Meetup 18 - Spark

Apache Spark

Matei Zaharia

ACM Doctoral Dissertation Award

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.,* cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the clus-

databricks

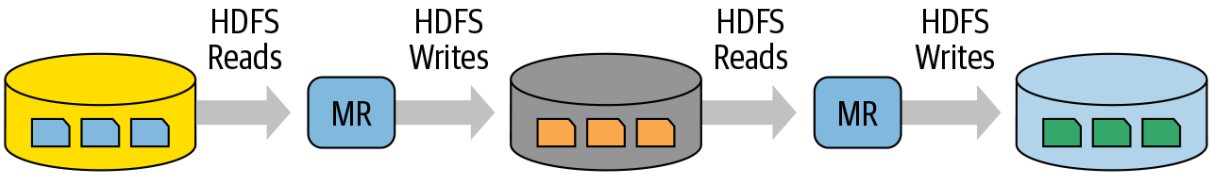
The book Learning Spark 2nd Edition

GFS: fault-tolerant and distributed file system in a cluster farm

Bigtable: scalable storage of structured data across GFS

MapReduce: parallel programming model for large-scale processing of data distributed over GFS and Bigtable

Apache Hadoop: Hadoop Common, MapReduce, HDFS (Hadoop File System), Apache Hadoop YARN. open source community



💡 Was there a way to make Hadoop and MR simpler and faster?

Researchers at UC Berkeley came up with Spark. They observed MR is inefficient for interactive or iterative computing jobs, and a complex framework to learn. Early paper.

