DTCC 2014中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2014

大数据技术探索和价值发现

# Spark Runtime Internals


连城
lian@databricks.com
cheng.lian@ciilabs.org

databricks™ intel®

# What is Spark

- A fast and general engine for large-scale data processing
- An open source implementation of Resilient Distributed Datasets (RDD)
- Has an advanced DAG execution engine that supports cyclic data flow and *in-memory* computing

# Why Spark

- Fast
  - Run machine learning like iterative programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk
  - Run HiveQL compatible queries 100x faster than Hive (with Shark/Spark SQL)

# Why Spark

- Easy to use
  - Fluent Scala/Java/Python API
  - Interactive shell
  - 2-5x less code (than Hadoop MapReduce)

# Why Spark

- Easy to use
  - Fluent Scala/Java/Python API
  - Interactive shell
  - 2-5x less code (than Hadoop MapReduce)

```
sc.textFile("hdfs://...")
  .flatMap(_.split(" "))
  .map(_ -> 1)
  .reduceByKey(_ + _)
  .collectAsMap()
```

# Why Spark

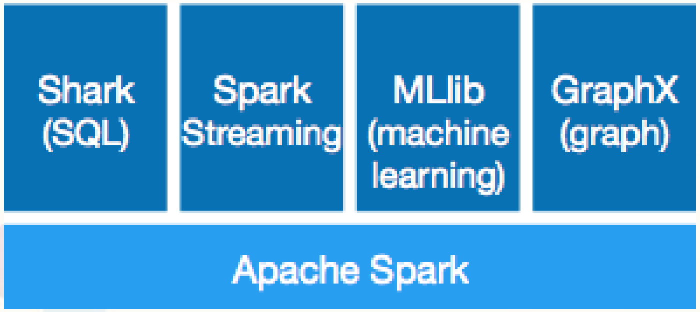
- Easy to use
  - Fluent Scala/Java/Python API
  - Interactive shell
  - 2-5x less code (than Hadoop MapReduce)

```
sc.textFile("hdfs://...")
  .flatMap(_.split(" "))
  .map(_ -> 1)
  .reduceByKey(_ + _)
  .collectAsMap()
```

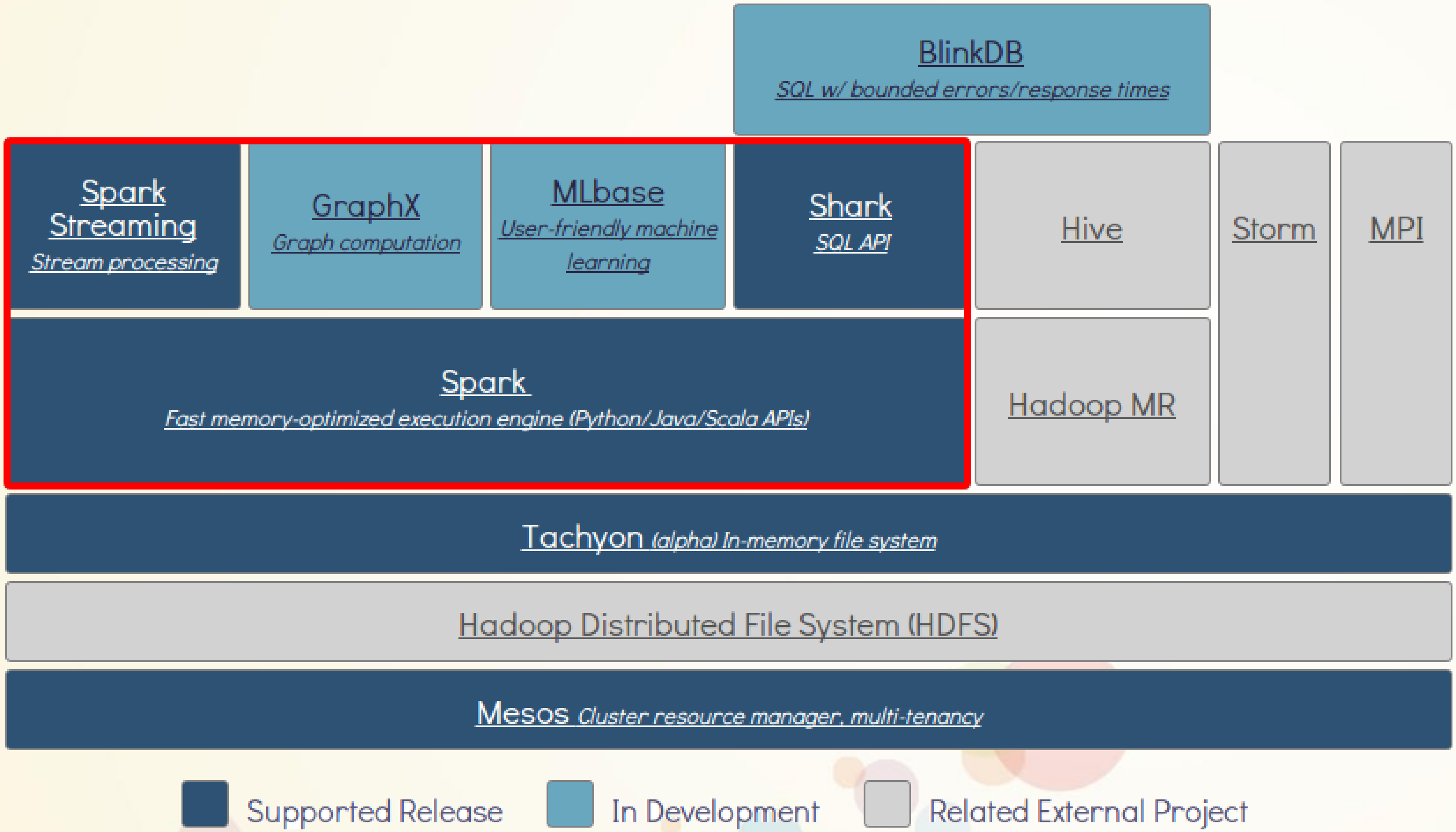
Can you write down
WordCount
in 30 seconds with
Hadoop MapReduce?

# Why Spark

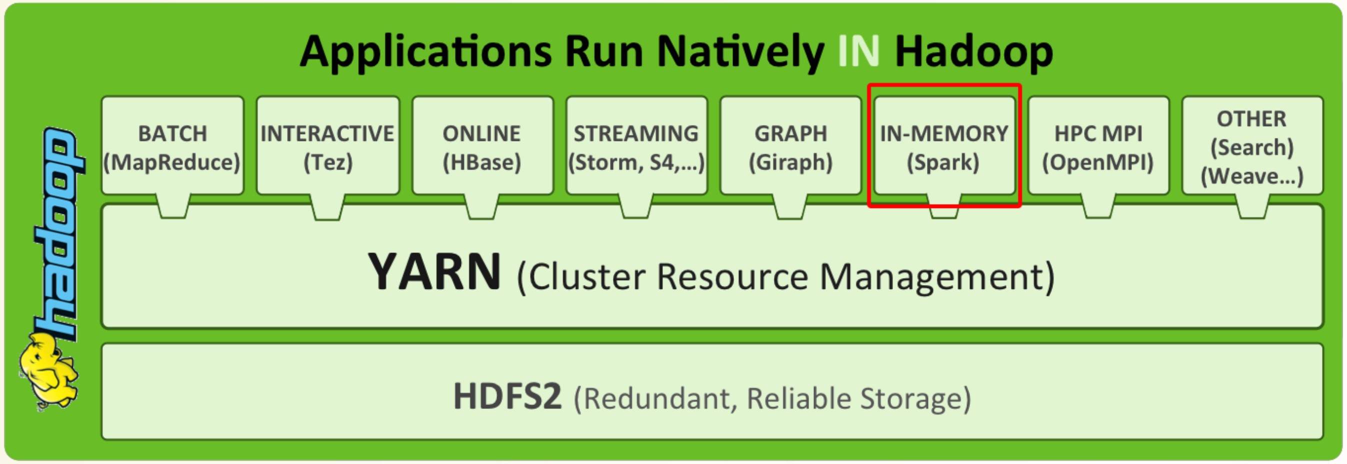
- Unified big data pipeline for:
  - Batch/Interactive (Spark Core vs MR/Tez)
  - SQL (Shark/Spark SQL vs Hive)
  - Streaming (Spark Streaming vs Storm)
  - Machine learning (MLlib vs Mahout)
  - Graph (GraphX vs Giraph)



