# Resilient Distributed Datasets:

A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Presented By : Ravi Bisla(301345992)

## About the Paper

**Some details about the paper :**

- This paper was presented in 9th USENIX Symposium on Networked Systems Design and Implementation
  - April 25 - 27, 2012
  - San Jose,CA
- Authors : Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, University of California, Berkeley
- Research supported by Berkeley AMP Labsponsors Google, SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp andVMWare, by DARPA (contract #FA8650-11-C-7136),by a Google PhD Fellowship, and by the Natural Sci-ences and Engineering Research Council of Canada

## Introduction

- **This paper presents** Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large  clusters in a fault-tolerant manner.
- **The RDDs were implemented** in a system called Spark. And was evaluated through a variety of user application and benchmarks.
- There are 4 main sections of this presentation:
    - Motivation and Challenges
    - RDD
    - Spark and Examples
    - Evaluation and Benchmarking

## Motivation

- MapReduce greatly simplifies “big data” analysis on large, unreliable clusters
  - Simple interface: map and reduce
  - Hides the details of parallelism, data partition, fault-tolerance, load-balancing…

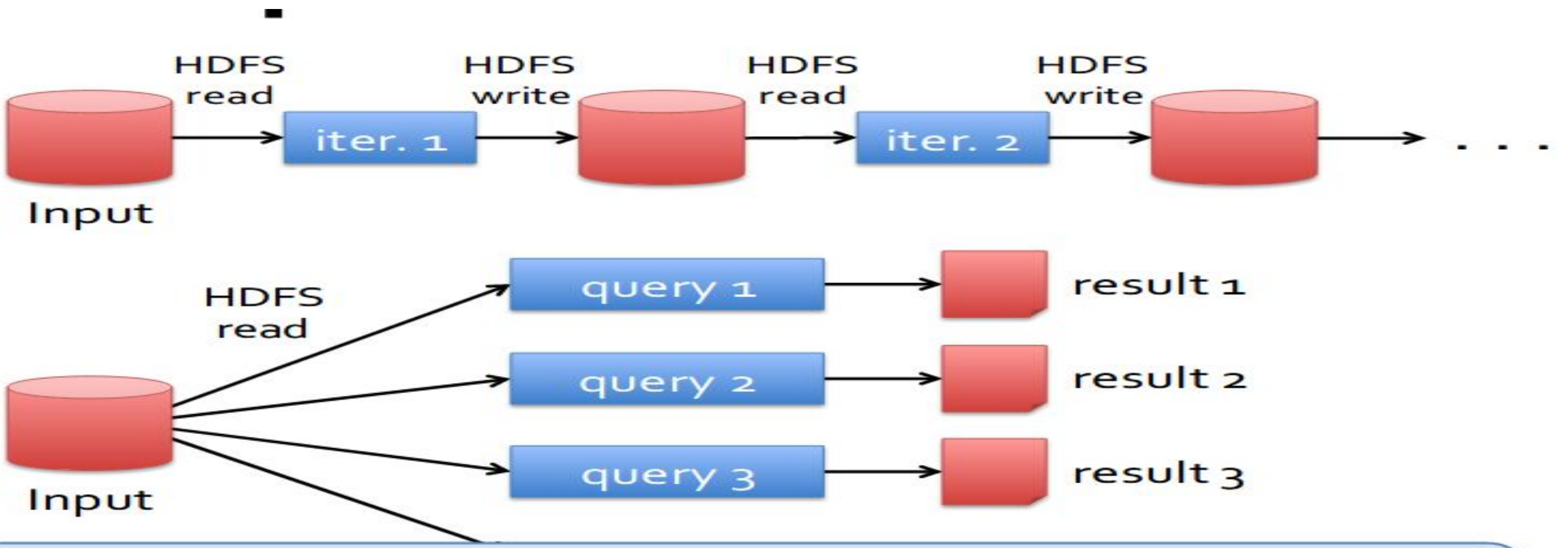
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage applications (e.g. iterative machine learning & graph processing)
  - More interactive ad-hoc queries(data mining)
    - In mapreduce only was to reuse data between computations is to write it to external storage system e.g a distributed file system.