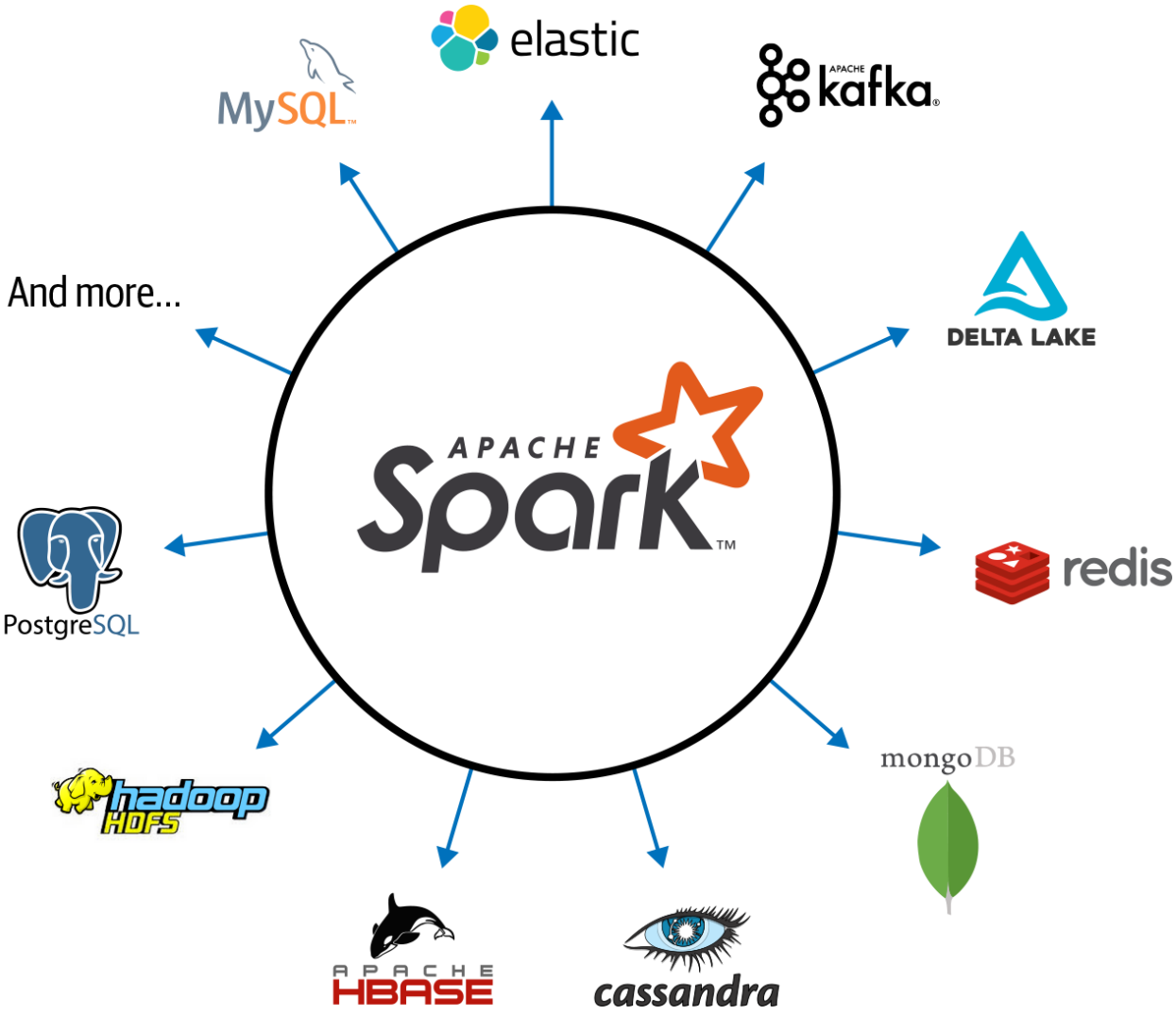
Spark website. Ideas

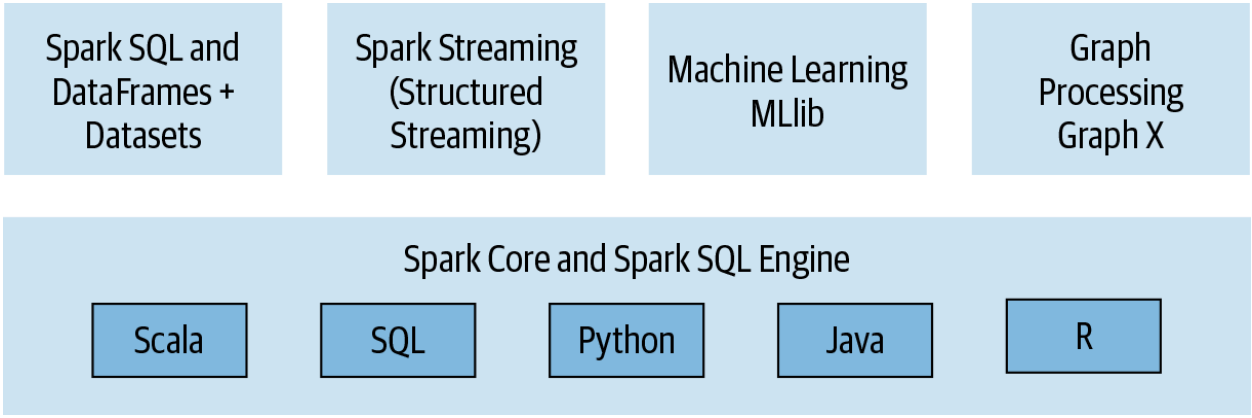
1. highly fault tolerant
2. parallel
3. in-memory storage for intermediate results between map and reduce computations
4. offer easy and composable APIs (MLlib, Spark SQL, Structured Streaming, GraphX)
5. support other workloads in a unified manner

The researchers donate the Spark project to ASF, and formed Databricks

Design Principles

1. **Speed**: in-memory, query computations in DAG, physical execution engine Tungsten uses whole-stage code gen.
2. **Ease of Use**: Resilient Distributed Datasets, transformations, actions as operations
3. **Modularity:** unified processing engine
4. **Extensibility**: decouples storage and compute


elastic
MySQL
kafka
And more...
DELTA LAKE
APACHE
Spark
redis
PostgreSQL
mongoDB
hadoop
HDFS
APACHE
HBASE
cassandra


Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R

Paper

RDD are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.

To achieve fault tolerance efficiently, RDD provide a restricted form of shared memory, based on coarse-grained transformations.

1. Intro

the falls of MapReduce

1. they lack abstractions for leveraging distributed memory → inefficient for those that reuse intermediate results across multiple computations → inefficient on iterative machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression.

   Another compelling use case is interactive data mining, where a user runs multiple ad-hoc queries on the same subset of the data.

2. To reuse data, MapReduce needs to write it to a distributed file system → overheads on data replication, disk I/O, and serialization

Entering resilient distributed datasets (RDDs)

1. efficient data reuse in a broad range of applications

> 💡 RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

How to persist?

How to control?

and what are the operators?

The challenge for designing RDD

🔥 how to define a programming interface that can provide fault tolerance *efficiently*.

❓ How does MapReduce provide fault tolerance?

Answer: any map task or reduce task in-progress failed are reset to idle state. completed map task needs to be re-execute on failed machine, but not reduce task, since the output is stored in GFS.

Existing solution has fine-grained updates to mutable state.

Distributed shared memory, databases, key-value stores and Piccolo

The only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network.

How's RDD different?

RDD provides coarse-grained transformations (e.g. map, filter, join) that apply the same operation to many data items.

This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (lineage) rather than the actual data.

There is a caveat to it, we will talk about it in chap 5.4

If a partition of an RDD is lost, the lineage knows how it was derived form other RDDs to recompute just that partition → recovering lost data without replication.

❓ How does lineage help to not replicate the data? What's the advantage here?

❓ How does coarse-grained operators fit into the use cases of interactive data mining?

Spark and RDD

RDDs are implemented in Spark, which provides a convenient language-integrated programming interface in Scala (later to Python and other PLs).

The author cites:

> Spark is the first system that allows a general-purposely programming language to be used at interactive speeds for in-memory data minings on clusters

Performance

20x faster than Hadoop for iterative application, speeds up a real-world data analytics report by 40x, scan 1TB dataset with 5-7s latency. (at that time)

2. Resilient Distributed Datasets (RDDs)

2.1 RDD Abstraction

💡 an RDD is a read-only, partitioned collection of records.

❓ from the definition, RDD is immutable. How does Spark represent versioned datasets?

Properties:

1. Can only be created through deterministic operations on either data in stable storage, or other RDDs by transformations, including *map, filter, join*.
2. RDDs **do not need to be materialized** at all times. Only the lineage is enough to reconstruct an RDD. (similar to Haskell lazy evaluated programs)

3. Users can control persistence and partitioning of RDD.

   Users can indicate which RDDs they will reuse and choose a storage strategy for them (in-memory).

