Large-scale Distributed Systems

Lecture 7: Cloud computing

LIÈGE université

Today

How do we program this thing?

- MapReduce
- Spark

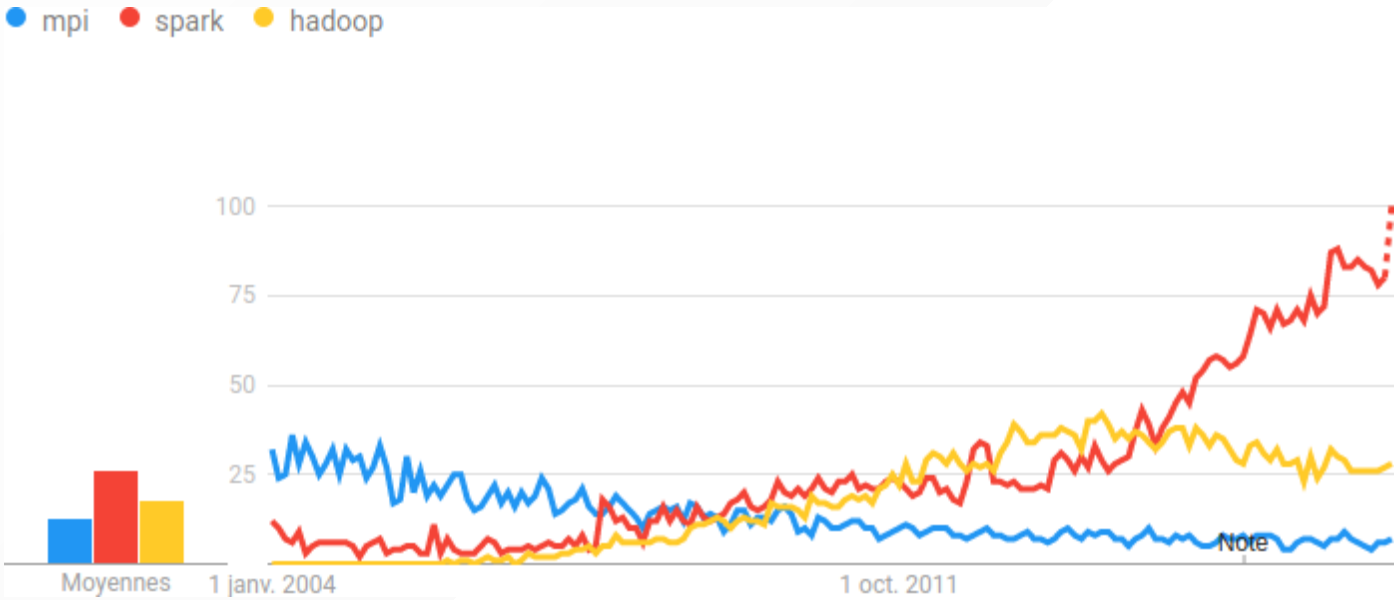
Dealing with lots of data

- Example: 130+ trillion web pages $\times$ $50\text{KB} = 6.5$ exabytes.
  - ~6500000 hard drives (1TB) just to store the web.
- Assuming a data transfer rate of $200\text{MB}/s$, it would require 1000+ years for a single computer to read the web!
  - And even more to make any useful usage of this data.
- Solution: spread the work over many machines.

Traditional network programming

- Message-passing between nodes (MPI, RPC, etc).
- Really hard to do at scale (for 1000s of nodes):
  - How to split problem across nodes?
    - Important to consider network and data locality.
  - How to deal with failures?
    - a 10000-node clusters sees 10 faults/day.
  - Even without failure: stragglers.
    - Some nodes might be much slower than others.

Traditional network programming



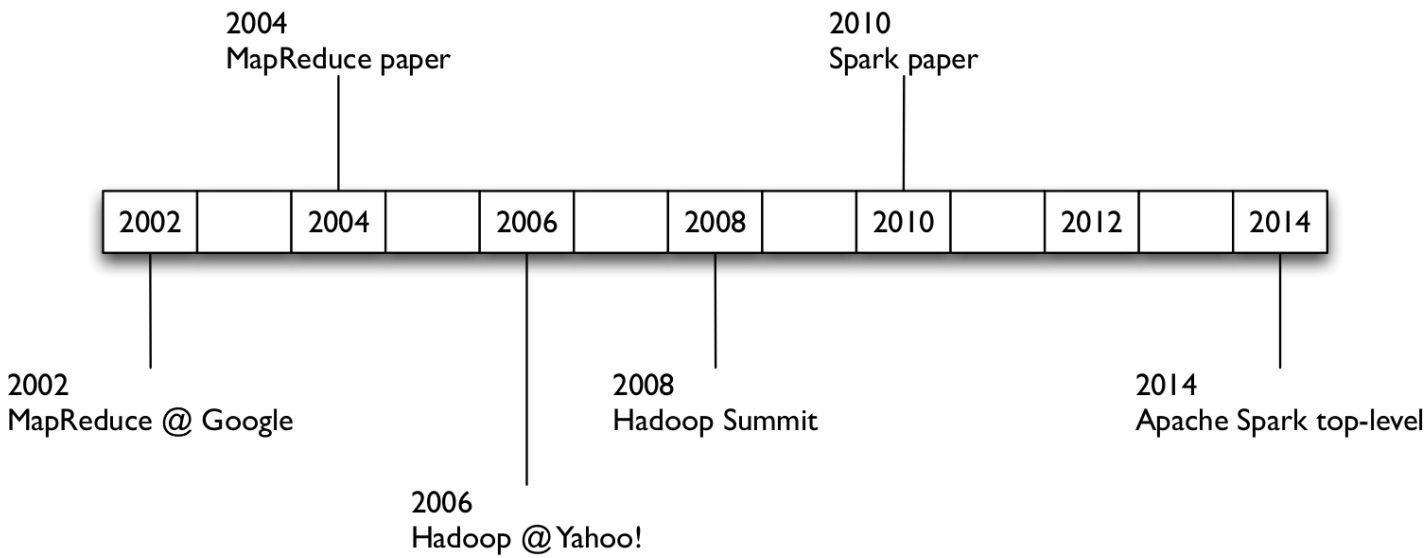
Almost nobody does message-passing anymore!*

*: except in niches, like scientific computing.

Data-Parallel models

- Restrict and simplify the programming interface so that the system can do more automatically.
- "Here is an operation, run it on all of the data".
  - I do not care where it runs (you schedule that).
  - In fact, feel free to run it twice on different nodes if that can help.

History


2004
MapReduce paper
2010
Spark paper
2002
2004
2006
2008
2010
2012
2014
2002
MapReduce @ Google
2006
Hadoop @ Yahoo!
2008
Hadoop Summit
2014
Apache Spark top-level

MapReduce

What is MapReduce?

- MapReduce is a parallel programming model for processing distributed data on a cluster.
- Simple high-level API limited two operations: map and reduce, as inspired by Lisp primitives:
  - `map`: apply function to each value in a set.
    - `(map 'length '(() (a) (a b) (a b c)))` → `(0 1 2 3)`
  - `reduce`: combines all the values using a binary function.
    - `(reduce #'+ '(1 2 3 4 5))` → `15`
- MapReduce is best suited for embarrassingly parallel tasks.
  - When processing can be broken into parts of equal size.
  - When processes can concurrently work on these parts.
- This abstraction makes it possible to not worry about handling
  - parallelization
  - data distribution
  - load balancing
  - fault tolerance

