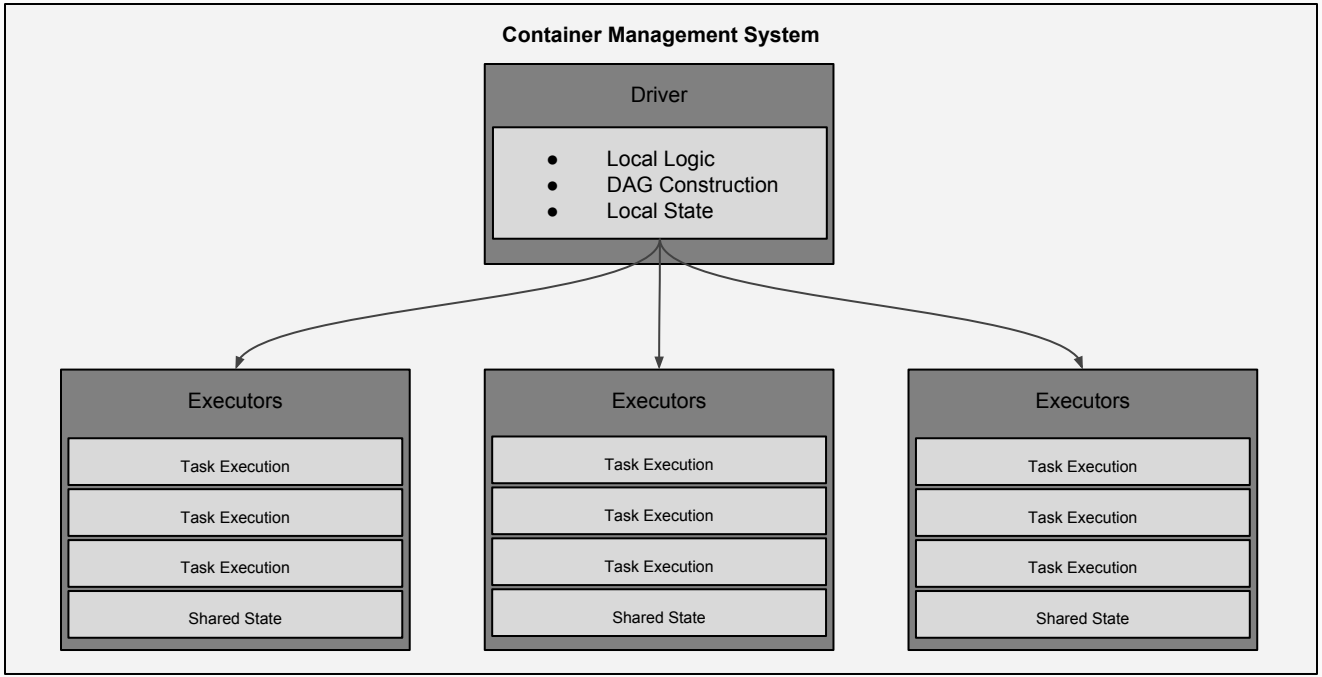
Spark

- Parts of Spark
- Communication between parts
- Parallelism
- Compared to MapReduce