## Motivation

Complex algorithms/apps and interactive queries both need one thing that MapReduce lacks:

**It lacked abstraction for leveraging distributed memory.**

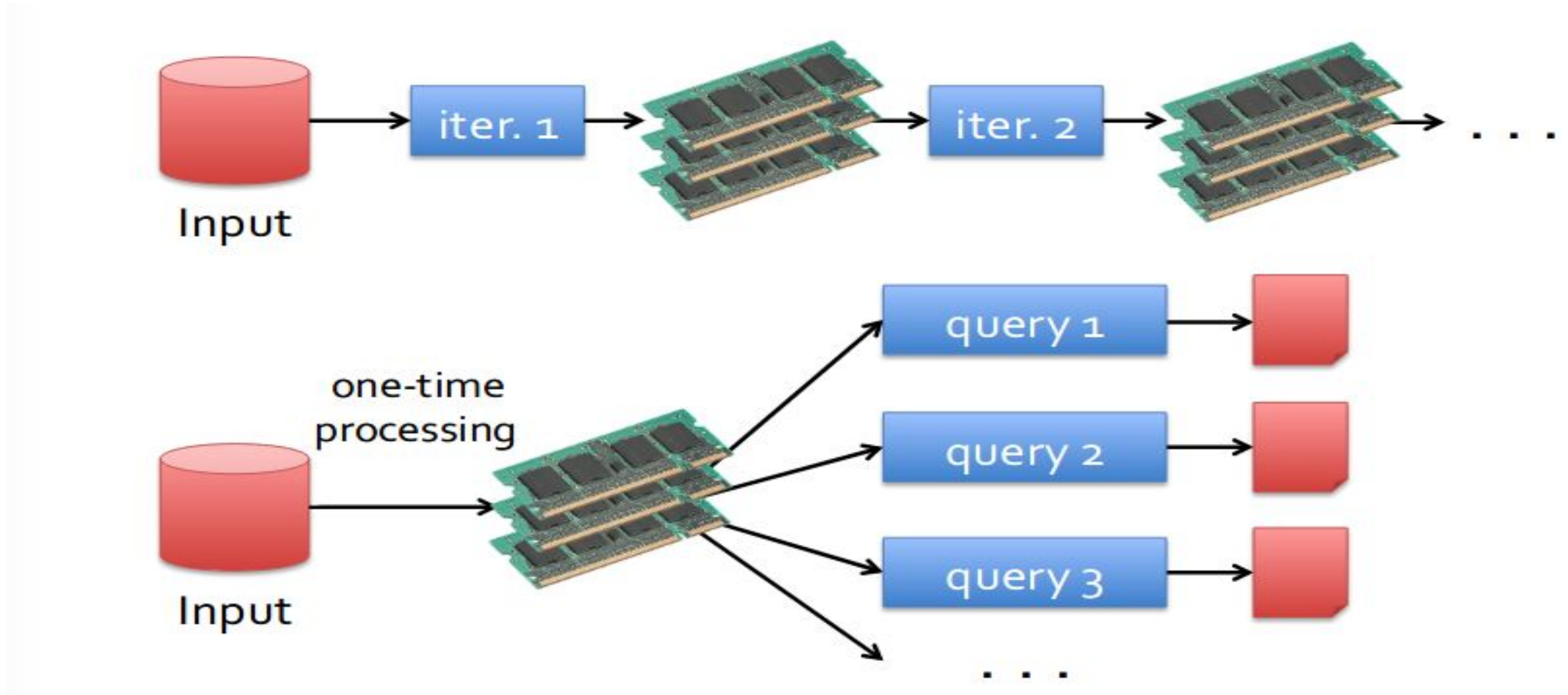
# Example



Slow due to replication and disk I/O,
but necessary for fault tolerance

# Goal: In-Memory Data Sharing

10-100 * faster than network/disk.



## Challenge

- How to design a distributed memory abstraction that is both :
  - Fault tolerant
  - Efficient

## Solution : Resilient Distributed Datasets(RDD)

- **Restricted form of distributed shared memory :**
    - Immutable, partitioned collections of records
    - Can only be built through coarse-grained deterministic transformations(map, filter, join)

## Solution : Resilient Distributed Datasets(RDD)

- **Efficient fault recovery using lineage**
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

## RDD  : Persistence and Partitioning

- Users can control two other aspects of RDDs: persistence and partitioning. Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage).

- They can also ask that an RDD’s elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

## Generality of RDDs

- Despite their restrictions, RDDs can express surprisingly many parallel algorithms.
    - These naturally apply the same operation to many items

- Unify many related programming models
    - Data flow models: MapReduce, Dryad, SQL
    - Specialized models for iterative  apps: BSP(Pregel),iterative MapReduce(Haloop)

- Support new functionality.