   They can also ask that an RDD's elements be partitioned across machines based on a key in each record → useful for *placement optimizations*.

> ❓ What does based on a key in each record mean?

2.2 RDD API

Spark 2.4.0 ScalaDoc

Spark 2.4.0 ScalaDoc

https://spark.apache.org/docs/2.4.0/api/scala/index.html#org.apache.spark.sql.Dataset

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

1. define one or more RDDs through transformations.
2. they can apply actions to RDD that return a value to the application, or export data to a storage system.

3. `persist` to tell Spark to persist RDDs in memory by default, but it can spill them to disk if there is not enough RAM.

2.2.1 Example: Console Log Mining

suppose a web service is experiencing errors and programmer wants to search TBs of logs in the HDFS (Hadoop file system, similar to GFS) to find the cause.

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

//---------------👆 no work performed on the cluster 👆-----------------------//

// count the number of messages
errors.count() //<------- Spark store the partitions of errors in memory


// count errors mentioning mysql
errors.filter(_.contains("MySQL"))
      .count()
// return the time fields of errors mentioning HDFS
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

1. `errors.count()` is where Spark store the partitions of errors in memory
2. `lines` is not loaded to memory!
3. what does the lineage graph look like?

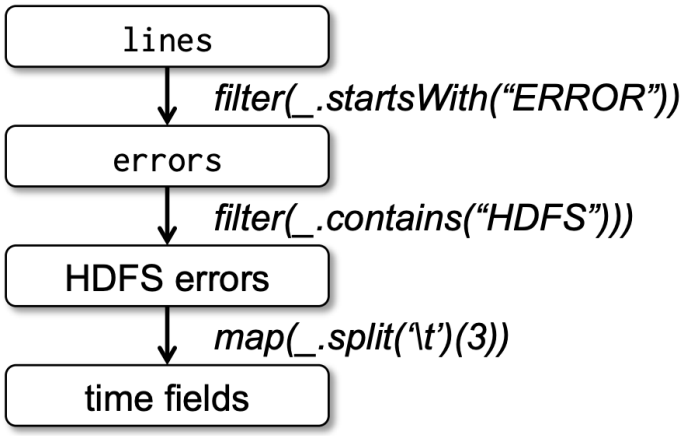


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

2.3 Advantages of the RDD Model

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

1. RDDs can only be created (writes) by transformations (coarse-grained). It does not allow writes to each memory location.

2. RDDs support fine-grained reads, behaving just like a large read-only lookup table.
3. only the partitions of an RDD need to be recomputed upon failure, and they can be recomputed in parallel on different nodes. (to the contrast, MapReduce assigns a worker to redo the work)
4. Since RDDs are deterministic and immutable, backup copies of slow tasks can run in parallel (to reduce stragglers). Similar to MapReduce. It is impossible to implement backup tasks in DSM

❓ What is memory consistency model?

🔥 My thought: the insight to Spark is that in data-intensive use cases, application typically does not need fine-grained control over the memory, especially writes.

2.4 Applications Not Suitable for RDDs

1. storage system for a web application
2. an incremental web crawler
3. any system that make synchronous fine-grained updates to shared state

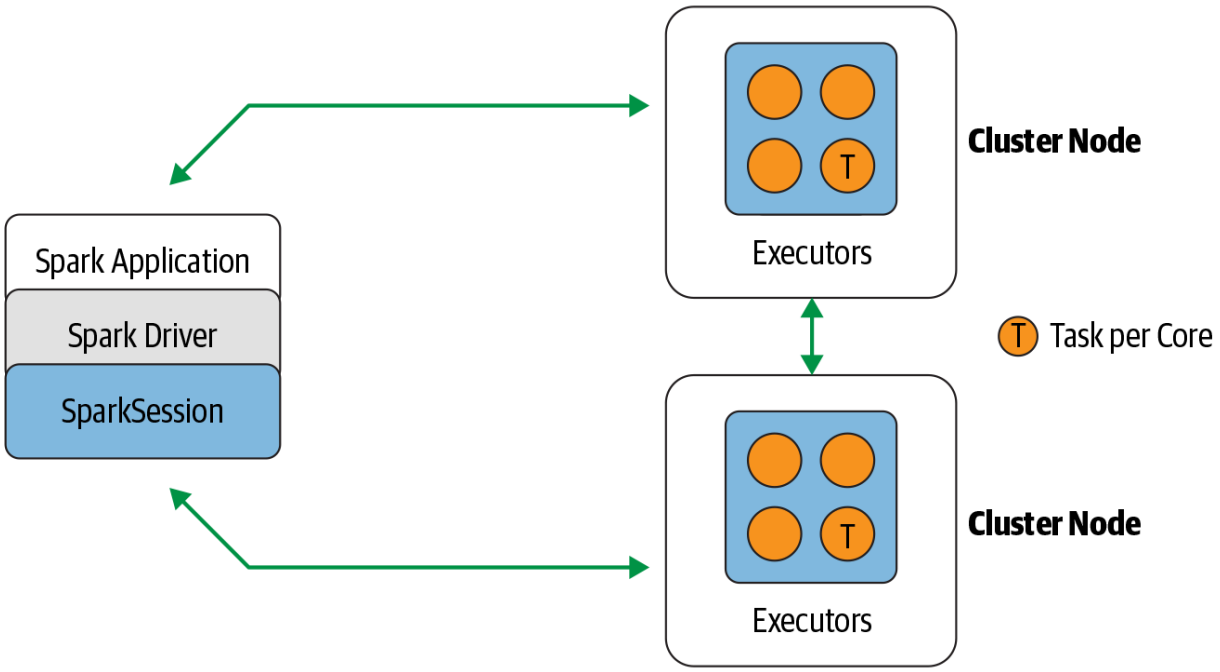
Spark Programming Interface ⚡

Why Scala?

