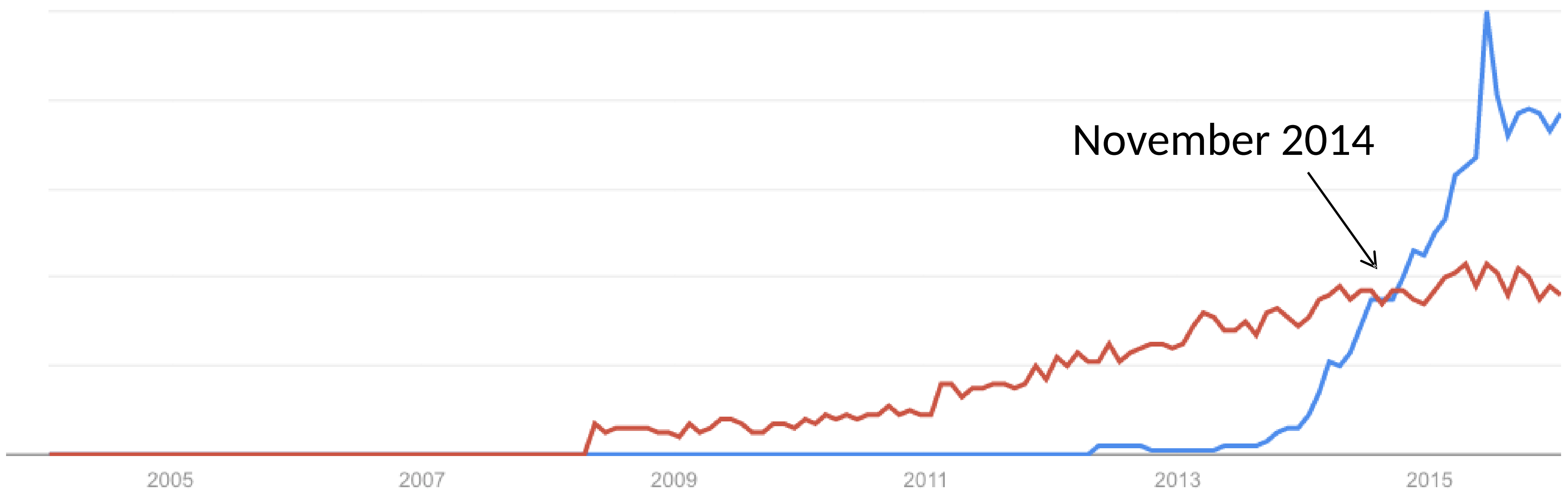
Apache Spark

Adapted from the slides by Dr. Zareen Alamgir

Sources for Slides are Spark 2.0.0 Documentation, Learning Spark(BOOK), Jimmy Lin notes, DataBricks (Holden Karau), Spark Tutorial by tutorialpoint

Spark vs. Hadoop



Google Trends

Source: Datanami (2014): http://www.datanami.com/2014/11/21/spark-just-passed-hadoop-popularity-web-heres/

Why Spark ?

Hadoop is extensively used to analyze data sets

- It provides computing solution that is scalable, flexible, fault-tolerant and cost effective.

Why Spark ?

- Fast Computations (in-memory cluster computing)
- Iterative Algorithms
- Interactive Queries
- Stream Processing
- Graph Algorithms
- Reduce management burden of maintaining separate tools.

Why Spark ?

Hadoop is great, but it's really way too low level!

(circa 2007)

What's the solution?

Design a higher-level language

What’s Spark

Apache Spark is a fast cluster computing platform designed for fast computation.

Built on top of Hadoop MapReduce and extends it to efficiently use more types of computations

- Interactive Queries
- Stream Processing

Up to 100 times faster in memory, and 10 times faster when running on disk.

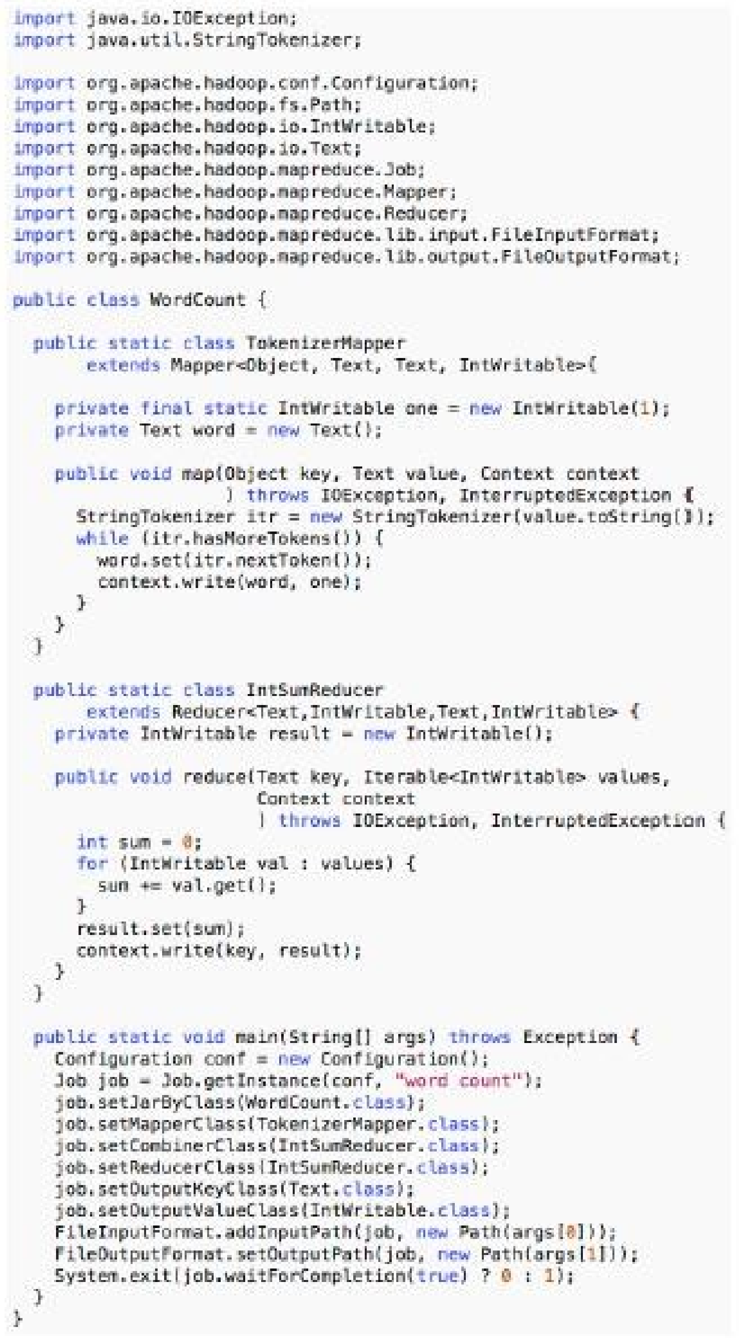
Difficulty of Programming in MR

Word Count implementations

- Hadoop MR – 61 lines in Java
- Spark – 1 line in interactive shell

```
sc.textFile('...').flatMap(lambda x: x.split())
    .map(lambda x: (x, 1)).reduceByKey(lambda x, y: x+y)
    .saveAsTextFile('...')
```