## Spark Programming Interface

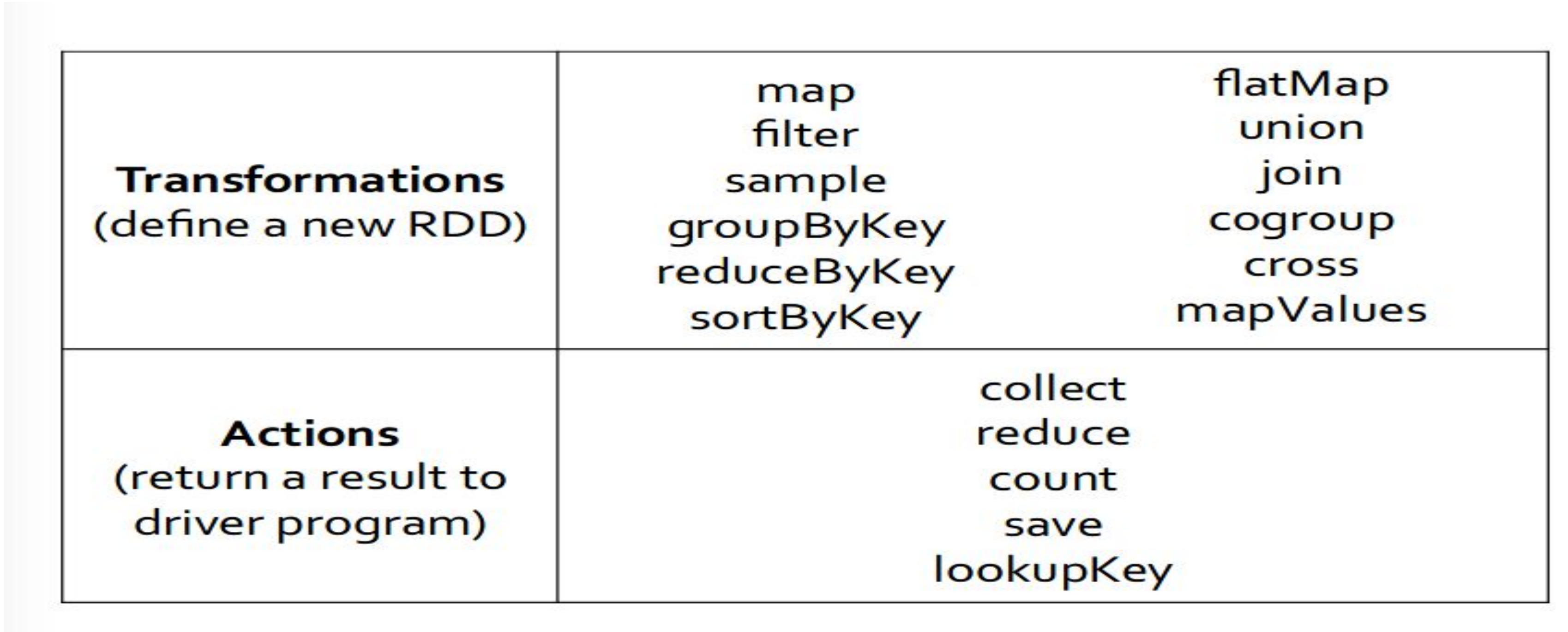
Spark exposes RDD through a language oriented API where each dataset is represented as an object and transformations are involved using method on these objects.

It provides :

- Resilient distributed datasets(RDDs)
- Control of each RDD partitioning (layout across nodes) and persistence (storage in RAM, on disk, etc)