1. conciseness, no much boilerplate
2. efficiency (static typing)

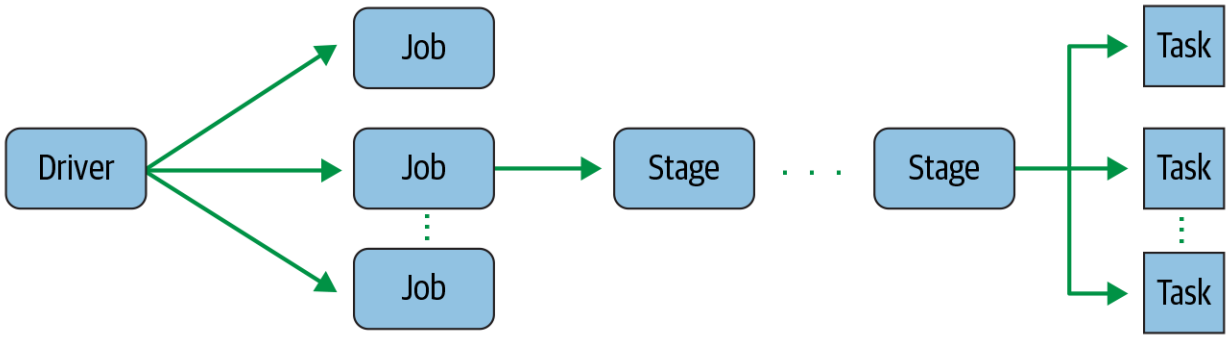
🔥 It is actually a surprise to me that Spark chooses Scala, a statically typed language. I would imagine dynamically typed language, like Python, fits more on the use cases of interactive data mining.

Spark Concepts



driver: it connects to a cluster of workers. It defines one or more RDDs and invoke actions on them. Driver also tracks the RDDs' lineage.

SparkSession: An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs.



Job is created when you perform actions. Each Job is a DAG of Task.

A Task is one core working on one partition of data. 16 core machine means 16 tasks on 16 partitions of data!

Scala and JVM

Scala represents each closure (in *map* etc.) as a Java object, and these objects can be serialized and send to another node across the network.

3.1 RDD Operations in Spark

Remember this

1. Transformations like `map` and `filter` are lazy! They do not compute/materialize RDDs. They build the lineage graph.
2. The compute is triggered when an action is performed. Examples are `count`, `collect`, `reduce`.

3.2 PageRank Example (Demo Time!)

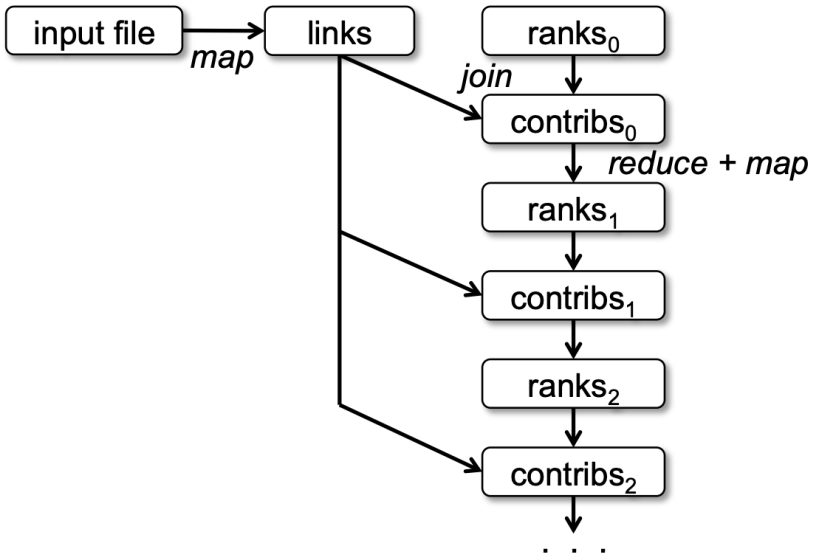


Figure 3: Lineage graph for datasets in PageRank.

> ❓ Spark can tolerate fault on failed jobs, but what if the machine that holds the lineage graph fails?

Answer: in the paper, they say "we do not yet tolerate scheduler failures, though replicating the RDD lineage graph would be straightfoward"

1. The lineage graph can grow very rapidly (in proportion to the loop iterations), it may be necessary to reliably replicate some of the versions of `ranks` to reduce fault recovery times.

   > ❓ How does this help?

2. `links` does not need to be replicated, because partitions of it can be rebuilt efficiently by rerunning a `map` on blocks of the input file.
3. optimization by controlling partitions of the RDDs.