VS

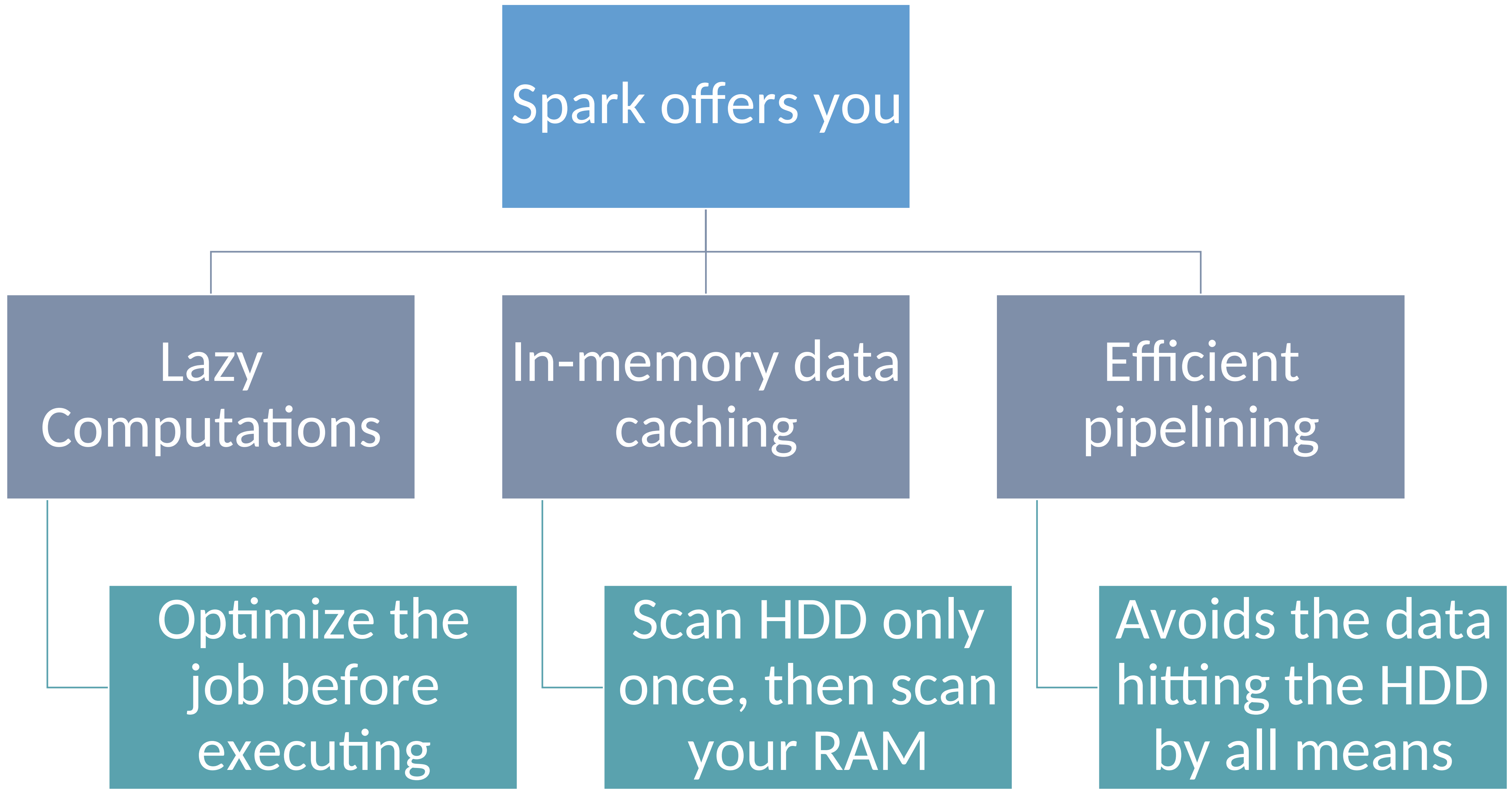


```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Limitations of Map Reduce

Map Reduce is designed for Batch Processing

- Cannot handle interactive queries
- Cannot handle iterative tasks
- Cannot handle stream processing


Spark offers you
Lazy Computations
In-memory data caching
Efficient pipelining
Optimize the job before executing
Scan HDD only once, then scan your RAM
Avoids the data hitting the HDD by all means

Spark & Hadoop

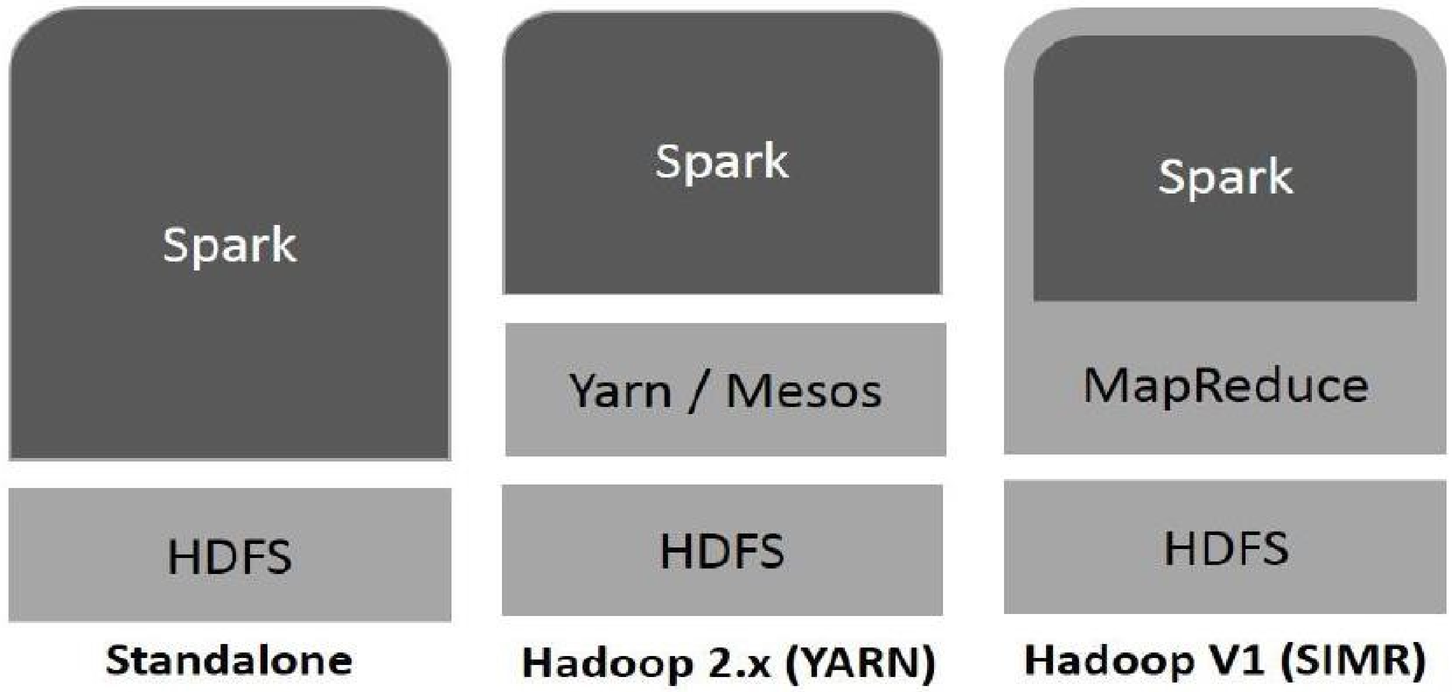
Spark is not a modified version of Hadoop nor is dependent on Hadoop

- It has its own cluster management.
- Hadoop is just one of the ways to implement Spark.

Spark uses Hadoop in two ways

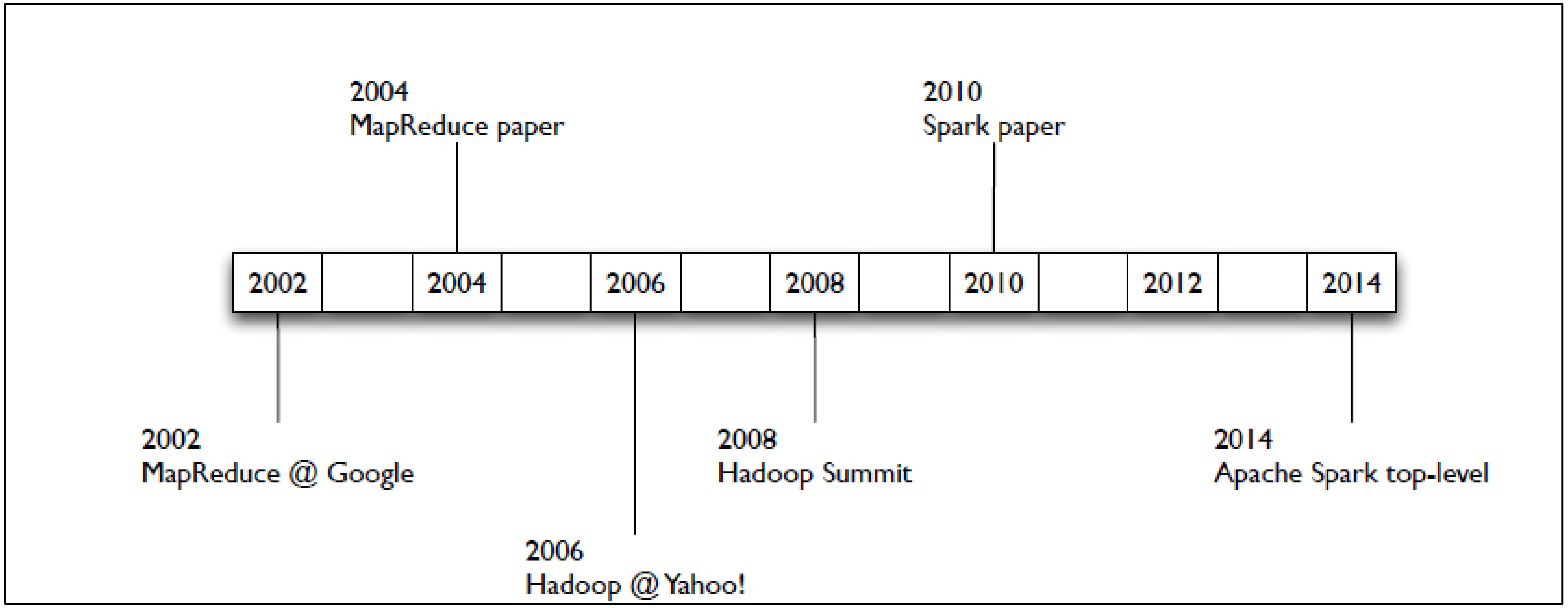
- one is **storage** and
- second is **processing**.

Spark Built on Hadoop



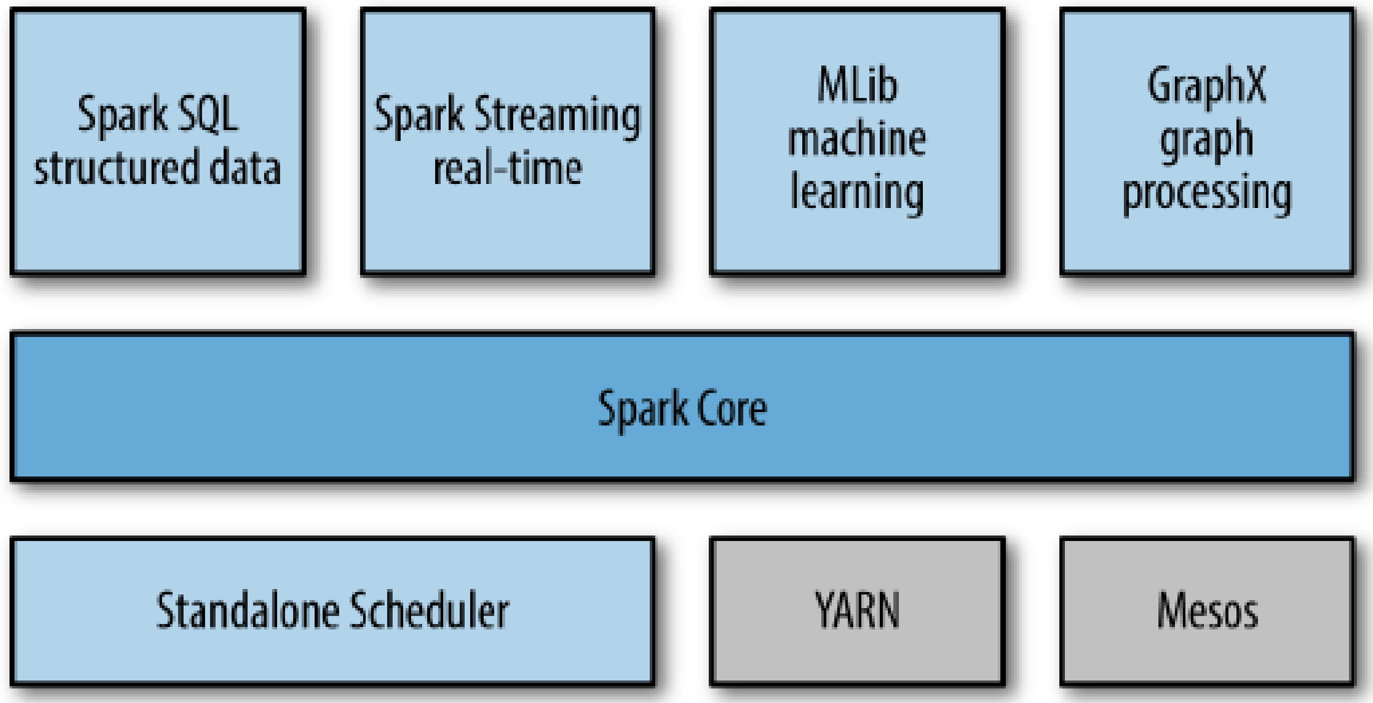
Brief History

- Developed at UC Berkeley AMPLab in 2009
- Open-sourced in 2010
- Commercial support provided by DataBricks



Components of Spark

- Spark core is a “computational engine”
  - Deals with scheduling, distributing, and monitoring applications across many worker machines, or a *computing cluster*
- Tight integration helps build applications that combine different processing models



Resilient Distributed Dataset (RDD)

- Fundamental data structure of Spark
- Immutable distributed collection of objects
- Partitioned collection of records that can be operated on in parallel

Spark use the concept of RDD to achieve faster and efficient MapReduce operations.

Let’s design a data processing language “from scratch”!

(Why is MapReduce the way it is?)

When and why they are not so efficient.

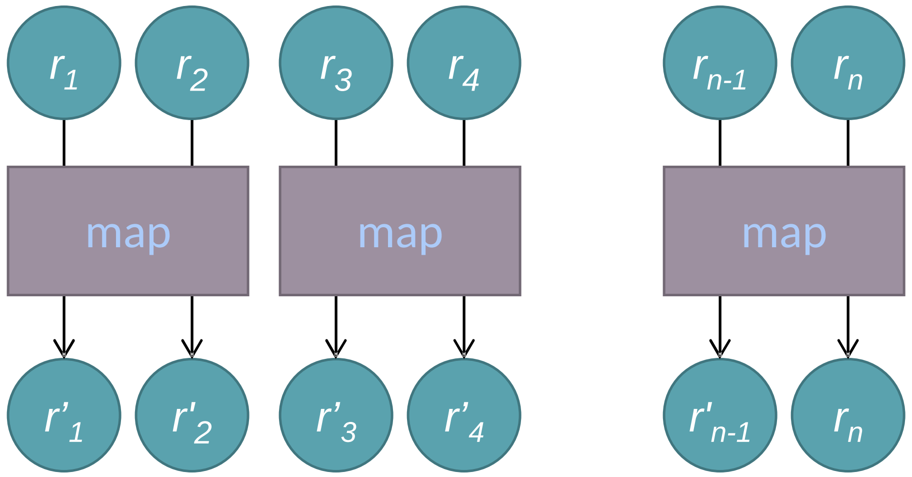
Data-Parallel Dataflow Languages

We have a collection of records,
want to apply a bunch of transformations
to compute some result

Assumptions: static collection,
records (not necessarily key-value pairs)

We need per-record processing

(note, not necessarily key-value pairs)



Remarks: Easy to parallelize maps,
record to “mapper” assignment is an implementation detail