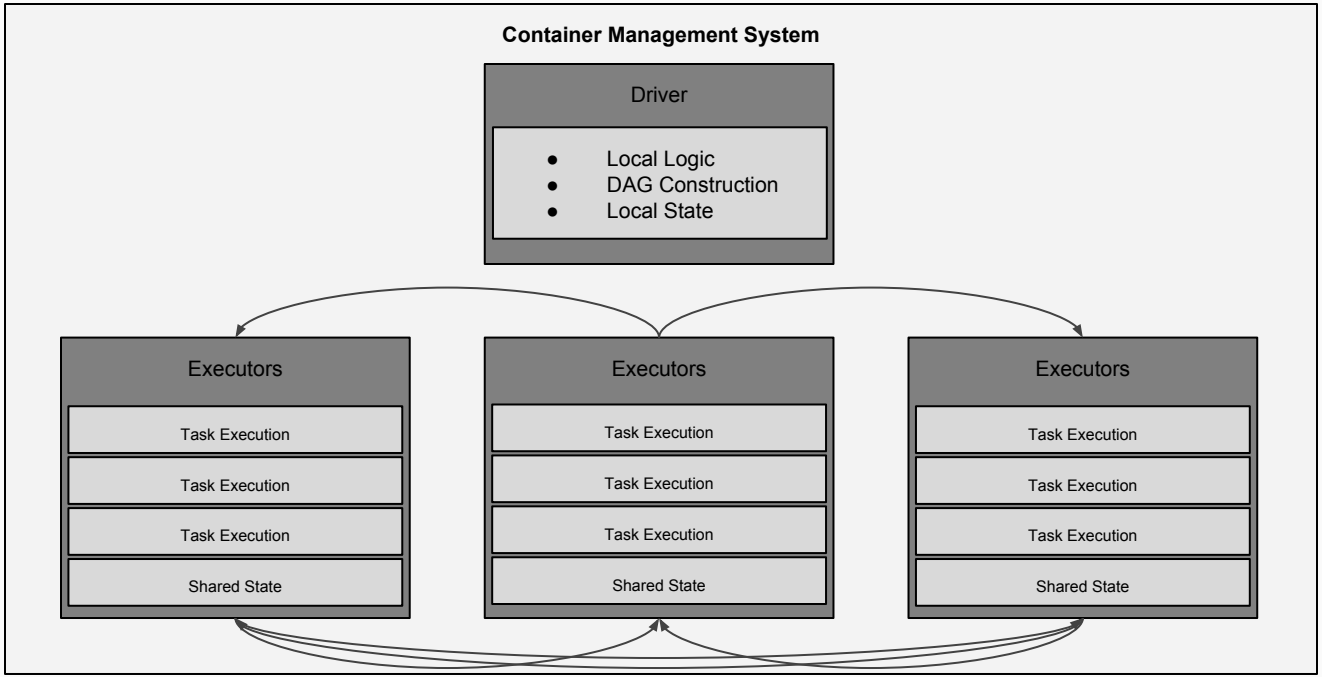
Parts of Spark



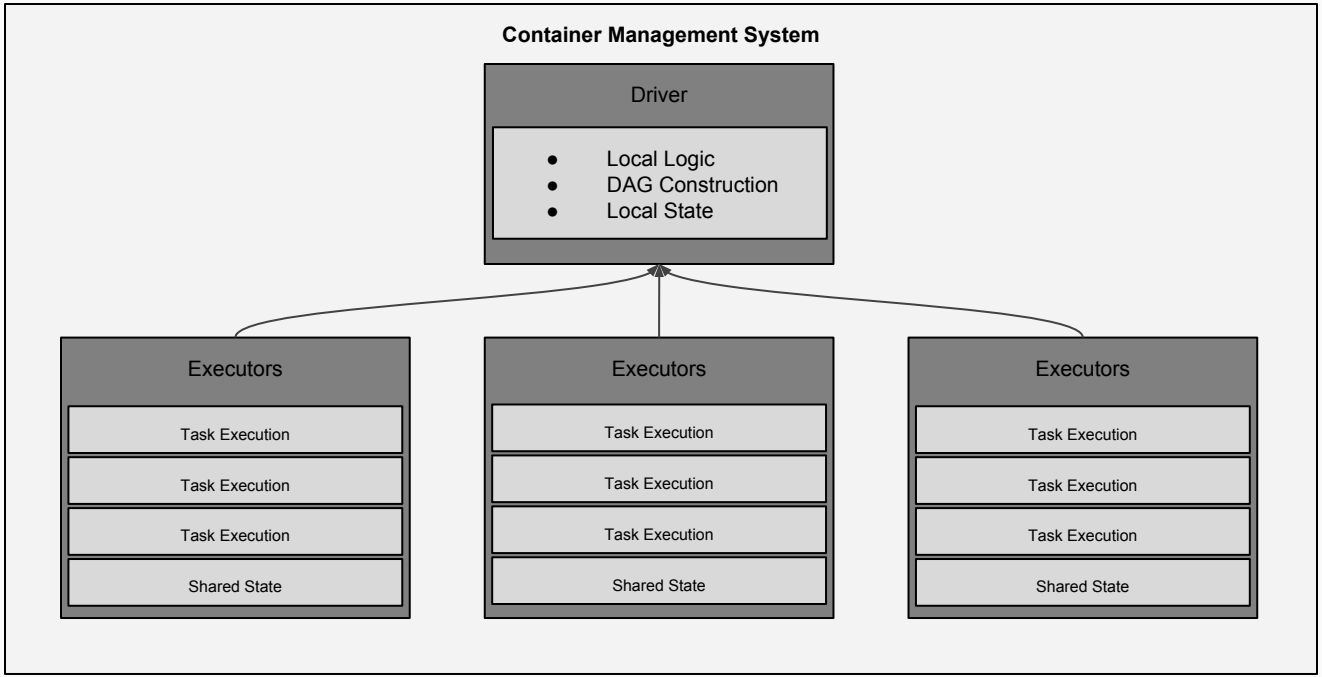
Communication: Broadcast



Communication: Shuffle or ~Repartition

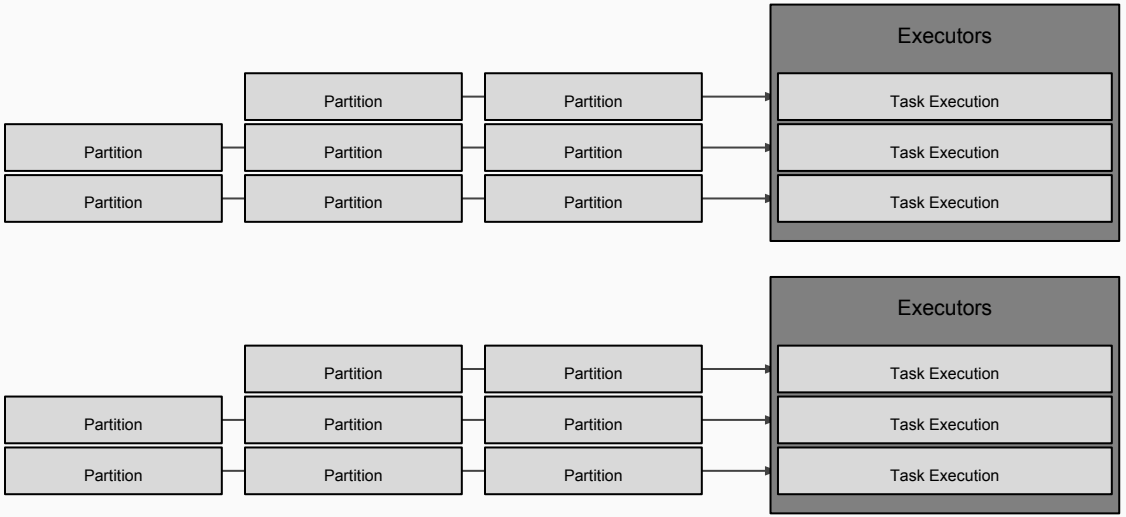


Communication: Take or Collect



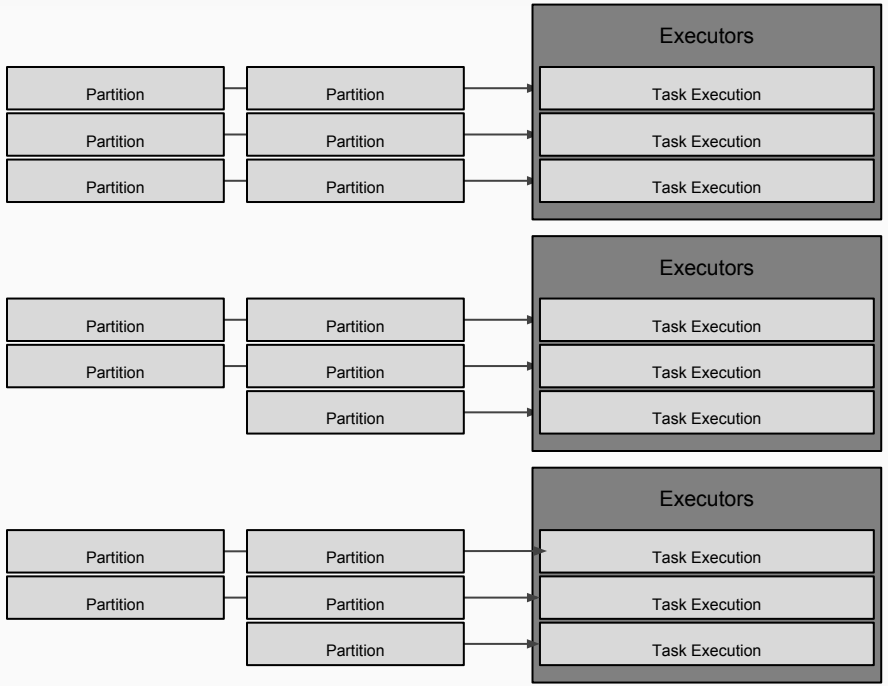
Parallelism

- 10x Partitions
- 2x Executors
- 3x Cores each



Parallelism - Dynamic Allocation

- 16x Partitions
- 2x Executors
- 3x Cores each
- An addition Executor added

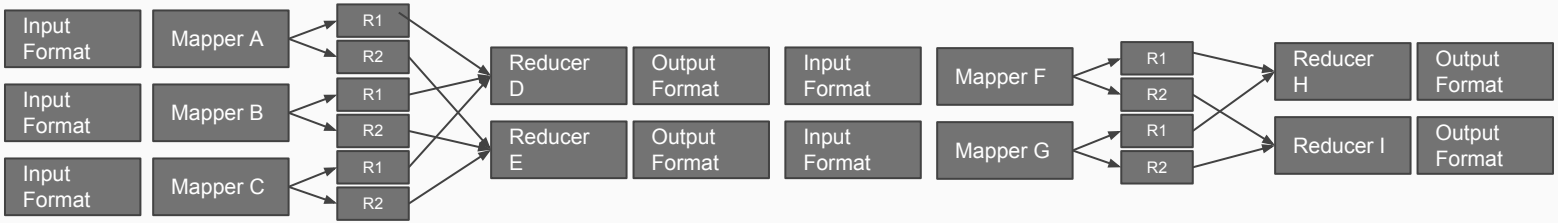


The Dream

The dream is more CPU faster results

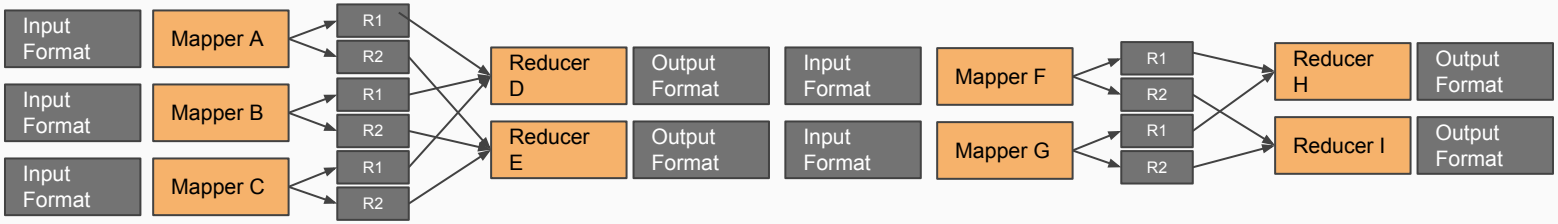
Comparing to MapReduce

- MapReduce Example: A join and a group by



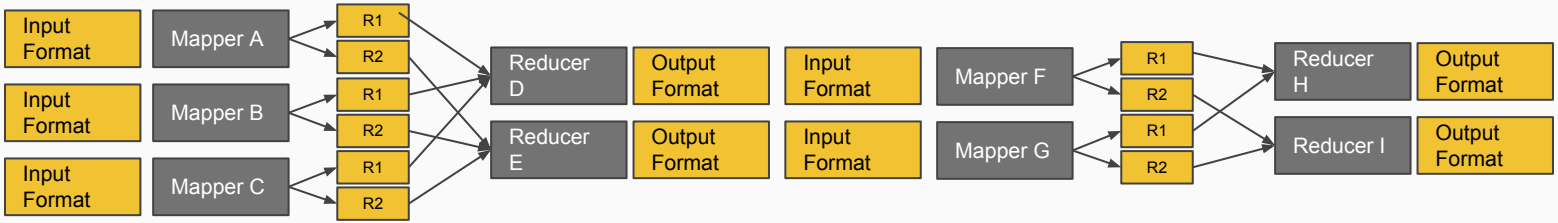
Comparing to MapReduce: JVM Starts

- MapReduce Example: A join and a group by



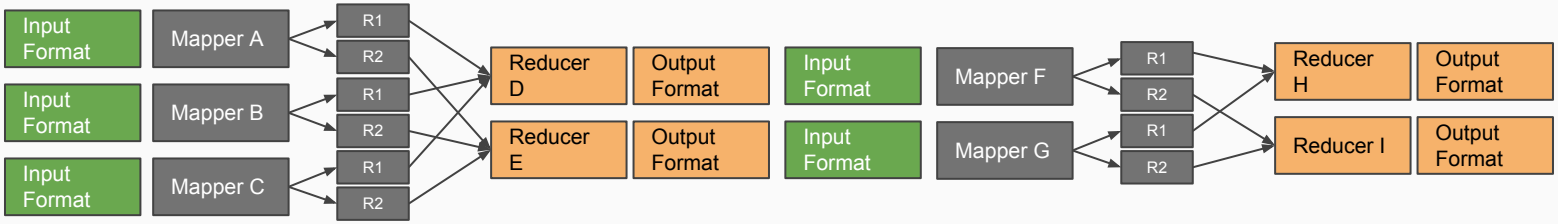