# A Bigger Picture...

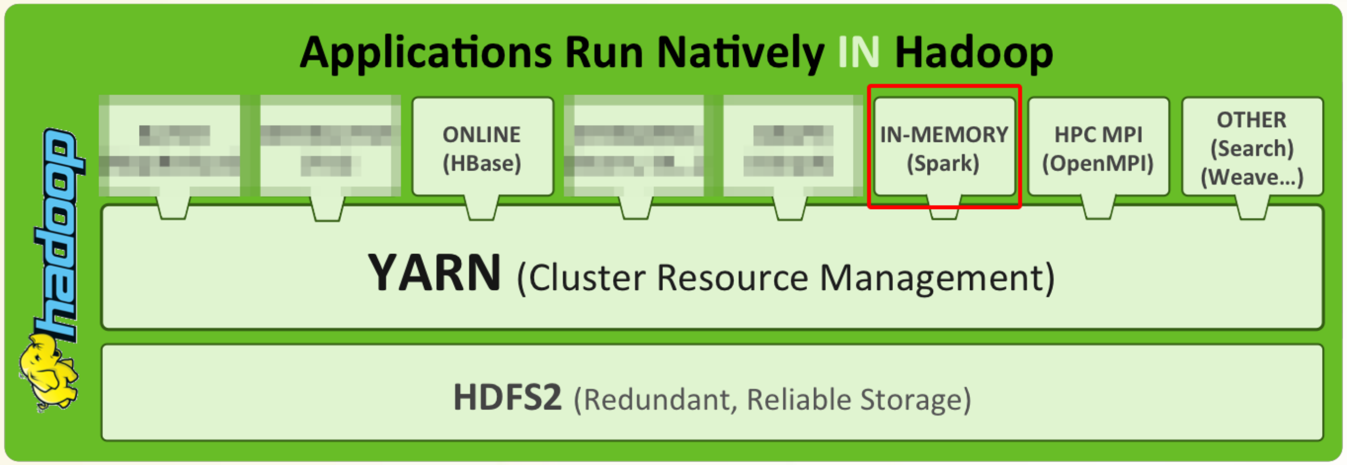


# An Even Bigger Picture...



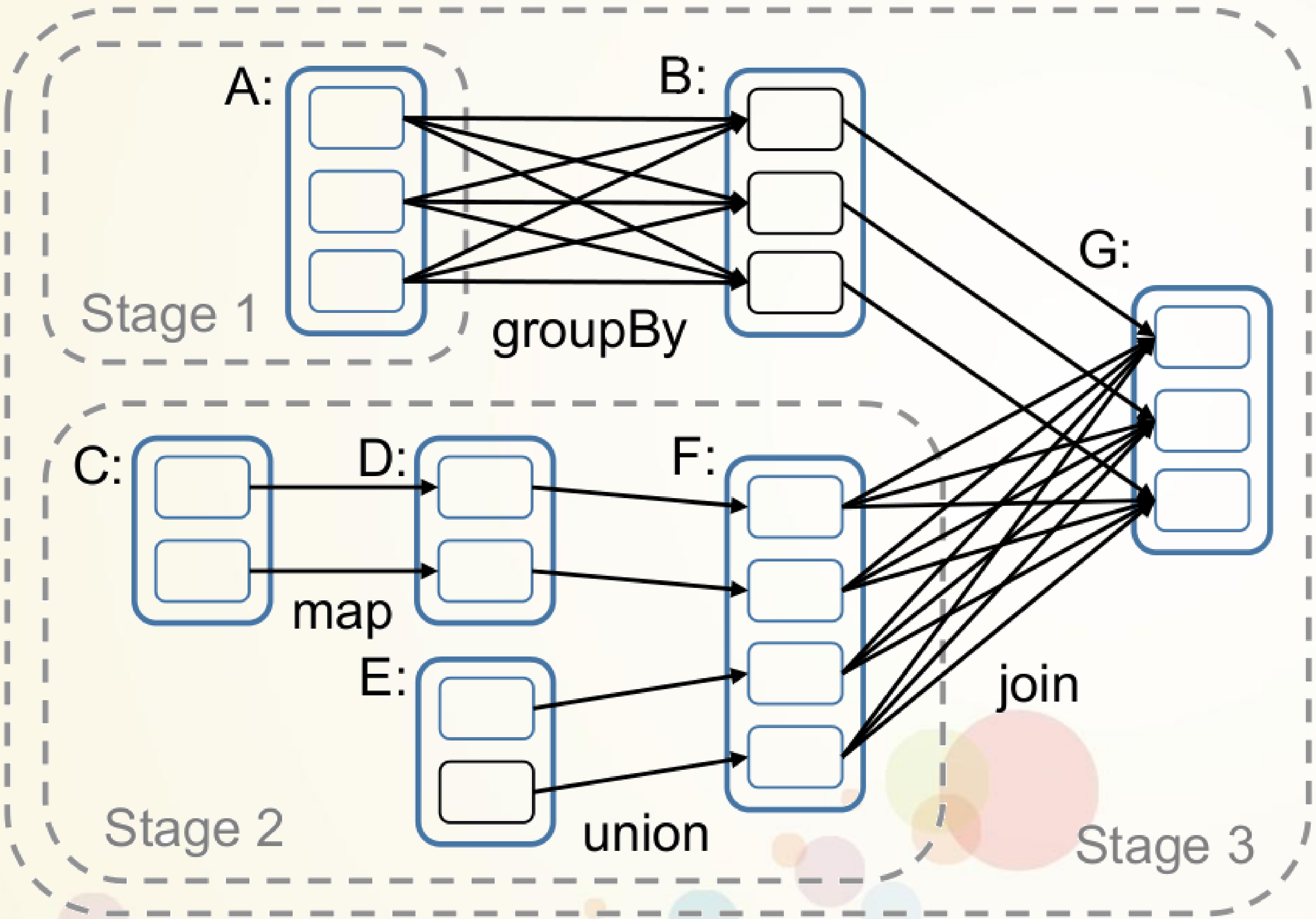

- Components of the Spark stack focus on big data analysis and are compatible with existing Hadoop storage systems
- Users don't need to suffer expensive ETL cost to use the Spark stack

# “One Stack To Rule Them All”




- Well, mostly :-)
- And don't forget Shark/Spark SQL vs Hive

# Resilient Distributed Datasets



# Resilient Distributed Datasets

- Conceptually, RDDs can be roughly viewed as partitioned, locality aware distributed vectors
- An RDD…
  - either points to a direct data source
  - or applies some transformation to its parent RDD(s) to generate new data elements
  - Computation can be represented by *lazy evaluated* lineage *DAGs* composed by connected RDDs

# Resilient Distributed Datasets

- Frequently used RDDs can be materialized and cached in-memory to accelerate computation
- Spark scheduler takes data locality into account

# The In-Memory Magic

- “In fact, one study [1] analyzed the access patterns in the Hive warehouses at Facebook and discovered that for the vast majority (96%) of jobs, the entire inputs could fit into a fraction of the cluster’s total memory.”