(If we want more than embarrassingly parallel processing)

Map alone isn't enough

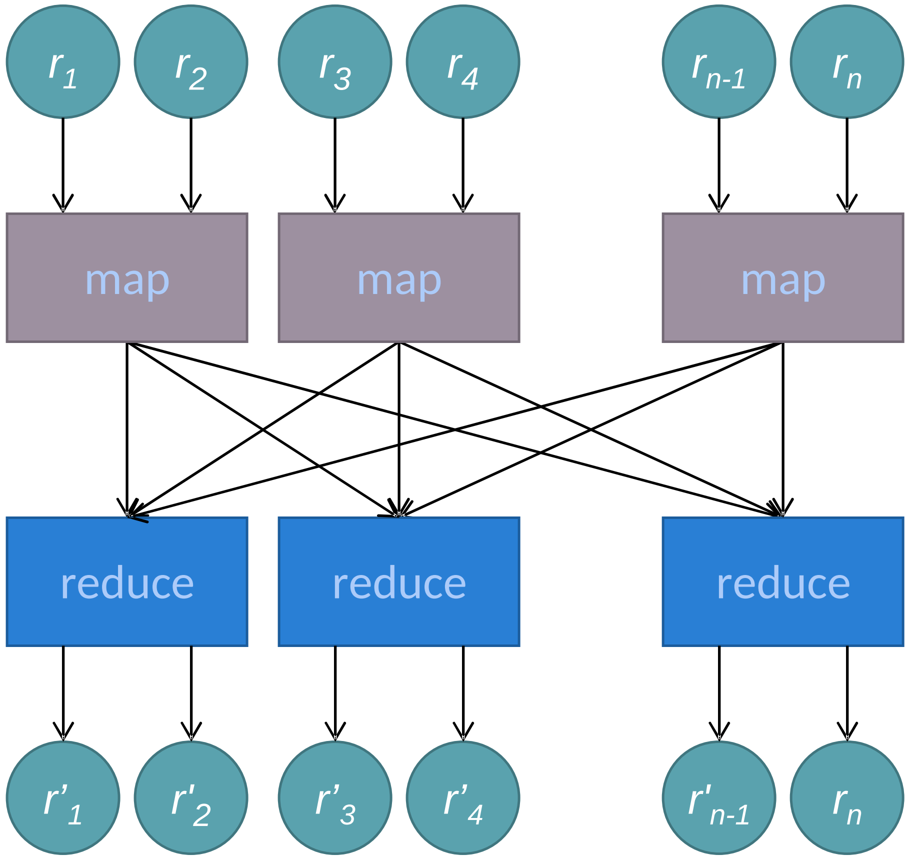
We need a way to group partial results

Intermediate (key, value) pairs

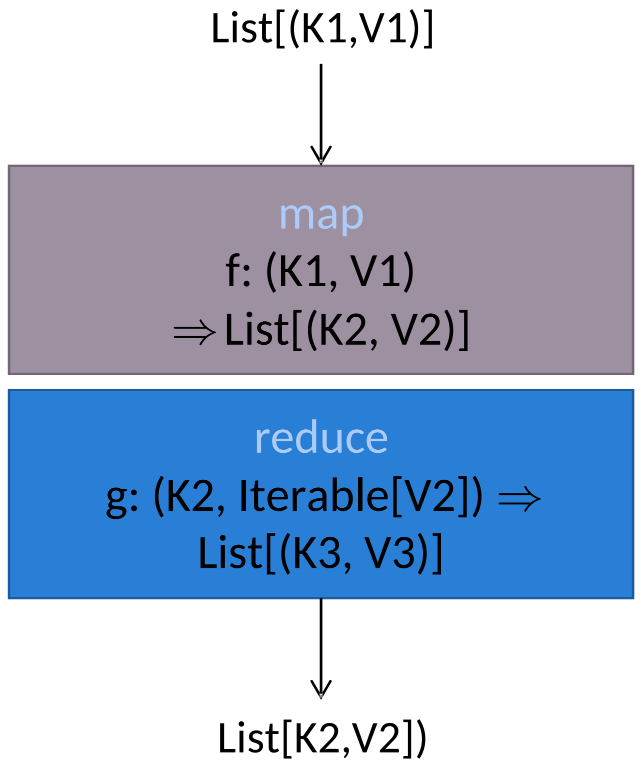
grouping key   partial result

For each key, we can apply some computation

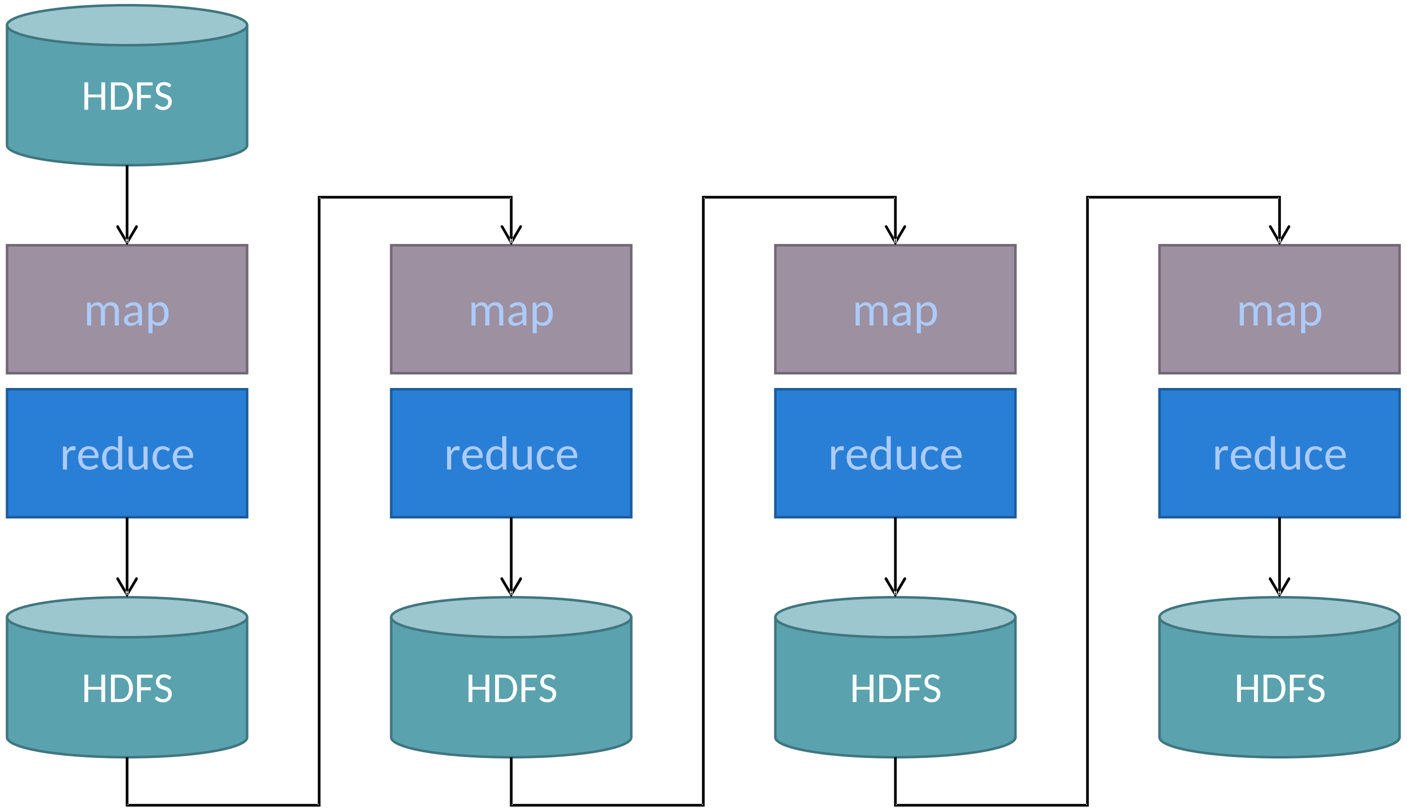
MapReduce


r_1
r_2
r_3
r_4
r_{n-1}
r_n
map
map
map
reduce
reduce
reduce
r'_1
r'_2
r'_3
r'_4
r'_{n-1}
r_n

MapReduce

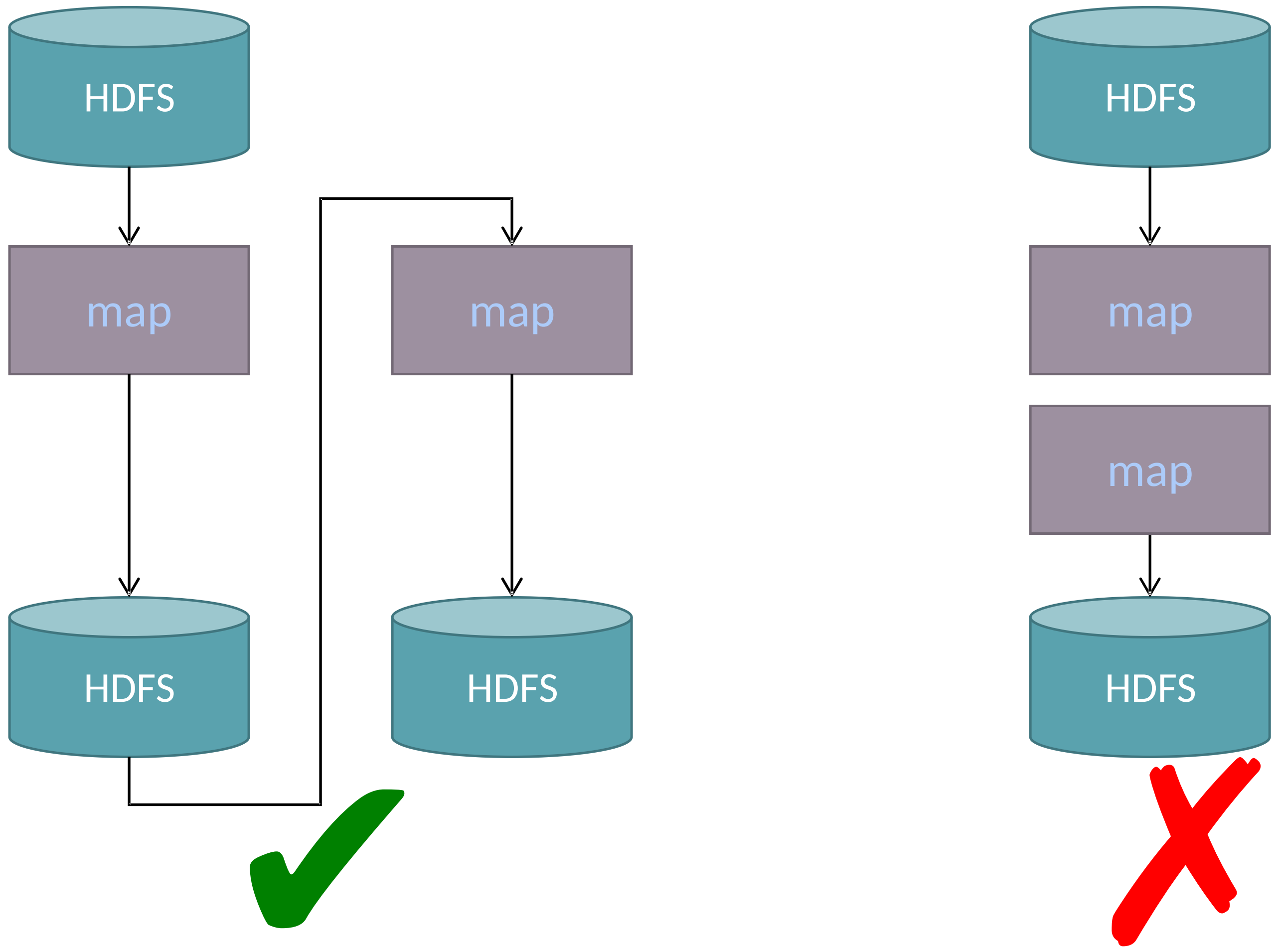

List[(K1,V1)]
map
f: (K1, V1)
⇒ List[(K2, V2)]
reduce
g: (K2, Iterable[V2]) ⇒
List[(K3, V3)]
List[K2,V2])

MapReduce Workflows

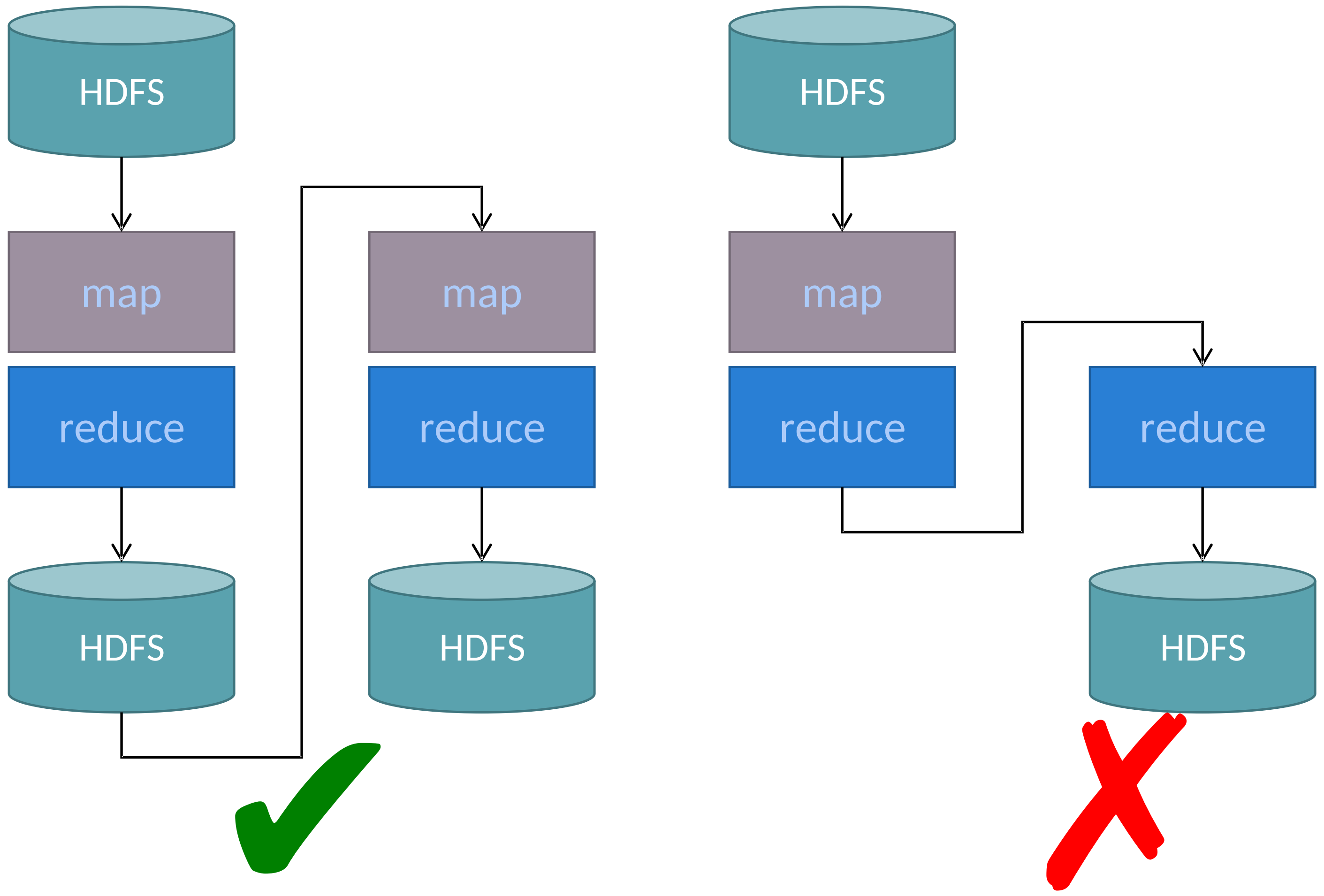


What’s wrong?

Want MM...?



Want MRR?


HDFS
map
reduce
HDFS
map
reduce
HDFS
✓
HDFS
map
reduce
reduce
HDFS
✗

Iterative Operations on MapReduce

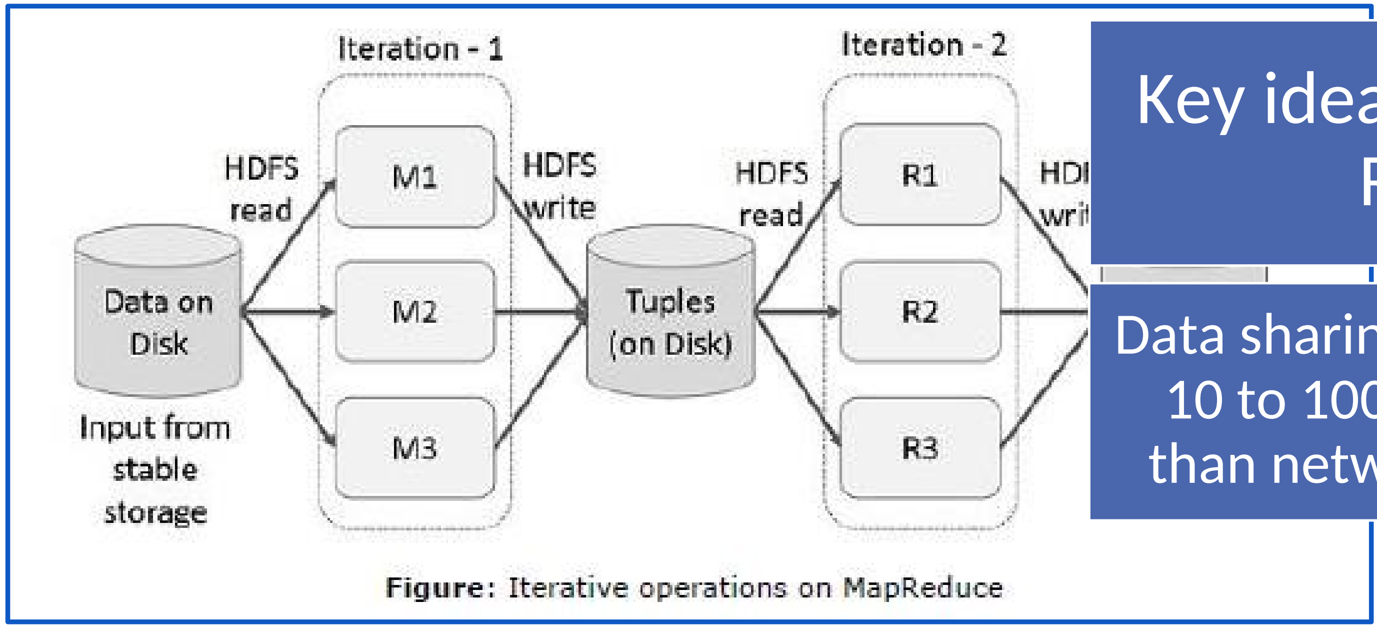


Figure: Iterative operations on MapReduce

Key idea of spark is RDD

Data sharing in memory is 10 to 100 times faster than network and Disk.

RDD supports in-memory processing

- it stores the state of memory as an object across the jobs
- the object is sharable between those jobs.

Iterative Operations on Spark RDD

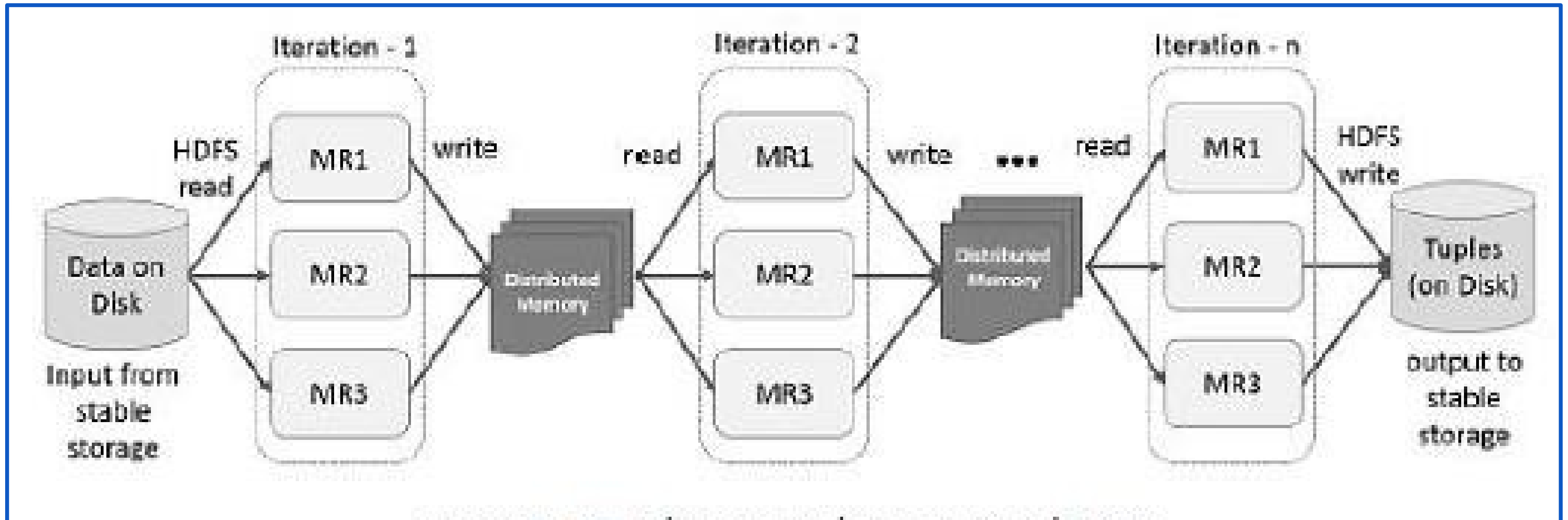


Figure: Iterative operations on Spark RDD

Interactive Operations

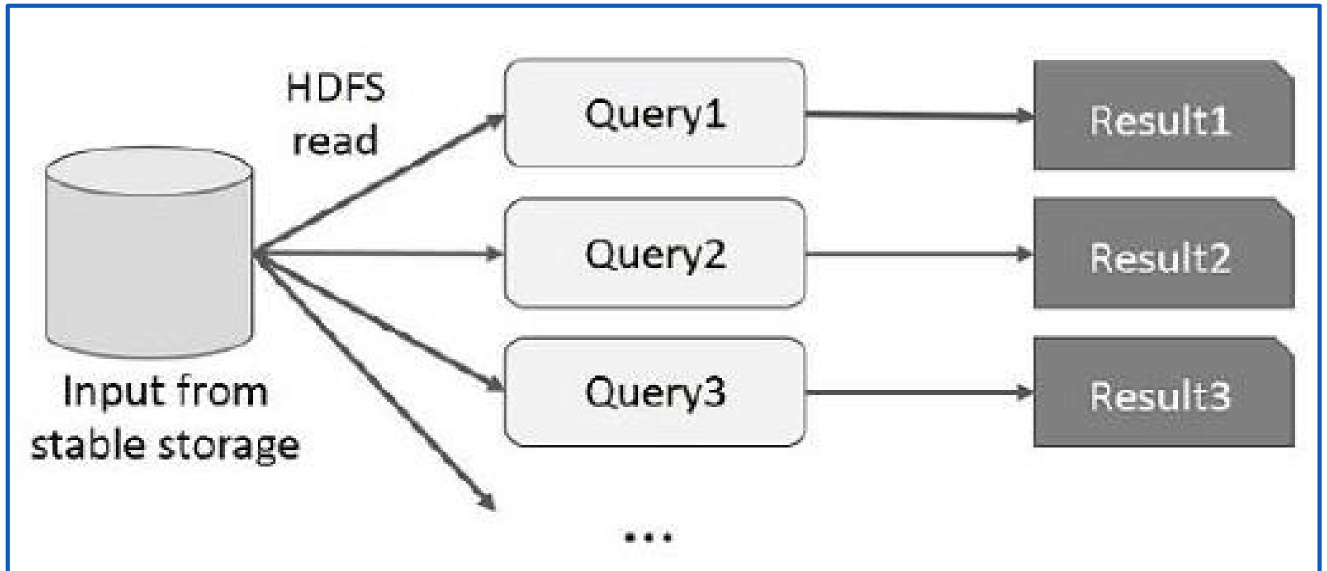


Figure: Interactive operations on MapReduce

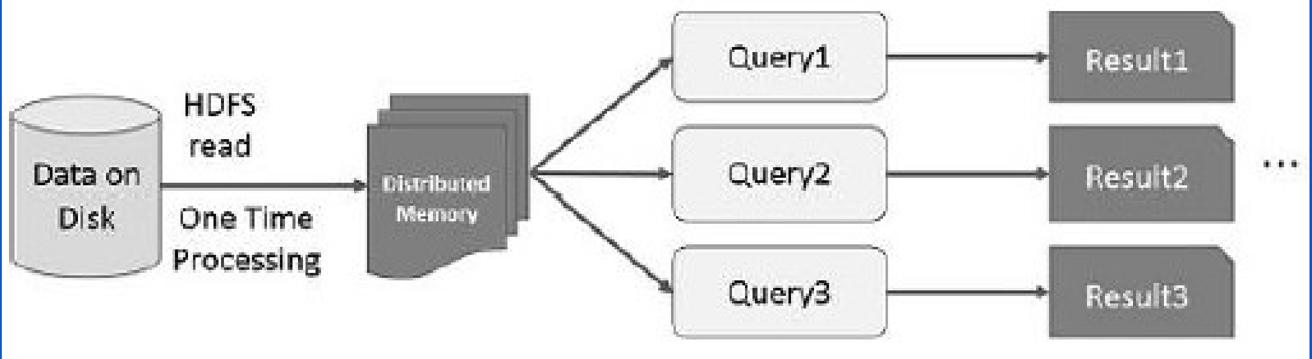


Figure: Interactive operations on Spark RDD

Interactive operations on Spark RDD

For different queries on the same data set keep data in memory

RDD's can be recomputed when an action is run on it

RDD can be persisted in memory, or on disk, or replicated across multiple nodes

Spark-Resilient Distributed Datasets (RDD)

- An RDD is simply a distributed collection of elements
    - Core abstraction
    - Immutable
    - Partitioned collection of objects spread across a cluster,
        - stored in RAM or on disk
    - Achieve fault tolerance through lineage
        - if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.

Spark-RDD

- Simple view
  - RDD is collection of data items split into partitions and stored in memory on worker nodes of the cluster