Comparing to MapReduce: Hitting Disk

- MapReduce Example: A join and a group by



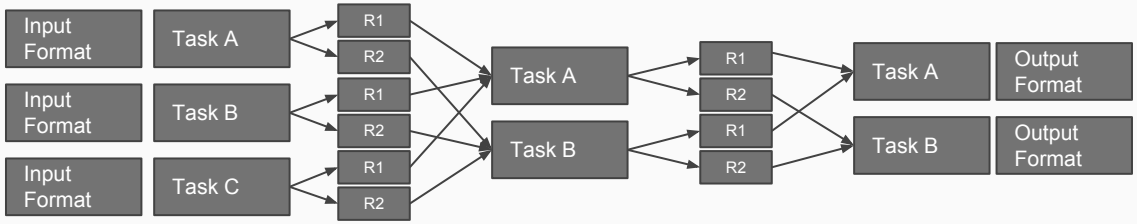
Comparing to MapReduce: Network

- MapReduce Example: A join and a group by



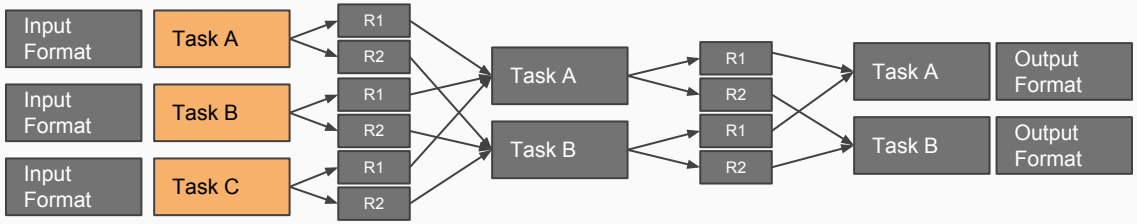
Comparing to MapReduce

- Spark Example: A join and a group by



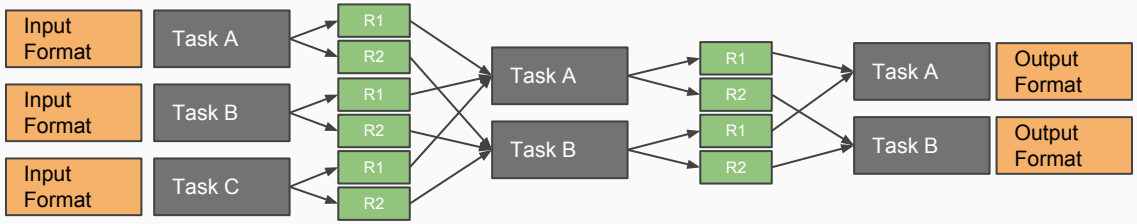
Comparing to MapReduce: JVM Starts

- Spark Example: A join and a group by



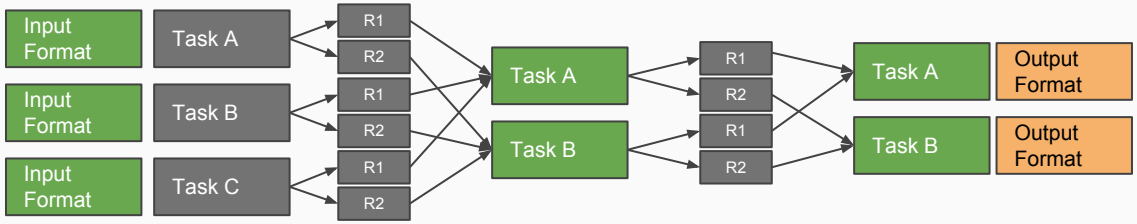
Comparing to MapReduce: Hitting Disk

- Spark Example: A join and a group by



Comparing to MapReduce: Network

- Spark Example: A join and a group by



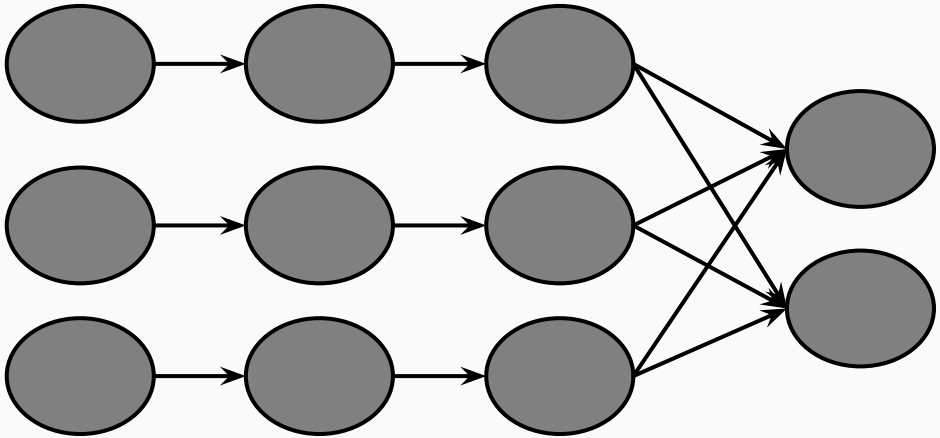
Spark

- Parts of Spark
- Communication between parts
- Parallelism
- Compared to MapReduce

DAG Management

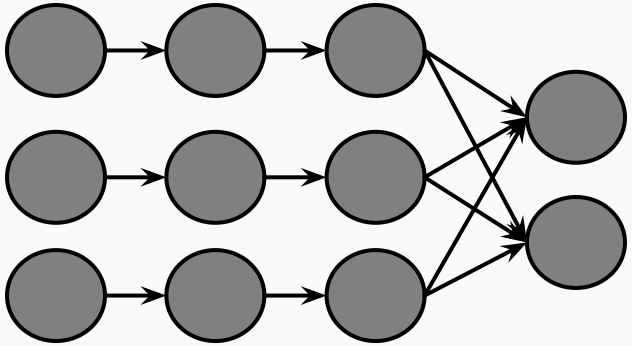
What is a DAG?

