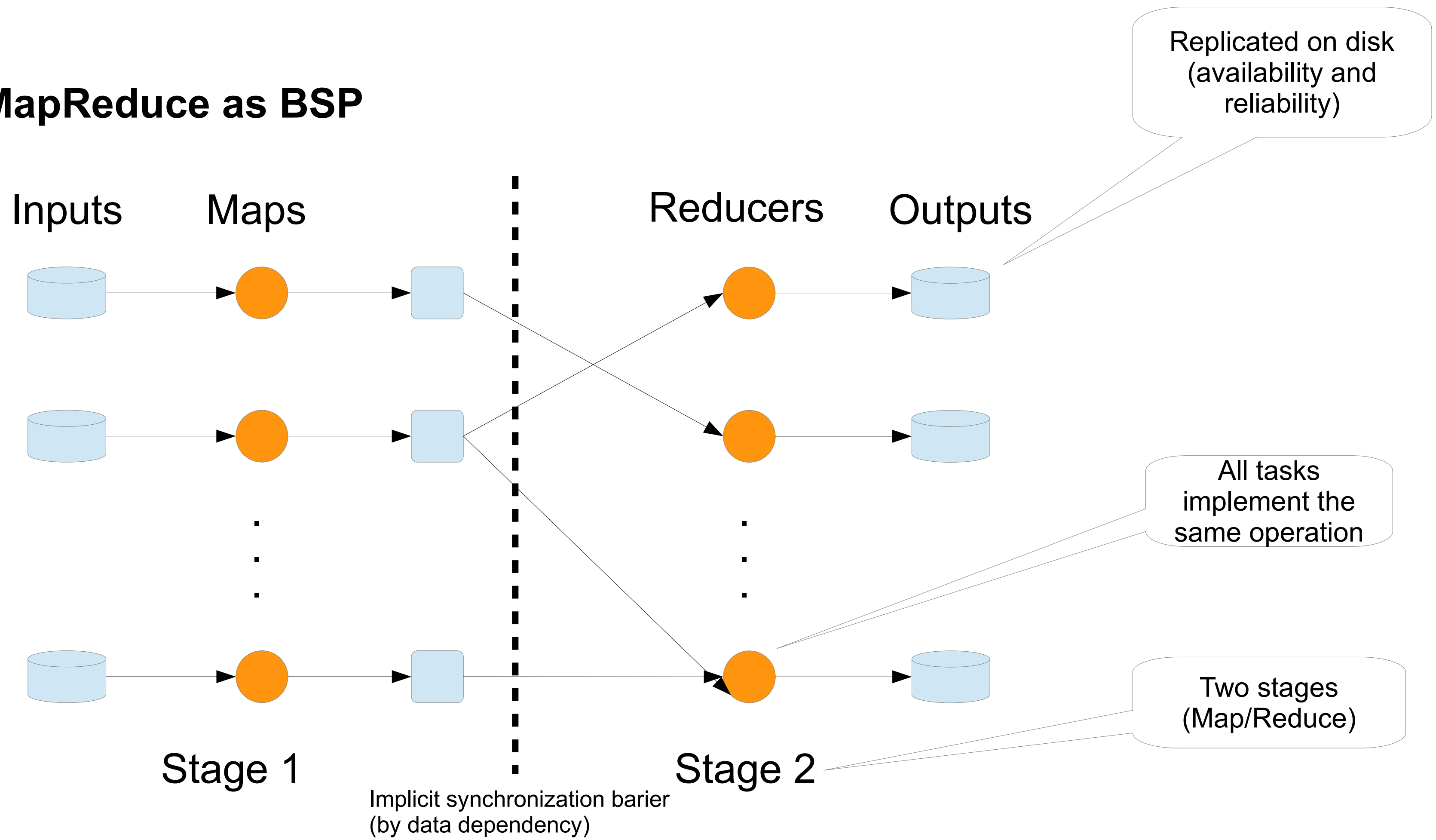
PDD

Lecture 2: Introduction to Apache Spark (and cluster computing)

Prepared by Jacek Sroka

Leslie G. **Valiant**. 1990. **A bridging model for parallel computation**.
Commun. ACM 33, 8 (Aug. 1990), 103–111.
DOI:https://doi.org/10.1145/79173.79181

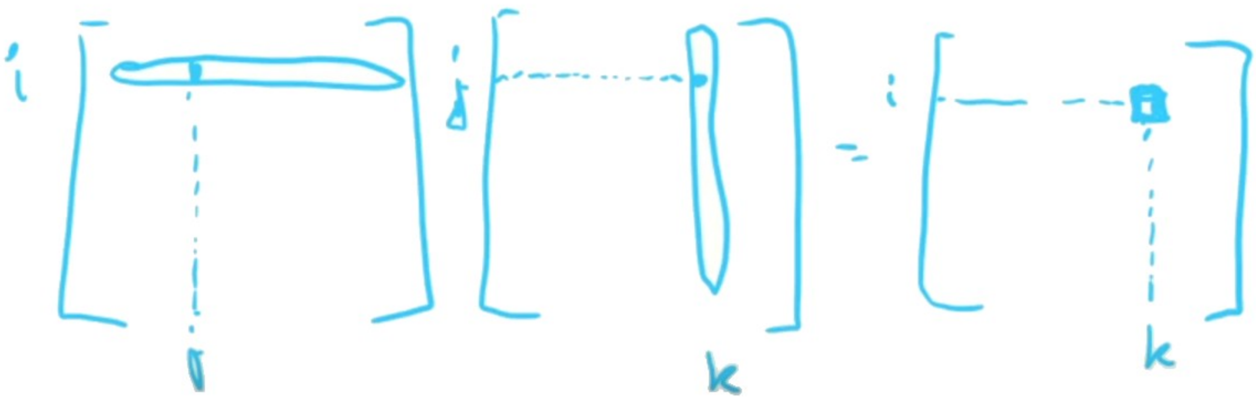
MapReduce as BSP



BTW: Massively Parallel Computation (MPC)
is simplification of BSP

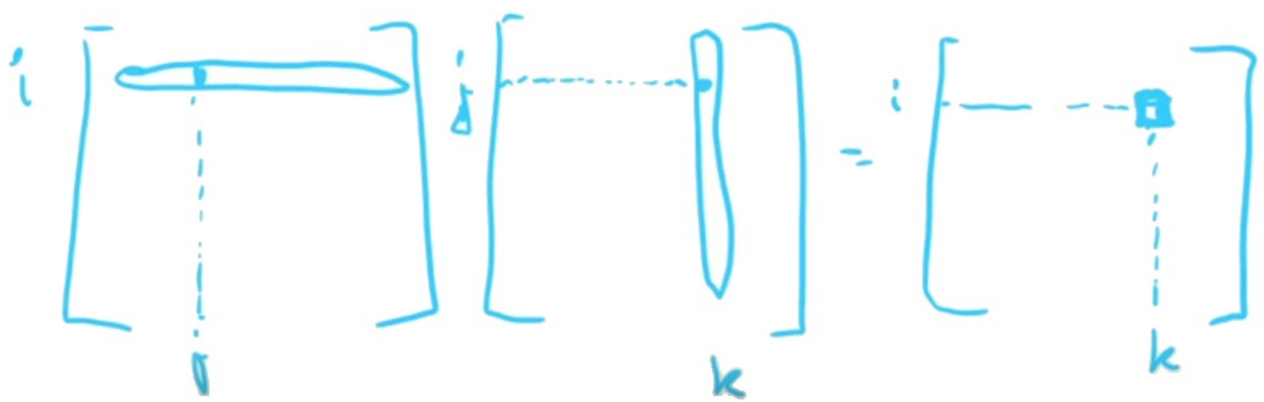
Matrix multiplication example

- Two good ideas to solve (both have advantages)
- **Multiplying matrix by matrix**
  Input: (M, N) as <(i, j), m(i, j)> and <(j, k), n(j, k)>
  Output: MN as <(i, j), $SUM_{j=1..n}$ m(i, j) * n(j, k)>
  Solution1: (two MR phases)
  Use J as the key both for mappers processing M and N
  Reduce result from all mappers together (group on J)
  In second MR phase group on (I, K) and sum values