- Complex view
  - RDD is an interface for data transformation
  - RDD refers to the data stored either in persisted store (HDFS, Cassandra, HBase, etc.) or in cache (memory, memory+disks, disk only, etc.) or in another RDD

Spark-Operations on RDD

- In Spark all work is expressed as either
  - creating new RDDs,
  - transforming existing RDDs, or
  - calling operations on RDDs to compute a result.

- Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them

Spark-**Transformations**

- *Transformations* construct a new RDD from a previous one.
- Example of transformation is filtering data that matches a predicate.

RDD transformations returns pointer to new RDD and allows you to create dependencies between RDDs.

Transformations in Spark are "lazy". They do not compute results immediately.

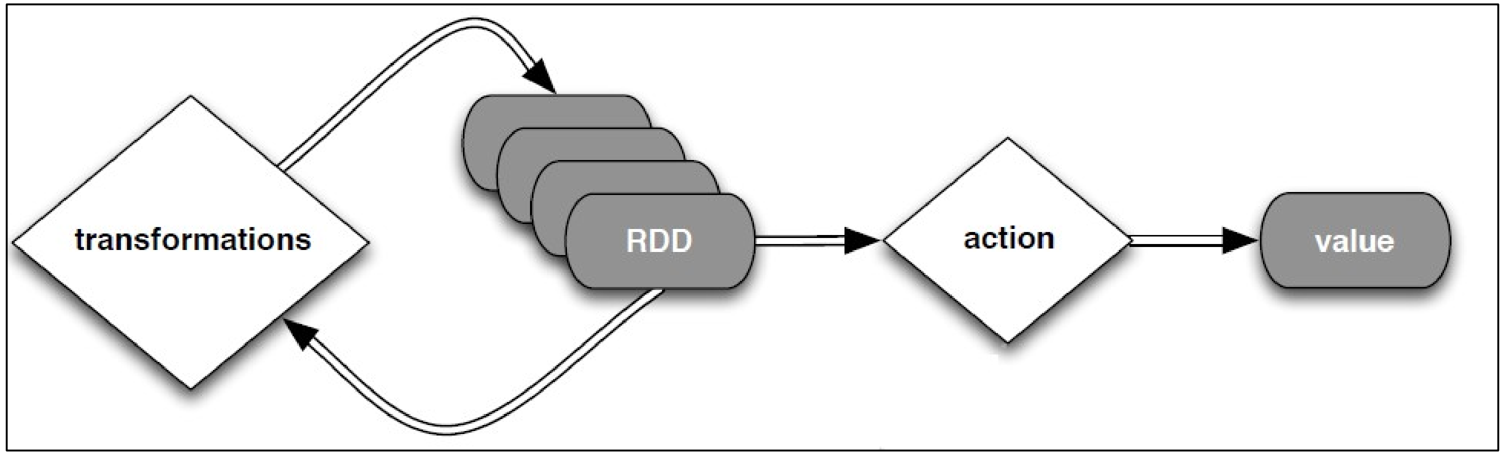
They just "remember" the operation to be performed and the dataset (e.g., file) to which the operation is to be performed

This design enables Spark to run more efficiently. WHY?

$$\begin{tikzcd}
\boxed{\text{RDD[T]}} \arrow[d] \\
\boxed{\begin{array}{c}\textbf{filter} \\ \text{f: (T)} \Rightarrow \\ \text{Boolean}\end{array}} \arrow[d] \\
\boxed{\text{RDD[T]}}
\end{tikzcd}$$

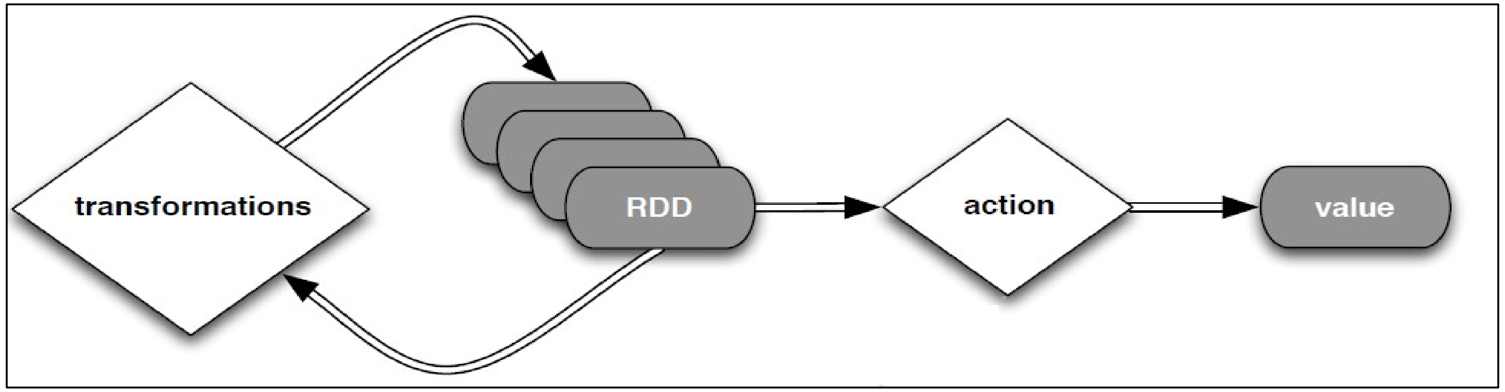
Spark-Actions

- *Actions* compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).
- Examples of actions include
  - count (which returns the number of elements in the dataset),
  - collect (which returns the elements themselves), and
  - save (which outputs the dataset to a storage system)



Spark-Transformation vs Actions

- Transformations and actions are different because of the way Spark computes RDDs
    - Spark computes RDDs in a *lazy* fashion—that is, the first time they are used in an action.
- Spark's RDDs are by default recomputed each time you run an action on them.
    - To reuse an RDD in multiple actions, you can ask Spark to *persist* it using **RDD.persist()**



Create RDD

- Spark provides an interactive shell: a powerful tool to analyze data interactively

Open spark shell