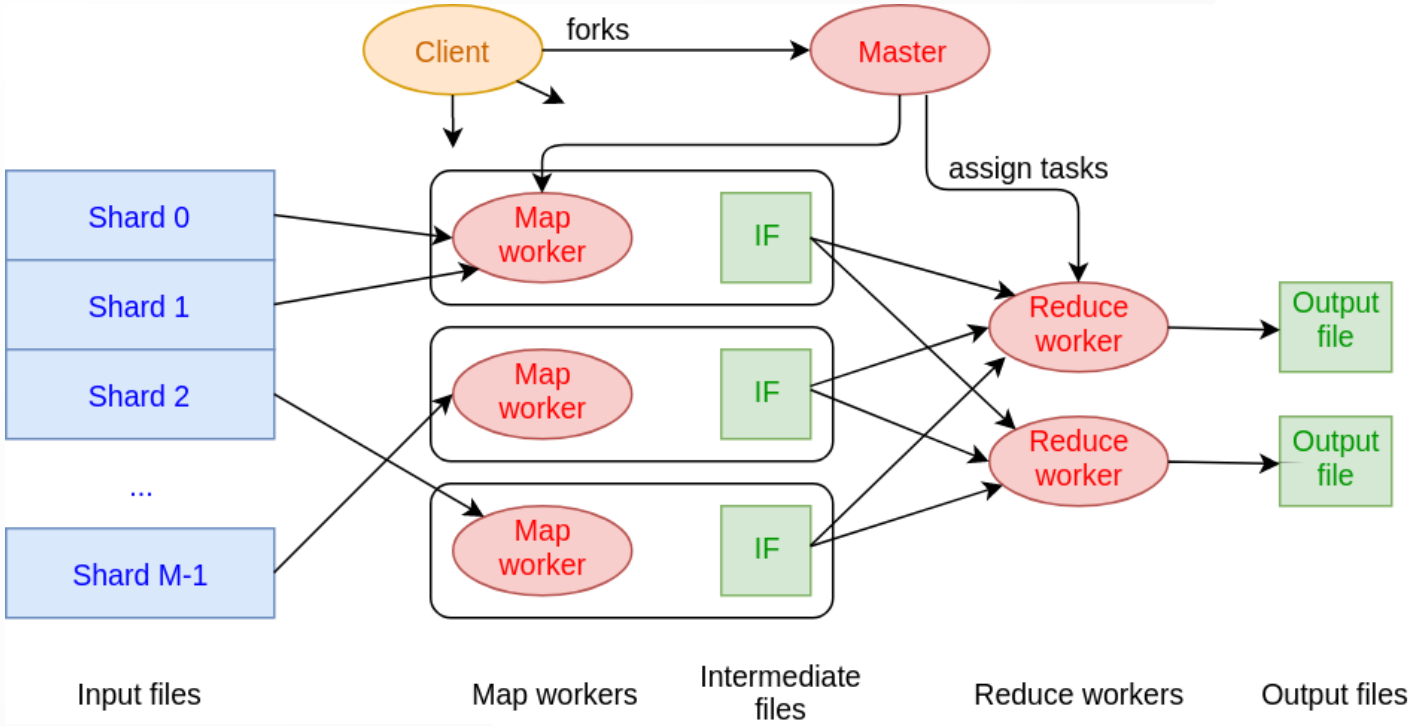
Programming model

- Map: input key/value pairs $\longrightarrow$ intermediate key/value pairs
  - User function gets called for each input key/value pairs.
  - Produces a set of intermediate key/value pairs.
- Reduce: intermediate key/value pairs $\longrightarrow$ result files
  - Combine all intermediate values for a particular key through a user-defined function.
  - Produces a set of merged output values.

What really happens

- Map worker:
  - Map:
    - Map calls are distributed across machines by automatically partitioning the input data into M shards.
    - Parse the input shards into input key/value pairs.
    - Process each input pair through a user-defined `map` function to produce a set of intermediate key/value pairs.
    - Write the result to an intermediate file.
  - Partition:
    - Assign an intermediate result to one of R reduce tasks based on a partitioning function.
      - Both R and the partitioning function are user defined.
- Reduce worker:
  - Sort:
    - Fetch the relevant partition of the output from all mappers.
    - Sort by keys.
      - Different mappers may have output the same key.
  - Reduce:
    - Accept an intermediate key and a set of values for the key.
    - For each unique key, combine all values through a user-defined `reduce` function to form a smaller set of values.

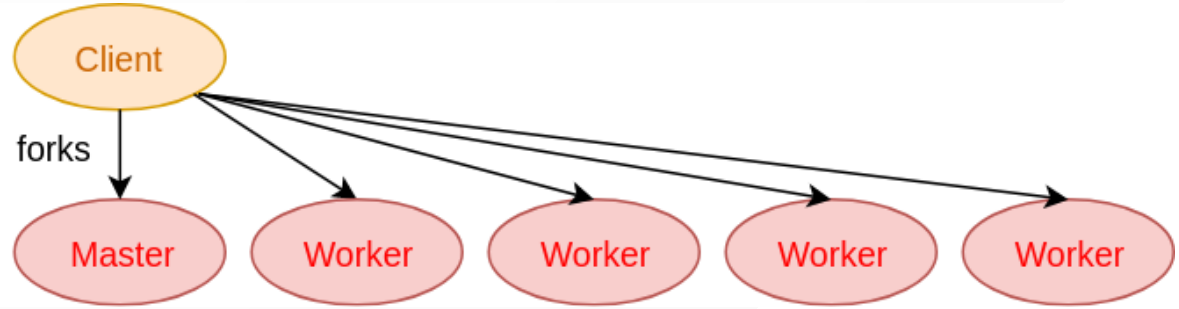
Overview


Client
forks
Master
assign tasks
Shard 0
Shard 1
Shard 2
...
Shard M-1
Map worker
Map worker
Map worker
IF
IF
IF
Reduce worker
Reduce worker
Output file
Output file
Input files
Map workers
Intermediate files
Reduce workers
Output files

Step 1: Split input files

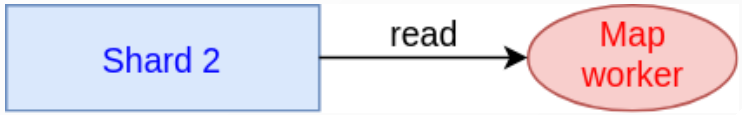
- Break up the input data into M shards (typically 64MB).

Step 2: Fork processes



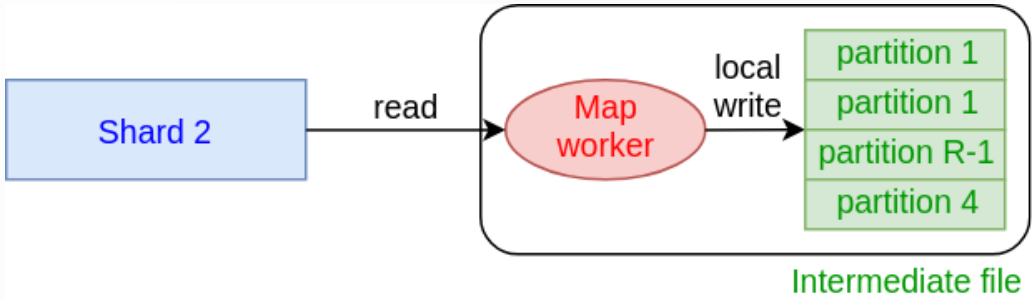
- Start up many copies of the program on a cluster of machines.
  - 1 master: scheduler and coordinator
  - Lots of workers
- Idle workers are assigned either:
  - map tasks
    - each works on a shard
    - there are M map tasks
  - reduce tasks
    - each works on intermediate files
    - there are R reduce tasks

Step 3: Map task



- Read content of the input shard assigned to it.
- Parse key/value pairs (k, v) out of the input data.
- Pass each pair to a user-defined `map` function.
  - Produce (one or more) intermediate key/value pairs (k', v').
  - These are buffered in memory.

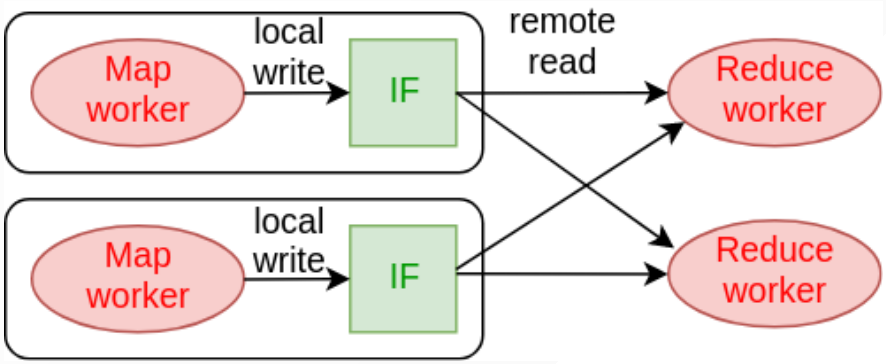
Step 4: Create intermediate files



- Intermediate key/value pairs (k', v') produced by the user's `map` function are periodically written to local disk.
  - These files are partitioned into R regions by a partitioning function, one for each reduce task.
  - e.g., `hash(key) mod R`
- Notify master when complete.
  - Pass locations of intermediate data to the master.
  - Master forwards these locations to the reduce workers.

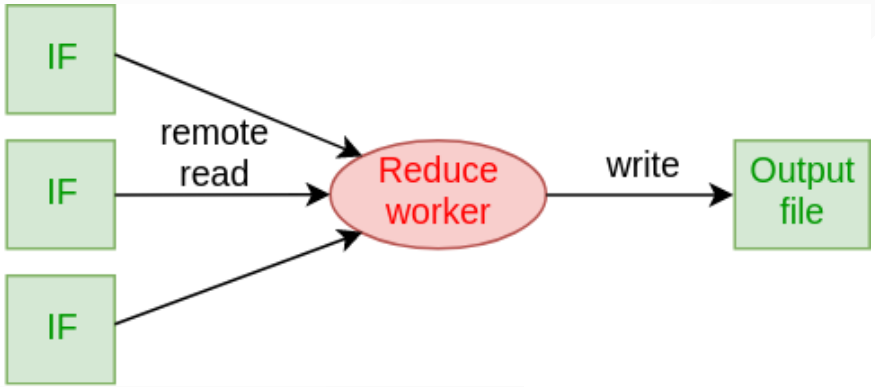
[Q] What is the purpose of the partitioning function?