- Directed Acyclic Graph

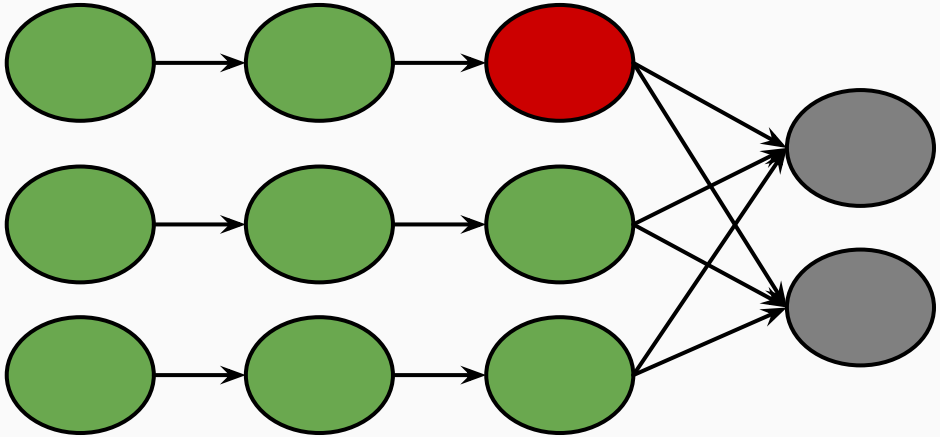


What’s different now

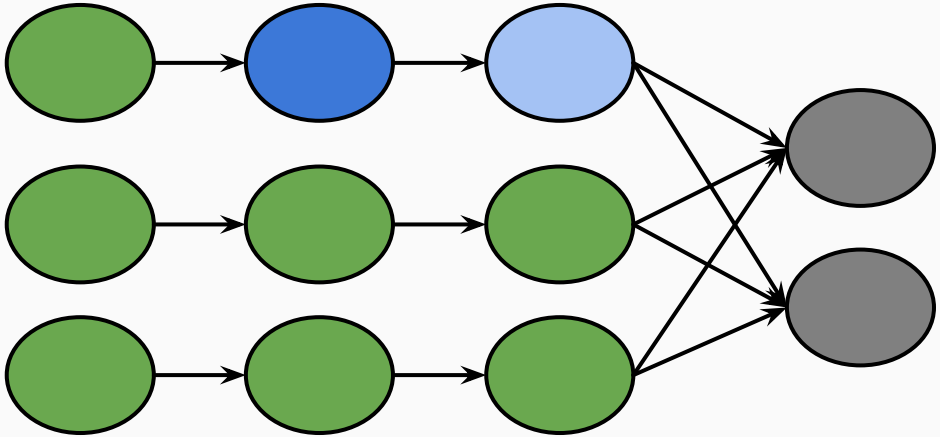
- Before DAG was managed externally
  - Hive
  - Ozzie
  - Pig, Cascading
- Now DAG is managed within the engine
  - Advanced failure recover
  - Advanced optimizations
  - Reuse of resources
  - Easier to debug
  - And many more



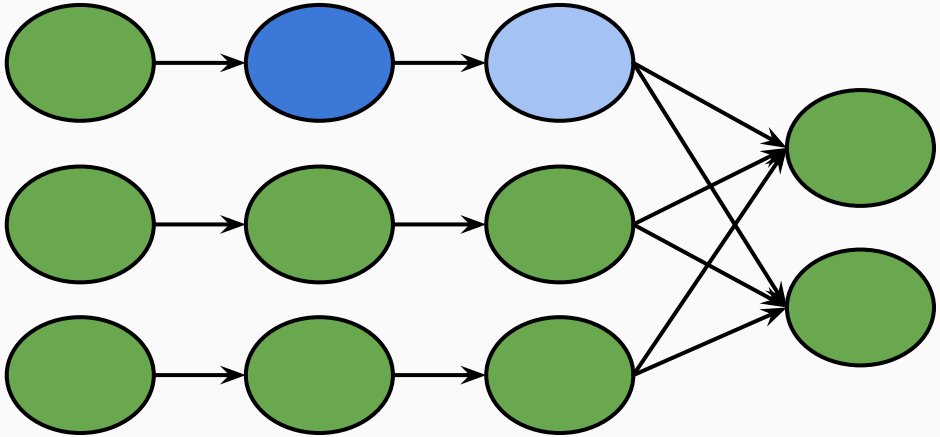
DAG and Failure Recovery



DAG and Failure Recovery



DAG and Failure Recovery



Things to think about with your DAG

- Bigger is not better
- Don’t shuffle over the same key more than once
- Think about the big and the small
- Right size partition counts
- SQL vs not SQL
- Understand you joins and repartitions

DAG: Bigger is not better

- Maps, Filters, FlapMap do not add to the DAG complexity
- It’s the shuffle that you should fear
-

DAG: Don’t Shuffle over the same key

- Combine steps
- Don’t be afraid to use complex structures and keys
- Think about ordering as a way of grouping
- SQL may optimize this for you

DAG: Think about the big and the small

- What is the high level goals
- How much data do you need to do each part of the job
- Can a group by or a partition ordering strategy help

DAG: Right size partitions

- Many thing will be impacted by partition size
  - Read/Write performance
  - Shuffle spilling
  - CPU usage
- How partition needs may change on you
  - Joins
  - Filters
  - FlapMaps

DAG: SQL vs not SQL

- Single SQL will most likely be more optimized
- Multiple SQL can produce a all over unoptimized outcome
- Not all problems can be expressed with SQL well

Understand you joins and repartitions

- Broadcast Join
- Coalesce
- Partition Order
- Bucketed and Ordered Join
- Shuffle
- Shuffle and Sort Join

Understand you joins and repartitions

- Broadcast Join: Only if one side is small
- Coalesce: Only is you are down partitioning
- Partition Order: Only if you are ordering within a partition
- Bucketed and Ordered Join: Needs data to be prepped before read
- Shuffle
- Shuffle and Sort Join

Transformation & Actions

A DAG is made up of these

- Transformations: Instructions to be added to the DAG
- Action: A call for the DAG to execute to get results

A DAG is made up of these

- Transformations
  - Map
  - MapPartition
  - FlapMap
  - Fliter
  - Repartition
  - Coalesce
  - Orderby
  - GroupByByKey
  - Join
  - ReduceByKey

Transformations

- Map: A transformation
- MapPartition: A transformation
- FlapMap: One in and 0 to many out
- Fliter: One in and one or 0 out

Transformations

- Repartition: When you want to spend your partitions out over more partitions
- Coalesce: When you want to combine partitions
- Orderby: When you want to order with in a partition
- GroupByKey: When you want to group by a Key
- ReduceByKey: When you want to reduce many rows with the same key to one
- Join: Joining two datasets