```
$ spark-shell
```

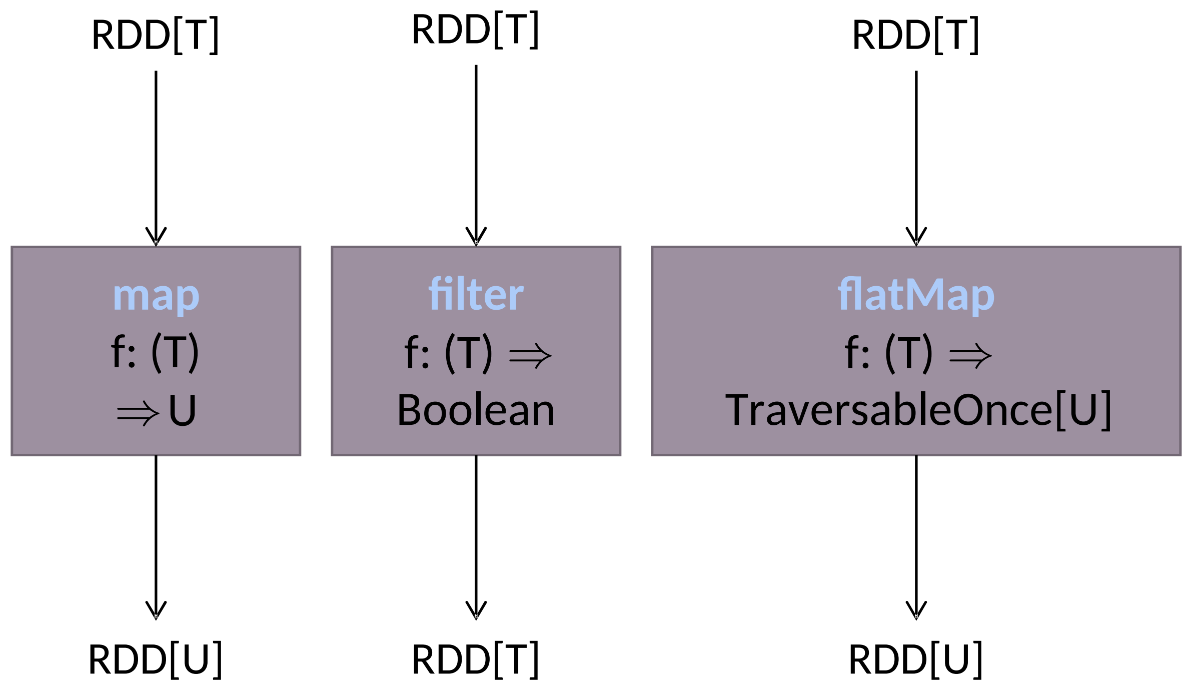
- RDDs can be created from
  - Hadoop Input Formats (such as HDFS files) or
  - Transforming other RDDs

Quick Scala Tutorial

Create Simple RDD