Matrix multiplication example

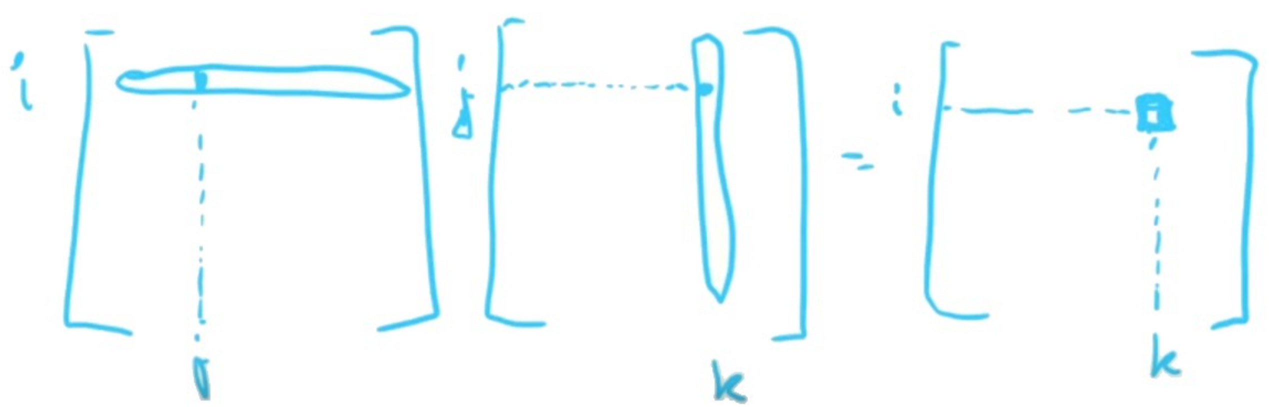
- Two good ideas to solve (both have advantages)
- **Multiplying matrix by matrix**
  Input: (M, N) as <(i, j), m(i, j)> and <(j, k), n(j, k)>
  Output: MN as <(i, k), $SUM_{j=1..n}$ m(i, j) * n(j, k)>
  Solution1: (two MR phases)
  Use J as the key:
  Map1 processes M and emits <j, (i, m(i, j), M)>
  Map2 processes N and emits <j, (k, n(j, k), N)>
  Red1 **for J** group generates pair**s** <(i, k), m(i, j)*n(j, k)>.
  In second MR group on (I, K) and sum values



Matrix multiplication example

- Two good ideas to solve (both have advantages)
- **Multiplying matrix by matrix**
  Input: (M, N) as <(i, j), m(i, j)> and <(j, k), n(j, k)>
  Output: MN as <(i, j), SUM$_{j=1..n}$ m(i, j) * n(j, k)>
  Solution1: (two MR phases)
  Use J as the key:
  Map1 processes M and emits <j, (i, m(i, j), M)>
  Map2 processes N and emits <j, (k, n(j, k), N)>
  Red1 **for J** group generates pair**s** <(i, k), m(i, j)*n(j, k)>.
  In second MR group on (I, K) and sum values

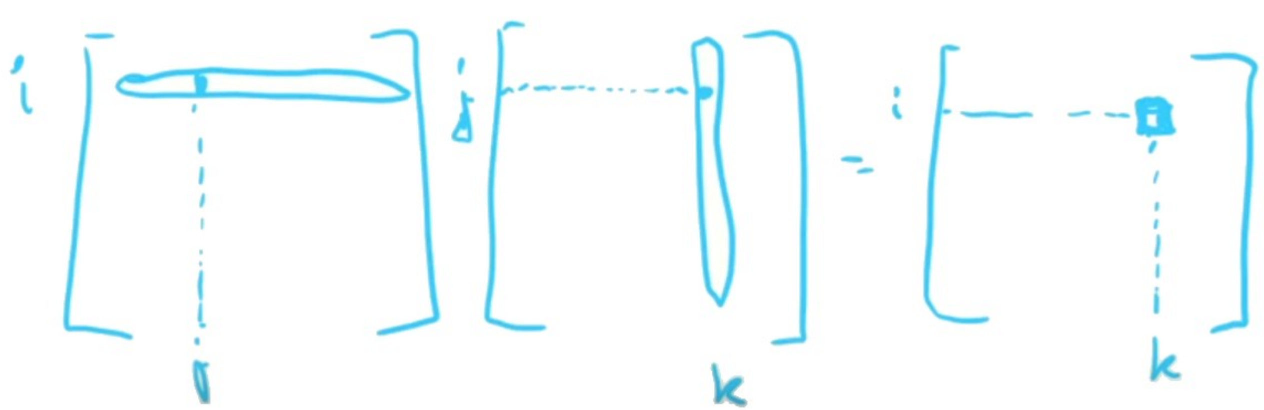
  Solution2: (one MR phase)
  Emit all values required to compute mn(i,k)
  For that emit many copies of each value with all missing index values



Matrix multiplication example

- Two good ideas to solve (both have advantages)
- **Multiplying matrix by matrix**
  Input: (M, N) as <(i, j), m(i, j)> and <(j, k), n(j, k)>
  Output: MN as <(i, j), $SUM_{j=1..n}$ m(i, j) * n(j, k)>
  Solution1: (two MR phases)
  - Use J as the key:
  - Map1 processes M and emits <j, (i, m(i, j), M)>
  - Map2 processes N and emits <j, (k, n(j, k), N)>
  - Red1 **for J** group generates pair**s** <(i, k), m(i, j)*n(j, k)>.
  - In second MR group on (I, K) and sum values

  Solution2: (one MR phase)
  - Map1: For m(i,j) emit <**(i, k)**,(j, m(i,j))> **k = 1,2 . . . up to col. no of N**
  - Map2: For n(j,k) emit <**(i, k)**,(j, n(j,k))> **i = 1,2 . . . up to row no of M**
  - Reduce: multiply values in (j, m(i,j)) and (j, n(j,k)) for the same j and sum



Question

- In first solution we have `J` groups and no replication
- In second solution we have `IxK` groups and data replication

- Is it good to have many reducers?

Question

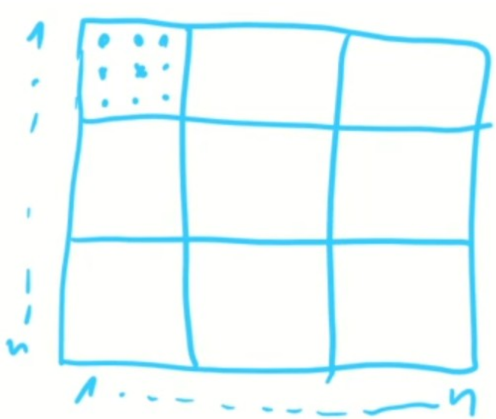
- In first solution we have `J` groups and no replication
- In second solution we have `IxK` groups and data replication

- Is it good to have many reducers?
  - Pros: if more than R.proc. e.g. 2-4 times then it helps with skew
  - Cons: may need to replicate a lot
    - Example: As input we have several thousands of elements of several MB each. We want to process pairs (e.g. to find similar elements, interaction between drugs, etc.)

Question

- In first solution we have `J` groups and no replication
- In second solution we have `IxK` groups and data replication

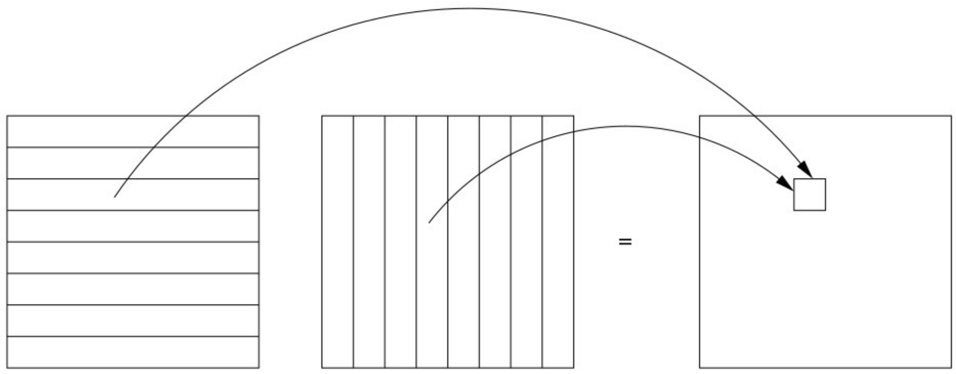
- Is it good to have many reducers?
  - Pros: if more than R.proc. e.g. 2-4 times then it helps with skew
  - Cons: may need to replicate a lot
    - Example: As input we have several thousands of elements of several MB each. We want to process pairs (e.g. to find similar elements, interaction between drugs, etc.)
    - key = pair with numbers of elements, then each block replicated thousands of times
    - key = pair with numbers of elements mod 100, then each block replicated tens of times