   If we specify a partitioning for links, we can partition ranks in the same way to ensure that the join operation between links and ranks require no communication.

   ```
   links = spark.textFile(...)
               .map(...)
               .partitionBy(myPartFunc)
               .persist()
   ```

4. Representing RDDs

Representing RDDs through a common interface (5 pieces):

1. a set of partitions
2. a set of dependencies on parent RDDs
3. a function for computing the dataset based on its parents
4. metadata about its partitioning scheme and data replacement

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

How to represent dependencies?

narrow dependencies: where each partition of the parent RDD is used by at most one partition of the child RDD. `map`

wide dependencies: multiple child partitions may depend on it. `join`

Why this distinction?

1. narrow dependencies allow pipelined execution: `map` followed by `filter`
2. wide dependencies require data from all parent partitions
3. recovery is efficient with narrow dependency. only the lost parent partitions need to be recomputed and they can be recomputed in parallel on other nodes.
4. wide dependency is hard! a single failed node might cause the loss of some partition from all the ancestors of an RDD - requiring a complete re-execution!

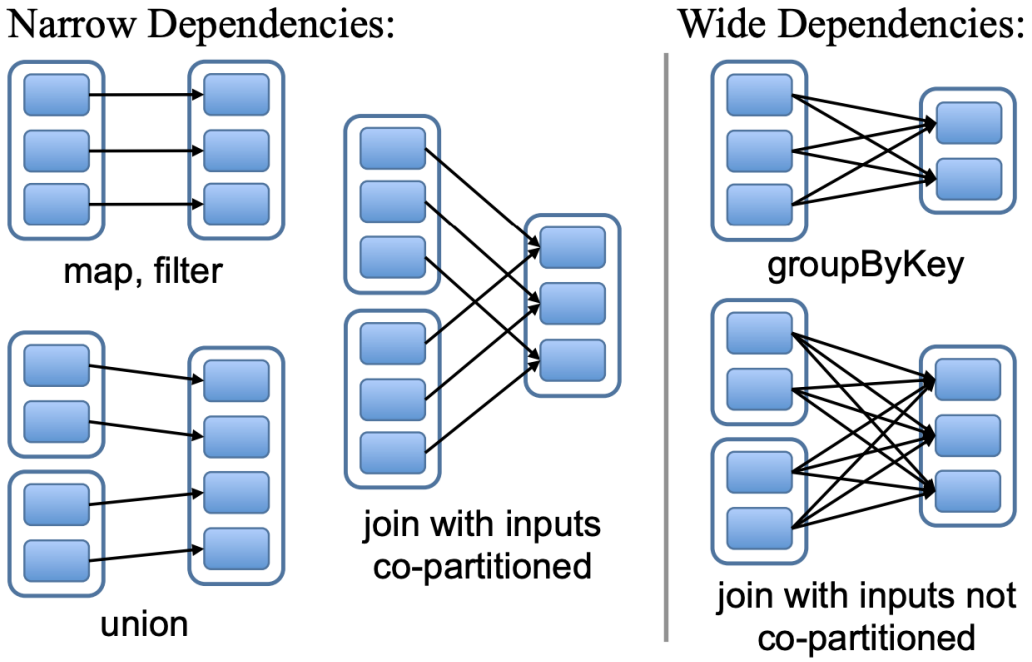


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

5. Implementation

Spark implemented in 14,000 lines of Scala (not bad, right?)

5.1 Job Scheduling

based on Dryad: here is a video if you are interested, and delay scheduling: another Matei's paper.

When action is performed on an RDD, the scheduler examine that RDD's lineage graph to build a DAG of stages to execute. The boundaries of stage are wide dependencies.

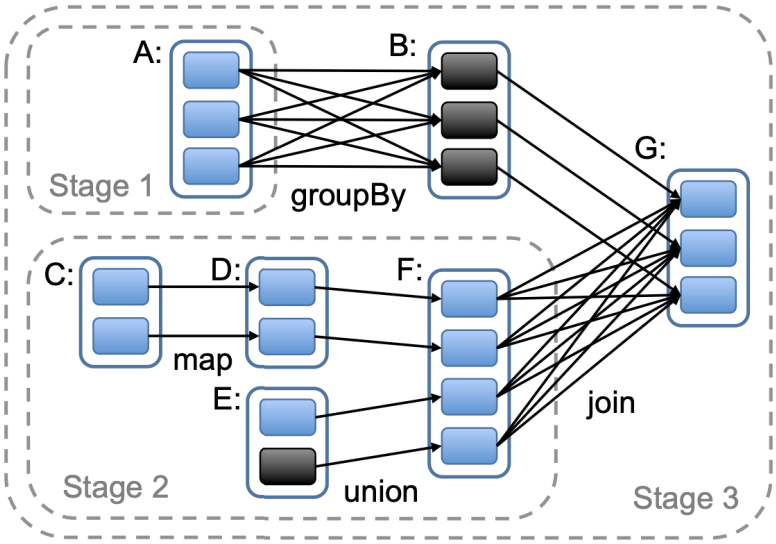


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

5.2 Interpreter Integration

5.3 Memory Management

1. in-memory storage as deserialized Java objects (fastest performance)
2. in-memory storage as serialized data (space-efficient)
3. on-disk storage (backup for no enough RAM)

5.4 Support for Checkpointing

wide dependency failure can be time-consuming for RDDs to recover.

checkpointing the RDDs in the background (immutability!) can be accomplished.