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In HotOS ’11, 2011.

# The In-Memory Magic

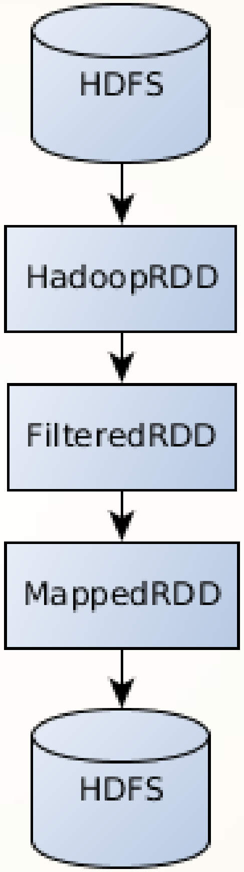
- Without cache
  - Elements are accessed in an iterator-based streaming style
  - One element a time, no bulk copy
  - Space complexity is almost O(1) when there’s only narrow dependencies

# The In-Memory Magic

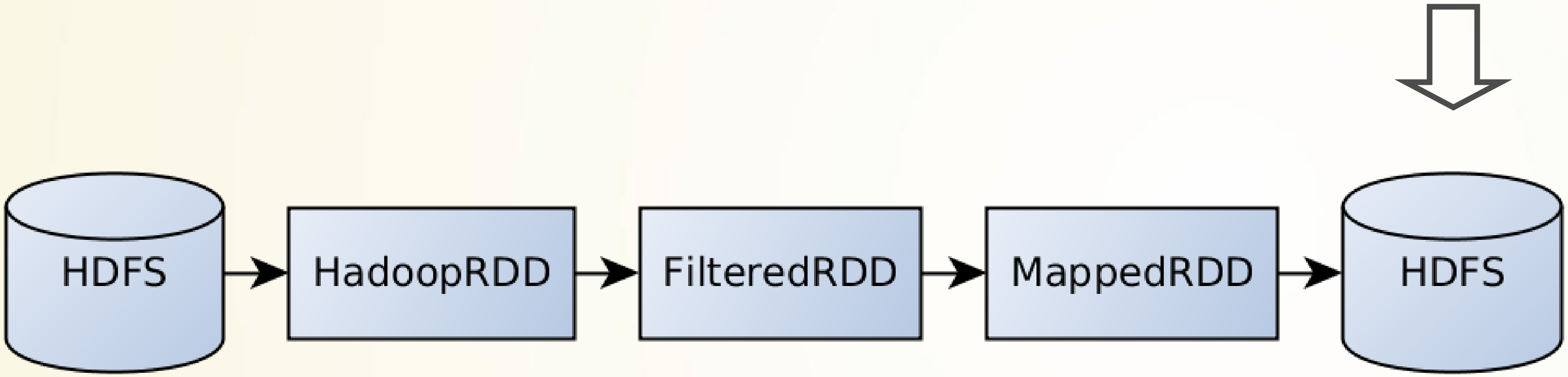
- With cache
  - One block per RDD partition
  - LRU cache eviction
  - Locality aware
  - Evicted blocks can be *recomputed in parallel* with the help of RDD lineage DAG

# Play with Error Logs

```
sc.textFile("hdfs://<input>")
  .filter(_.startsWith("ERROR"))
  .map(_.split(" ")(1))
  .saveAsTextFile("hdfs://<output>")
```

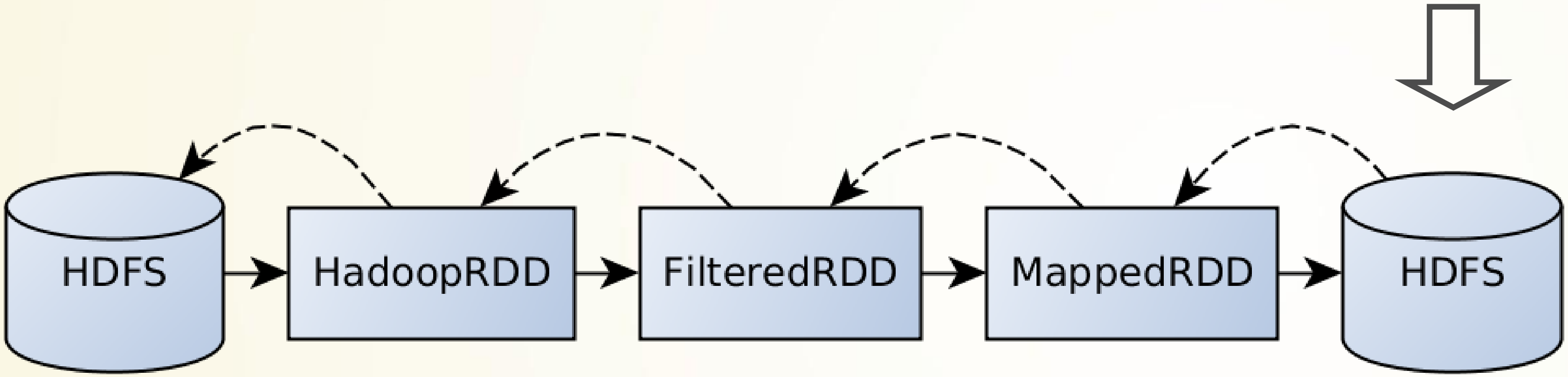


# Step by Step



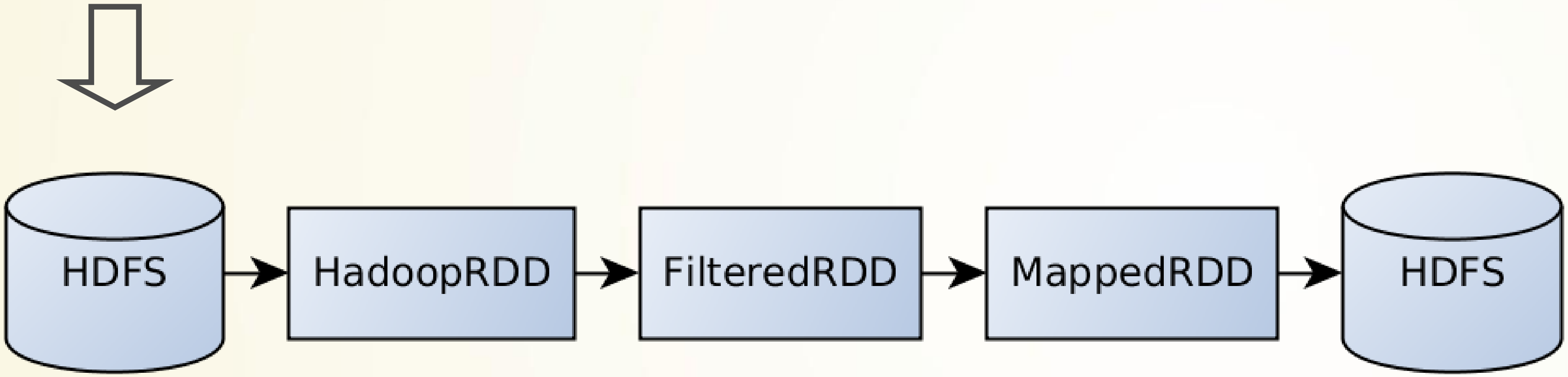

The RDD.saveAsTextFile() action triggers a job. Tasks are started on scheduled executors.