Step 5: Sorting/Shuffling



- Reduce worker get notified by master about the location of the intermediate files associated to their partition.
- RPC to read the data from the local disks for the map workers.
- When the reduce worker reads intermediate data for its partition:
  - it sorts the data by intermediate keys k'.
  - all occurrences v'_i associated to a same key are grouped together.

Step 6: Reduce tasks

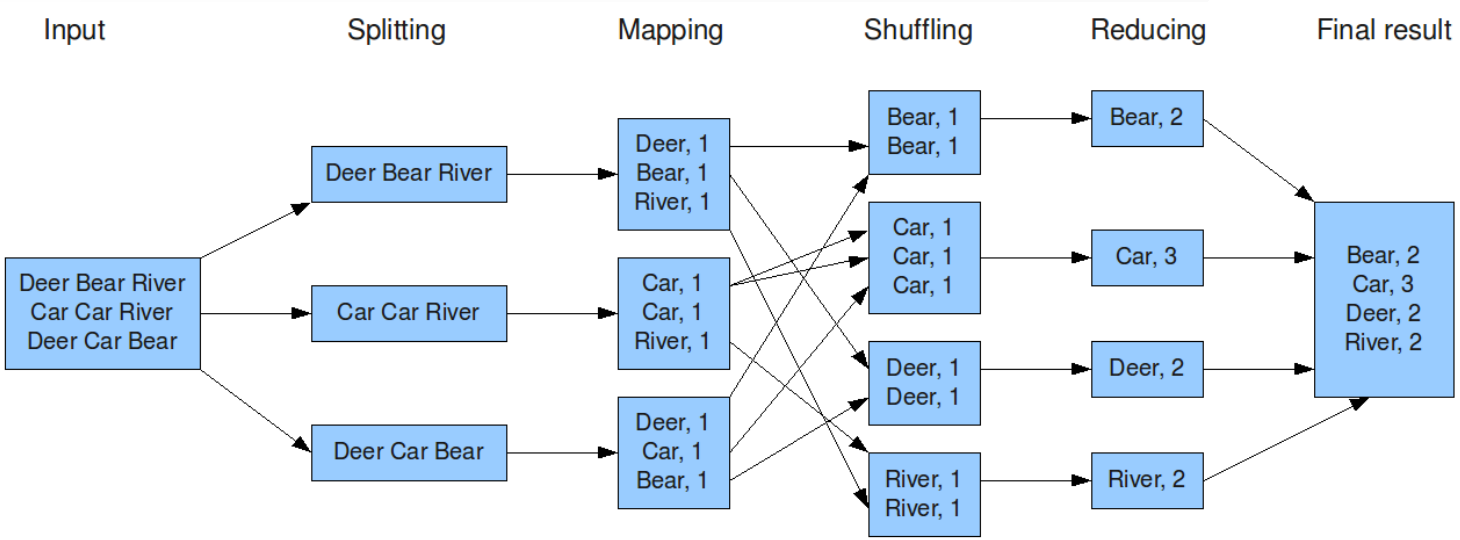


- The sorting phase grouped data sharing a unique intermediate key.
- The user-defined `reduce` function is given the key and the set of intermediate values for that key.
  - $(k', (v'_1, v'_2, v'_3, ...))$
- The output of the `reduce` function is appended to an output file.

Step 7: Return to user

- When all Map and Reduce tasks have completed, the master wakes up the user program.
- The MapReduce call in the user program returns and the program can resume execution.
    - The output of the operation is available in R output files.

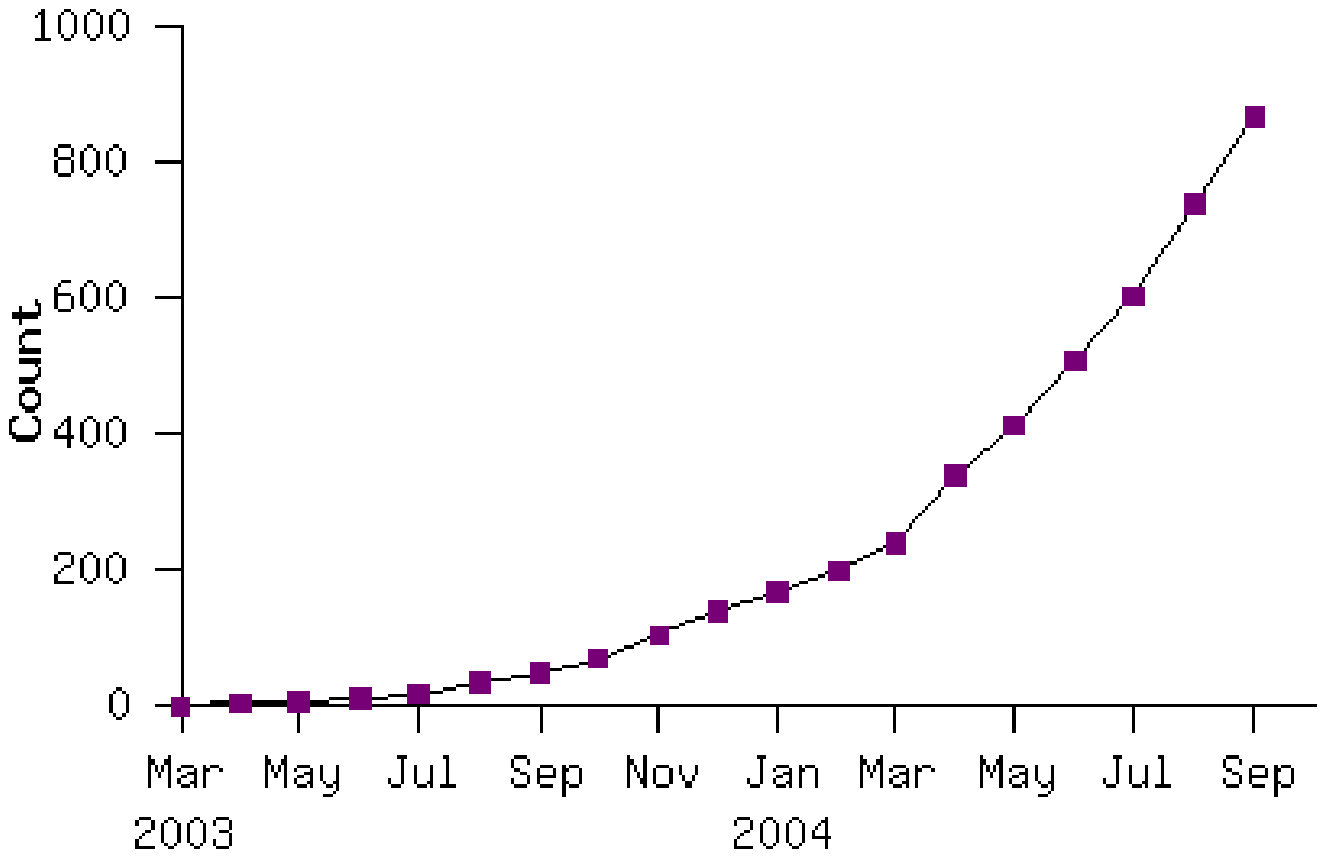
Example: Counting words



Other examples

- Distributed grep
  - Search for words in lots of documents.
  - Map: emit a line if it matches a given pattern. Produce $(file, line)$ pairs.
  - Reduce: copy the intermediate data to the output.
- Count URL access frequency
  - Find the frequency of each URL in web logs.
  - Map: process logs of web page access. Produce $(url, 1)$ pairs.
  - Reduce: add all values for the same URL.
- Reverse web-link graph
  - Find where page links come from.
  - Map: output $(target, source)$ pairs for each link $target$ in a web page $source$.
  - Reduce: concatenate the list of all source URLs associated with a target.

MapReduce is widely applicable



Number of MapReduce programs in Google code source tree.

Fault tolerance

- Master pings each worker periodically.
  - If no response is received within a certain delay, the worker is marked as failed.
  - Map or Reduce tasks given to this worker are reset back to the initial state and rescheduled for other workers.
  - Task completion is committed to master to keep track of history.

[Q] What abstraction does this use?

[Q] What if the master node fails? How would you fix that?