```
scala> val inputfile = sc.textFile(“input.txt”)

scala> sc.parallelize (List(1,2,3))
```

RDD Transformations



Map-like Operations

(Not meant to be exhaustive)

Spark Basic Transformations

- val rdd = sc. parallelize(List(1,2,3))
- val rdd2 = rdd.filter(x=> x%2 == 0)
  - Pass each element through a function, keep those fulfil the predicate
  - rdd2 will be {2}

- val inputRDD = sc.textFile("log.txt")
- val errorsRDD = inputRDD.filter(line => line.contains("error"))

Spark Basic Transformations

- val rdd = sc.parallelize(List(1,2,3))
- val rdd1 = rdd.map(x=>x*2)
  - Pass each element through a function, rdd1 will be {1 , 4, 6}
- val rdd2 = rdd.flatMap(x=> 0 to x)
  - Map each element to 0 or more other elements and flatten the result into one array,
  - rdd2 will be { 0, 1, 0, 1, 2, 0, 1, 2, 3)
- val rdd3 = rdd. map(x=> 0 to x)
  - Map each element to 0 or more other elements and returns a list of arrays
  - rdd3 will be { 0, 1}, {0, 1, 2} , {0, 1, 2, 3)

Spark Basic Transformations

- val sData = sc.parallelize(List("Summer days", "Autumn hues", "Winter cold", "Spring flowers"))

- val D3= sData.flatMap(x=>x.split(" "))
  - **D3 is** {Summer, days, Autumn, hues, Winter, cold, Spring, flowers}

- val D4 = sData.map(x=>x.split(" "))
  - **D4 is** Array(Summer, days), Array(Autumn, hues), Array(Winter, cold), Array(Spring, flowers)

map() vs flatMap()

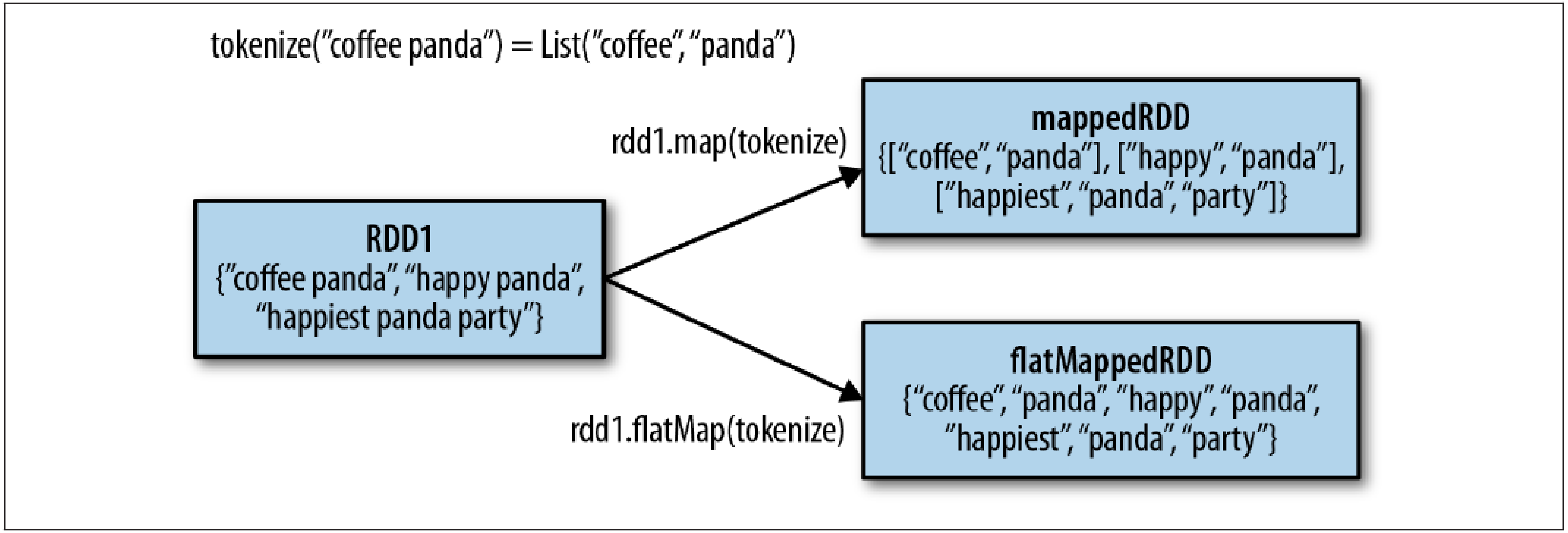
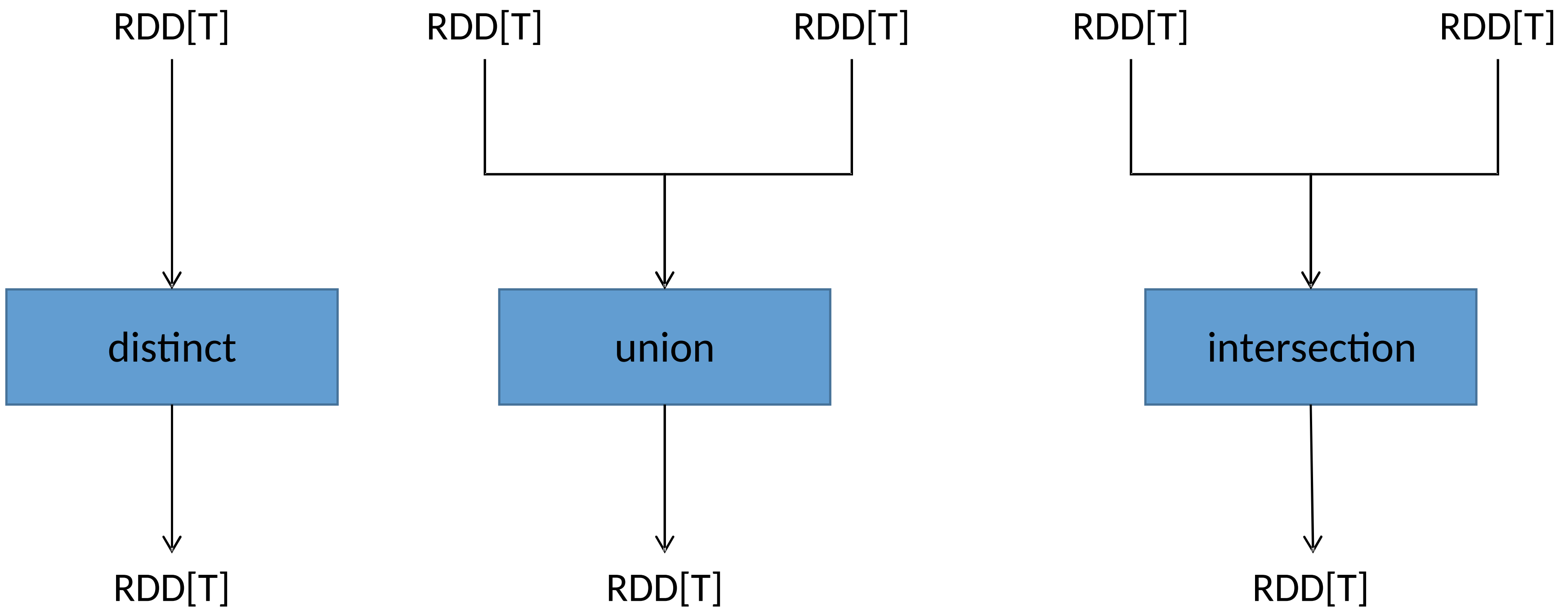

tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
rdd1.flatMap(tokenize)
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

Figure 3-3. Difference between flatMap() and map() on an RDD

Set Operations



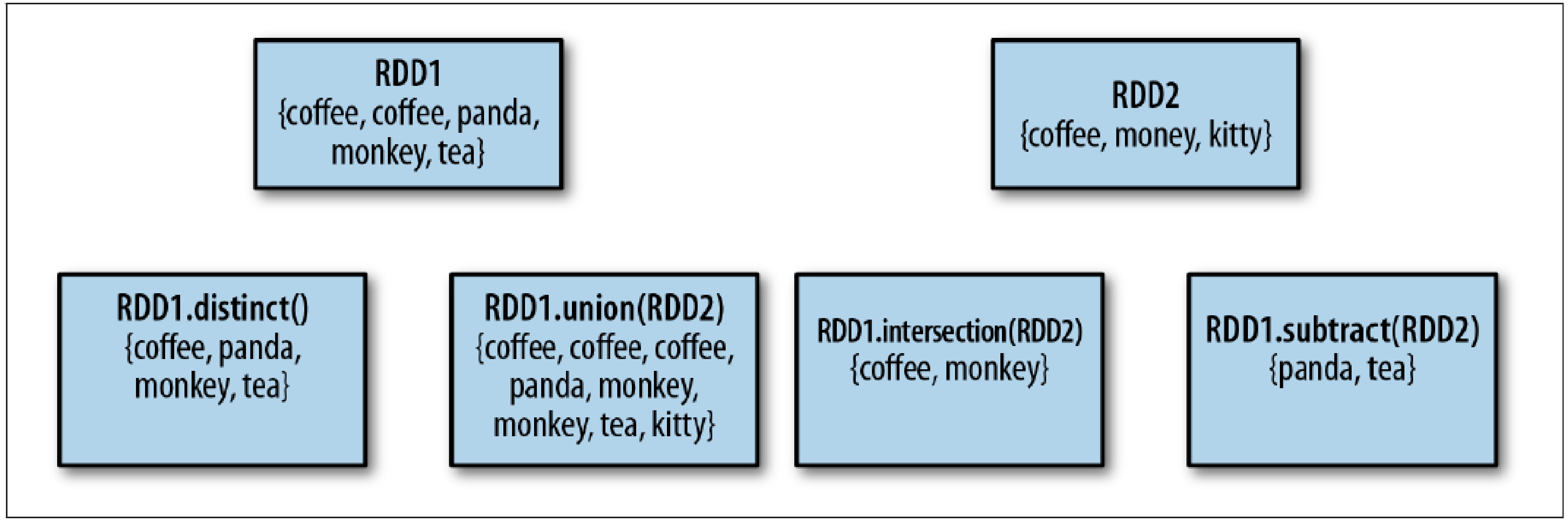


Figure 3-4. Some simple set operations

The set property most frequently missing from UNION is the uniqueness of elements.

The performance of intersection() is much worse than union() since it requires a shuffle over the network to identify common elements.

Set Operations

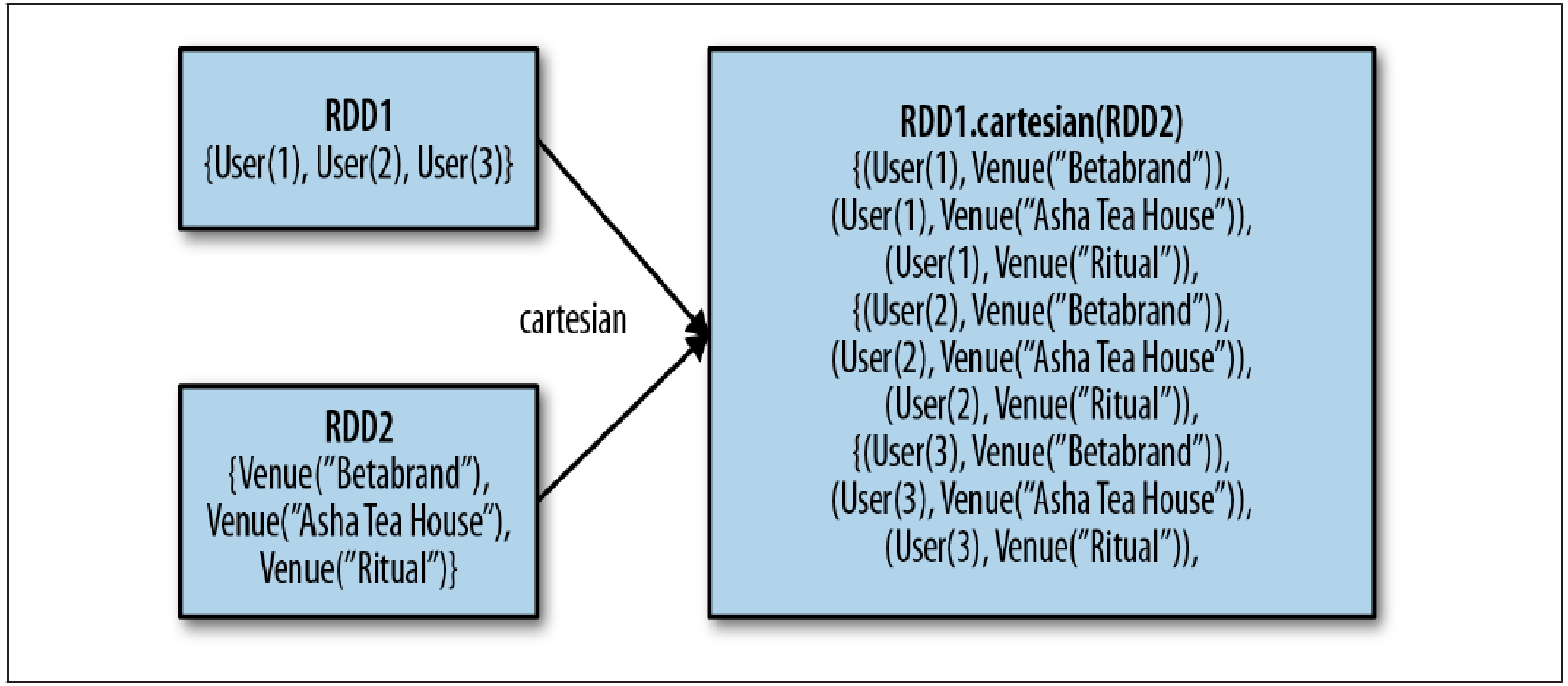
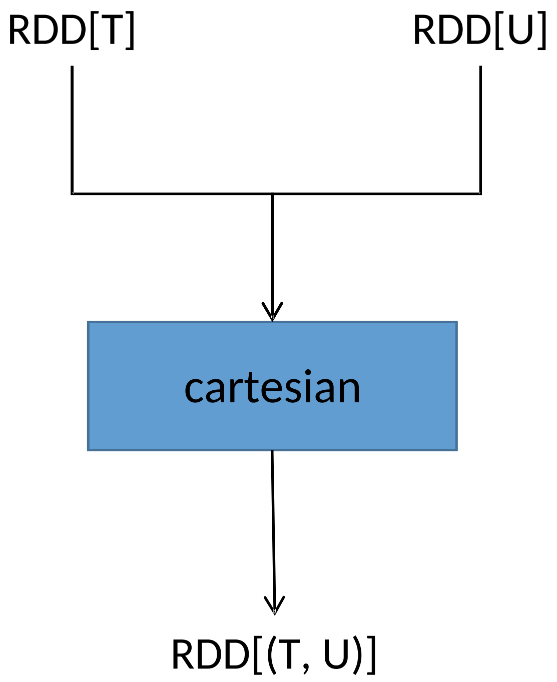

RDD1
{User(1), User(2), User(3)}
RDD2
{Venue("Betabrand"), Venue("Asha Tea House"), Venue("Ritual")}
cartesian
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

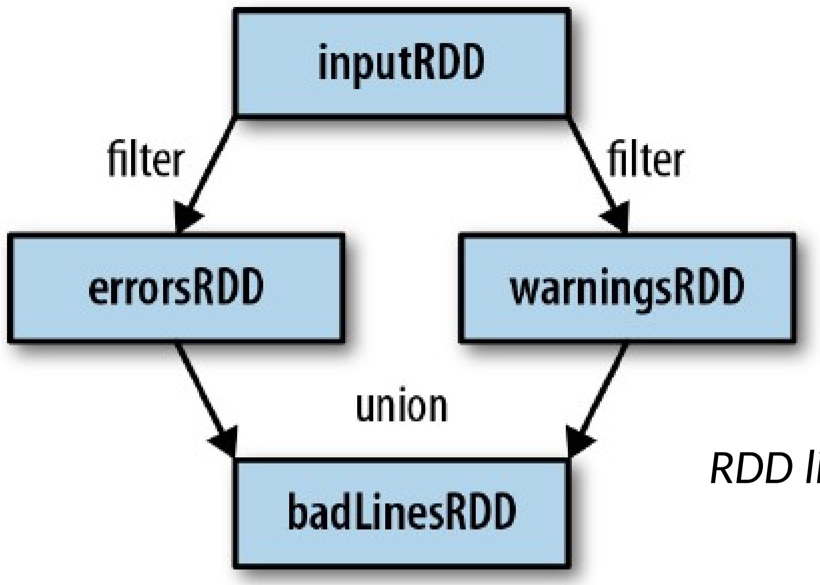
Figure 3-5. Cartesian product between two RDDs



(Not meant to be exhaustive)

Transformations And Lineage Graph

- Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*.
- It uses *lineage graph* to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost.

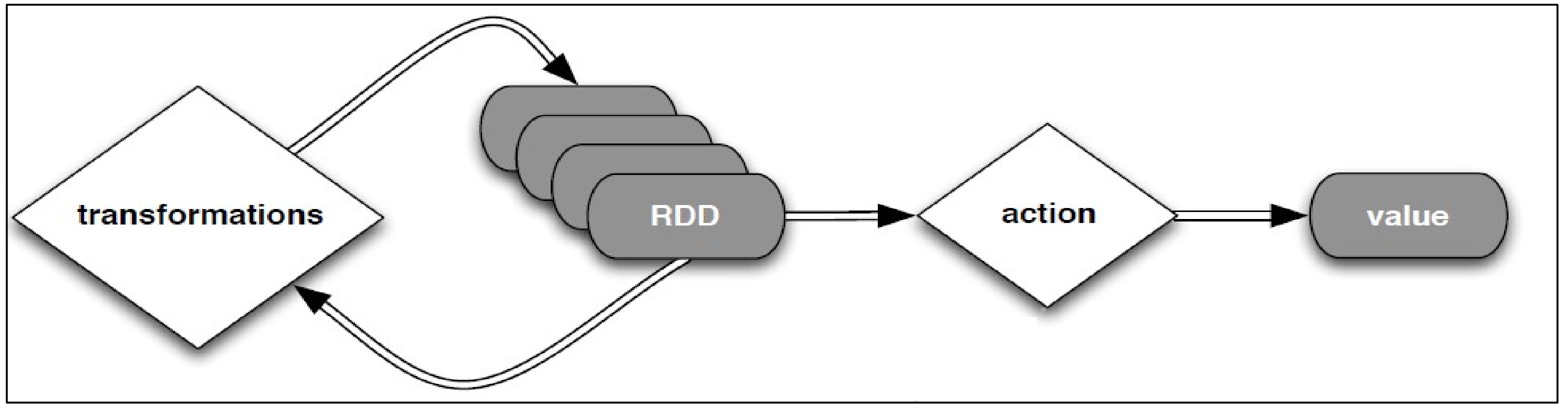


RDD lineage graph created during log analysis

ACTIONS

Actions

- Actions force the evaluation of the transformations required for the RDD they were called on, to produce output.



Basic Actions

- val nums = sc.parallelize(List(1,2,3,3,4))
- val odd=nums.filter(x =>x%2==1)
- odd.collect()
  - retrieve the entire RDD
- val rddT= nums.take(2)
- rddT.foreach(println)

Keep in mind that your entire dataset must fit in memory on a single machine to use collect() on it, so collect() shouldn't be used on large datasets.

take(n) returns *n* elements from the RDD

It's important to note that these operations do not return the elements in the order you might expect.

foreach(func) apply the provided function to each element of the RDD.

Basic Actions

- val nums = sc.parallelize(List(1,2,3,3,4))

1. nums.saveAsTextFile(“filename.txt")
2. nums.foreach(println)
3. nums.count()
4. nums.countByValue()
   - Number of times each element occurs in the RDD.
   - {(1, 1),(2, 1),(3, 2),(4,1)}
5. val sum = nums. reduce((x, y) => x + y)
   - Combine the elements of the RDD together in parallel (e.g., sum).

Basic Actions

- val nums = sc.parallelize(List(1,2,3,3,4))

- val agg = nums.aggregate((0,0)) (
  ```
  (x, value)=>(x._1 + value, x._2 + 1),
  (x1, x2)=>(x1._1 + x2._1, x1._2 + x2._2))
  println(agg)
  ```
- val average = agg._1 /agg._2.toDouble
- println(average)

Computing sum and count in one action

- First provide a function to combine the elements from our RDD with the accumulator.
- Next we need a second function to merge two accumulators

Lazy Evaluation

Transformations on RDDs are lazily evaluated

- A transformation is not immediately performed.
- Spark records metadata to indicate that this operation has been requested
- **For example**: when we call sc.textFile(), the data is not loaded until it is necessary.

WHY?

- Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together

Persistence (Caching)

- val result = nums.map(x => x*x)
- println(result.count())
- println(result.collect().mkString(","))

Especially expensive for iterative algorithms

Issue ??

RDD is computed multiple times

Solution??

Persist the data

Persistence (Caching)

- val result = nums.map(x => x*x)
- result.persist()
- println(result.count())
- println(result.collect().mkString(","))

What happened when a node that has data persisted on it fails ?

Spark will recompute the lost partitions of the data

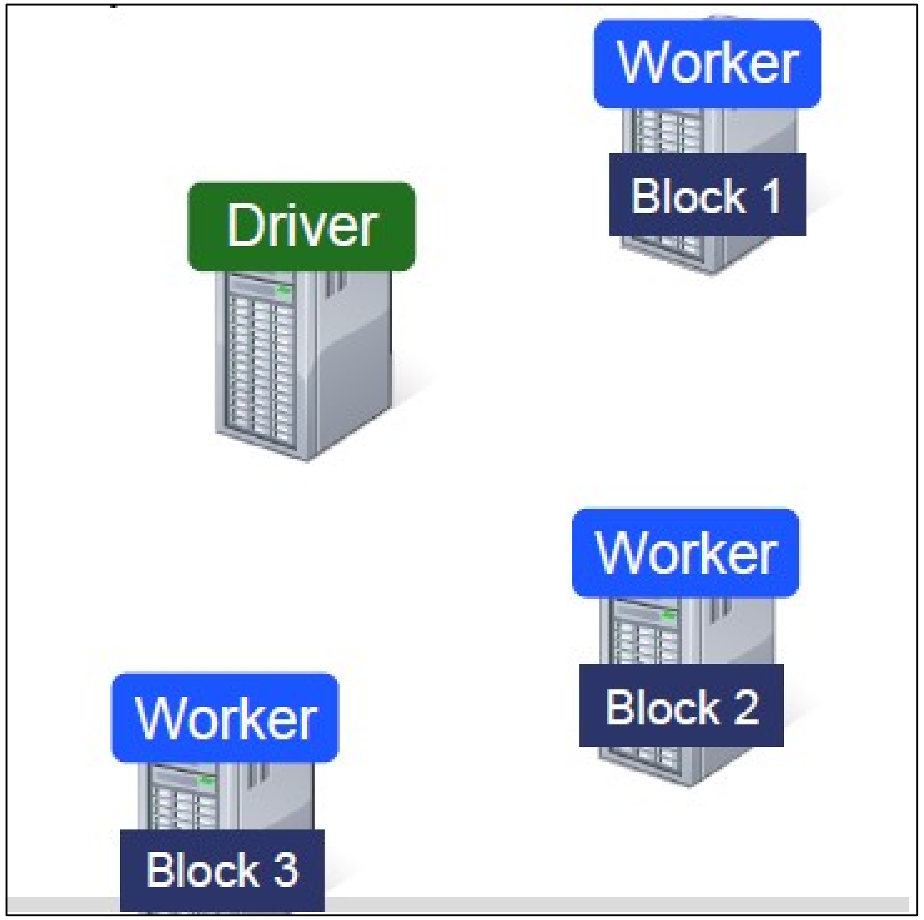
Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