# Spark Programming Interface

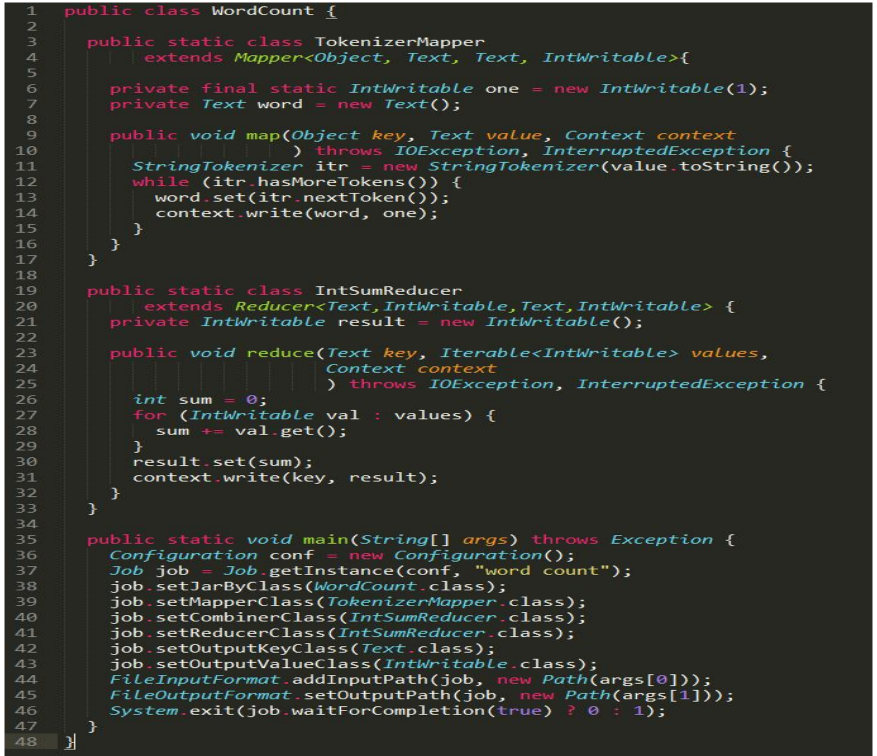
- Operations on RDDs: Transformations (build new RDDs and are **lazy operation**), Actions (compute and output results)



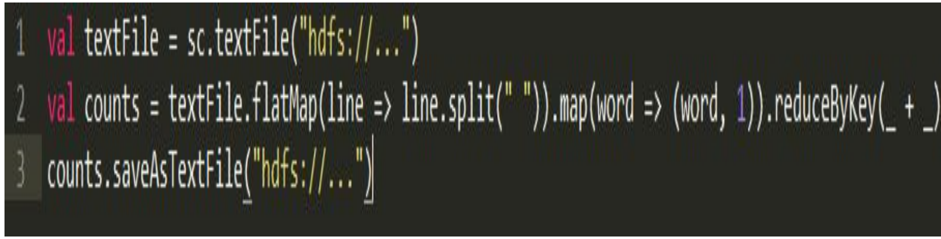
| | | |
|---|---|---|
| **Transformations** (define a new RDD) | map<br>filter<br>sample<br>groupByKey<br>reduceByKey<br>sortByKey | flatMap<br>union<br>join<br>cogroup<br>cross<br>mapValues |
| **Actions** (return a result to driver program) | collect<br>reduce<br>count<br>save<br>lookupKey | |

## Example : WordCount

WordCount Implementation: Hadoop vs. Spark



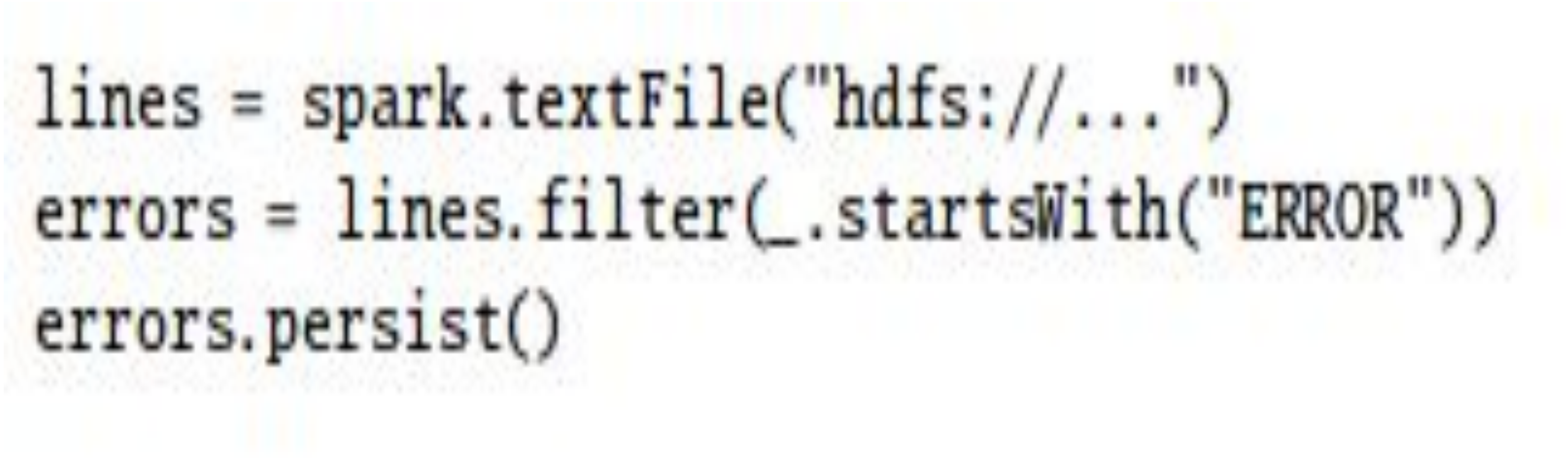
```java
public class WordCount {

  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);
  }
}
```



```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _
counts.saveAsTextFile("hdfs://...")
```