6. Evaluation

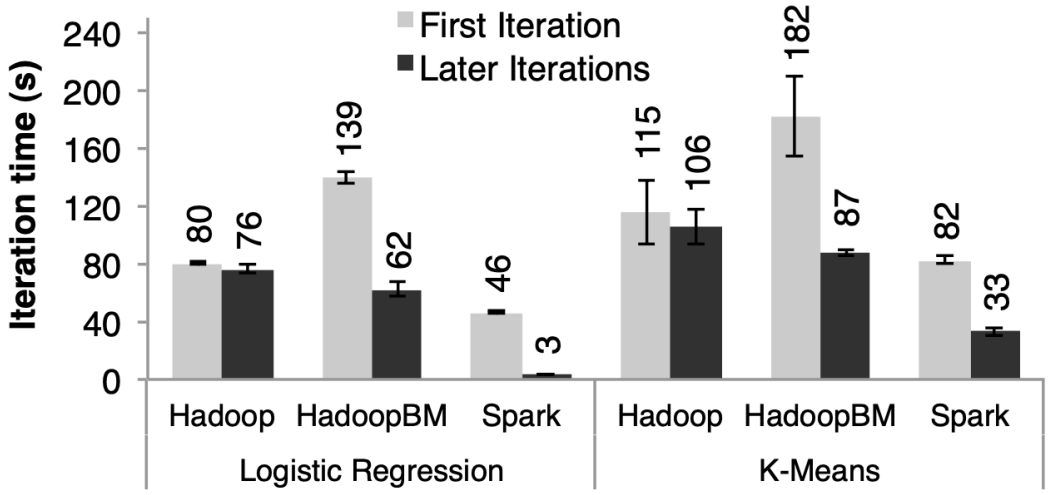


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

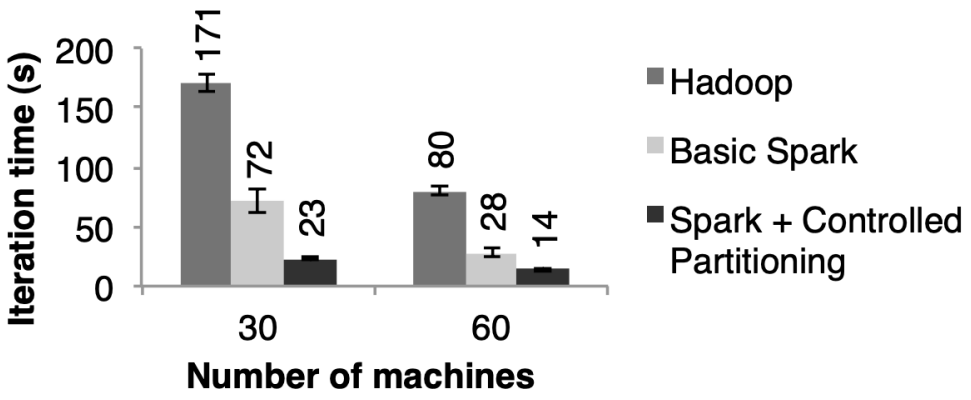


Figure 10: Performance of PageRank on Hadoop and Spark.