Redundant execution

- Slow workers significantly lengthen completion time
  - Because of other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

Locality

- Input and output files are stored on a distributed file system.
  - e.g., GFS or HDFS.
- Master tries to schedule Map workers near the data they are assigned to.
  - e.g., on the same machine or in the same rack.
- This results in thousands of machines reading input at local disk speed.
  - Without this, rack switches limit read rate.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

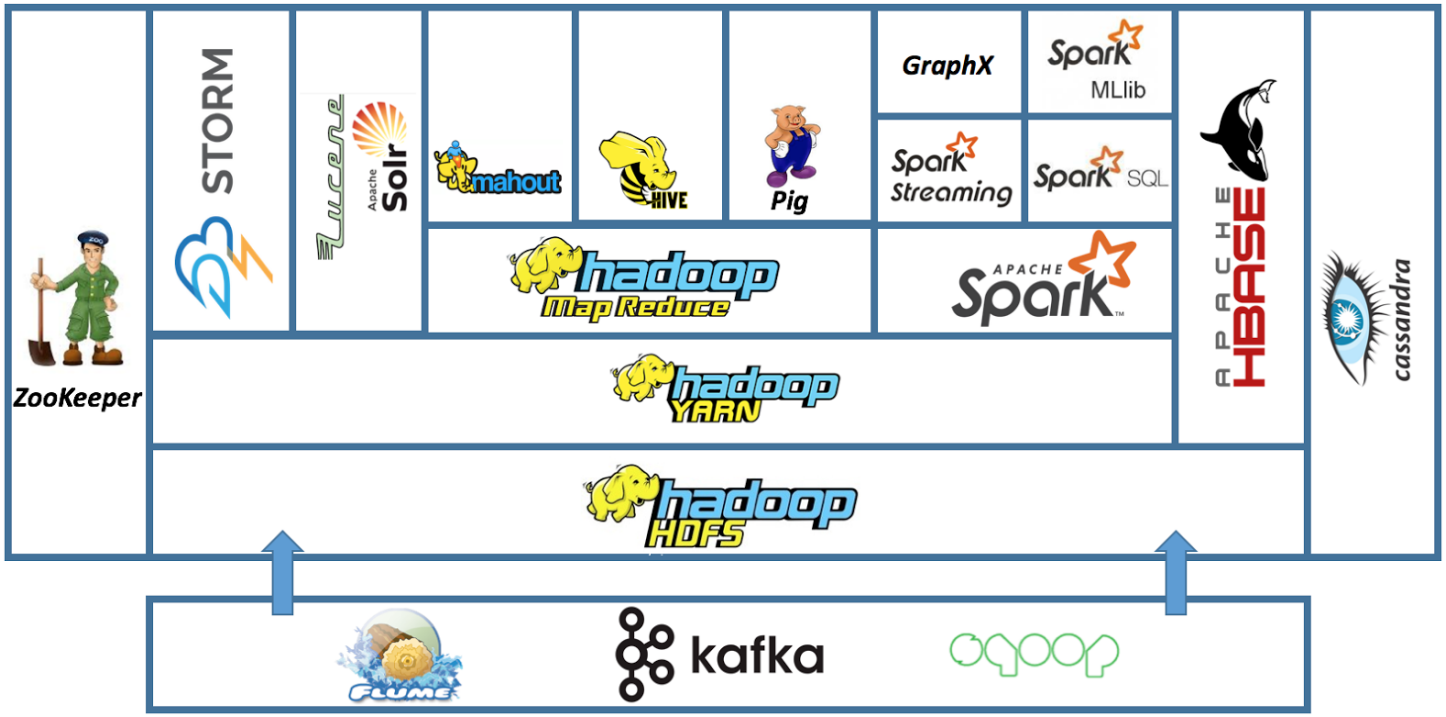
Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then

Google, 2004.

Hadoop Ecosystem (1)



Hadoop Ecosystem (2)

- Hadoop HDFS: A distributed file system for reliably storing huge amounts of unstructured, semi-structured and structured data in the form of files.
- Hadoop MapReduce: A distributed algorithm framework for the parallel processing of large datasets on HDFS filesystem. It runs on Hadoop cluster but also supports other database formats like Cassandra and HBase.
- Cassandra: A key-value pair NoSQL database, with column family data representation and asynchronous masterless replication.
    - Cassandra is built upon an architecture similar to a DHT.
- HBase: A key-value pair NoSQL database, with column family data representation, with master-slave replication. It uses HDFS as underlying storage.
- Zookeeper: A distributed coordination service for distributed applications.
    - It is based on a Paxos algorithm variant called Zab.

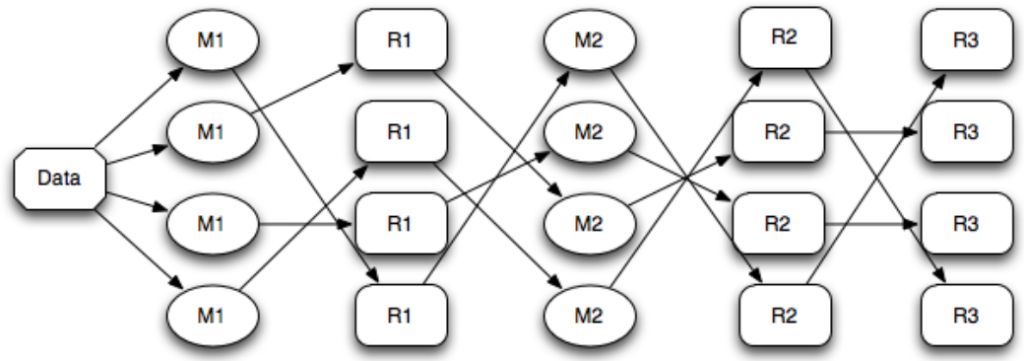
Hadoop Ecosystem (3)

- Pig: Pig is a scripting interface over MapReduce for developers who prefer scripting interface over native Java MapReduce programming.
- Hive: Hive is a SQL interface over MapReduce for developers and analysts who prefer SQL interface over native Java MapReduce programming.
- Mahout: A library of machine learning algorithms, implemented on top of MapReduce, for finding meaningful patterns in HDFS datasets.
- Yarn: A system to schedule applications and services on an HDFS cluster and manage the cluster resources like memory and CPU.
- Flume: A tool to collect, aggregate, reliably move and ingest large amounts of data into HDFS.
- ... and many others!

Spark

MapReduce programmability

- Most applications require multiple MR steps.
  - Google indexing pipeline: 21 steps
  - Analytics queries (e.g., count clicks and top-K): 2-5 steps
  - Iterative algorithms (e.g., PageRank): 10s of steps
- Multi-step jobs create spaghetti code
  - 21 MR steps $\rightarrow$ 21 mapper + 21 reducer classes
  - Lots of boilerplate code per step



Chaining MapReduce jobs.

Problems with MapReduce

- Over time, MapReduce use cases showed two major limitations:
  - not all algorithms are suited for MapReduce.
    - e.g., a linear dataflow is forced.
  - it is difficult to use for exploration and interactive programming.
  - there are significant performance bottlenecks in iterative algorithms that need to reuse intermediate results.
    - e.g., saving intermediate results to stable storage (HDFS) is very costly.
- That is, MapReduce does not compose so well for large applications.
- For this reason, dozens of high level frameworks and specialized systems were developed.
  - e.g., Pregel, Dremel, FI, Drill, GraphLab, Storm, Impala, etc.

Spark

- Like Hadoop MapReduce, Spark is a framework for performing distributed computations.
- Unlike various earlier specialized systems, the goal of Spark is to generalize MapReduce.
- Two small additions are enough to achieve that goal:
  - fast data sharing
  - general direct acyclic graphs (DAGs).
- Designed for data reuse and interactive programming.