# Step by Step



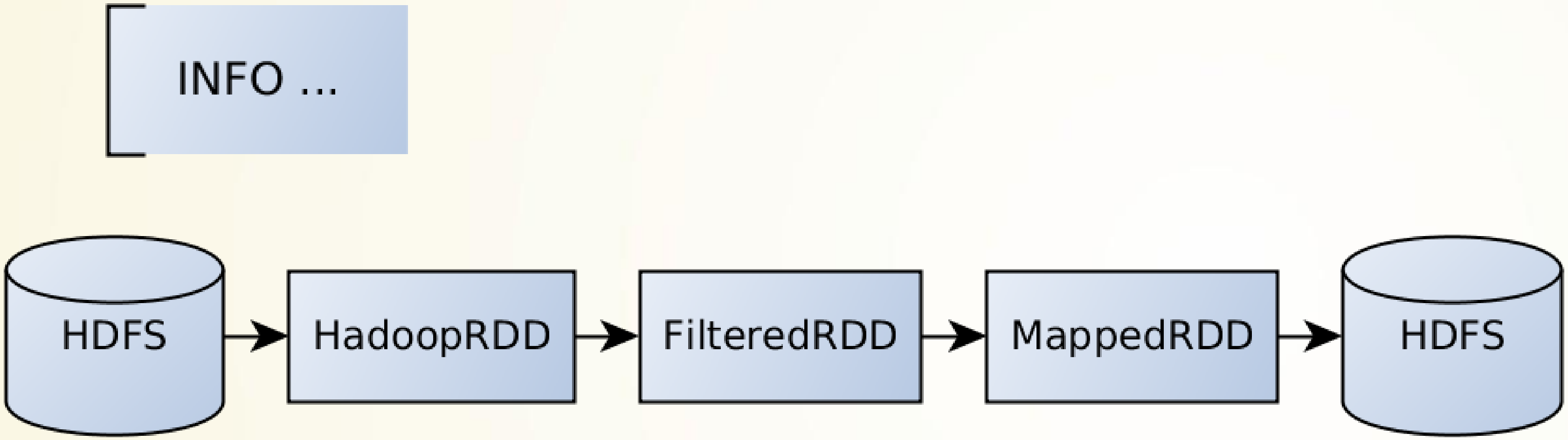

The task pulls data lazily from the final RDD (the `MappedRDD` here).

# Step by Step



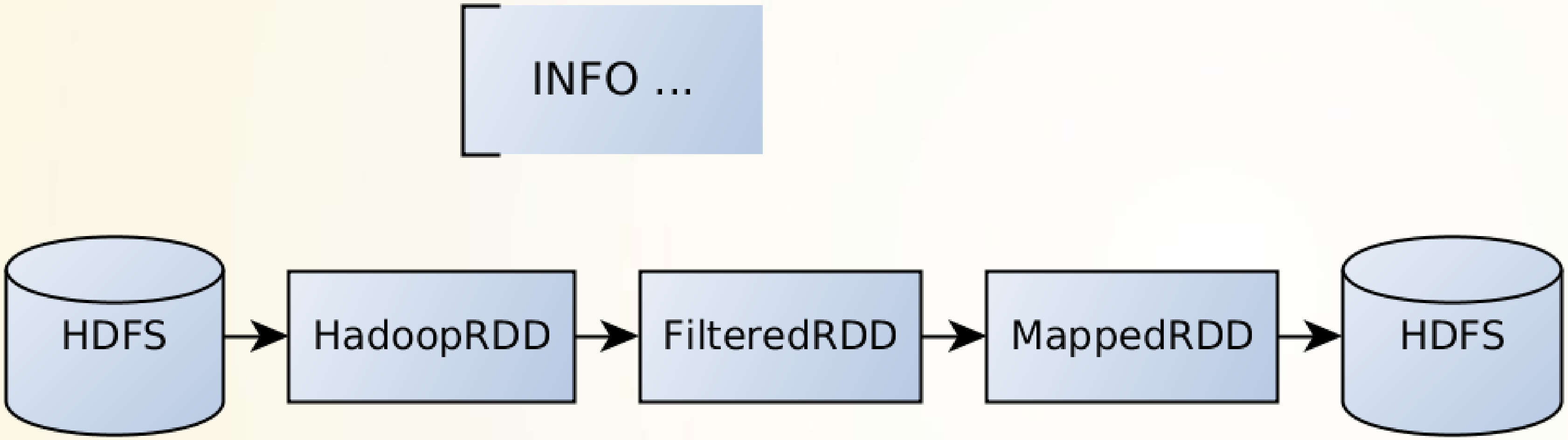

A record (text line) is pulled out from HDFS...