GroupByKey vs ReduceByKey

- GroupByKey
  - Requires all records with same key to be in memory at one time
- ReduceByKey
  - Only requires two records to be in memory at one time

Actions

- Actions
  - SQL Execution
  - Collect
  - Take
  - Foreach*
  - Reduce
  - TreeReduce

Functional programming and distributed computing

Word Count

1. val textFile = sc.textFile("hdfs://...")
2. val words = textFile.flatMap(line => line.split(" "))
3. val wordPairCount = words.map(word => (word, 1))
4. val counts = wordPairCount.reduceByKey(_ + _)
5. val localCollection = counts.collect
6. localCollection.foreach(println)

Adding some Color

1. Transfermations
2. Actions
3. Normal Scala to run on driver
4. Normal Scala to run on Executor
5. Distributed state
6. Local driver state

Word Count with Color

```
1. val textFile = sc.textFile("hdfs://...")
2. val words = textFile.flatMap(line => line.split(" "))
3. val wordPairCount = words.map(word => (word, 1))
4. val counts = wordPairCount.reduceByKey(_ + _)
5. val localCollection = counts.collect
6. localCollection.foreach(println)
```

Word Count with Color (Java)

```
1.  JavaRDD<String> textFile = sc.textFile("hdfs://...");
2.  JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {
3.    public Iterator<String> call(String s) { return Arrays.asList(s.split(" ")).iterator(); }
4.  });
5.  JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() {
6.    public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s, 1); }
7.  });
8.  JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
9.    public Integer call(Integer a, Integer b) { return a + b; }
10. });
11. counts.saveAsTextFile("hdfs://...");
```

Word Count with Color (Java)

```
1.  JavaRDD<String> textFile = sc.textFile("hdfs://...");
2.  JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {
3.    public Iterator<String> call(String s) { return Arrays.asList(s.split(" ")).iterator(); }
4.  });
5.  JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() {
6.    public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s, 1); }
7.  });
8.  JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
9.    public Integer call(Integer a, Integer b) { return a + b; }
10. }).collect;
11. ???
```

Examples of Big and Small

Use Case:

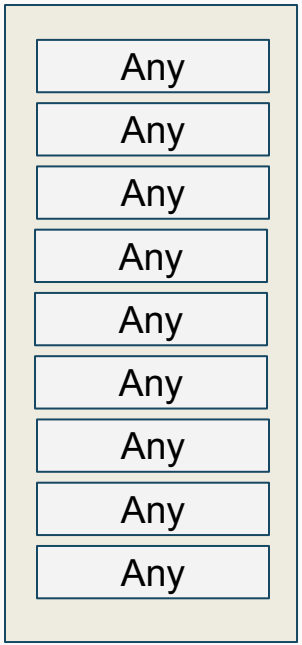
- **Small:** We want to review credit card statements and produce net change from last 3 months statements for customers
- **Big:** We want to review credit card balance of card holders and produce rolling 100 transition average max min spend.
- **Super Big:** We want to review the rolling 1000 average and max for the stock price of all stocks

Distributed Collections

Collections

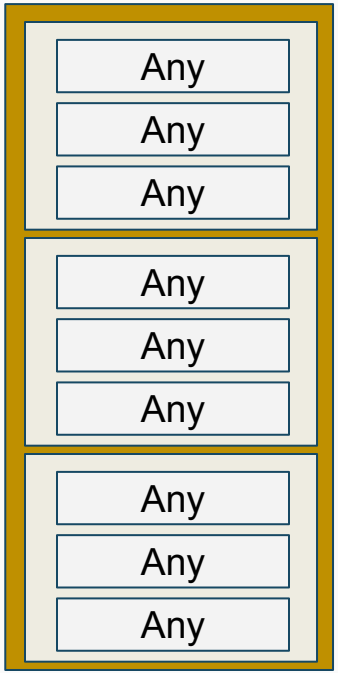
- ArrayList
- RDD
- DataFrame
- DataSet

Collection: Array List



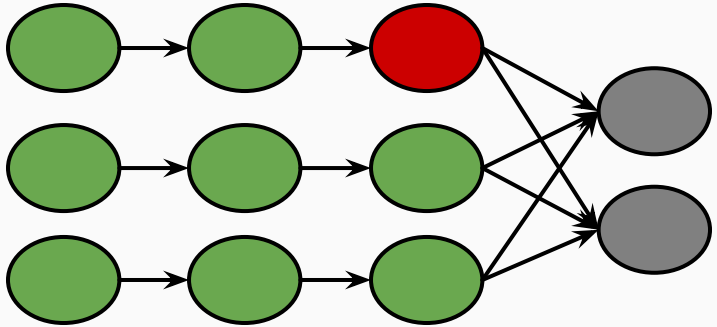
RDD (Resilient Distributed Dataset)

- Partitioned
- Immutable
- Deterministic



RDD is

- Immutable
- Deterministic
- Partitioned

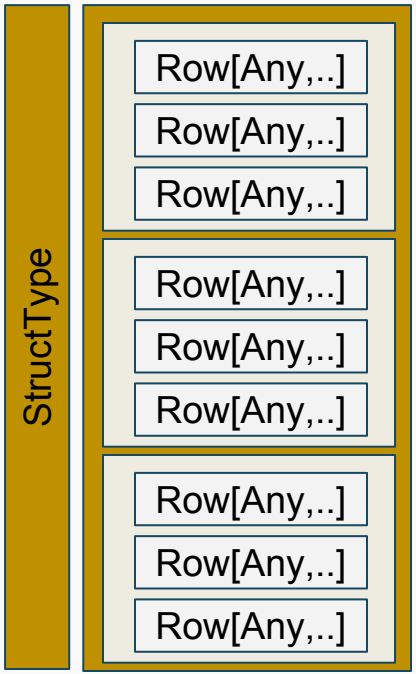


RDD (Creating one)

- Demo Code

DataFrame

- Schema
- Defined Types

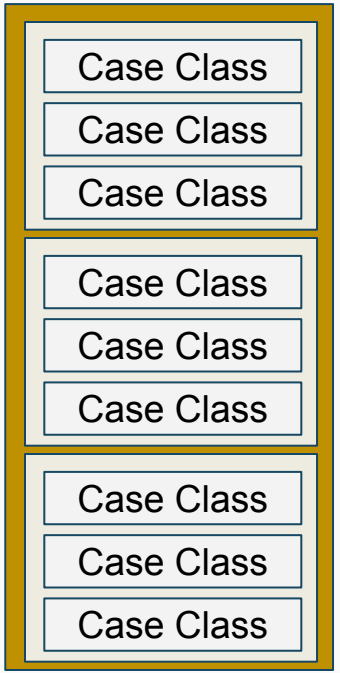


DataFrame (Creating/Getting a Schema)

- Demo Code

DataSet

- Case Class
- Less code
- More auto complete
- Less memory
- More room for Optimization



DataSet (Creating)

- Demo Code

Nested Types

What is a Nested Type

- { “person”:”bob”, "cars":
- {"title": "mustang", "maker": "ford", "tires":
- {"type": "P235/55R17", "pressure":”36psi”}
- {"type": "P235/55R17", "pressure":”36psi”}
- {"type": "P235/55R17", "pressure":”36psi”}
- {"type": "P235/55R17", "pressure":”36psi”}
- },
- {"title": "odyssey", "maker": "honda", "tires":
- {"type": "Goodyear Viva 3 All-Season Tire 205/55R16 91H", "pressure":”36psi”}
- {"type": "Goodyear Viva 3 All-Season Tire 205/55R16 91H", "pressure":”36psi”}
- {"type": "Goodyear Viva 3 All-Season Tire 205/55R16 91H", "pressure":”36psi”}
- {"type": "Goodyear Viva 3 All-Season Tire 205/55R16 91H", "pressure":”36psi”}
- }
- }
-