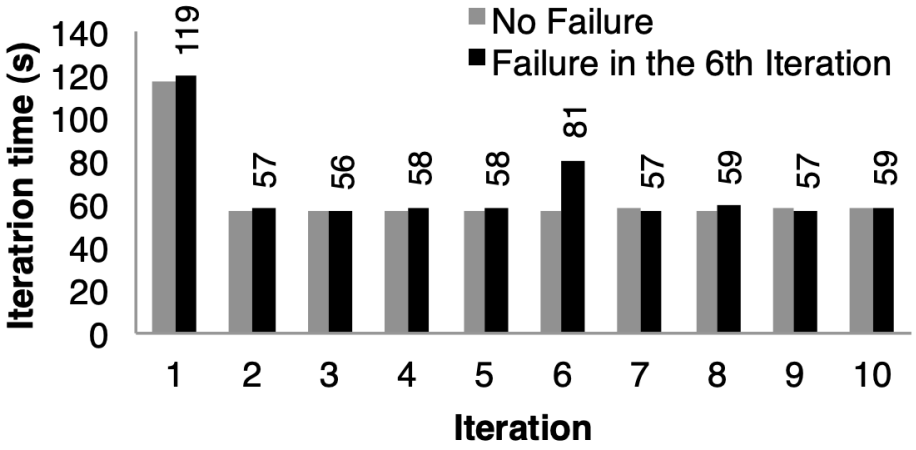


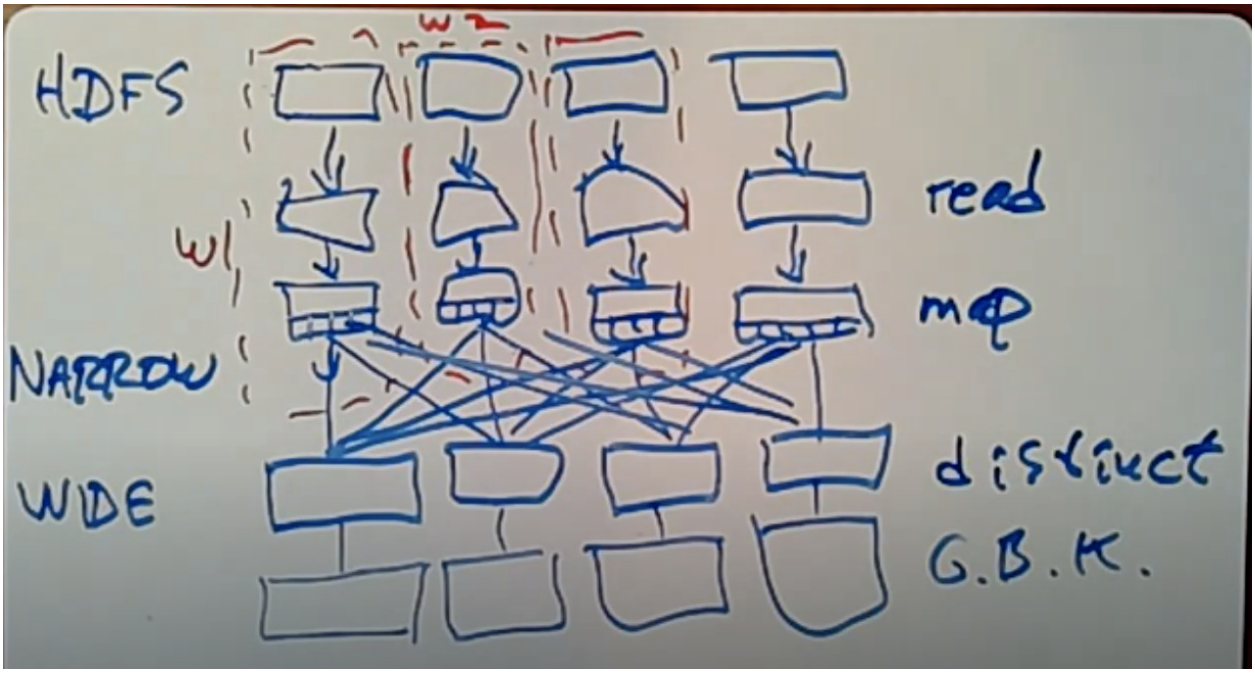
Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Video 💻

Why are we looking at Spark?

1. widely-used for datacenter operations
2. generalized MapReduce into dataflow
3. supports iterative applications better than MapReduce
4. successful research: ACM doctoral thesis award

```
1      val lines = spark.read.textFile("in").rdd
2      val links1 = lines.map{ s =>
3        val parts = s.split("\\s+")
4        (parts(0), parts(1))
5      }
6      val links2 = links1.distinct()
7      val links3 = links2.groupByKey()
8      val links4 = links3.cache()
9      var ranks = links4.mapValues(v => 1.0)
10
11     for (i <- 1 to 10) {
12       val jj = links4.join(ranks)
13       val contribs = jj.values.flatMap{
14         case (urls, rank) =>
15           urls.map(url => (url, rank / urls.size))
16       }
17       ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
18     }
19
20     val output = ranks.collect()
21     output.foreach(tup => println(s"${tup._1} has rank:  ${tup._2} ."))
```



Generating a lineage graph for datasets. It does not process the data.

The lineage graph can tell the driver a lot of information about how to transform the data.

1. stream records, one at a time, though sequence of narrow transformations
    1. increases locality, good for CPU data caches
    2. avoids having to store entire partition of records in memory
2. notice when shuffles aren't needed because inputs already partitioned in the same way.

`distinct`, `groupByKey` and `join`: these need to look at data from *all* partitions, not just one because all recods with a given key must be considered together. These are the paper's **wide** dependencies (as opposed to "narrow"

How are wide dependencies implemented?