# Step by Step



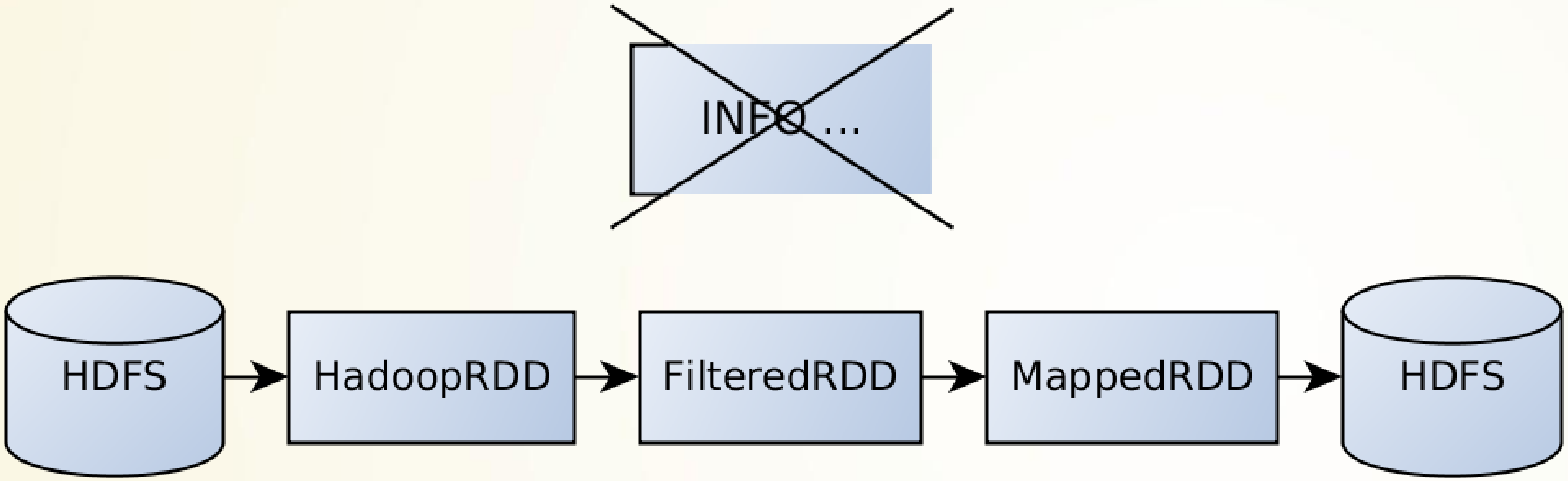

... into a HadoopRDD

# Step by Step



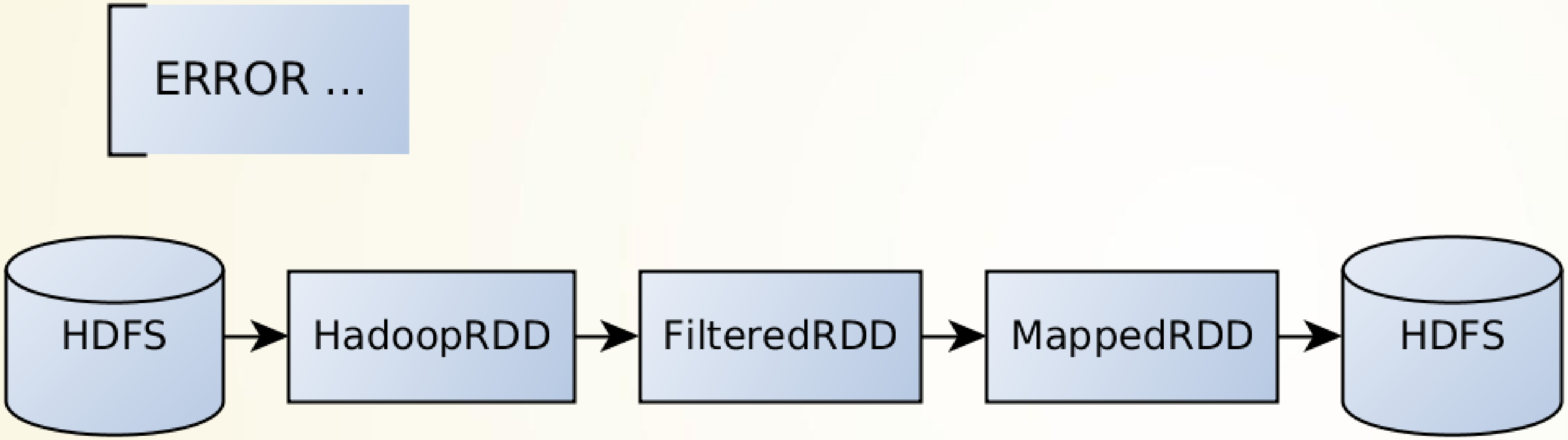

Then filtered with the `FilteredRDD`

# Step by Step



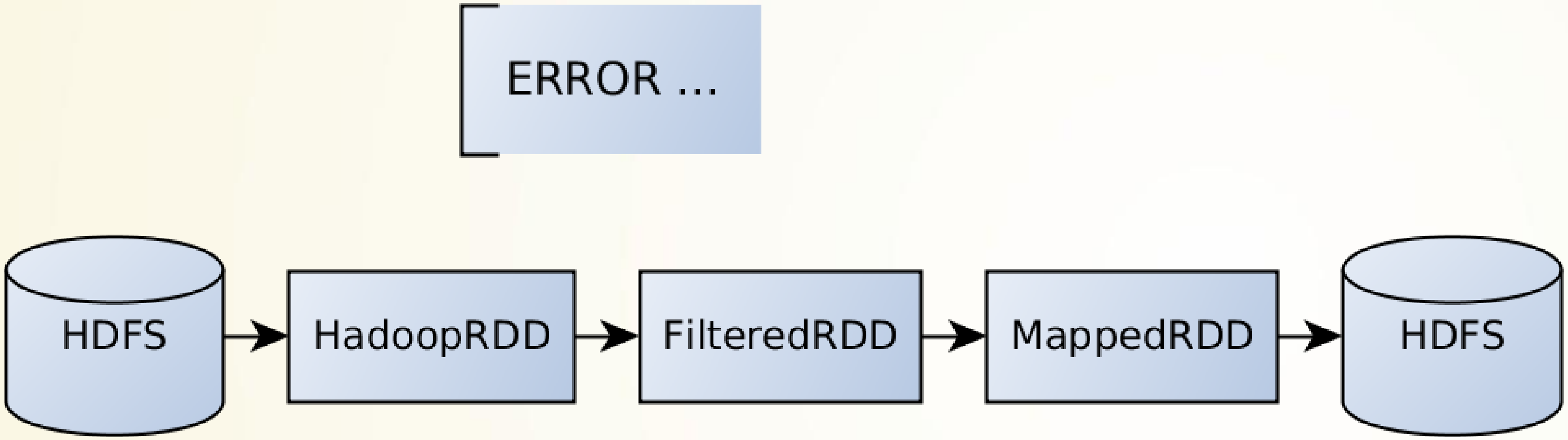

Oops, not a match

# Step by Step



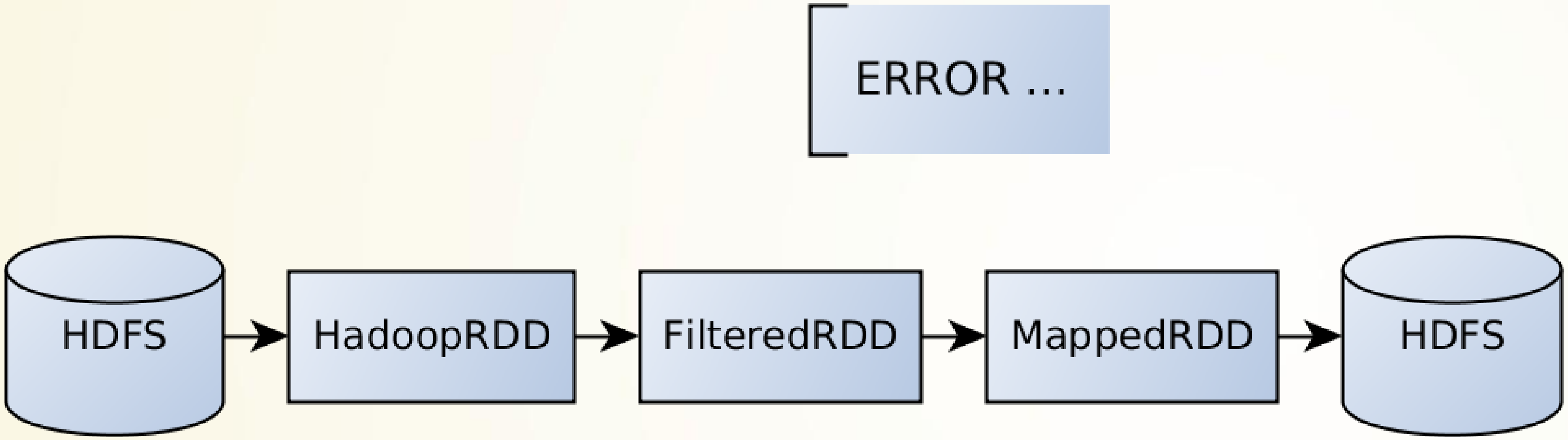

Another record is pulled out...