Why Nested Types

- Parent Child Joins
- More focused reduce by key
- Cartesian Joins

In Hive

```
Create table car_ownership (
  Person string,
  Cars ARRAY < STRUCT <
        Title: STRING,
        Maker: STRING,
        Tires: ARRAY < STRUCT <
              Size: String,
              Pressure: String
        >>
  >>
)
```

In a DataFrame

- Row(“bob”, Seq(
- Row(“mustang”, “ford”, Seq(
- Row(“P235/55R17”, “36psi”),
- Row(“P235/55R17”, “36psi”),
- Row(“P235/55R17”, “36psi”),
- Row(“P235/55R17”, “36psi”)),
- Row(“odyssey”, “honda”, Seq(
- Row(“Goodyear Viva 3 All-Season Tire 205/55R16 91H”, “36psi”),
- Row(“Goodyear Viva 3 All-Season Tire 205/55R16 91H”, “36psi”),
- Row(“Goodyear Viva 3 All-Season Tire 205/55R16 91H”, “36psi”),
- Row(“Goodyear Viva 3 All-Season Tire 205/55R16 91H”, “36psi”))))

Partitioning

What is a Partition

- Isolated block of data
- On read it is supplied to you
  - You have a lot of control depending on you source
- You have full control of partition after first read

What is a Partition

- Repartition
- Coalesce

How are things partitioned on Read

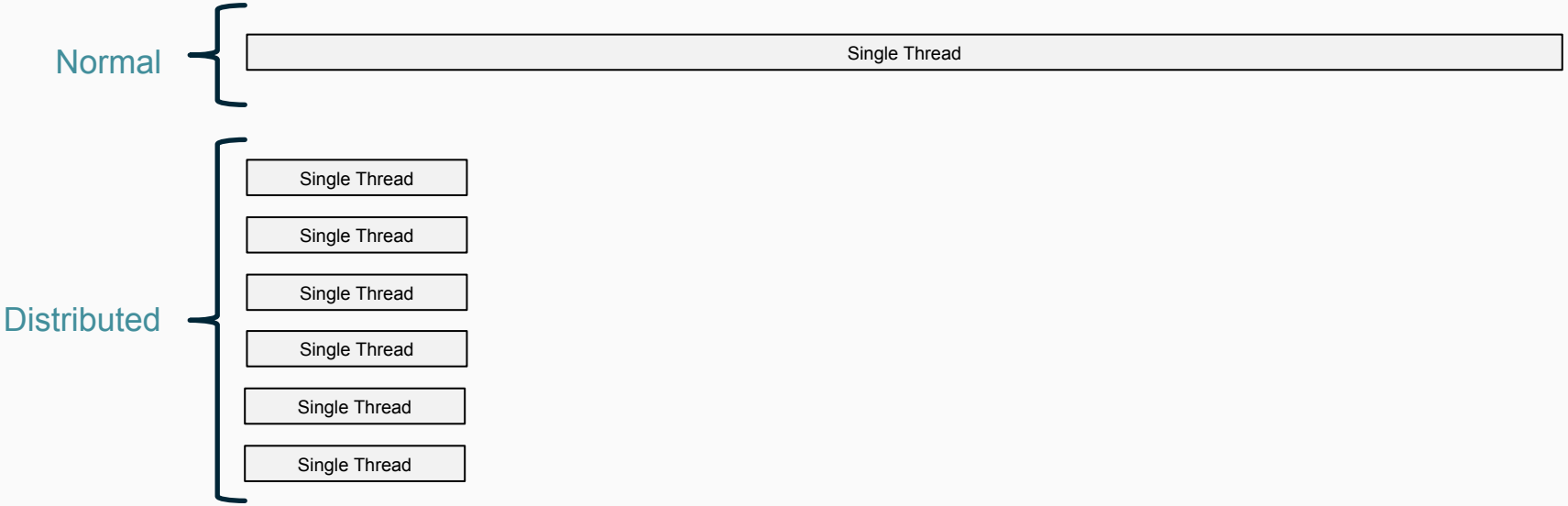
- Blocks (HDFS)
- HBase (Regions)
- Cassandra (Partitions)

Partitioners

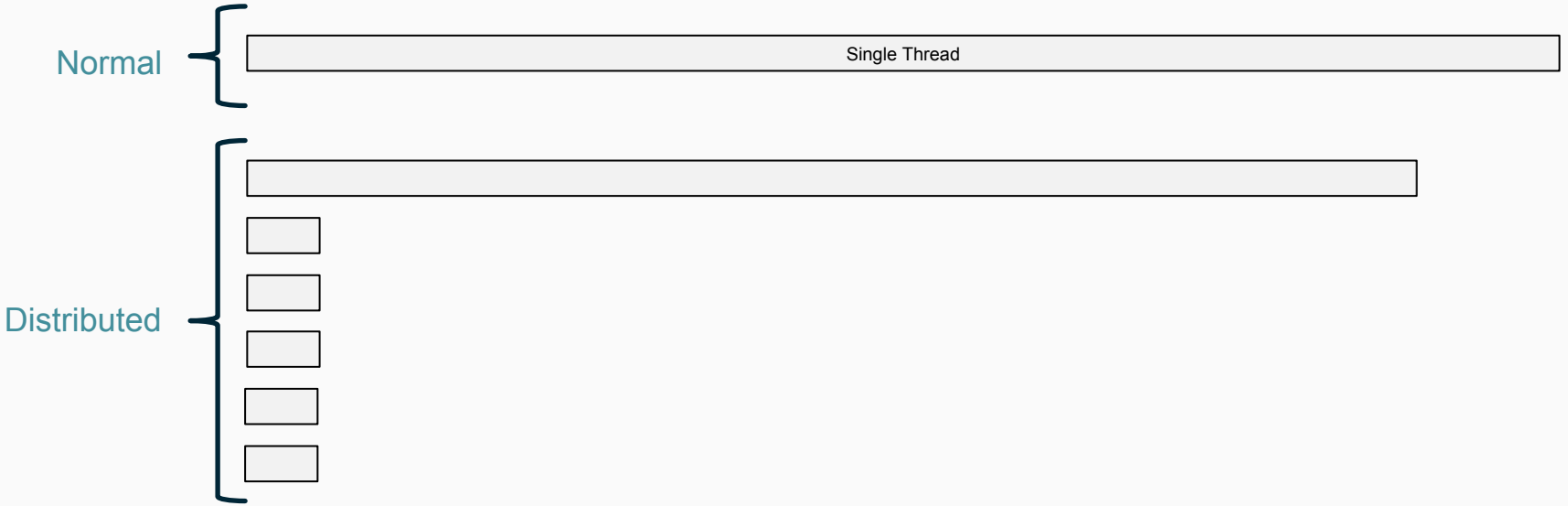
```
class SimpleCustomPartitioner(numOfParts:Int) extends Partitioner {
 override def numPartitions: Int = numOfParts

 override def getPartition(key: Any): Int = {
   val k = key.asInstanceOf[(String, Long)]
   Math.abs(k._1.hashCode) % numPartitions
 }
}
```

Shew

Remember the Perfect Distributed Program



Sometimes Life is not Perfect



Why does Shew happen

- One key makes up a large % of the data when doing joins
- Cartesian joins

Can we solve Shew?