Programmability

```
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();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(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);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

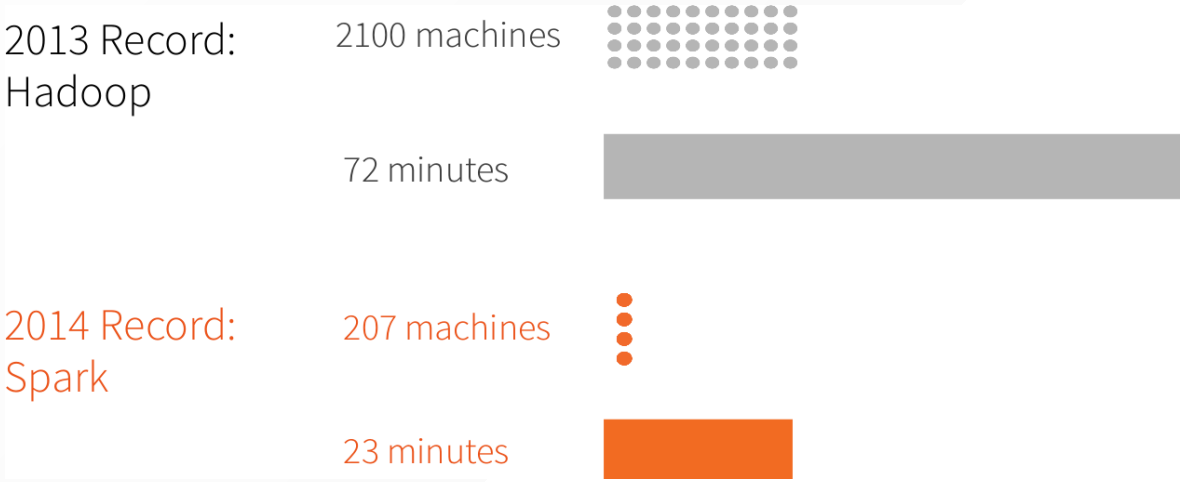
```
1  val f = sc.textFile(inputPath)
2  val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3  w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Performance

Time to sort 100TB:

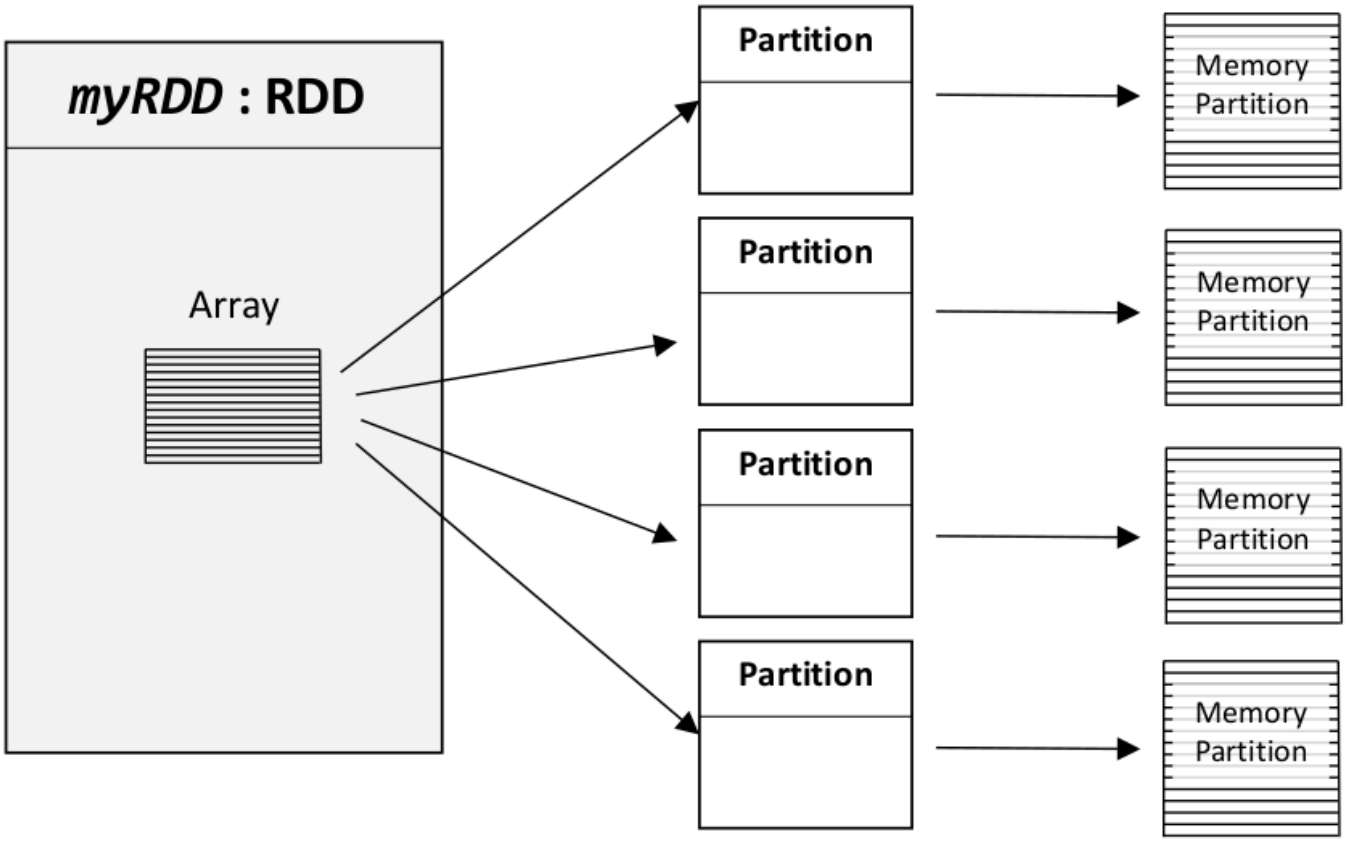


Credits: sortbenchmark.org

RDD

- Programs in Spark are written in terms of a Resilient Distributed Dataset (RDD) abstraction and operations on them.
- An RDD is a fault-tolerant read-only, partitioned collection of records.
  - Resilient: built for fault-tolerance (it can be recreated).
  - Distributed: content is divided into atomic partitions, usually stored in memory and across multiple nodes.
  - Dataset: collection of partitioned data with primitive values or values of values.
- RDDs can only be created through deterministic operations on either:
  - data in stable storage, or
  - other RDDs.

RDD

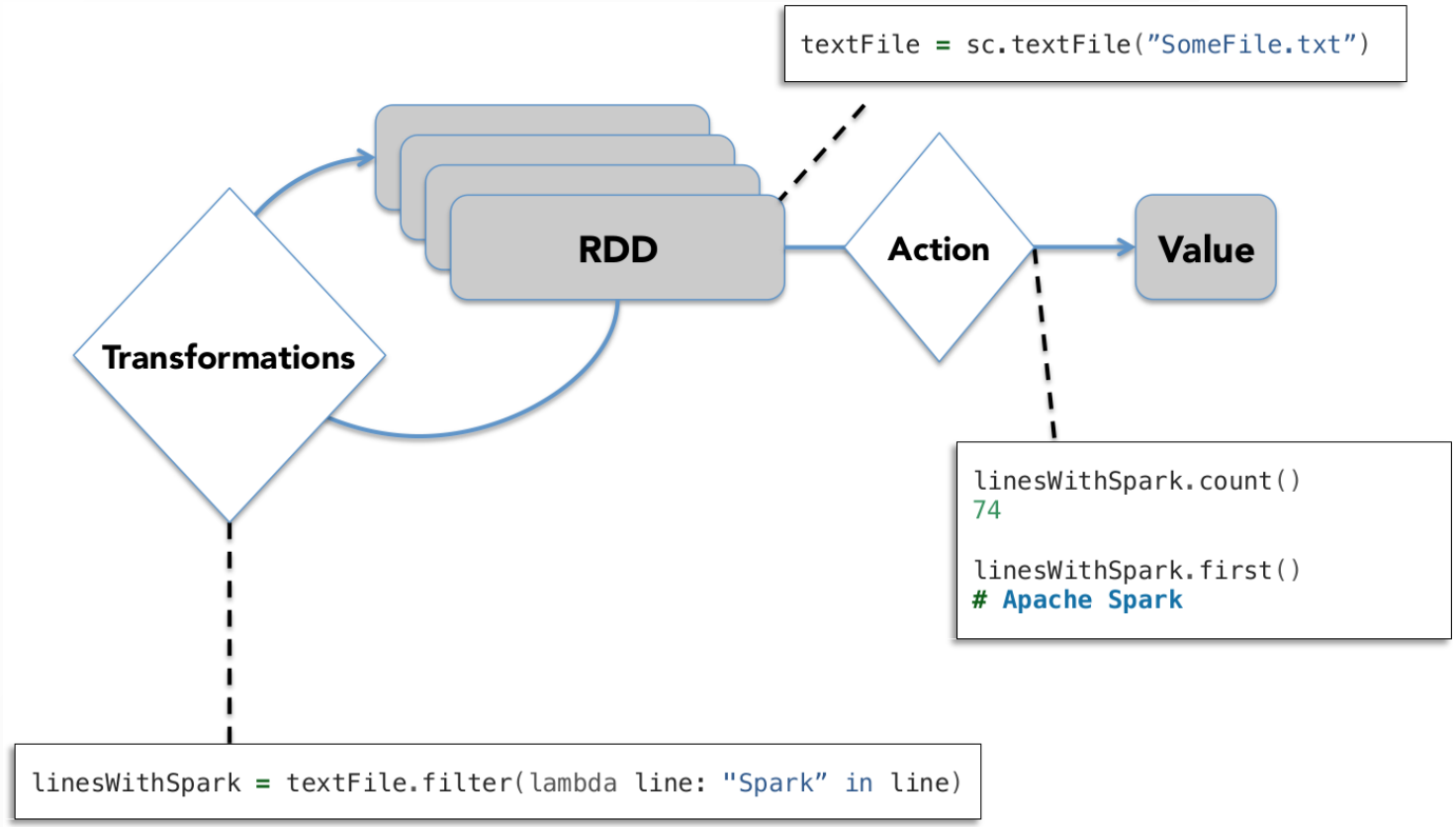


Credits: Tony Duarte

Operations on RDDs

- Transformations: $f(\text{RDD}) \rightarrow \text{RDD'}$
  - Coarse-grained operations only (à la pandas/numpy).
    - It is not possible to write to a single specific location in an RDD.
  - Lazy evaluation (not computed immediately).
  - e.g., `map` or `filter`.
- Actions: $f(\text{RDD}) \rightarrow v$
  - Triggers computation.
  - e.g., `count`.
- The interface also offers explicit persistence mechanisms to indicate that an RDD will be reused in future operations.
  - This allows for significant internal optimizations.