# Step by Step



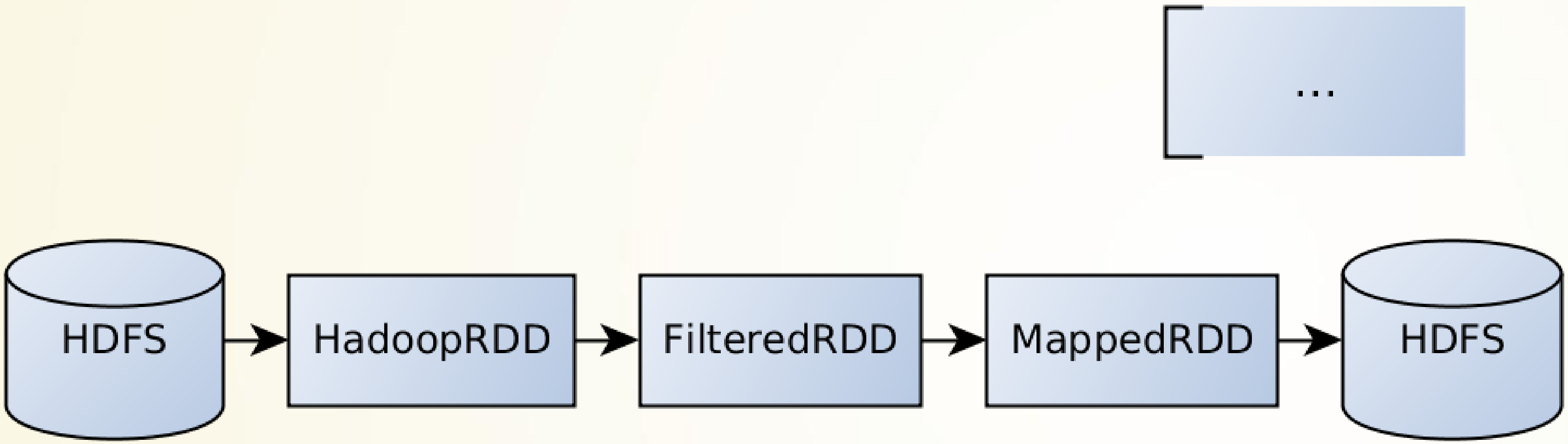

Filtered again...

# Step by Step



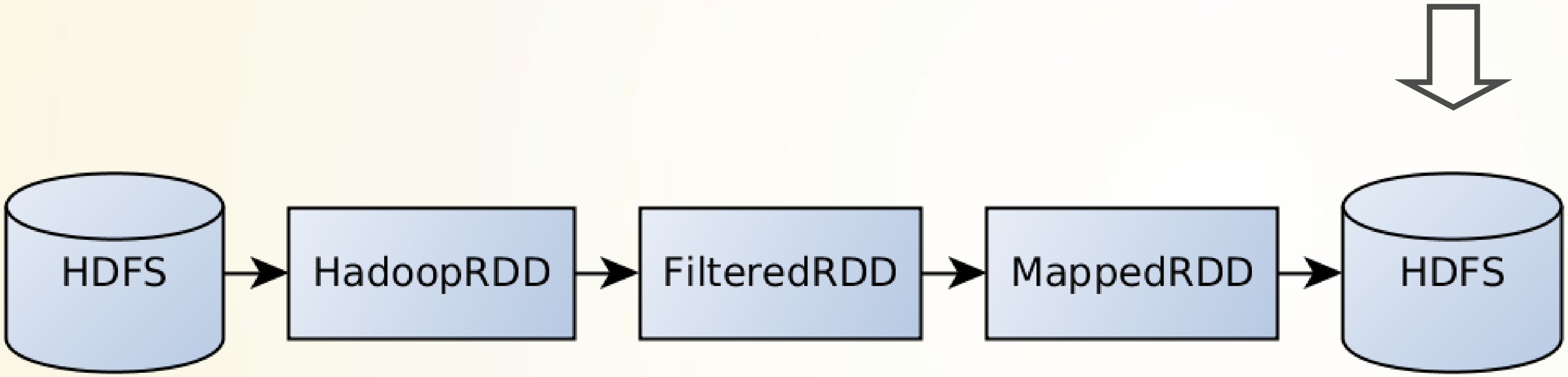

Passed to the `MappedRDD` for next transformation

# Step by Step




Transformed by the `MappedRDD` (the error message is extracted)

# Step by Step




Saves the message to HDFS

# A Synthesized Iterator

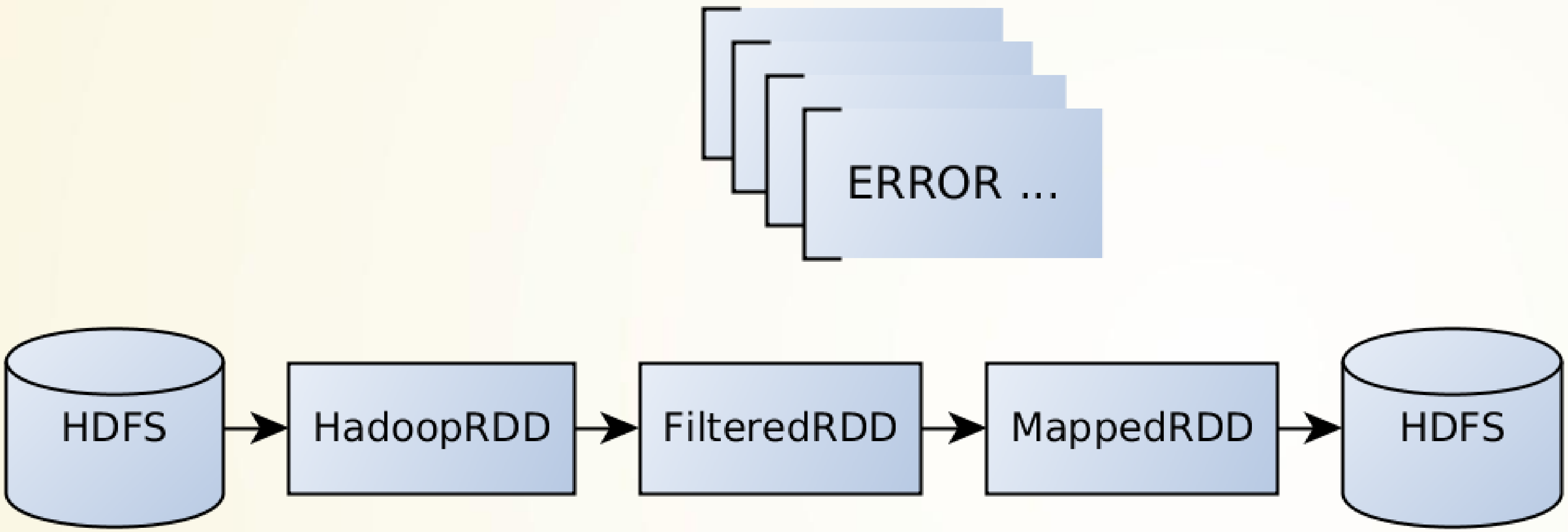
Constant Space Complexity!

```
new Iterator[String] {
  private var head: String = _
  private var headDefined: Boolean = false

  def hasNext: Boolean = headDefined || {
    do {
      try head = readOneLineFromHDFS(...)     // (1) read from HDFS
      catch {
        case _: EOFException => return false
      }
    } while (!head.startsWith("ERROR"))       // (2) filter closure
    true
  }

  def next: String = if (hasNext) {
    headDefined = false
    head.split(" ")(1)                        // (3) map closure
  } else {
    throw new NoSuchElementException("...")
  }
}
```