Extending this idea to matrix multiplication

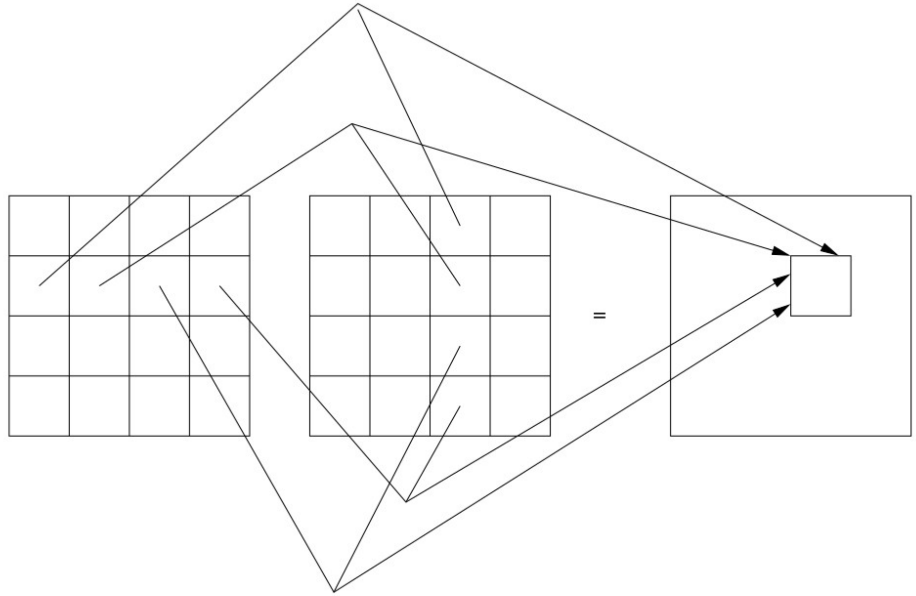
<u>Single phase</u>

- Divide bands of g col/rows
- Maps generate g copies
- Reducers have all values necessary to compute a square of MN
- Replication rate is g



<u>Two phases</u>

- Divide into squares
- First phase:
  compute the products of squares (I, J) of M with the square (J, K) of N
- Second phase:
  for each I and K we sum the products over all possible sets J

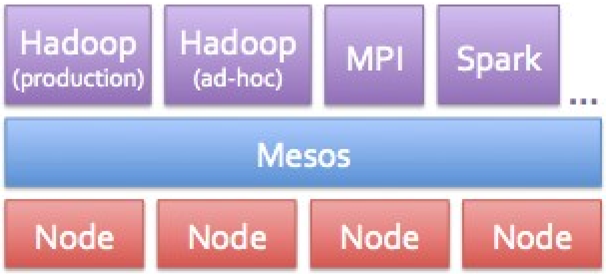


MapReduce pros/cons

- Scalable: scales to clusters of thousands of nodes
- Cheap: Runs on commodity hardware, Open source Hadoop
- Simple: simple API, fault tolerant, straggler mitigation
- General: expressive enough for large variety of data proc. (arbitrary code in map/reduce)
- Great for batch processing of logs (multiple HDD read simultaneously)

- Performance
  - Not great for data exploration (a job takes minutes or more)
  - Not great for iterative processing (ML?)
- Hard to develop end-to-end data pipelines
  - Stitch together different systems & manage them (e.g. preprocess data in Hadoop and then analyze in Pig or Pregel)
  - Learn different APIs
  - Need to move data between systems

Spark



source: Spark documentation

- Started at UC Berkeley in 2009
  - prof. Ion Stoica
  - Matei Zaharia
  - Mesos: Mesos is built using the same principles as the Linux kernel, only at a different level of abstraction. The Mesos kernel runs on every machine and provides applications (e.g., Hadoop, Spark, Kafka, Elasticsearch) with API’s for resource management and scheduling across entire datacenter and cloud environments.
  - ML students in RADLab got bad performance using Hadoop
- Initial Goals (2009-2010)
  - Support workloads not handled (well) by Hadoop MR (Iterative computations, Interactive processing)
  - Address industry need to do ad-hoc processing
  - Leverage hardware and workload trends (rapid increase in memory capacity, working sets in Big Data clusters fit in memory, 96% of working set at Facebook fit in RAM despite data being 200x larger)
- Open Source: 2010, Apache Project: 2013
- For some time: the most popular and active Big data project
- Databricks

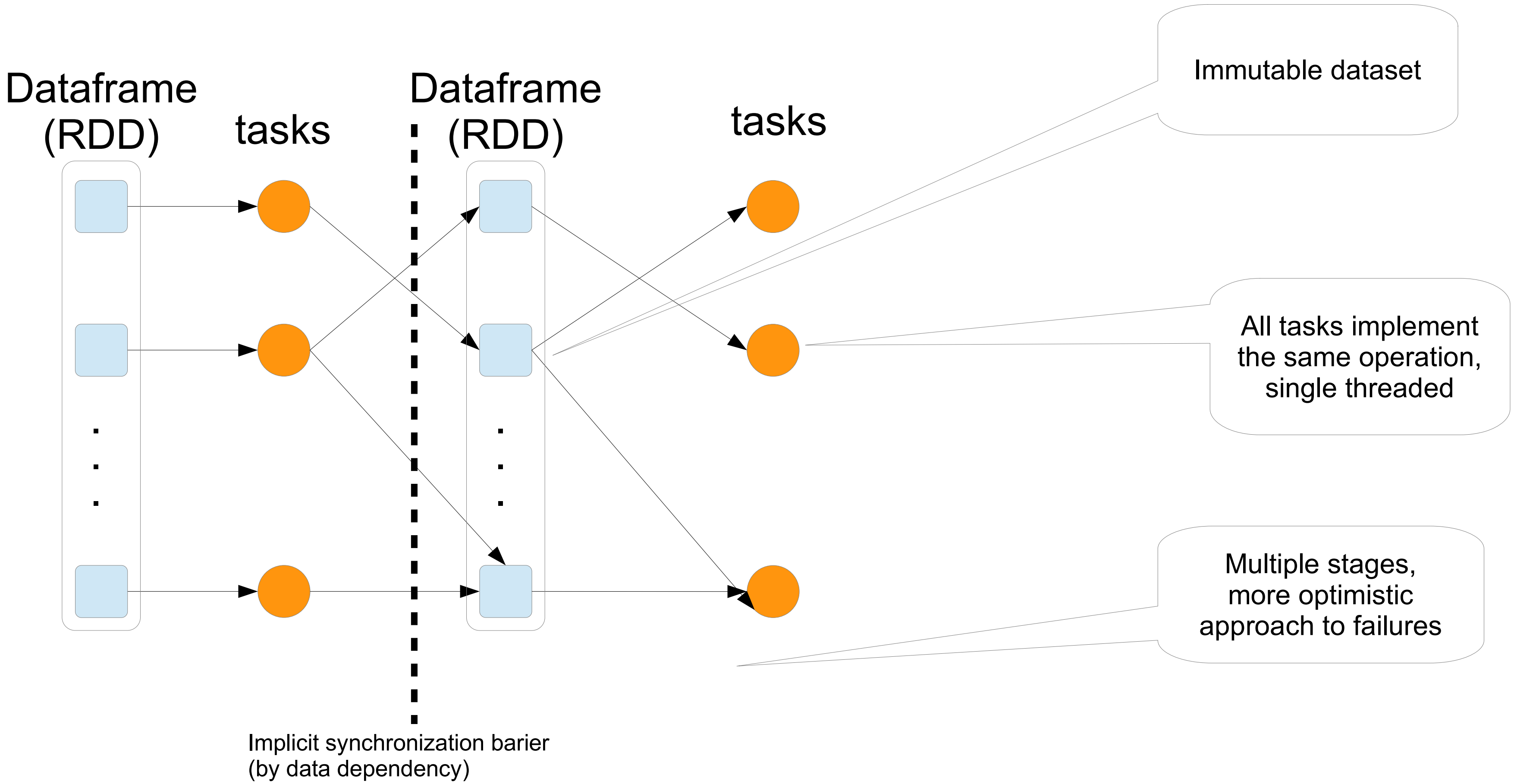
Differences

	Spark	Hadoop (MR)
Computation model	Multi-stage BSP task → thread	Two-stage BSP; task → process (JVM)
Data sharing	in-memory, across stages	on-disk, across jobs
API	Expressive(80+ functions)	Simple (2 functions)