- One Key Problem
  - Salting
  - SubSpliting
  - ReduceByKey
- Cartesian Problem
  - Nesting

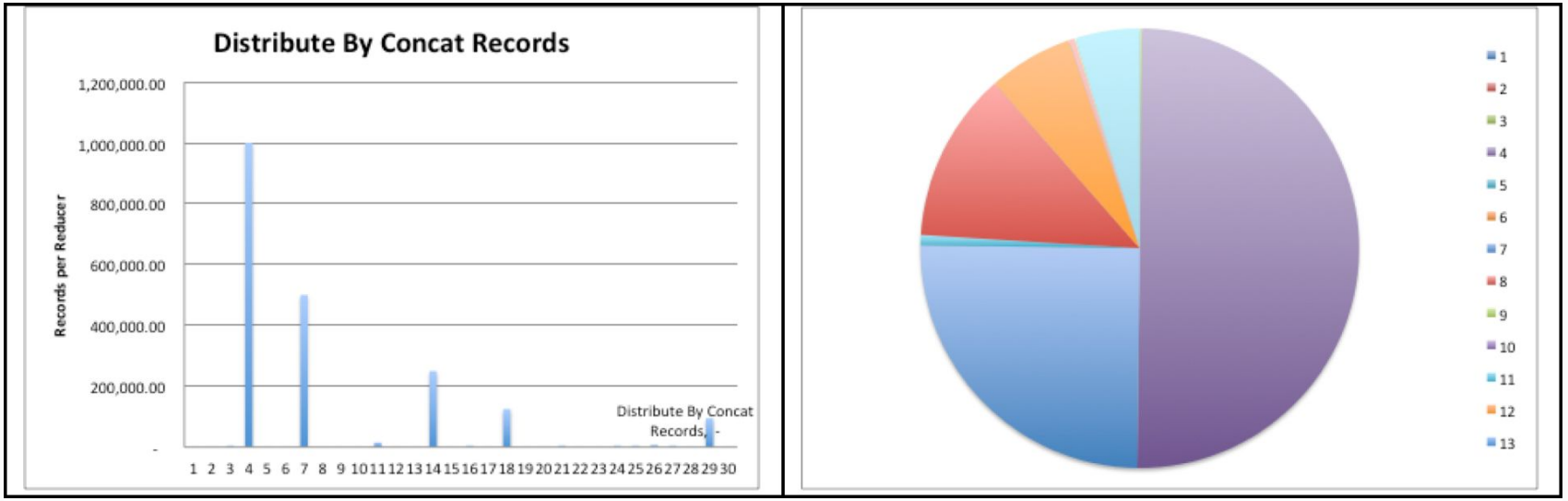
Why does Shew happen

- One Key Problem
  - **Salting**: Will talk about next
  - **SubSpliting**: Super Big Windowing Code
  - **ReduceByKey**: Will talk about next
- Cartesian Problem
  - **Nesting**: Example Nesting Code

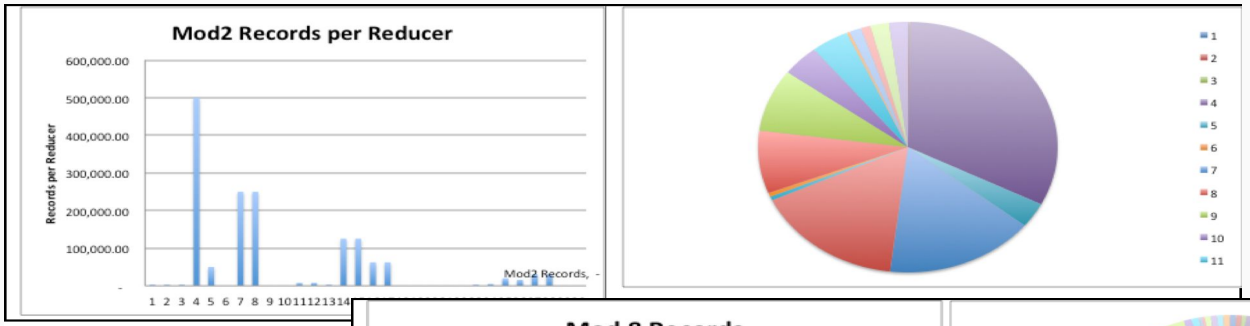
How to Salt

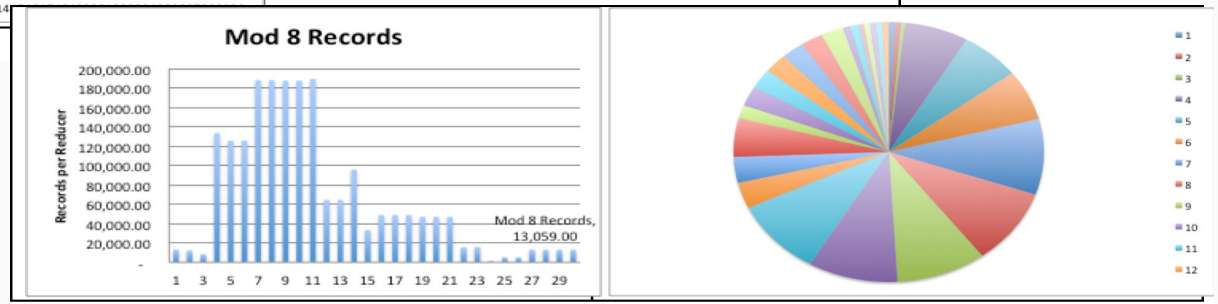
- Normal Key: “Foo”
- Salted Key: “Foo” + random.nextInt(saltFactor)