Workflow



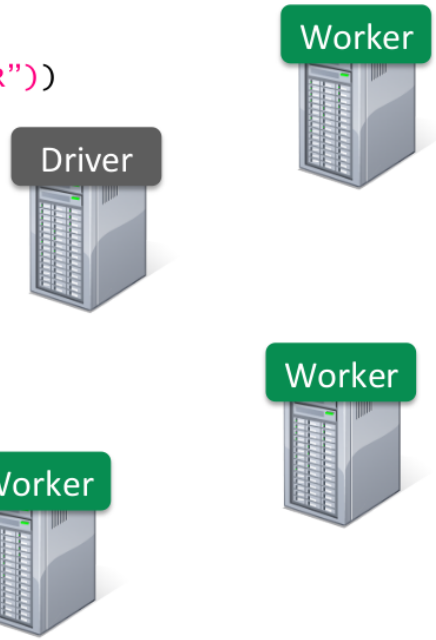
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (1)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



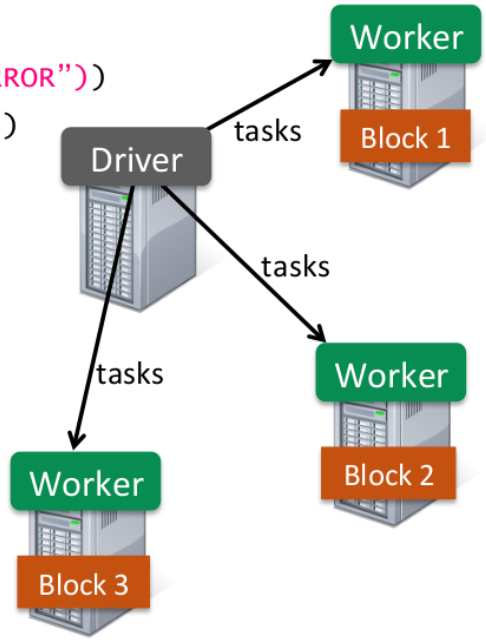
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (2)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



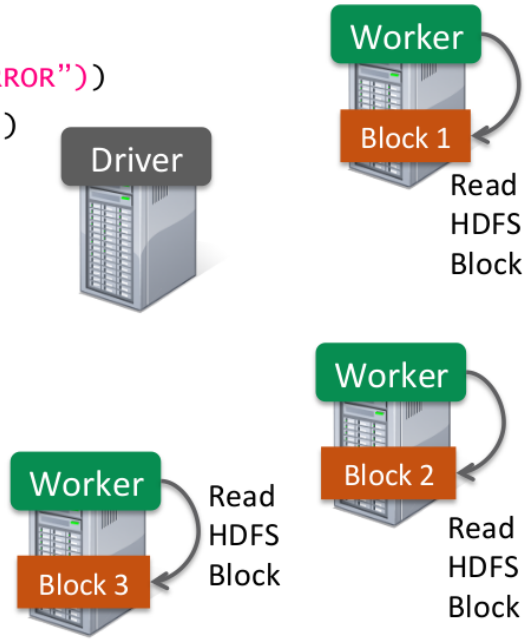
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (3)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



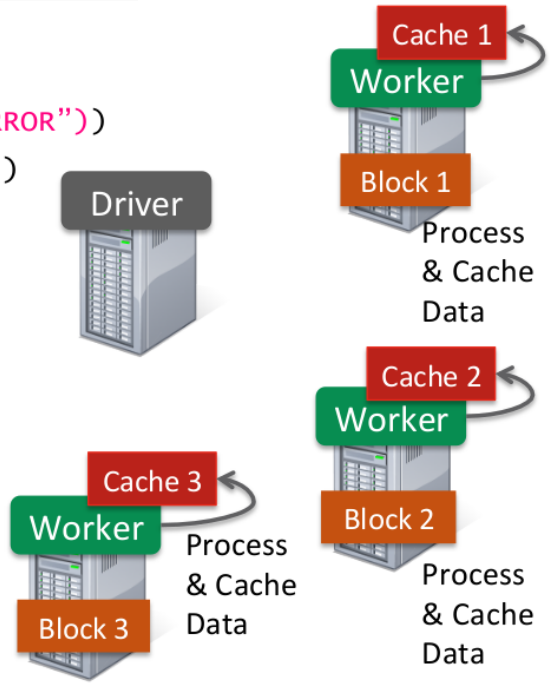
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (4)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



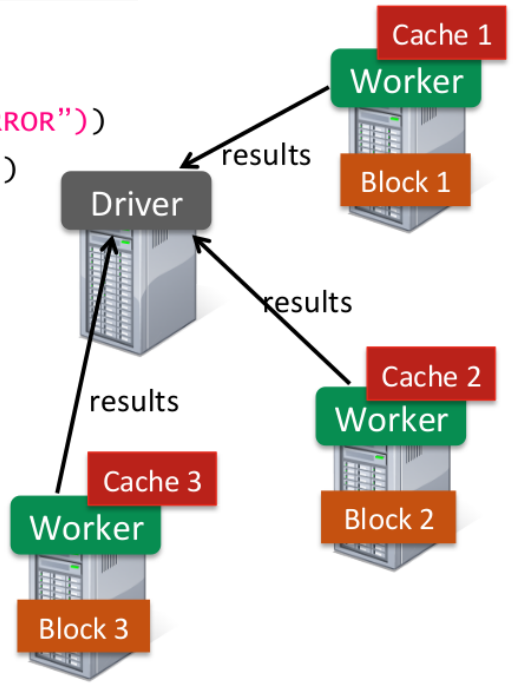
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (5)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



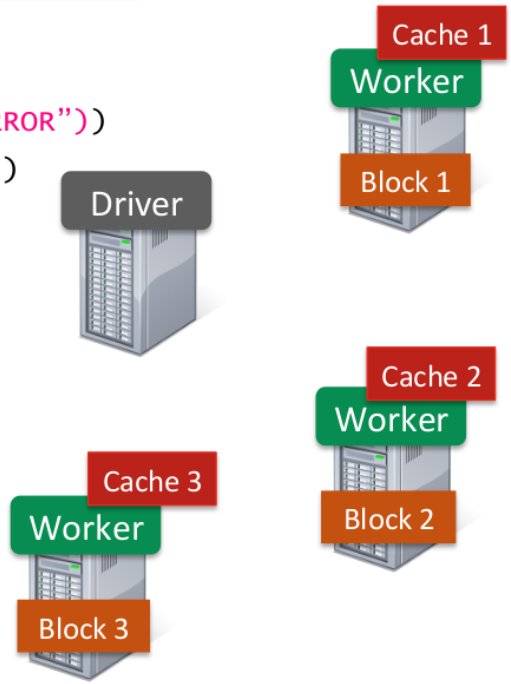
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (6)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



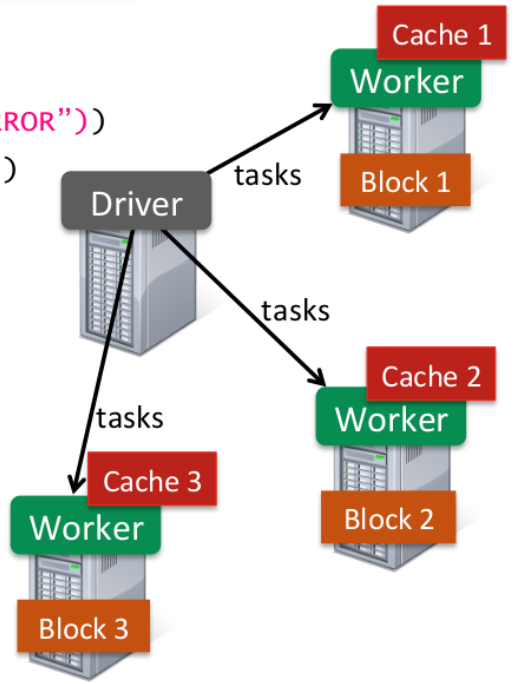
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (7)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