# If Cached...

```
val cached = sc
  .textFile("hdfs://<input>")
  .filter(_.startsWith("ERROR"))
  .cache()

cached
  .map(_.split(" ")(1))
  .saveAsTextFile("hdfs://<output>")
```

# If Cached...

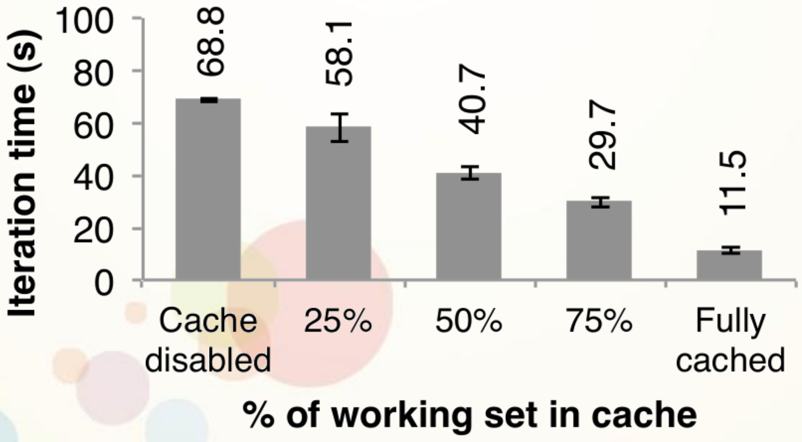



All filtered error messages are cached in memory before being passed to the next RDD. LRU cache eviction is applied when memory is insufficient

# But… I Don’t Have Enough Memory To Cache *All* Data

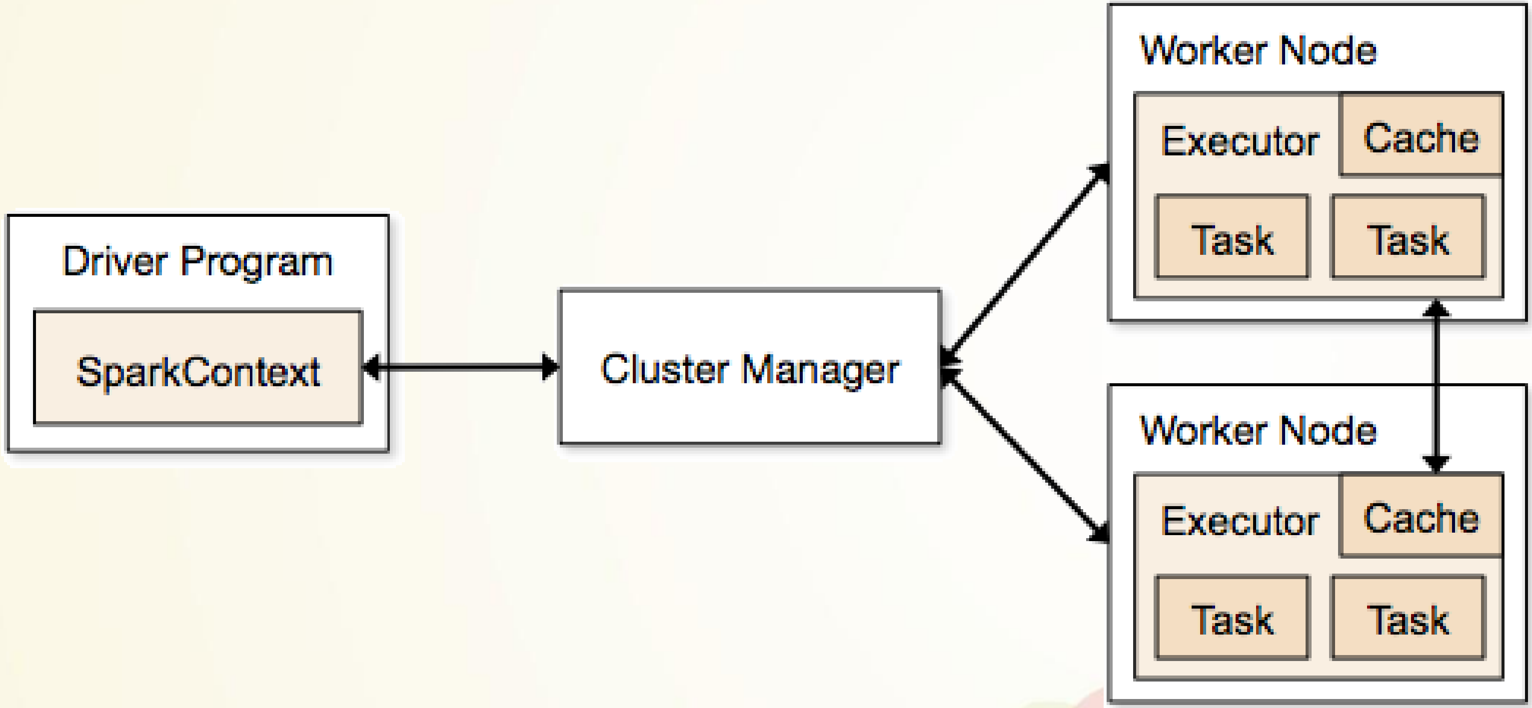
# Don't Worry

- In most cases, you only need to cache a fraction of hot data extracted/transformed from the whole input dataset
- Spark performance downgrades linearly & gracefully when memory decreases



# How Does The Job Get Distributed?

# Spark Cluster Overview



Driver, Cluster Manager,
Worker, Executor, ...

So... Where THE HELL
Does My Code Run?