Before Salting Example



After Salting

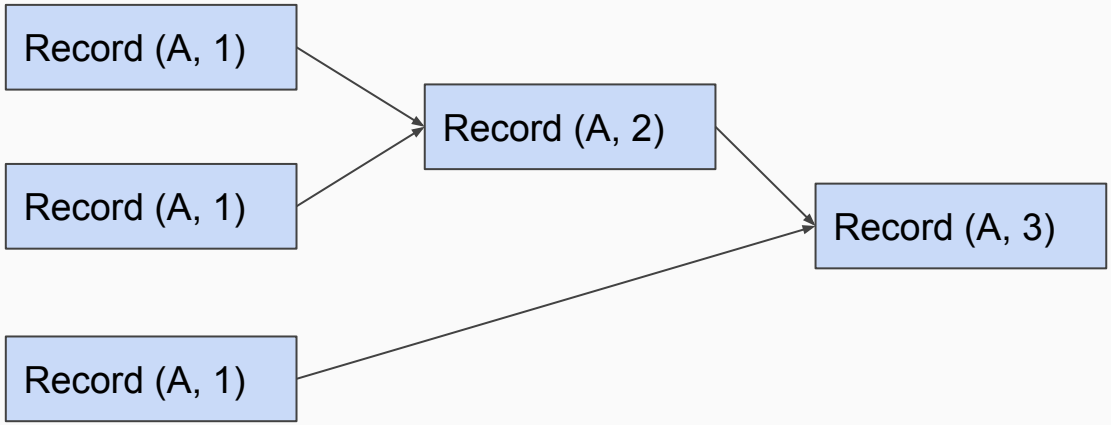




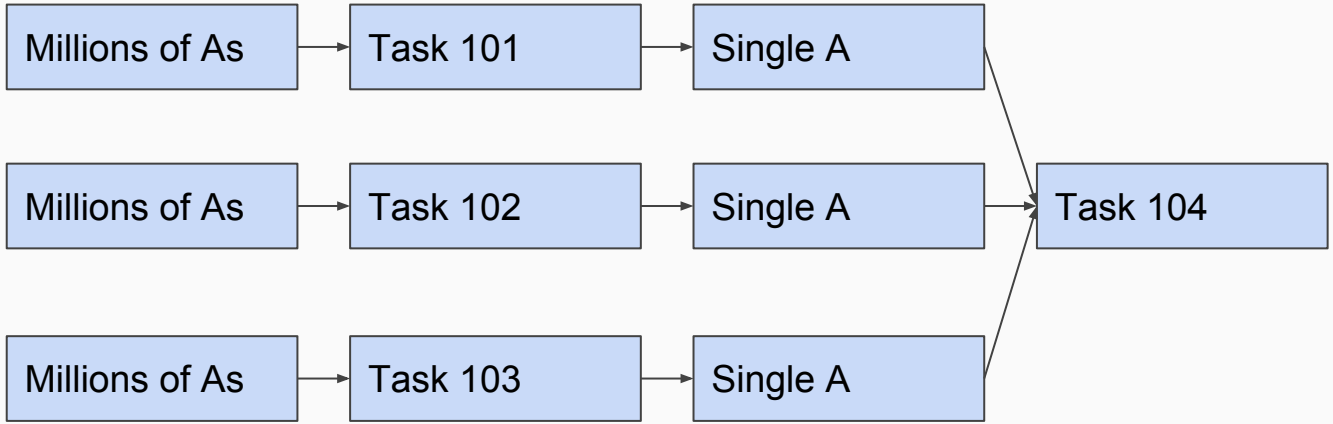
Why ReduceByKey

- Only sees two records at a time
- Reducing happens on all nodes in parallel
- A smaller set of key make reduce by key even better

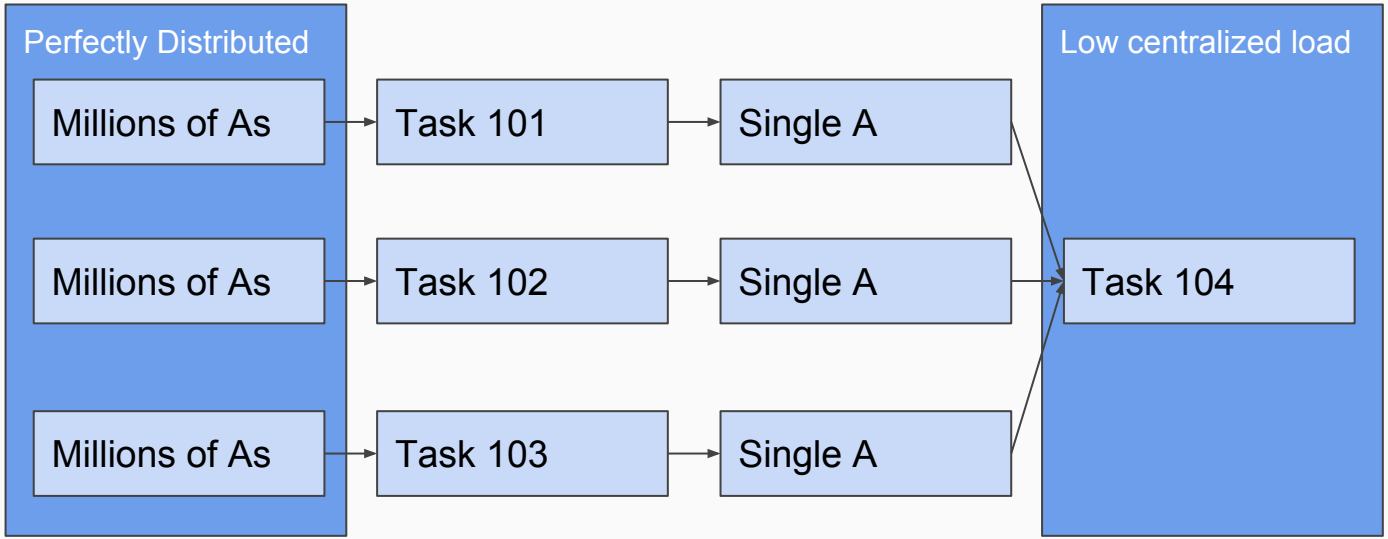
Visional of Reduce By Key



Visional of Reduce By Key



Visional of Reduce By Key



Examples of running the code locally for debugging

Debugging Demo

- Demo

Big and Small Windowing

Small Windowing

- Demo Code

Big Windowing

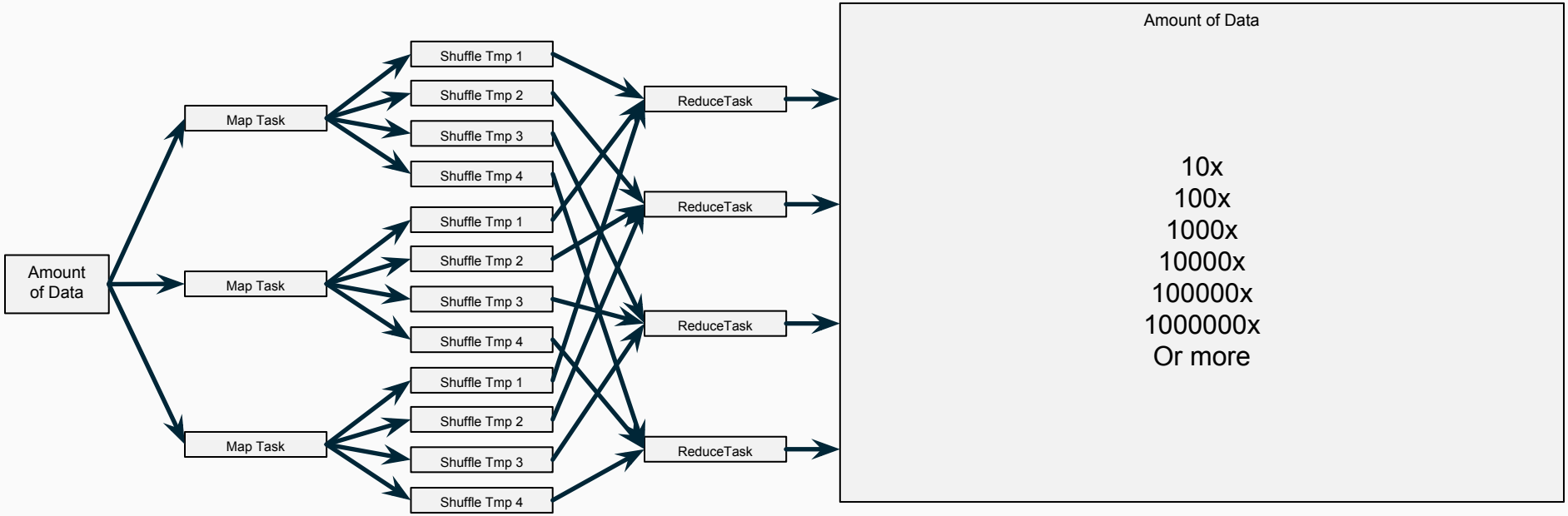
- Demo Code

Super Big Windowing

- Demo Code

Cartesian Joins

What is a Cartesian Joins



Use Case

- N-Gram counts
- 100,000 words
- Will be up to 1,000,000,000 combinations
- Data Growth of 100,000x
- Key Size Grow of 100,100x

Adding Order

- Ordered pairs
- Now up to 50,000(100,001) = 5,050,000 combinations
- Data size growth of about 50x
- Key size growth of 50x

Adding in nesting

- Now up to 50,000(100,001)/2 = 2,525,000 combinations
- Data size growth of about 50x/2 = 25x
- Key size growth of 1x
-
- This means only 1x number of ReduceByKey operations

Multithreaded Driver Code

Why Multithreaded Driver Code

- Story about the Eggs
- More DAGs in a single application

Questions