## Example: Log Mining

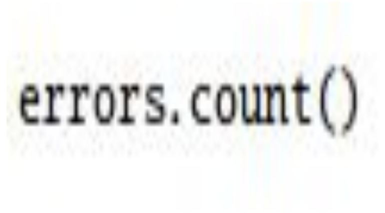
Load error messages from a log into memory, then interactively search for various patterns



```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

## Example: Log Mining

At this point, no work has been performed on the cluster. However, the user can now use the RDD in actions, e.g.,to count the number of messages:

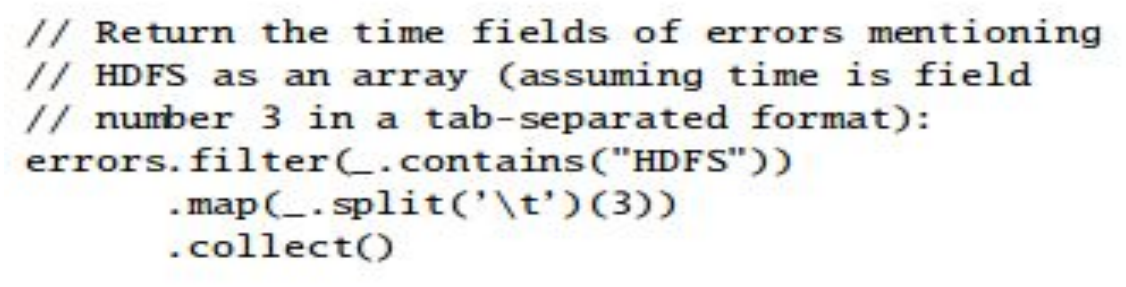


```
errors.count()
```

The user can also perform further transformations on the RDD and use their results.

## Example: Log Mining

After the first action involving errors runs, Spark will store the partitions of errors in memory, greatly speeding up subsequent computations on it.

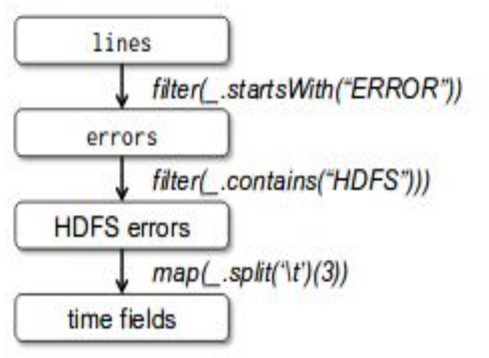


```
// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

Note that base RDD, *lines*, is not loaded into RAM. This is desirable because the error messages might only be a small fraction of the data (small enough to fit into memory).

## Example: Log Mining

To illustrate how the model achieves fault tolerance, below is the lineage graph for the RDDs in our third query



RDDs track the graph of transformations that built them (their lineage) to rebuild lost data.

## Applications Not Suitable For RDD Model

RDDs would be less suitable for applications that make asynchronous fine-grained updates to shared state, such as a storage system for a web application or an incremental web crawler.

For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases, RAMCloud [25], Percolator and Piccolo

## Representing RDDs

A RDD representing an HDFS file has a partition for each block of the file and knows which machines each block is on.

Each RDD is represented through a common interface that exposes five pieces of information.

## Representing RDDs

The five functions are :

1. partition—represents atomic pieces of the dataset.
2. dependencies—list of dependencies that an RDD has on its parent RDDs or data sources
3. iterator —a function that computes an RDD based on its parents
4. partitioner—whether data is range/hash partitioned.
5. preferredLocation—nodes where a partition can be accessed faster due to data locality.