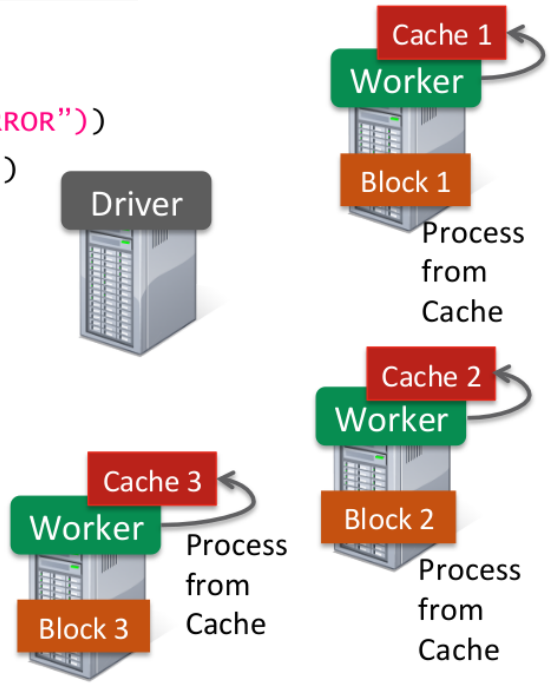
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (8)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



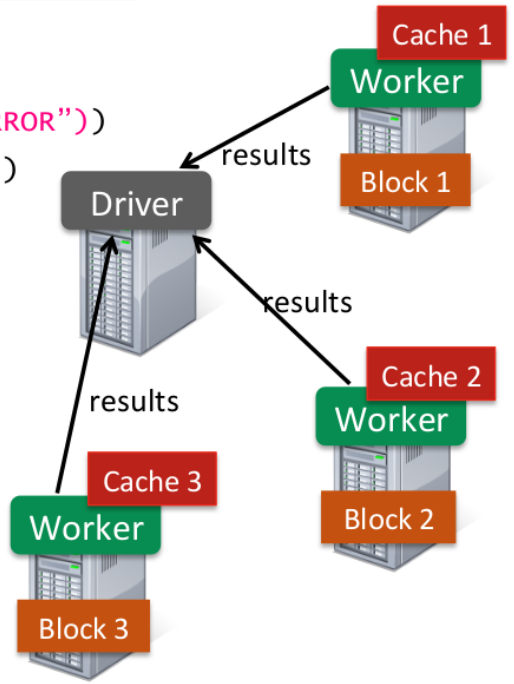
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Example: Log mining (9)

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



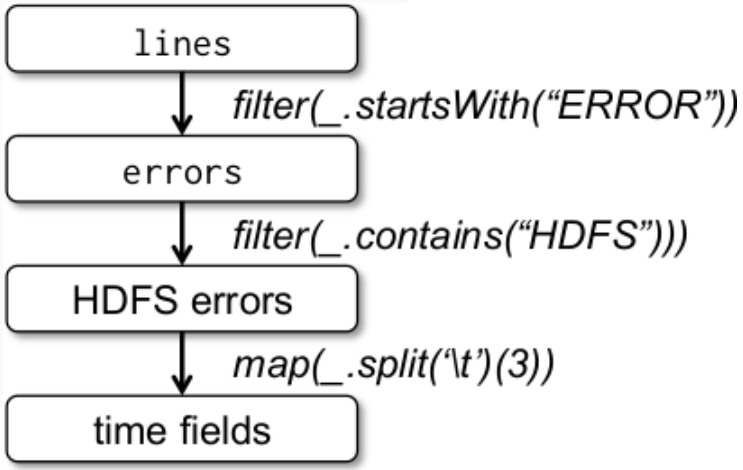
Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Rich, high-level API

map	reduce	sample
filter	count	take
sort	fold	first
groupBy	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
…	zip	save
	…	…

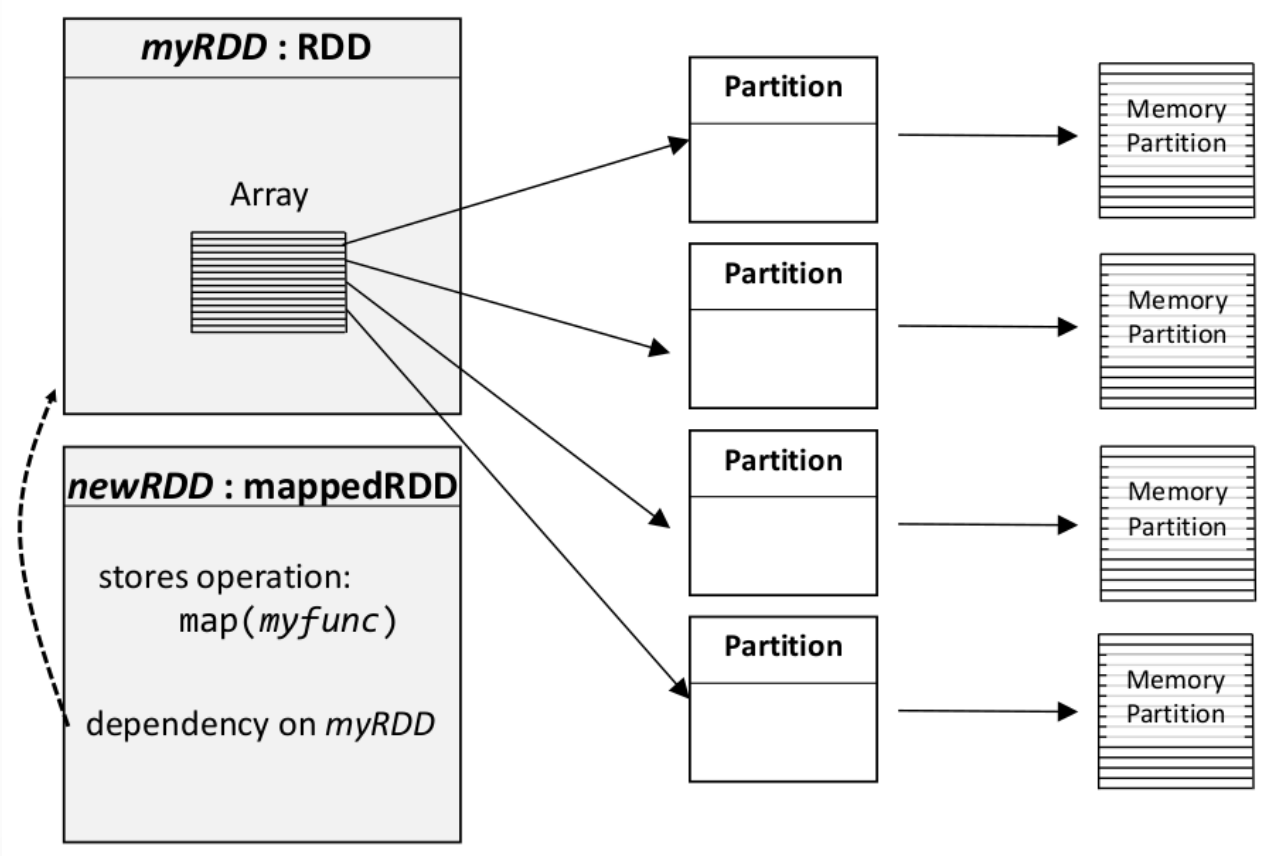
Lineage

- RDDs need not be materialized at all times.
- Instead, an RDD internally stores how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage.
  - This derivation is expressed as coarse-grained transformations.
- Therefore, a program cannot reference an RDD that it cannot reconstruct after a failure.