```
import org.apache.spark.storage.StorageLevel
result.persist(StorageLevel.DISK_ONLY)
```

Example: Log Mining

- Load error messages from a log into memory and search for various patterns
- Data is separated into partitions on different machines

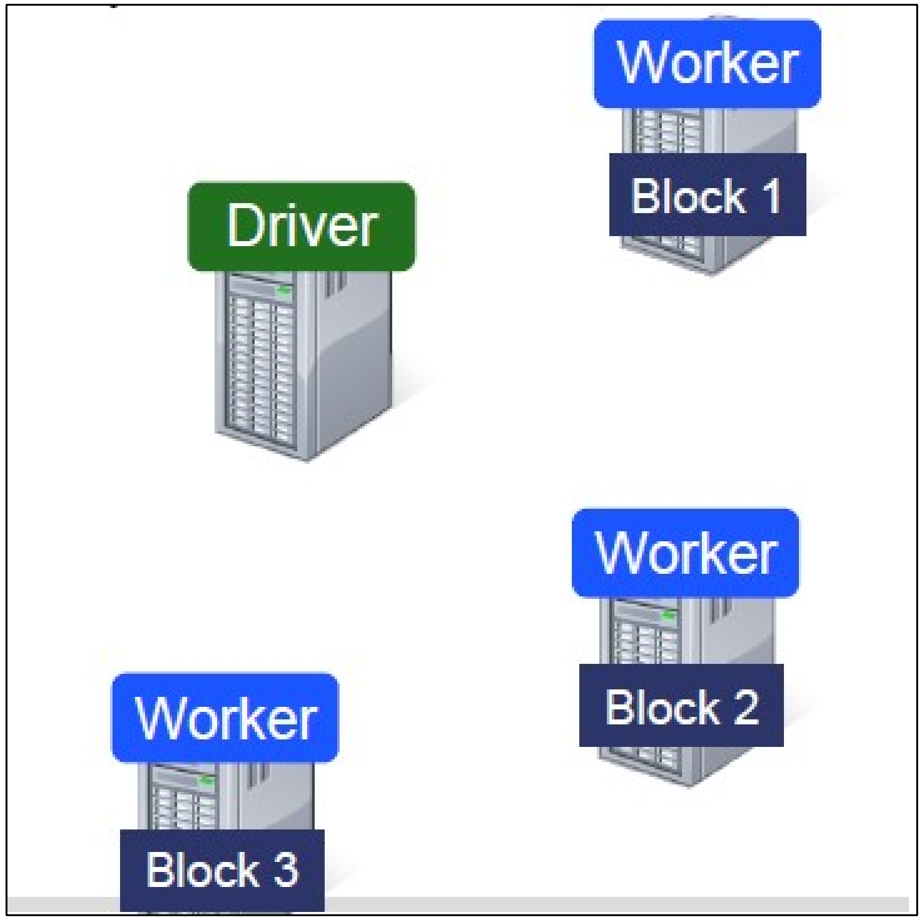


Example: Log Mining

Load error messages from a log into memory and search for various patterns

- Suppose that we wish to count the lines containing errors in a large log file stored in HDFS.
  - val file = spark.textFile("hdfs://...")
  - val errs = file.filter(_. contains("ERROR"))
  - val cachedErrs = errs.cache()
  - val ones = cachedErrs.map(_ => 1)
  - **val count = ones.reduce(_+_)**

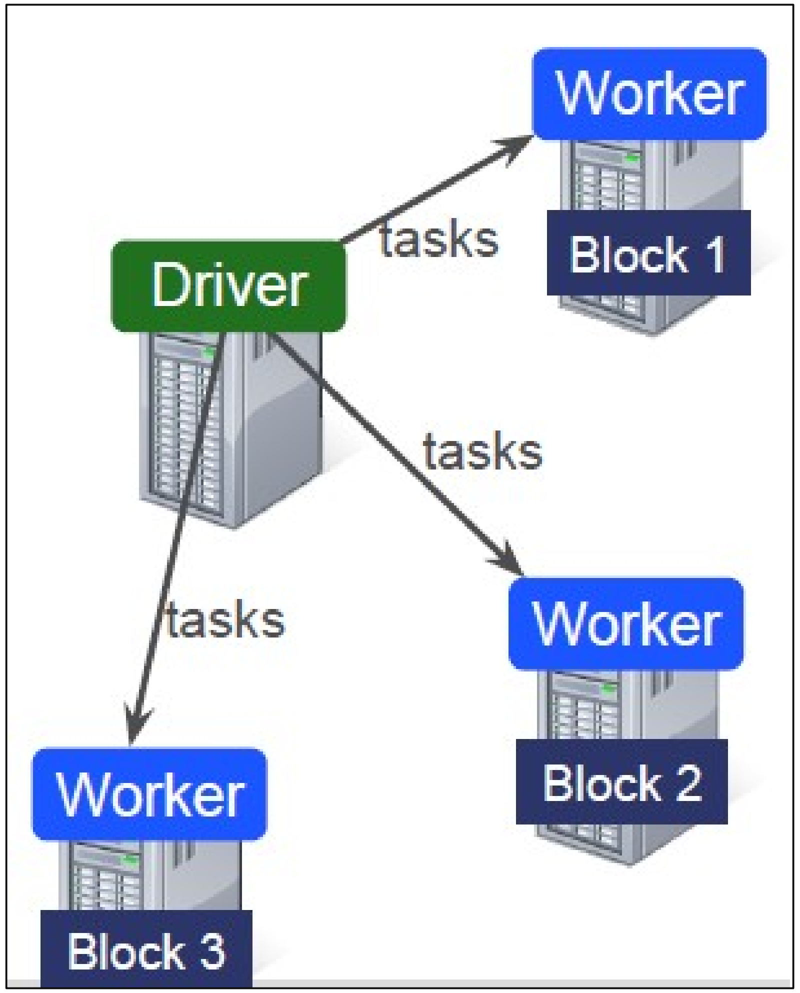
  - cachedErrs.filter(_.contains(“mysql”)).count()



Example: Log Mining

Load error messages from a log into memory and search for various patterns

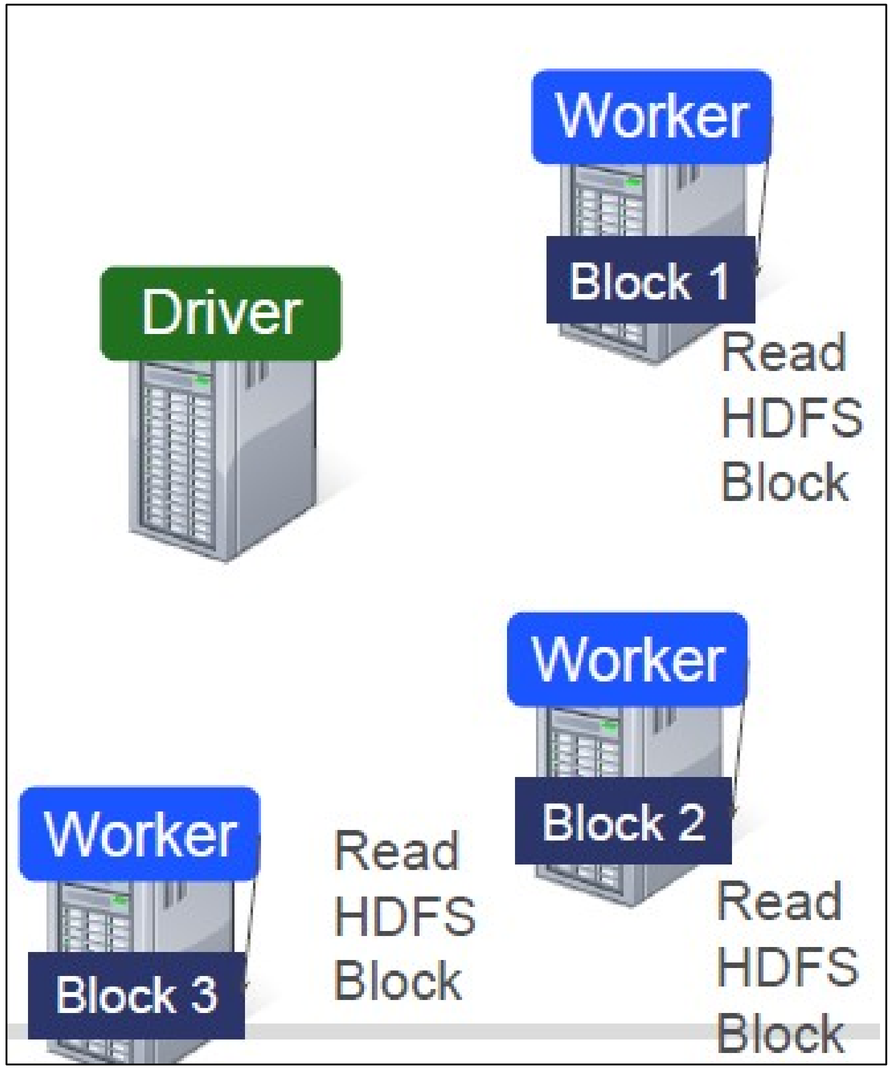
- Suppose that we wish to count the lines containing errors in a large log file stored in HDFS.
  - **val file = spark.textFile("hdfs://...")**
  - val errs = file.filter(_.contains("ERROR"))
  - val cachedErrs = errs.cache()
  - val ones = cachedErrs.map(_ => 1)
  - val count = ones.reduce(_+_)
  - cachedErrs.filter(_.contains("mysql")).count()



Example: Log Mining

Load error messages from a log into memory and search for various patterns

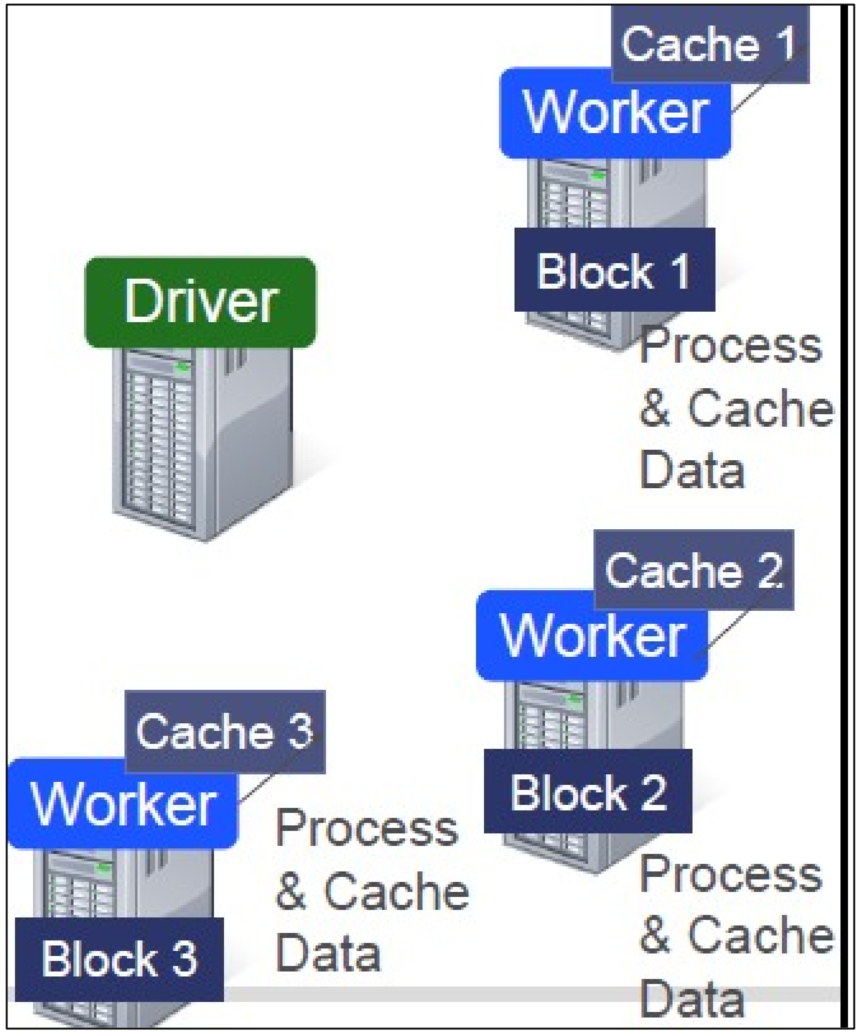
- Suppose that we wish to count the lines containing errors in a large log file stored in HDFS.
  - **val file = spark.textFile("hdfs://...")**
  - val errs = file.filter(_.contains("ERROR"))
  - val cachedErrs = errs.cache()
  - val ones = cachedErrs.map(_ => 1)
  - val count = ones.reduce(_+_)
  - cachedErrs.filter(_.contains(“mysql”)).count()



Example: Log Mining

Load error messages from a log into memory and search for various patterns

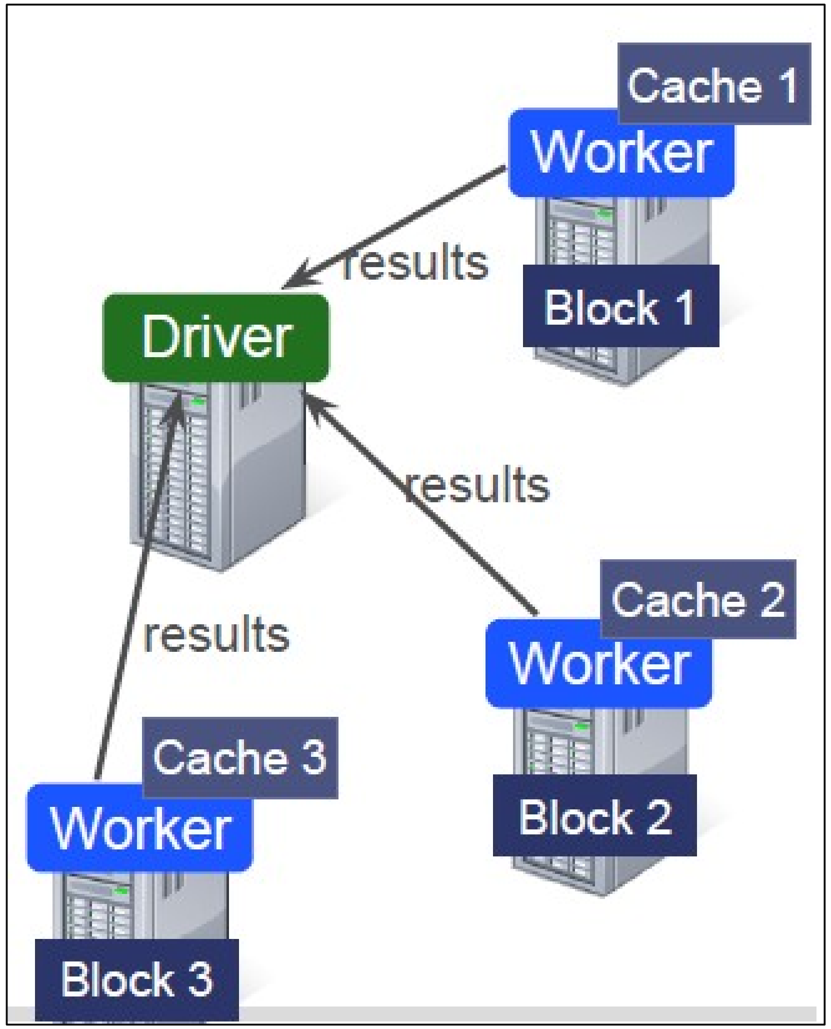