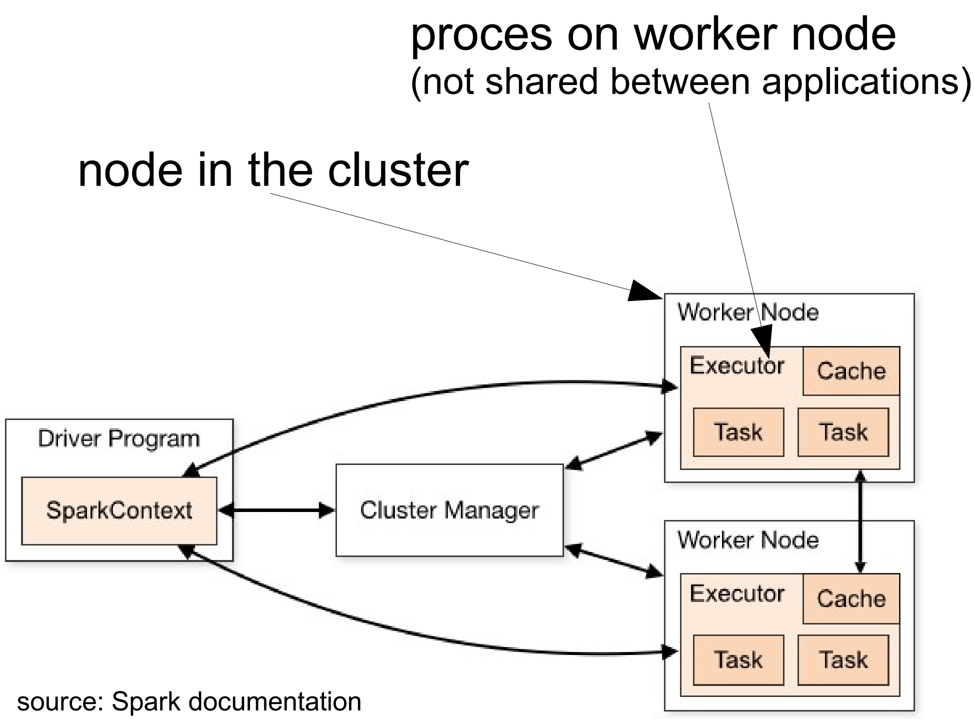
- Immutable datasets
  - Consistency problems go away
  - Fault tolerance much easier
  - Good enough
- RDD: Resilient Distributed Datasets (now replaced by Datasets)
  - Collections of objects partition & distributed across a cluster
  - Stored in RAM or on Disk
  - Resilient to failures
  - Operations
    - Transformations: `map, filter, groupBy` (lazy evaluation)
    - Actions: `count, collect, saveAsTextFile` (triggers computation)

Leslie G. **Valiant**. 1990. **A bridging model for parallel computation**.
Commun. ACM 33, 8 (Aug. 1990), 103–111.
DOI:https://doi.org/10.1145/79173.79181

Spark as BSP



Cluster managers, drivers, executors, ...



source: Spark documentation

- Task: a unit of work
- Job: a parallel computation of multiple tasks spawned in response to `save`, `collect`, etc (see driver's logs)
- Driver program must be network addressable from the worker nodes (best to run it on the same network and connect by RPC)
  - Drivers can be monitored on http://<driver-node>:4040
- Apart from Standarlone can run on cluster managers that support other applications (YARN, Kubernetes)

Examples of launching applications

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a Spark standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a YARN cluster in cluster deploy mode
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000

# Run a Python application on a Spark standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000

# Run on a Kubernetes cluster in cluster deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://xx.yy.zz.ww:443 \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 50 \
  http://path/to/examples.jar \
  1000
```

Shell presentation with examples from http://spark.apache.org/docs/latest/quick-start.html

Creating Dataset:

```
scala> val textFile = spark.read.textFile("README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

In Python called DataFrame to be consistent with the data frame concept in Pandas and R:

```
>>> textFile = spark.read.text("README.md")
```

We get results by calling some actions:

```
scala> textFile.count() // Number of items in this Dataset
res0: Long = 126 // May be different from yours as README.md will change over time, similar to other outputs

scala> textFile.first() // First item in this Dataset
res1: String = # Apache Spark
```

```
>>> textFile.count()  # Number of rows in this DataFrame
126

>>> textFile.first()  # First row in this DataFrame
Row(value=u'# Apache Spark')
```

By transforming a Dataset/DataFrame we create another Dataset/DataFrame:

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
```

We can chain together transformations and actions:

```
scala> textFile.filter(line => line.contains("Spark")).count() // How many lines contain "Spark"?
res3: Long = 15
```

```
>>> textFile.filter(textFile.value.contains("Spark")).count()  # How many lines contain "Spark"?
15
```

Most words in a line:

```
scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
res4: Long = 15
```

```
scala> import java.lang.Math
import java.lang.Math

scala> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
res5: Int = 15
```

The arguments to select and agg are both Column (we can use df.colName to get a column from a DataFrame)
import pyspark.sql.functions to access a lot of convenient functions for transforming Columns

```
>>> from pyspark.sql.functions import *
>>> textFile.select(size(split(textFile.value, "\s+")).name("numWords")).agg(max(col("numWords"))).collect()
[Row(max(numWords)=15)]
```

Caching:

```
scala> linesWithSpark.cache()
res7: linesWithSpark.type = [value: string]

scala> linesWithSpark.count()
res8: Long = 15

scala> linesWithSpark.count()
res9: Long = 15
```

```
>>> linesWithSpark.cache()

>>> linesWithSpark.count()
15

>>> linesWithSpark.count()
15
```

Self contained applications:

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.2.1"
```

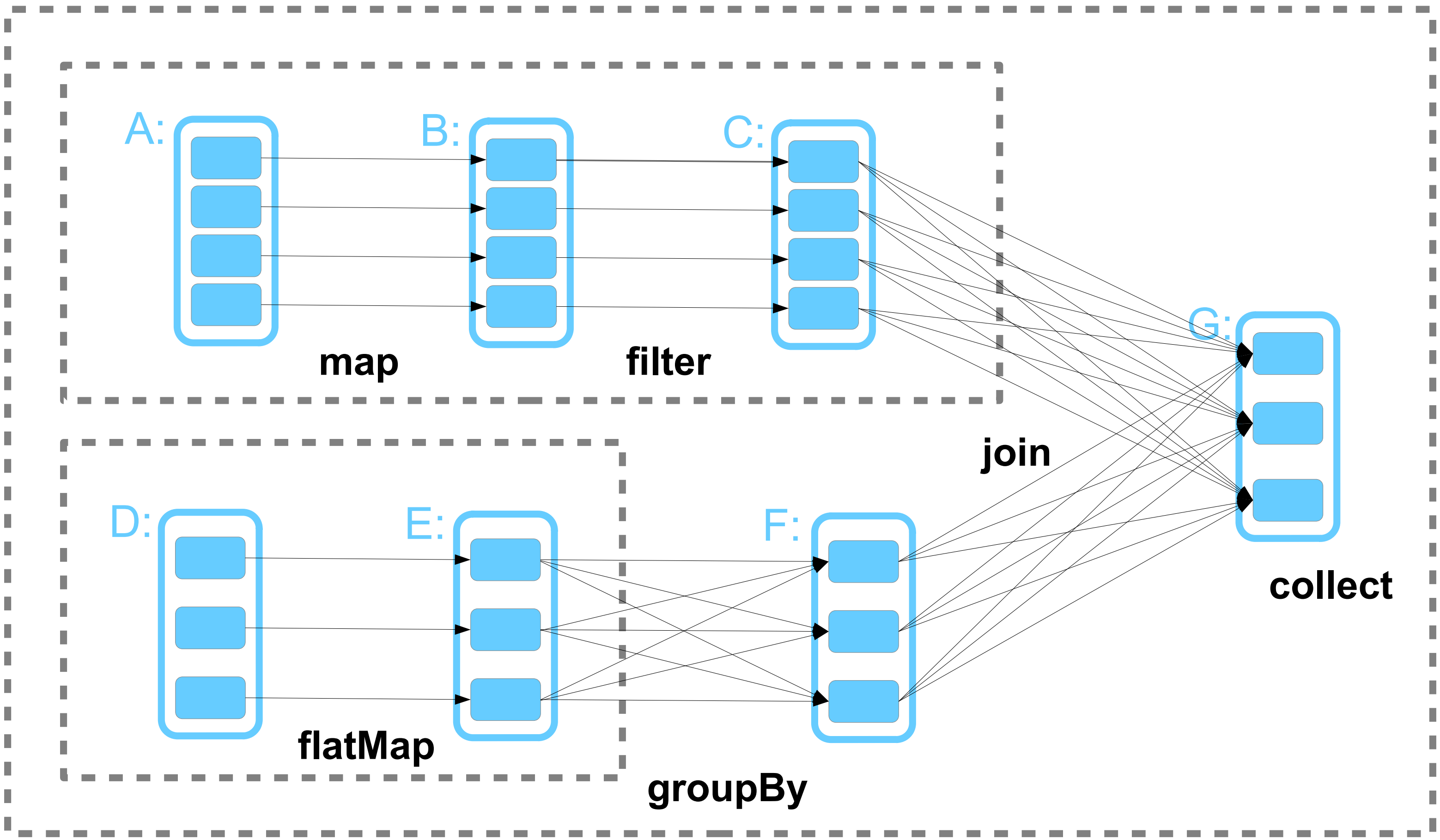
```scala
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

Self contained applications:

