SJTU SAIL (Software Architecture and Infrastructure Lab)

SHANGHAI JIAO TONG UNIVERSITY

Resilient Distributed Datasets

A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Hongyi Long 2020/10/09

Introduction

History of Spark APIs

Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

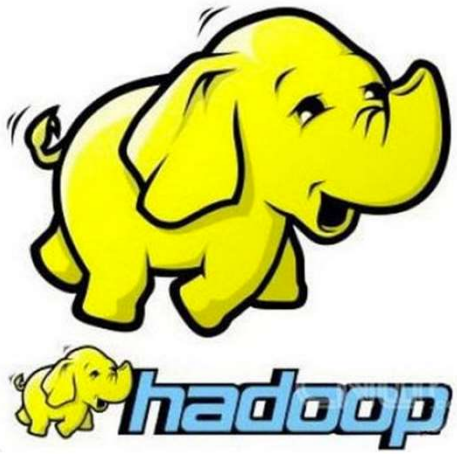
Internally rows, externally JVM objects

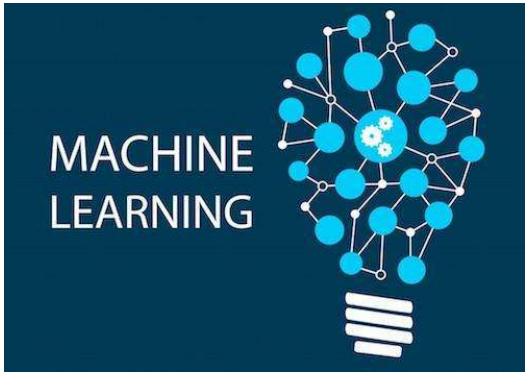
Almost the "Best of both worlds": **type safe + fast**

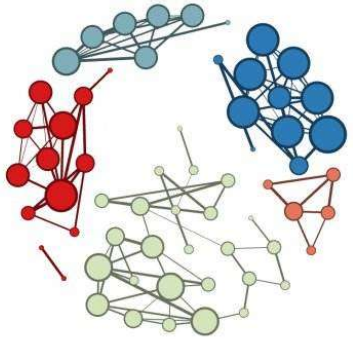
But slower than DF
Not as good for interactive analysis, especially Python