- Suppose that we wish to count the lines containing errors in a large log file stored in HDFS.
    - val file = spark.textFile("hdfs://...")
    - **val errs = file.filter(_.contains("ERROR"))**
    - **val cachedErrs = errs.cache()**
    - **val ones = cachedErrs.map(_ => 1)**
    - val count = ones.reduce(_+_)
    - cachedErrs.filter(_.contains("mysql")).count()



Example: Log Mining

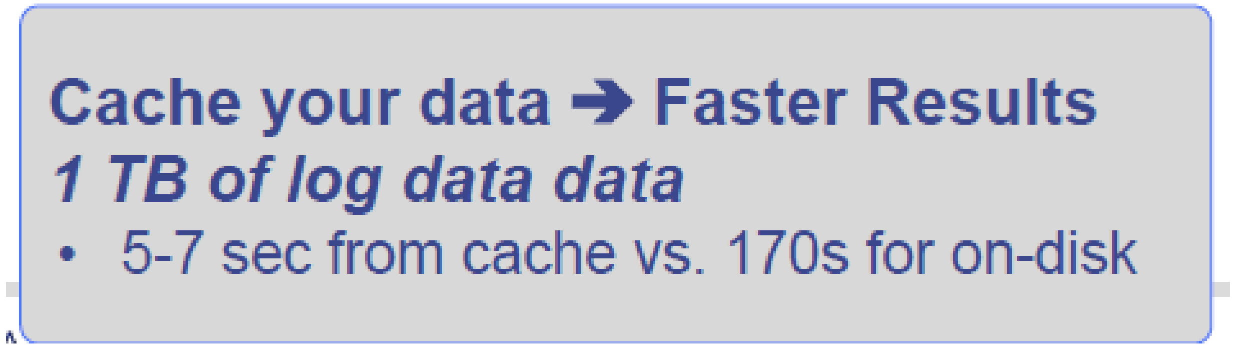
Load error messages from a log into memory and search for various patterns

- Suppose that we wish to count the lines containing errors in a large log file stored in HDFS.
  - val file = spark.textFile("hdfs://...")
  - val errs = file.filter(_.contains("ERROR"))
  - val cachedErrs = errs.cache()
  - val ones = cachedErrs.map(_ => 1)
  - **val count = ones.reduce(_+_)**
  - cachedErrs.filter(_.contains(“mysql”)).count()



Example: Log Mining

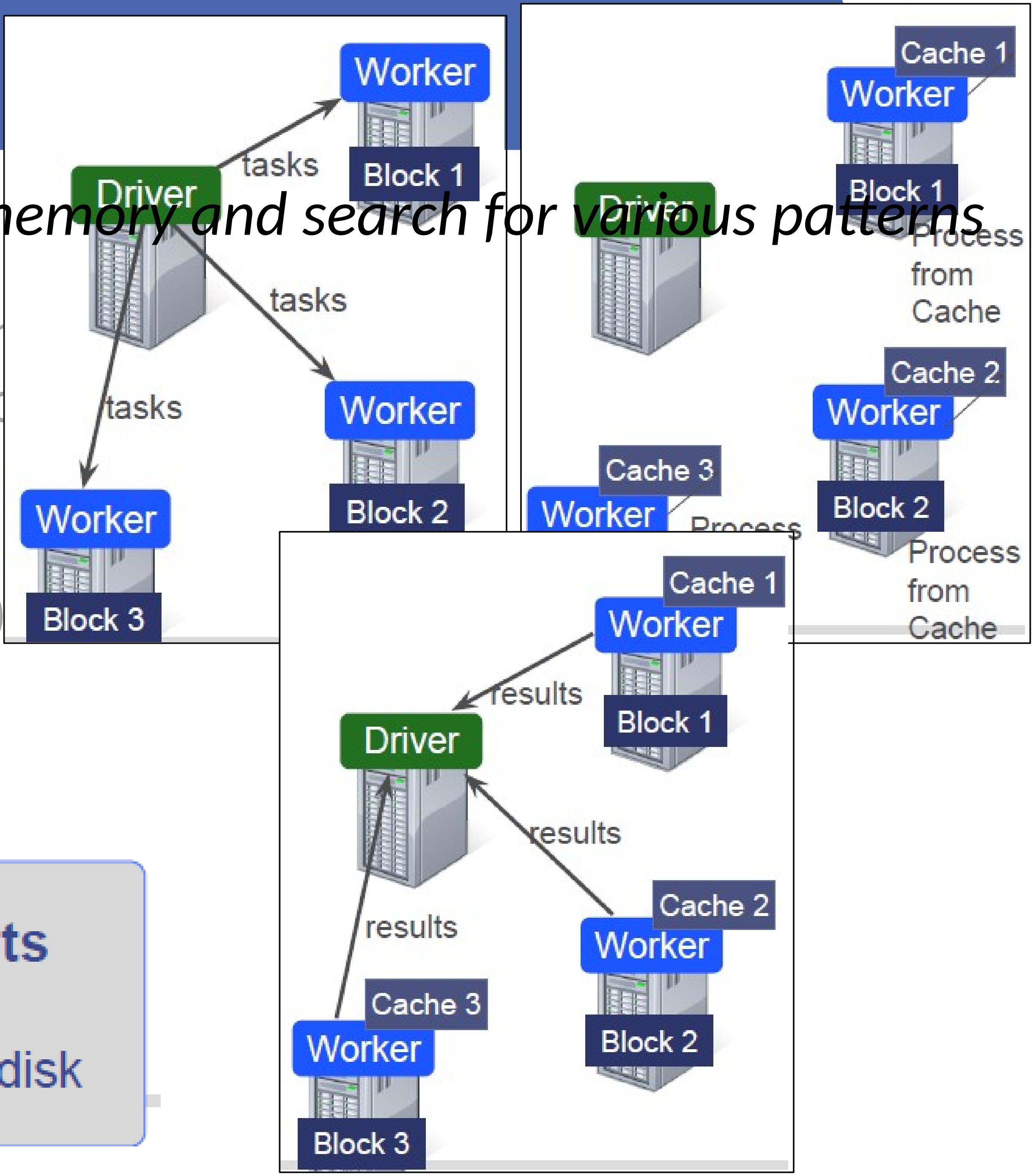
Load error messages from a log into memory and search for various patterns

- Suppose that we wish to count the li containing errors in a large log file sto HDFS.
  - val file = spark.textFile("hdfs://...")
  - val errs = file.filter(_.contains("ERROR"))
  - **val cachedErrs = errs.cache()**
  - **val ones = cachedErrs.map(_ => 1)**
  - val count = ones.reduce(_+_)
  - 

Cache your data ➔ Faster Results

1 TB of log data data

- 5-7 sec from cache vs. 170s for on-disk



Example: Log Mining

- Suppose that we wish to count the lines containing errors in a large log file stored in HDFS.
  - val file = spark.textFile("hdfs://...")
  - val errs = file.filter(_.contains("ERROR"))
  - val cachedErrs = errs.cache()
  - val ones = cachedErrs.map(_ => 1)
  - val count = ones.reduce(_+_)
  - cachedErrs.filter(_.contains(“mysql”)).count()

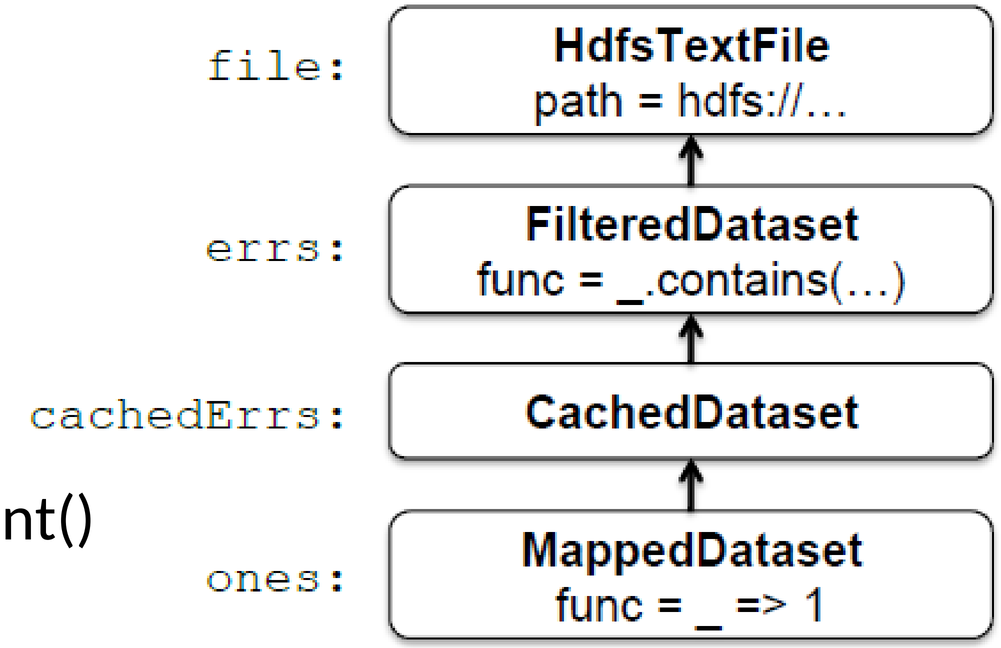


Figure 1: Lineage chain for the distributed dataset objects