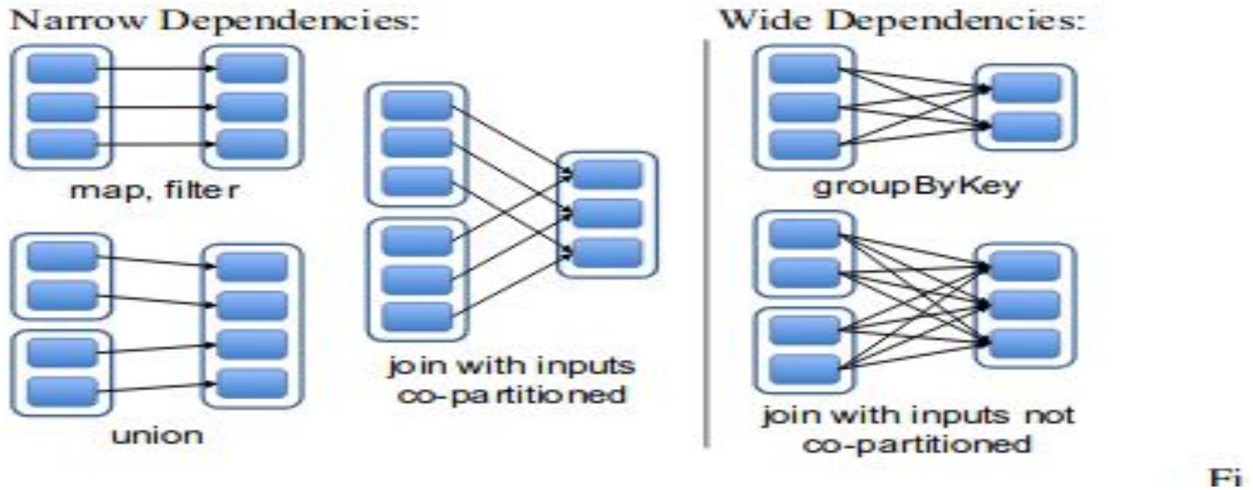
## Representing RDDs

- The most interesting question in designing this interface is how to represent dependencies between RDDs.

- We can classify these dependencies into two types:
    - **Narrow dependencies**, where each partition of the parent RDD is used by at most one partition of the child RDD,
    - **Wide dependencies**, where multiple child partitions may depend on it.

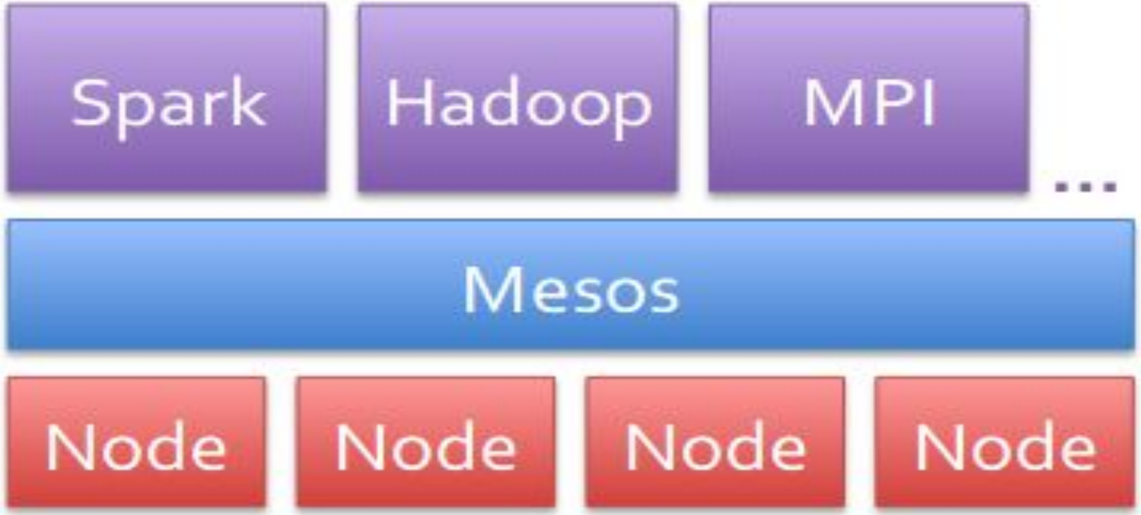
## Representing RDDs

- For example,map leads to a narrow dependency, while join leads to wide dependencies  (unless the parents are hash-partitioned).


Narrow Dependencies:
map, filter
union
join with inputs co-partitioned
Wide Dependencies:
groupByKey
join with inputs not co-partitioned
Fi

## Implementation

- Around 14,000 lines of Scala
- Runs on Mesos to share clusters w/Hadoop



## Implementation

- Each Spark program runs as a separate Mesos application, with its own driver (master) and workers, and resource sharing between these applications is handled by Mesos.