```
    install_requires=[
        'pyspark=={site.SPARK_VERSION}'
    ]
```

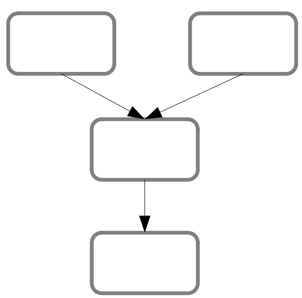
```python
"""SimpleApp.py"""
from pyspark.sql import SparkSession

logFile = "YOUR_SPARK_HOME/README.md"  # Should be some file on your system
spark = SparkSession.builder().appName(appName).master(master).getOrCreate()
logData = spark.read.text(logFile).cache()

numAs = logData.filter(logData.value.contains('a')).count()
numBs = logData.filter(logData.value.contains('b')).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

spark.stop()
```

Partitions and stages (and shuffling)



RDD → Stages → Task

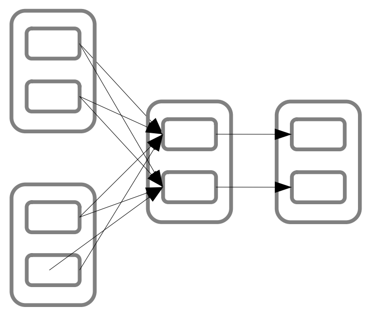
RDD objects



rdd1.join(rdd2)
.groupBy(...)
.filter(...)

results in opertor DAG

DAG scheduler



splits graph into stages of tasks

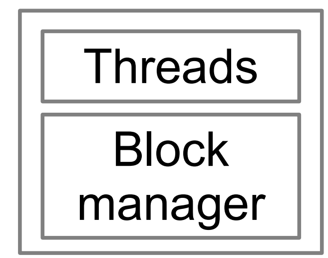
submits each stage as ready

Task scheduler

retry failed or struggling tasks

lunch tasks

Worker



execute tasks

store and serve blocks

Optimization (by program rewriting)

- For each letter find number of distinct products starting with that letter

```
sc.textFile("hdfs://products.txt")
.map(p => (p.charAt(0), p))
.groupByKey()
.mapValues(p => p.toSet.size)
.collect()
```

Optimization (by program rewriting)

- For each letter find number of distinct products starting with that letter