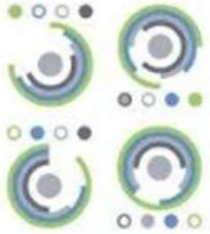
databricks

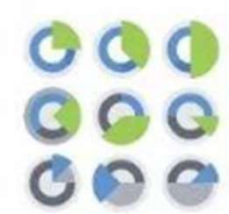
Inspiration





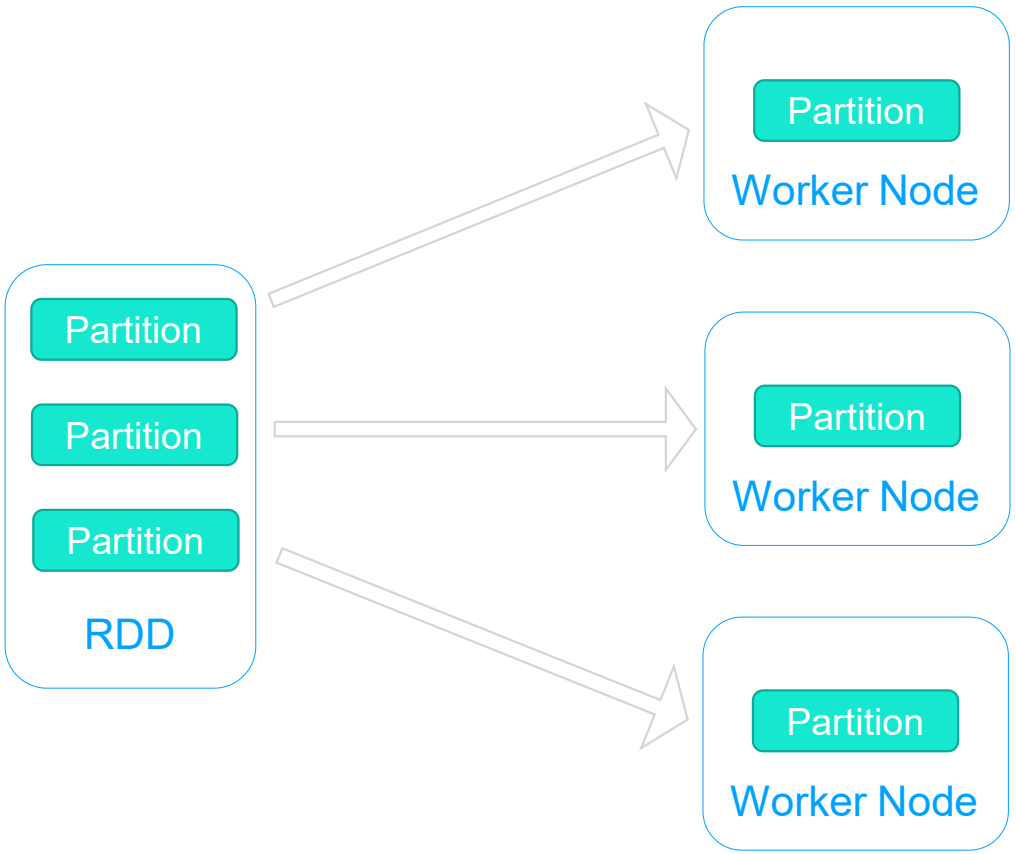






RDD

Abstraction



Transformations and Actions

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.

Example Applications

Example: Console Log Mining

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

// 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()
```

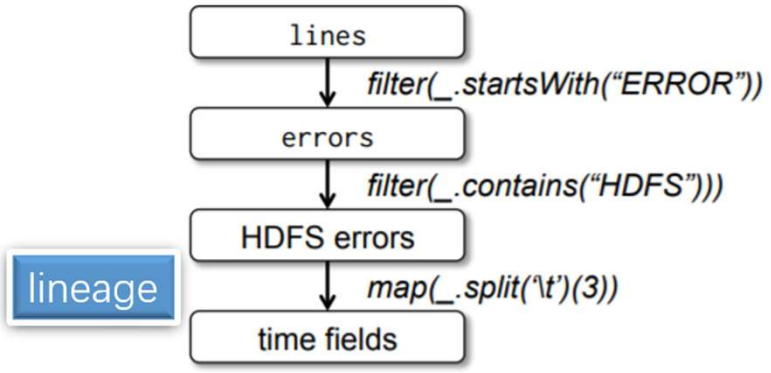


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Example: Logistic Regression

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

20x Speed up

Hadoop 0.20.2 stable release

Example: Page Rank

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
               .mapValues(sum => a/N + (1-a)*sum)
}
```

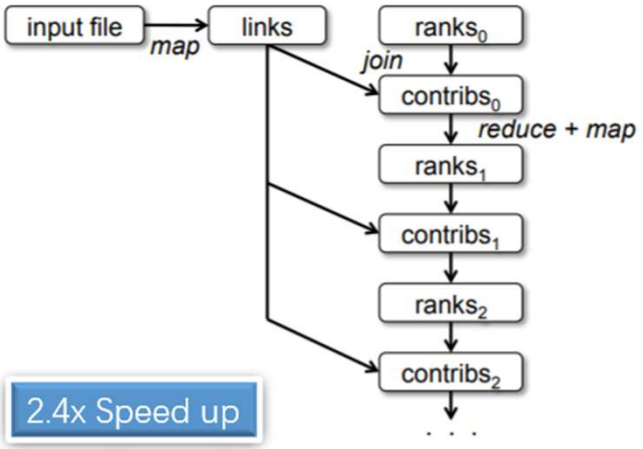


Figure 3: Lineage graph for datasets in PageRank.