Lineage

```
newRDD = myRDD.map(myfund)
```

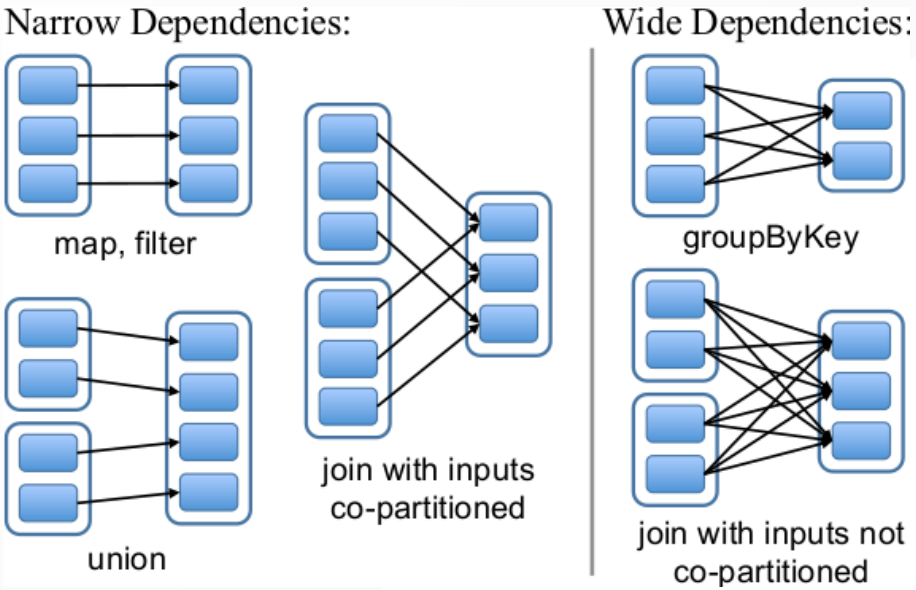


Credits: Tony Duarte

Representing RDDs

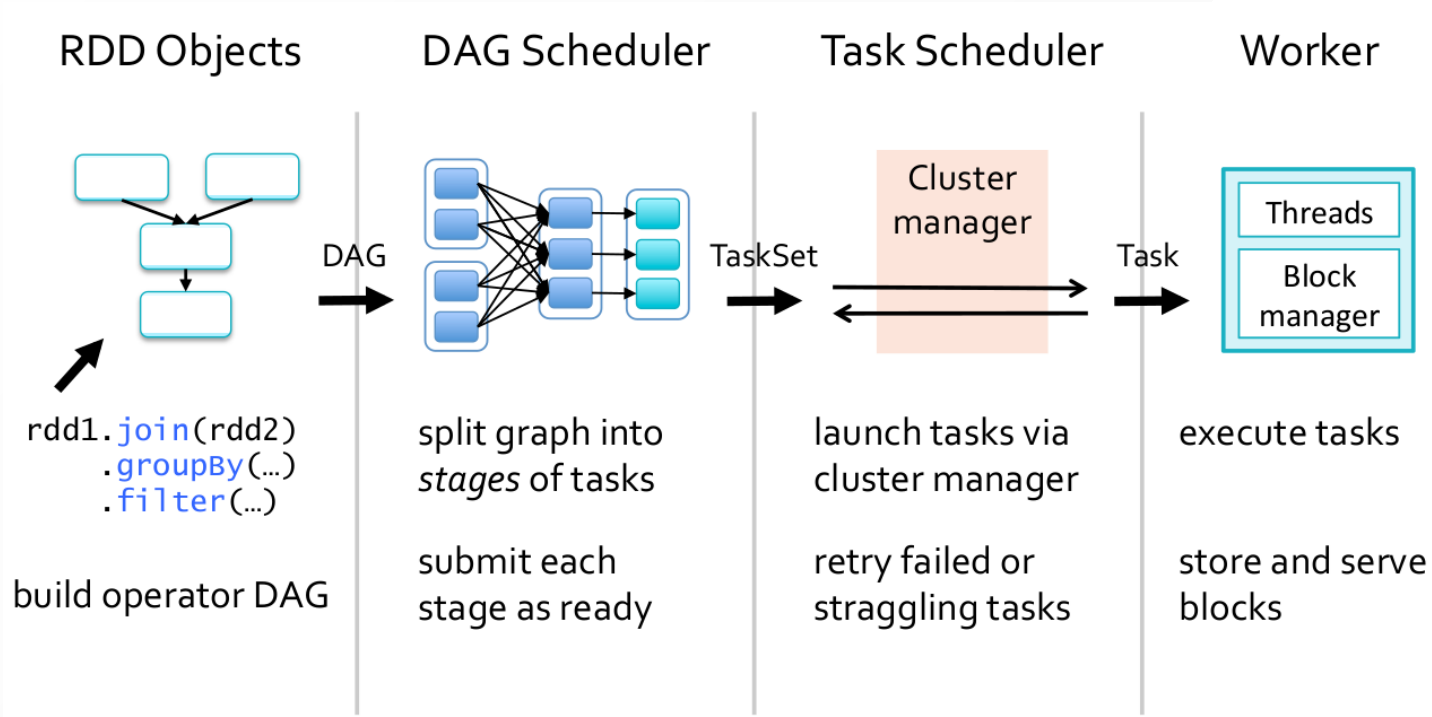
- RDDs are built around a graph-based representation (a DAG).
- RDDs share a common interface:
  - Lineage information:
    - Set of partitions.
    - List of dependencies on parents RDDs.
    - Function to compute a partition (as an iterator) given its parents.
  - Optimized execution (optional):
    - Preferred locations for each partition.
    - Partitioner (hash, range)

Dependencies



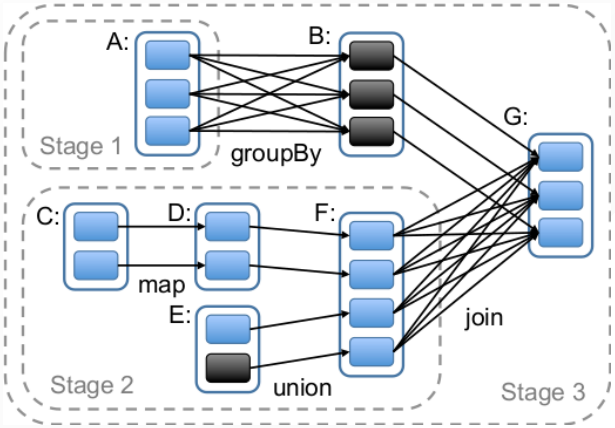
- Narrow dependencies: each partition of the parent RDD is used by at most one partition of the child RDD.
  - Allow for pipelined execution on one node.
  - Recovery after failure is more efficient with a narrow dependency, as only the lost parents partitions need to be recomputed.
- Wide dependencies: multiple child partitions may depend on a parent partition.
  - A child partition requires data from all its parents to be recomputed.

Execution process



Credits: Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.

Job scheduler



- Whenever an action is called, the scheduler examines that RDD's lineage graph to build a DAG of stages to execute.
- Each stage contains as many pipeline transformations with narrow dependencies as possible.
- The boundaries of the stages are
  - the shuffle operations required for wide dependencies, or
  - already computed partitions that can short-circuit the computation of a parent RDD.
- The scheduler launches tasks to a lower-level scheduler to compute missing partitions from each stage until it has computed the target RDD.
  - One task per partition.
- Tasks are assigned to machines based on data locality.

Fault tolerance

- If a task fails, it is rescheduled on another node, as long as its stage's parents are still available.
- If some stages have become unavailable, all corresponding tasks are resubmit to compute the missing partitions in parallel.

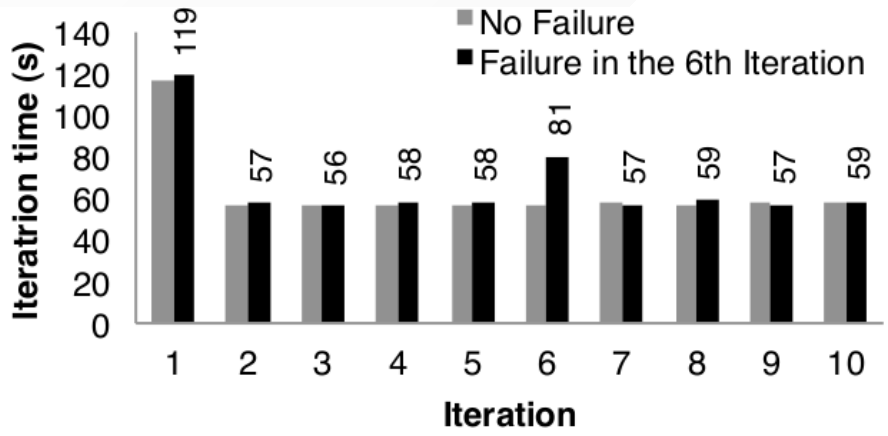


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Dataflow programming

- Spark builds upon the dataflow programming paradigm.
- Dataflow programming models a program as a directed graph of the data flowing between operations.
- An operation runs as soon as all of its inputs become valid.
- Dataflow languages are inherently parallel and work well in large, decentralized systems.
- Modern examples:
  - Scala
  - Spark
  - Tensorflow

Summary

- High-level abstractions enable cloud programming over clusters.
  - Without having to handle parallelization, data distribution, load balancing, fault tolerance, ...
- MapReduce is a parallel programming model based on map and reduce operations.
  - Best suited for embarrassingly parallel and linear tasks.
  - Its simplicity is a disadvantage for complex iterative programs for interactive exploration.
- Spark generalizes MapReduce by making use of:
  - fast data sharing (data resides in memory)
  - general direct acyclic graphs of operations.

References

- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
- Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
- Xin, Reynold. "Stanford CS347 Guest Lecture: Apache Spark". 2015.