Well... It Depends

# Driver and SparkContext

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sc = new SparkContext(...)
```

- A `SparkContext` initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors
- Since then, the driver takes full responsibilities

# WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

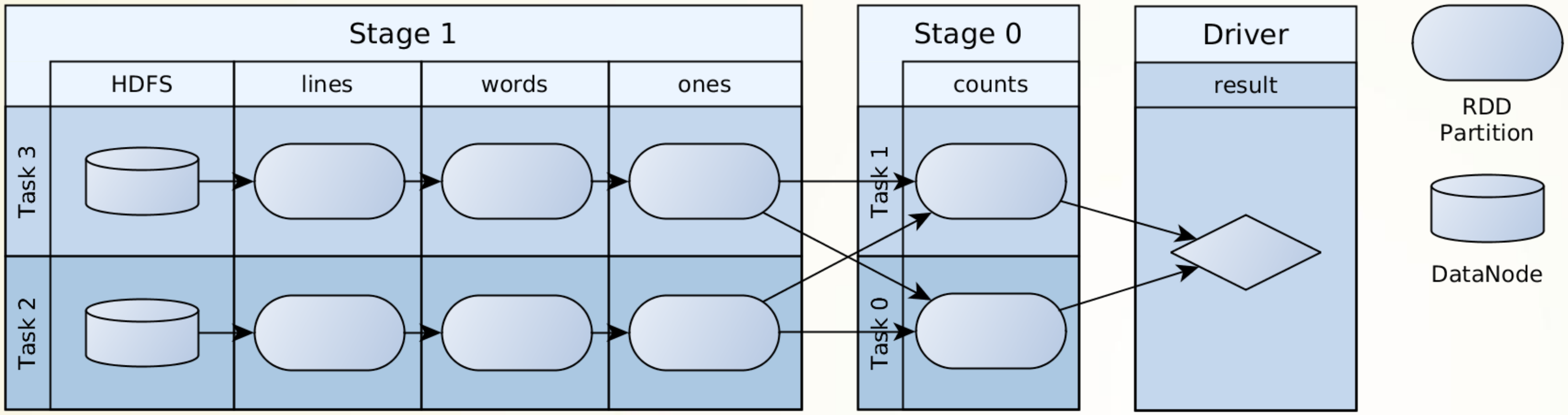
- RDD lineage DAG is built on *driver* side with:
  - Data source RDD(s)
  - Transformation RDD(s), which are created by transformations

# WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

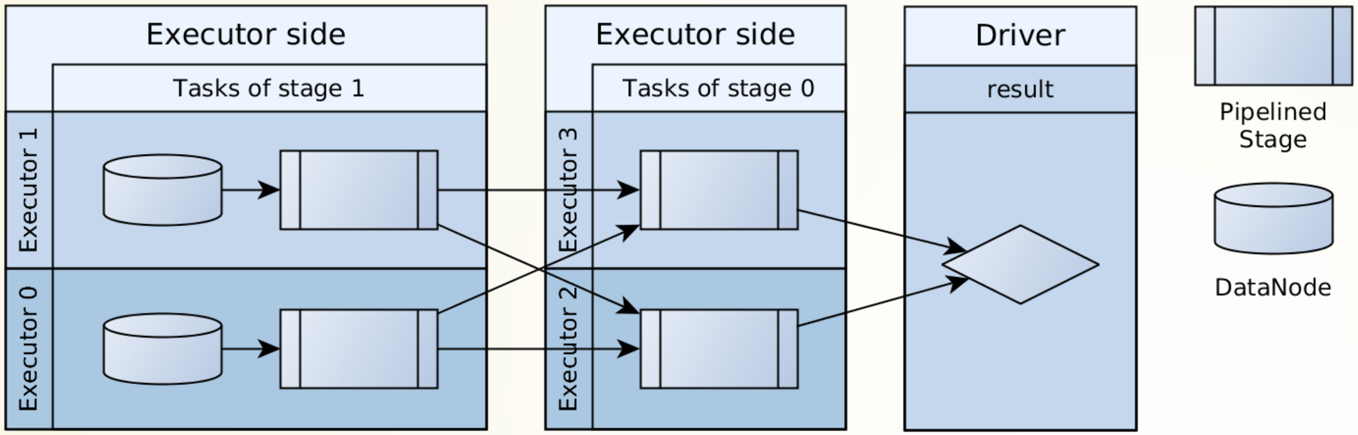
- Once an action is triggered on *driver* side, a job is submitted to the *DAG scheduler* of the driver

# WordCount Revisited




- DAG scheduler cuts the DAG into stages and turns each *partition* of a stage into a single task.
- DAG scheduler decides *what to run*

# WordCount Revisited




- Tasks are then scheduled to executors by driver side *task scheduler* according to *resource* and *locality* constraints
- Task scheduler decides *where to run*

# WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- Within a task, the lineage DAG of corresponding stage is serialized together with closures of transformations, then sent to and executed on scheduled *executors*

# WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- The `reduceByKey` transformation introduces in a shuffle
- Shuffle outputs are written to local FS on the mapper side, then downloaded by reducers

# WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- ReduceByKey automatically combines values within a single partition locally on the mapper side and then reduce them globally on the reducer side.

# WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- At last, results of the action are sent back to the driver, then the job finishes.

# What About Parallelism?

# How To Control Parallelism?

- Can be specified in a number of ways
  - RDD partition number
    - `sc.textFile("input", minSplits = 10)`
    - `sc.parallelize(1 to 10000, numSlices = 10)`
  - Mapper side parallelism
    - Usually inherited from parent RDD(s)
  - Reducer side parallelism
    - `rdd.reduceByKey(_ + _, numPartitions = 10)`
    - `rdd.reduceByKey(partitioner = p, _ + _)`
    - …

# How To Control Parallelism?

- "Zoom in/out"
  - `RDD.repartition(numPartitions: Int)`
  - `RDD.coalesce(`
    `numPartitions: Int,`
    `shuffle: Boolean)`

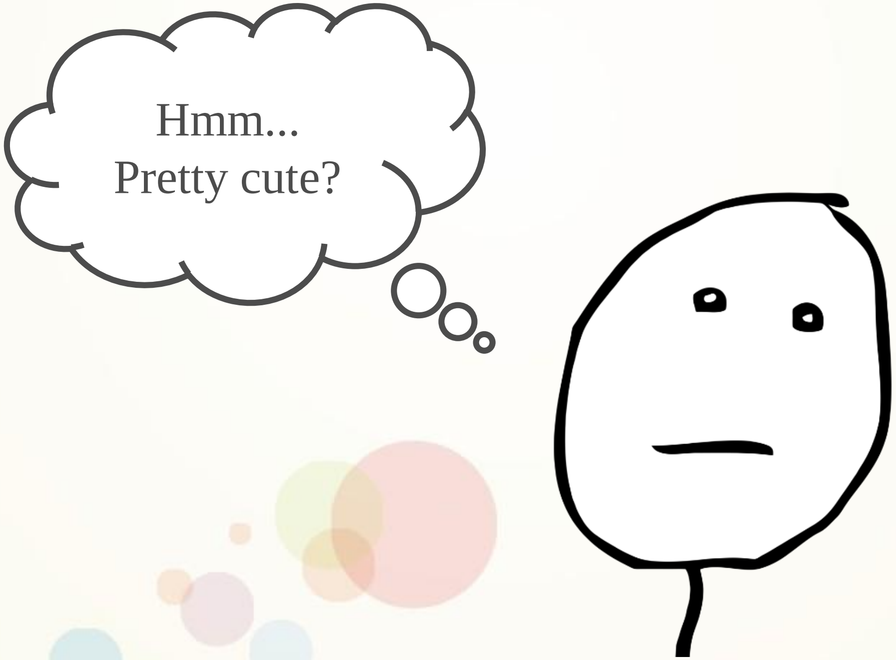
# A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- Run in *local* mode
- 60+GB input file

# A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```



# A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```


- Actually, this may generate 4,000,000 temporary files...
- Why 4M? (Hint: $2K^2$)

# A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- 1 partition per HDFS split
- Similar to mapper task number in Hadoop MapReduce, the `minSplits` parameter is only a *hint* for split number

# A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- Actual split number is also controlled by some other properties like minSplitSize and default FS block size

# A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- In this case, the final split size equals to local FS block size, which is 32MB by default, and 60GB / 32MB ≈ 2K
- `ReduceByKey` generates $2K^2$ shuffle outputs

# A Trap of Partitions

```
sc.textFile("input", minSplits = 2).coalesce(2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- Use `RDD.coalesce()` to control partition number precisely.

# Summary

- In-memory magic
  - Iterator-based streaming style data access
  - Usually you don't need to cache all dataset
  - When memory is not enough, Spark performance downgrades linearly & gracefully

# Summary

- Distributed job
  - Locality aware
  - Resource aware
- Parallelism
  - Reasonable default behavior
  - Can be finely tuned

# References


- Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing
- An Architecture for Fast and General Data Processing on Large Clusters
- Spark Internals (video, slides)
- Shark: SQL and Rich Analytics at Scale
- Spark Summit 2013

# Q&A

THANKS