Optimization

7.4x Speed up

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

Dependency

Narrow / Wide Dependency

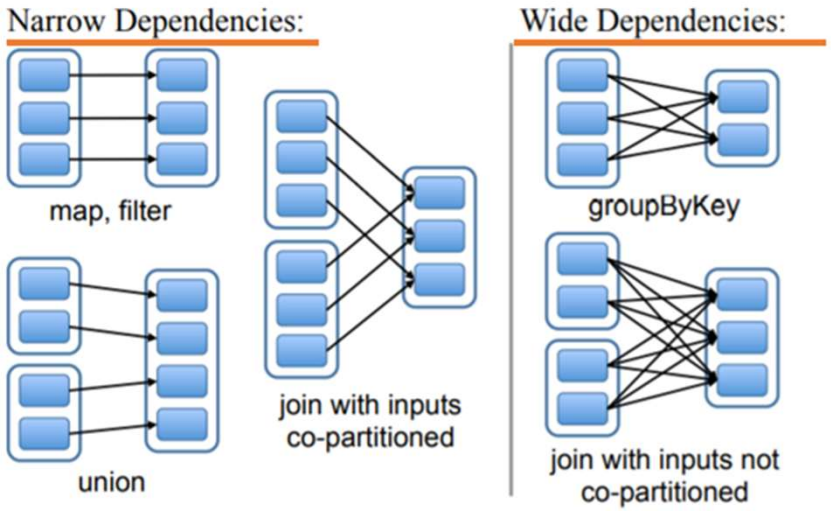


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

No Scheduler Fault-Tolerance

Implementation

Job Scheduling

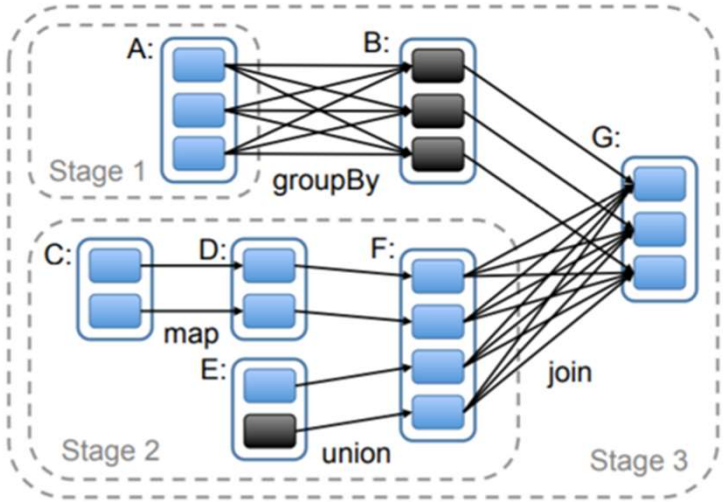


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

Interpreter Integration

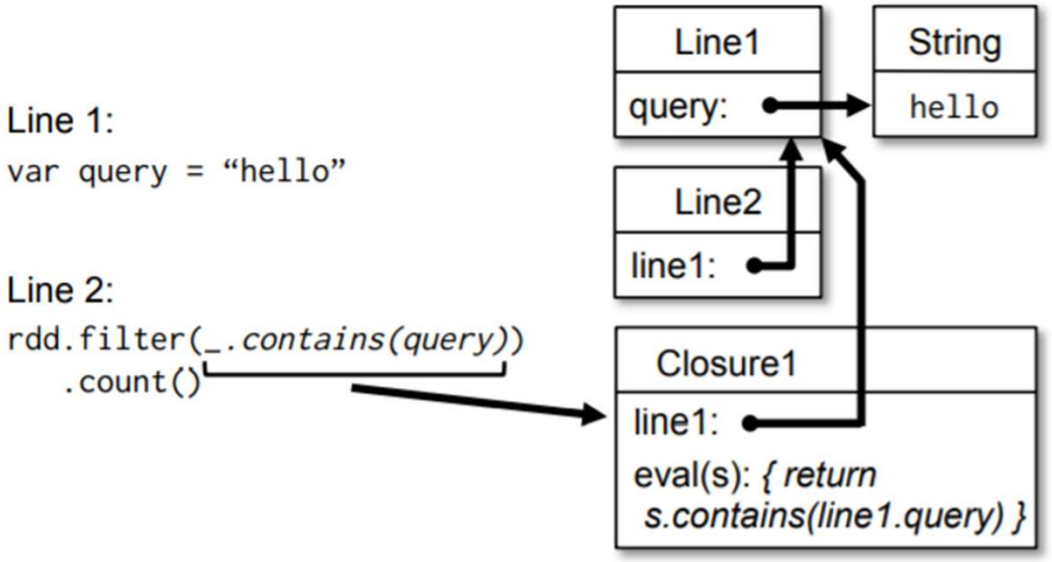


Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

Memory Management & Checkpointing

- **In-memory Deserialized Java objects**
- **In-memory Serialized data**
- **On-disk storage**
    - **LRU**
    - **Priority**