```
sc.textFile("hdfs://products.txt")
.distinct()
.map(p => (p.charAt(0), p))
.groupByKey()
.mapValues(p => p.size)
.collect()
```

Optimization (by program rewriting)

- For each letter find number of distinct products starting with that letter

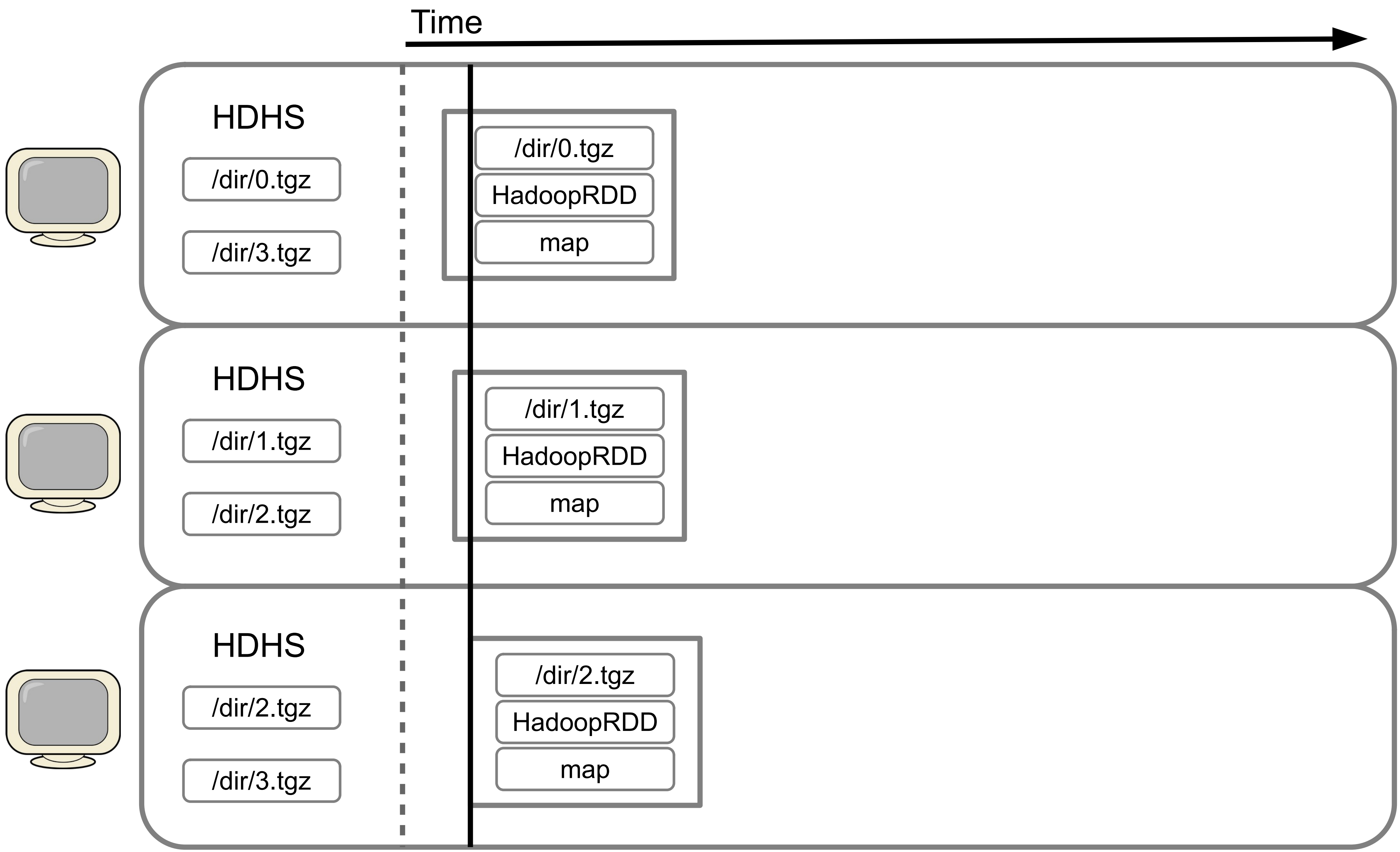
```
sc.textFile("hdfs://products.txt")
.distinct()
.map(p => (p.charAt(0), 1))
.reduceByKey(_+_)
.collect()
```

Optimization (by program rewriting)

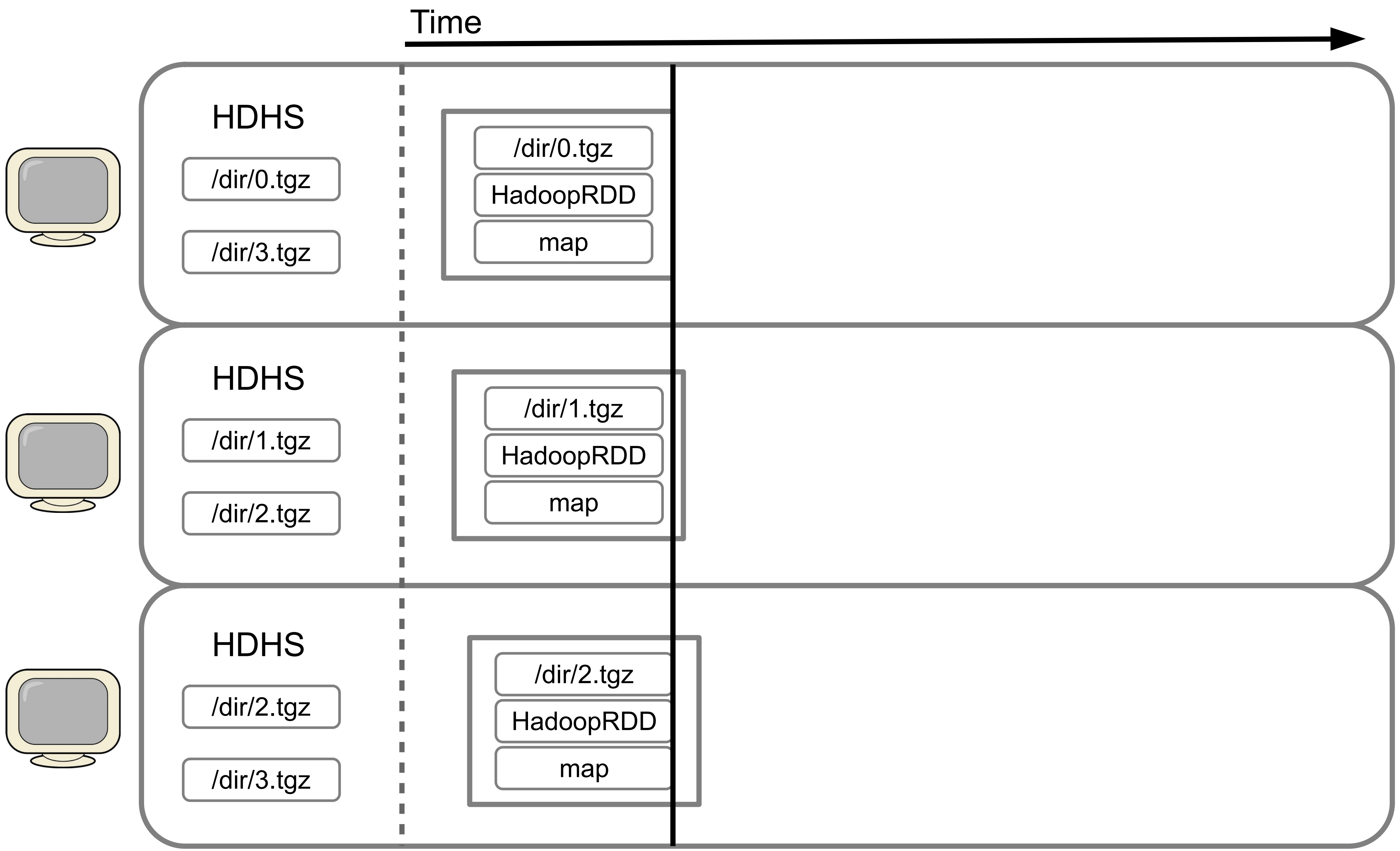
- For each letter find number of distinct products starting with that letter

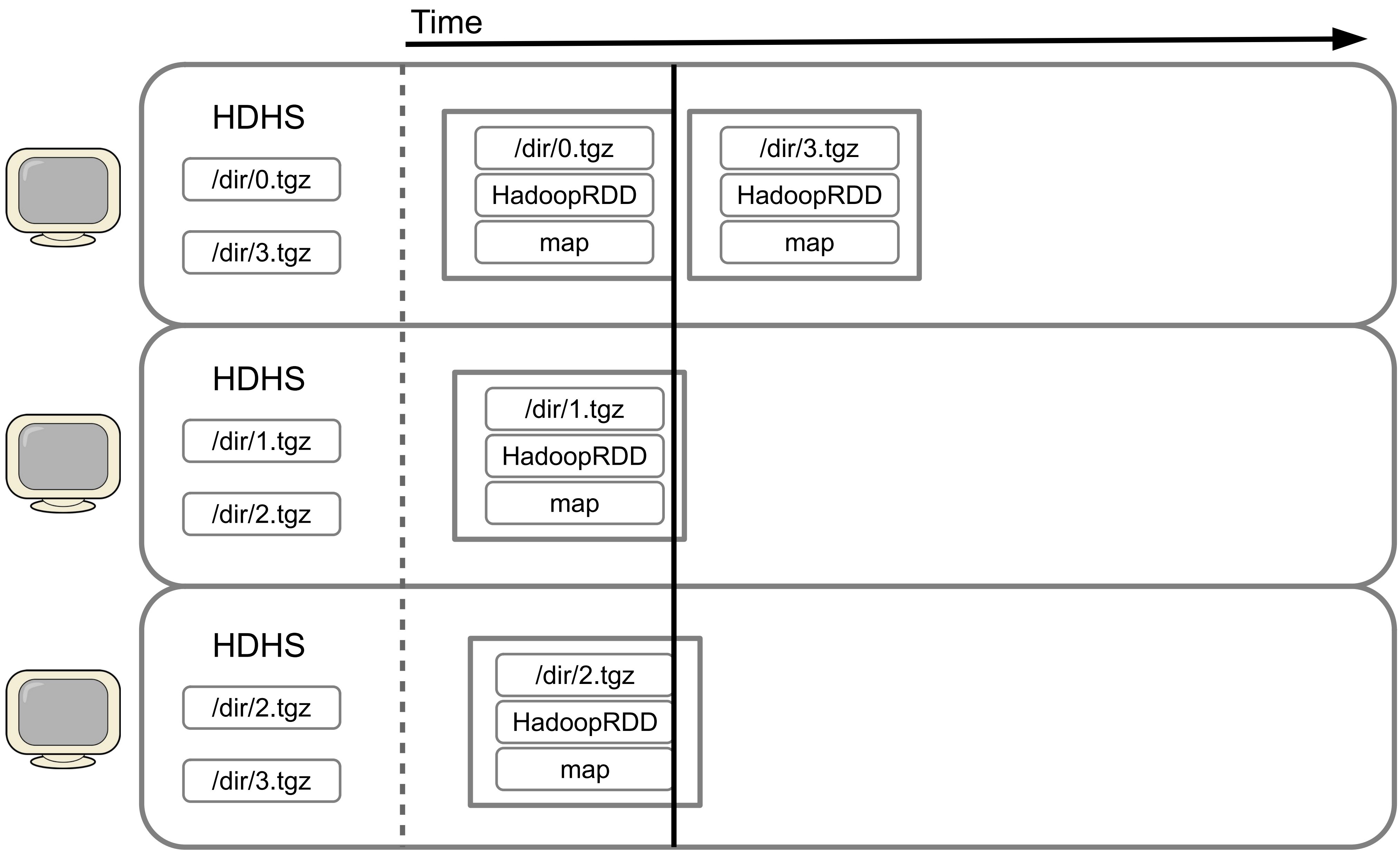
```
sc.textFile("hdfs://products.txt")
.distinct(numPartitions = 100)
.map(p => (p.charAt(0), 1))
.reduceByKey(_+_)
.collect()
```

- How many partitions?
  - More partitions often help with memory errors
  - Reasonable partition processing time (>100ms)
  - 2 to 4 times more than cores

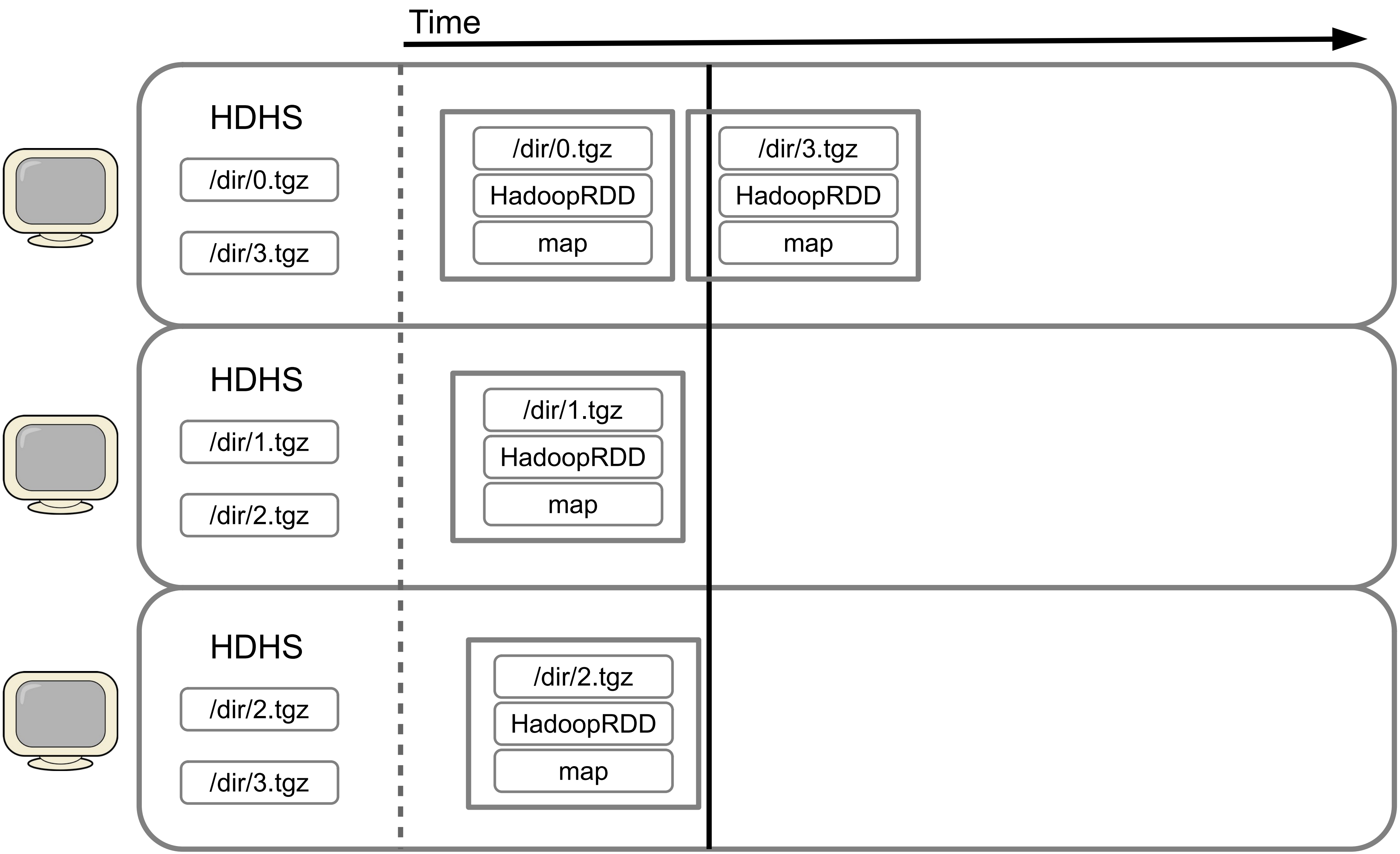


Time
HDHS
/dir/0.tgz
/dir/3.tgz
/dir/0.tgz
HadoopRDD
map
HDHS
/dir/1.tgz
/dir/2.tgz
/dir/1.tgz
HadoopRDD
map
HDHS
/dir/2.tgz
/dir/3.tgz
/dir/2.tgz
HadoopRDD
map


Time
HDHS
/dir/0.tgz
/dir/3.tgz
/dir/0.tgz
HadoopRDD
map
HDHS
/dir/1.tgz
/dir/2.tgz
/dir/1.tgz