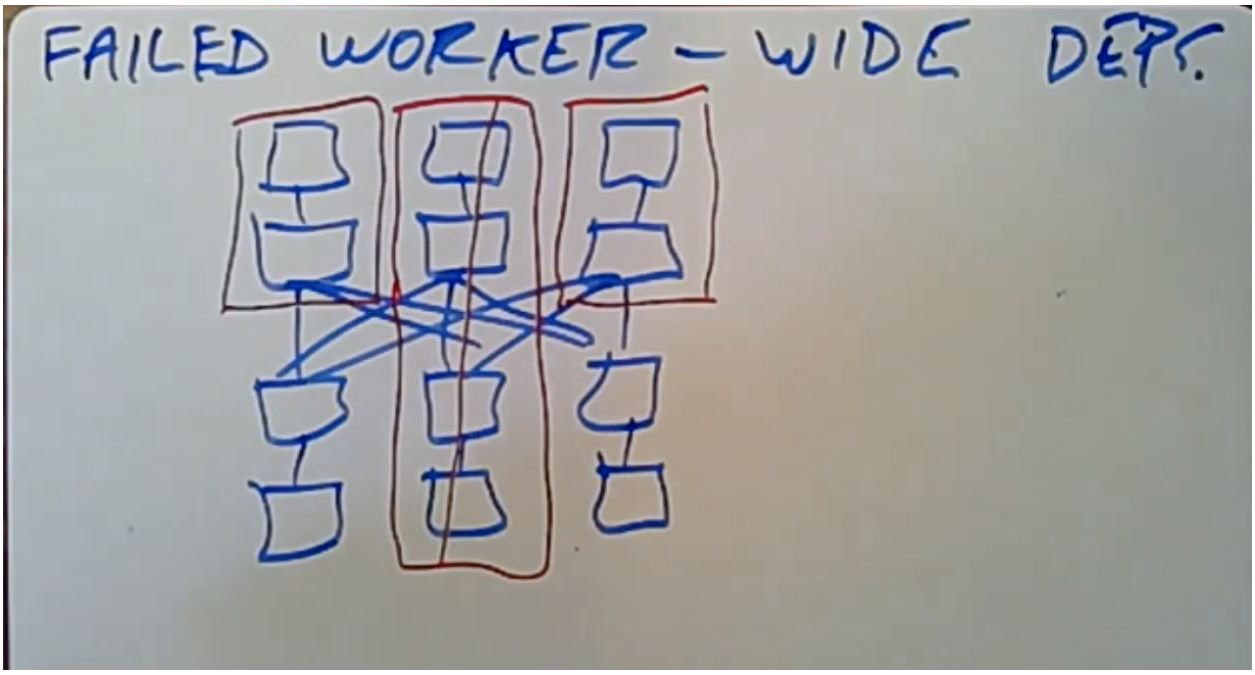
- This looks like Map intermediate output in MapReduce
- The driver knows where the wide depenencies are (between `map` and `distinct`
- The data must be shuffled into new partitions e.g. bring all of a given key together after the upstream transformation: split output up by shuffle criterion, and arrange into buckets in memory, one per downstream parition.
- Before the downstream transformationL (wait until upstream transformation completes — driver manages this). Each worker fetches its bucket from each upstream worker and now the data is partitioned in a differnt way.
- "wide" operation is expensive! All data is moved across the network

Fault Tolerance

Why we need FT: it's expensive to repeat the process (unlike the database which absolutely cannot loss data)

One worker fail: re-compute what the worker is responsible for on some other machines. Each machine responsible for multiple partitions. So load can be spread and re-computation is pretty fast for narrow dependencies, only lost paritions have to be re-executed.

How about failures when there are wide dependencies?



- re-computation one failed partition requires information from **all** partitions, so **all** partitions may need to re-execute from the start!
- Spark supports checkpoints to HDFS to cope with this. Driver only has to recompute along lineage from latest checkpoint. For page-rank, perhaps checkpoint ranks every 10th iteration.

Lessons

- When the cluster grows bigger, the change of a machine failure approaches 1.
- Spark computations be deterministic, RDD are immutable → recover from failure by simply re-computing one partition.
- Non-determinism → no good recovery strategy. Shared memory is being critized.

Limitations

- Gearted up for batch procssing of bulk data
- If you want to process bank transfer, bank query, or amazon orders, Spark is not a good choice for online processing.
- Stream processing (Spark streaming)

- Transformations are functional — turn input into output

Summary

- Spark as an evolution of MapReduce.
- It makes data-flow graph explicit
- RDD is kind of deprecated, replaced by DataFrames and Datasets. Further reading here

Further Reading

1. Spark: Cluster Computing with Working Sets. Z. Matei
2. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Z. Matei
3. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. E. Cliff
4. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. Z. Matei
5. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Z. Matei
6. Shark: SQL and Rich ANalytics at Scale. X. Reynold
7. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. X. Reynold
8. Spark SQL: Relational Data Processing in Spark. A. Michael
9. MLlib: Machine Learning in Apache Spark. M. Xiangrui
10. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark A. Michael
11. Making Big Data Processing Simple with Spark

    https://www.youtube.com/watch?v=d9D-Z3-44F8

12. Matei on The future of Big Data

https://www.youtube.com/watch?v=oSj2vYw5RLs

13. Matei on Spark

https://www.youtube.com/watch?v=7k4yDKBYOcw

14. Reynold Xin

https://www.youtube.com/watch?v=QGj_mxD7Q2I

15. 连城的采访

连城：大数据场景下的"搔到痒处"和"戳到痛处"

非商业转载请注明作译者、出处，并保留本文的原始链接：http://www.ituring.com.cn/article/179495 连城，Databricks工程师，Apache Spark committer。 《Erlang/OTP并发编程

https://segmentfault.com/a/1190000002591709

Databricks连城谈Spark的现状-InfoQ

连城目前就职于DataBricks，曾工作于网易杭州研究院和百度，也是《Erlang/OTP并发编程实战》及《Erlang并发编程（第一部分）》的译者。近日，InfoQ中文站编辑跟连城进行了邮件沟通，连城在邮件中分享了自己对Spark现状的解读。InfoQ：有专家侧重Storm，您则是侧重Spark，请简单谈谈这两者的区别

https://www.infoq.cn/article/2014/09/spark-status

16. MLlib 孟祥瑞

https://www.youtube.com/watch?v=Q0VXllYilM0

Discussion

1. azure hdinsights vs kusto vs snapse vs data lake
2. snowflake vs. redshift
3.

🌟 spark cmd