- Can read from any Hadoop input source (HDFS, S3, ...)

# Evaluation

**Overall results shows following :**

- Spark outperforms Hadoop by up to 20× in iterative machine learning and graph applications.
    - By avoiding I/O and deserialization costs by storing data in memory as Java objects.

## Evaluation

- Applications written in using Spark perform and scale well. In particular, we used Spark to speed up an analytics report that was running on Hadoop by 40×.
- When nodes fail, Spark can recover quickly by re-building only the lost RDD partitions.
- Spark can be used to query a 1 TB dataset interactively with latencies of 5–7 second

## Evaluation

**Iterative Machine Learning Applications:**

We implemented two iterative machine learning applications, logistic regression and k-means, to compare the performance of the following systems:

- **Hadoop:** The Hadoop 0.20.2 stable release
- **HadoopBinMem:** A Hadoop deployment that converts the input data into a low-overhead binary format in the first iteration to eliminate text parsing in laterones, and stores it in an in-memory HDFS instance.
- **Spark**: Our implementation of RDDs.

## Evaluation

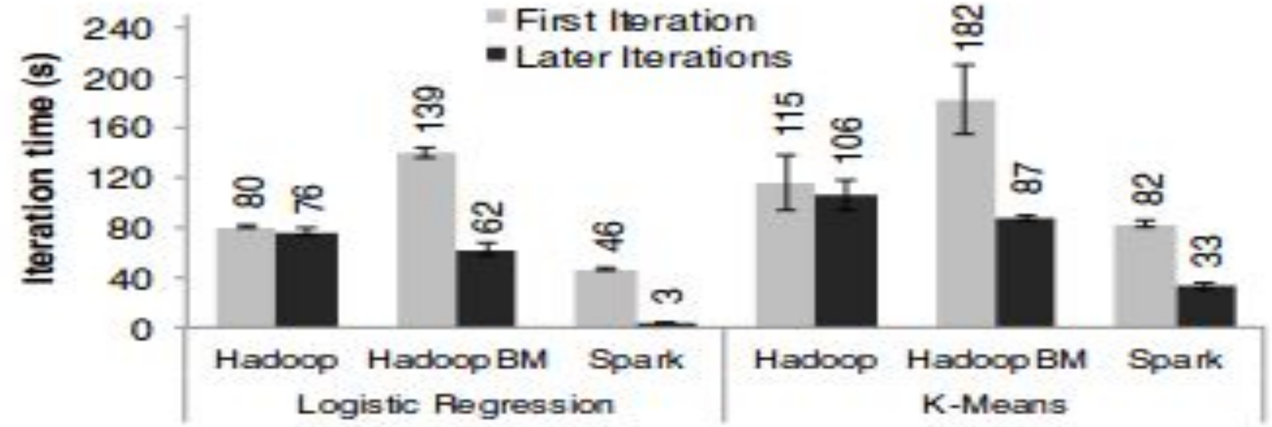


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

## Evaluation

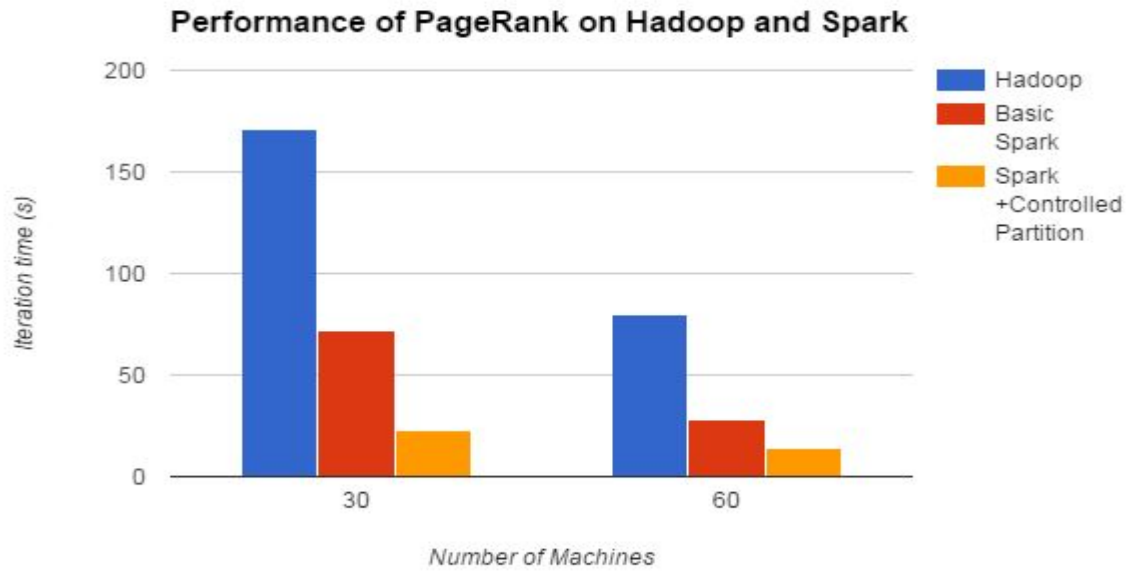
- First Iterations : This difference was due to **signaling overheads in Hadoop's heartbeat protocol between its master and worker.**