HadoopRDD
map
HDHS
/dir/2.tgz
/dir/3.tgz
/dir/2.tgz
HadoopRDD
map


Time
HDHS
/dir/0.tgz
/dir/3.tgz
/dir/0.tgz
HadoopRDD
map
/dir/3.tgz
HadoopRDD
map
HDHS
/dir/1.tgz
/dir/2.tgz
/dir/1.tgz
HadoopRDD
map
HDHS
/dir/2.tgz
/dir/3.tgz
/dir/2.tgz
HadoopRDD
map

How to remedy this type of skew?



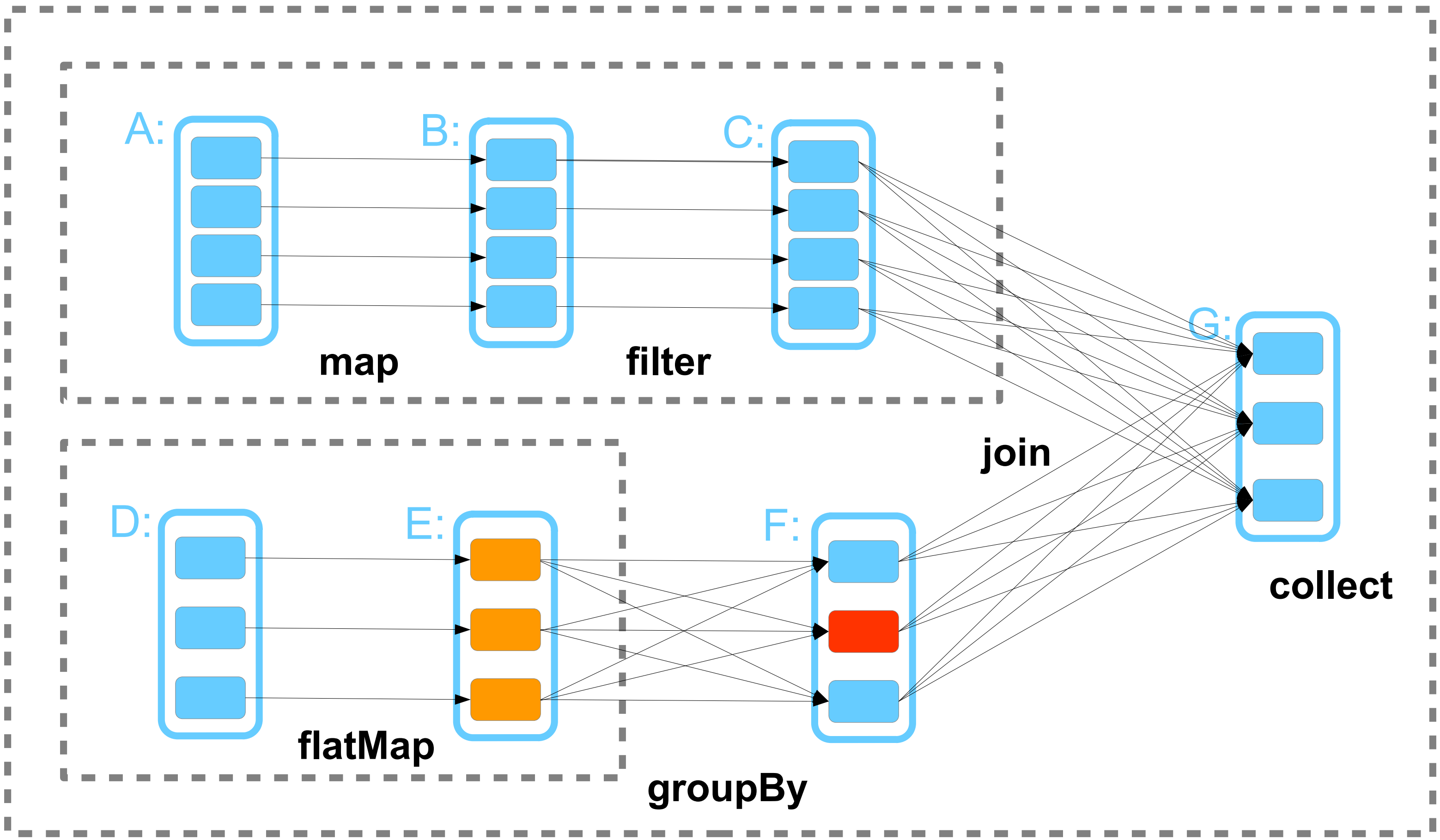
Warning

- But more tasks can increase data transfer for some algorithms
  - Recall our pair processing example (after matrix multiplication)

Fault Recovery: Design Alternatives

- In-memory replication:
  - Slow: need to write data over network
  - Memory inefficient
- Backup on persistent storage:
  - Persistent storage still (much) slower than memory
  - Still need to go over network to protect against machine failures
- Spark choice:
  - Lineage: track computation lineage to reconstruct lost RRD partitions
  - Enabled by deterministic execution and data immutability

Fault recovery with lineage



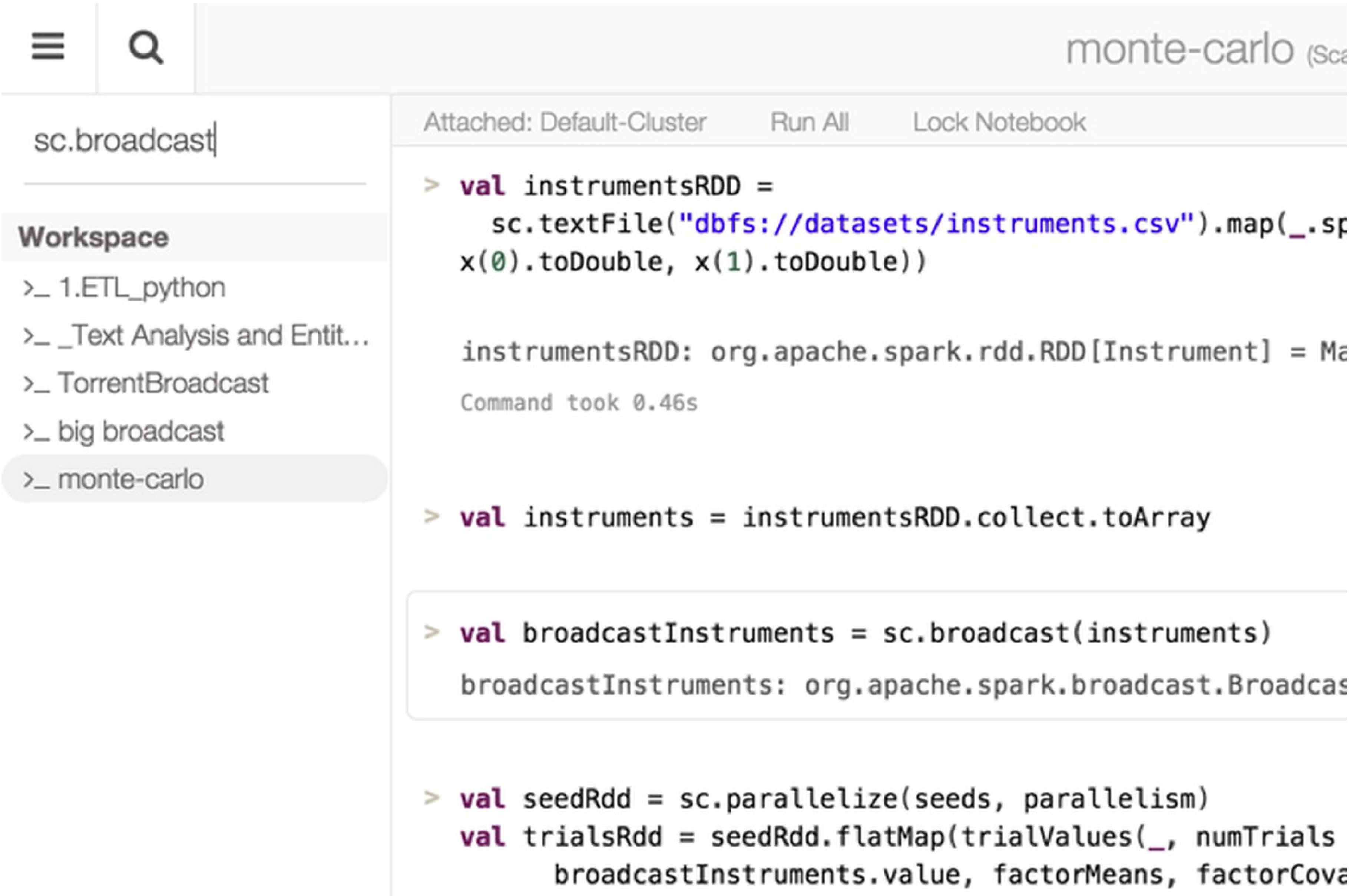
Language support

- Scala
- Java
- Python
  - sometimes not as fast
- R

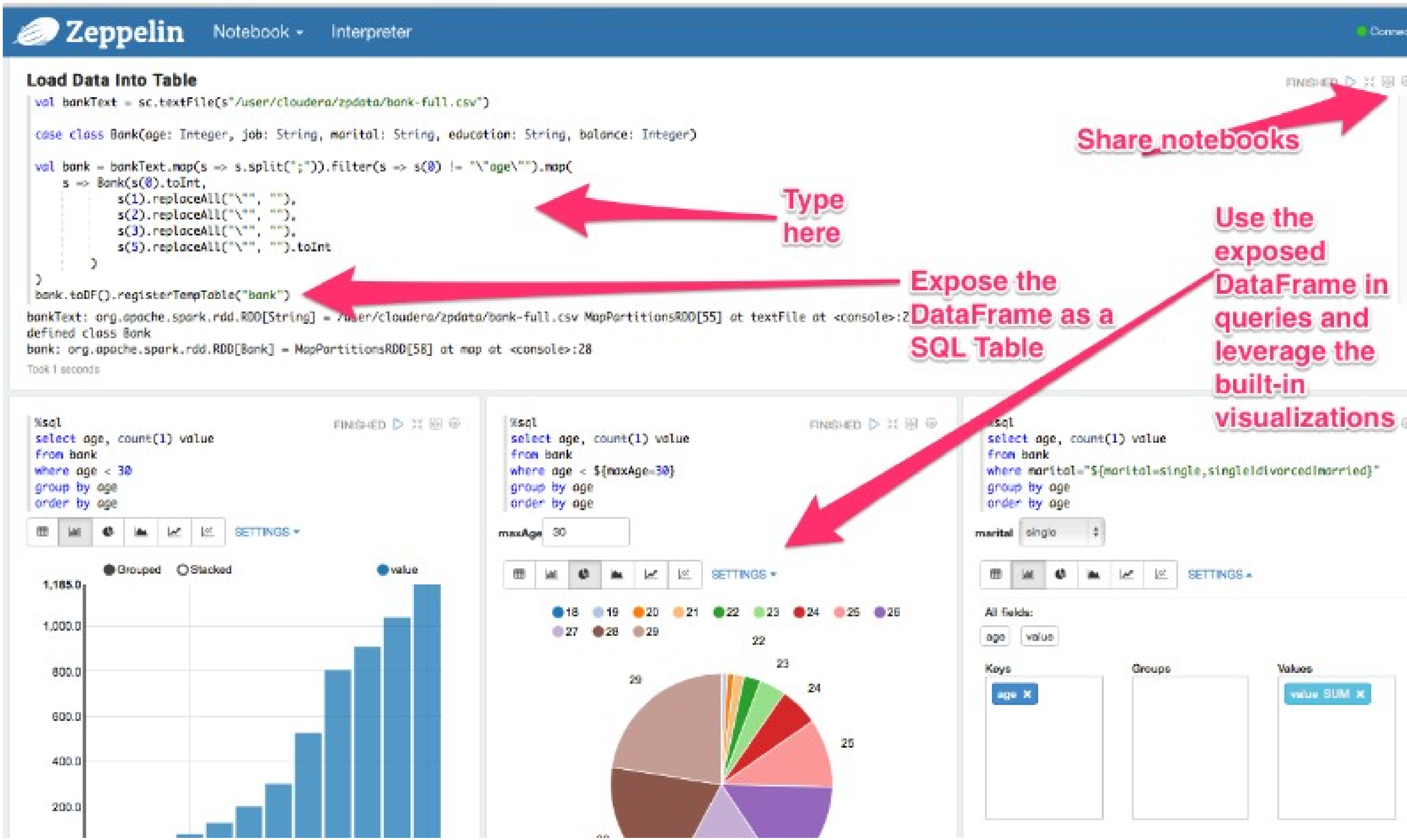
Synergy

- Spark core
  - SaprkSQL
  - Spark Streaming
  - MLlib
  - GraphX
  - SparkR

Databricks Unified Analytics Platform



Apache Zeppelin notebooks



When to use Hadoop/Spark and when not to use it?

- Use when:
  - Data does not fit into RAM and reading it from disk will take too long, e.g., 200TB with 50MB/s takes 4M sec = 46+ days
    - on 1000 nodes only 4000 seconds
  - It is possible to distribute work to many cores without excessive data dependency
  - Want to combine structured queries, machine learning, graph processing, streaming (only Spark)
- Consider other options when:
  - Indexing the data would be useful
    - Initial overhead
    - For many applications directories and filenames are enough
  - It is possible to rent machine with enough RAM (e.g. 4 TB)
    - Especially useful for graph algorithms or algorithms with some global data structure
- You have a problem when:
  - Complexity of the algorithm is worse than linearithmic and you don’t want approximations (note that rate of data production is growing faster than the processing speed)
    - Warning: new measure of algorithm complexity for data intensive distributed processing, see Massively Parallel Communication (MPC) model
  - Consider combining many simple approaches