- HadoopBinMem still ran slower :
  - Deserialization cost to convert binary records to usable in-memory Java objects.

## Evaluation

**Pagerank :** 10 iterations on 54GB data with approximately 4M articles

Spark with a 2.4×speedup over Hadoop, controlled partition 7.4 * speedup



## Evaluation

**Fault Recovery**

Evaluated the cost of reconstructing RDD partitions using lineage after a node failure in the k-means application.

Next figure compares the running times for 10 iterations of k-means on a 75-node cluster in normal operating scenario, with one where a node fails at the start of the 6th iteration.Without any failure, each iteration consisted of 400 tasks working on 100 GB of data

## Evaluation

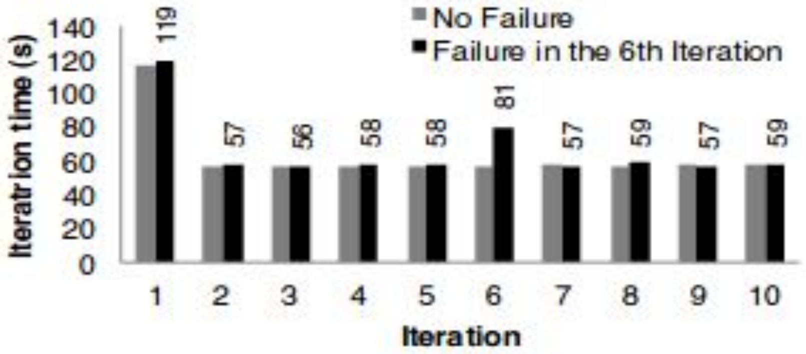


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

## Evaluation

**Behaviour with insufficient memory :**

So far, we ensured that every machine in the cluster had enough memory to store all the RDDs across iterations.

A natural question is how Spark runs if there is not enough memory to store a job’s data.

# Evaluation

In this experiment,we configured Spark not to use more than a certain percentage of memory to store RDDs on each machine.

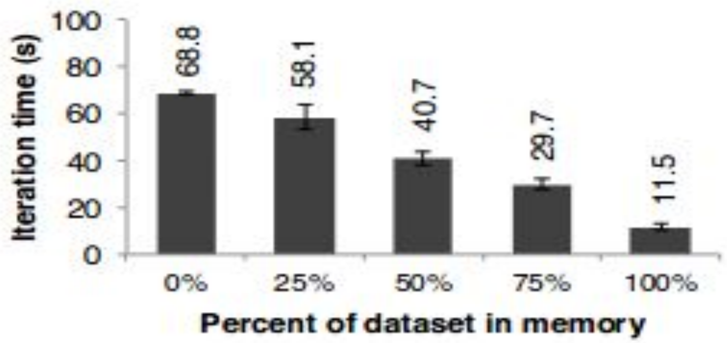


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

## Comparison with existing systems

RDDs and Spark learn from and improve the existing systems in many ways :

- Piccolo and DSM do not provide a high-level programming interface like RDDs. Moreover, they use checkpointing and roll back which are more expensive than lineage based approach.
- Nectar, Ceil and FlumeJava do not provide in-memory caching.
- MapReduce and Dryad use lineage based recovery within a computation, but this information is lost after a job ends. In contrast, RDDs persists lineage information across computations.

## Interesting Insight

The paper also offers an interesting insight on the question of why previous frameworks could not offer the same level of generality.

It says previous frameworks did not observe that **“the common cause of these problems was a lack of data sharing abstractions”.**

## Key Takeaways

- RDDs are an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications.
- Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that  lets them recover data efficiently using lineage.
- Spark (RDD) outperforms  Hadoop  by  upto 20× in iterative applications and can be used interactively to query hundreds of gigabytes of data.

THANK YOU!

Questions?