- **Checkpointing for lineage**

Evaluation

Logistic Regression & Page Rank

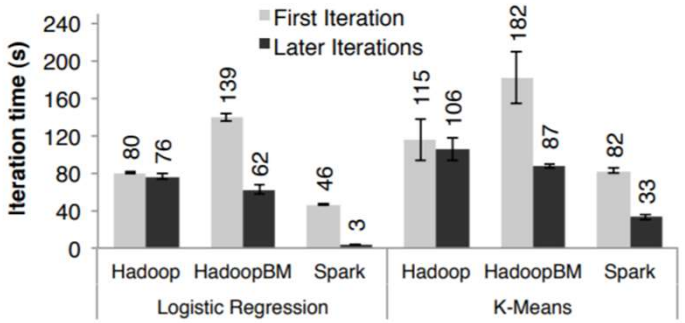


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

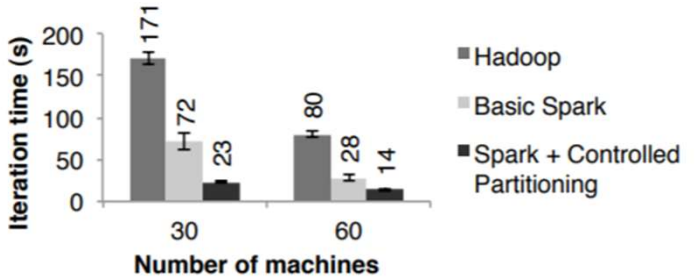
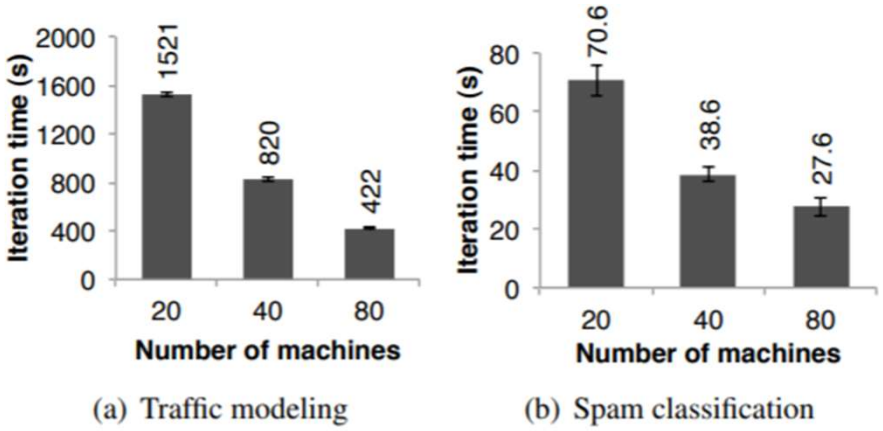


Figure 10: Performance of PageRank on Hadoop and Spark.

- Hadoop software stack overhead
- HDFS Overhead while serving data
- Deserialization cost: convert binary records to usable in-memory Java objects

- In-memory
- No serialization (Raw Java Object)

Interpreter Integration



(a) Traffic modeling (b) Spam classification

Figure 13: Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

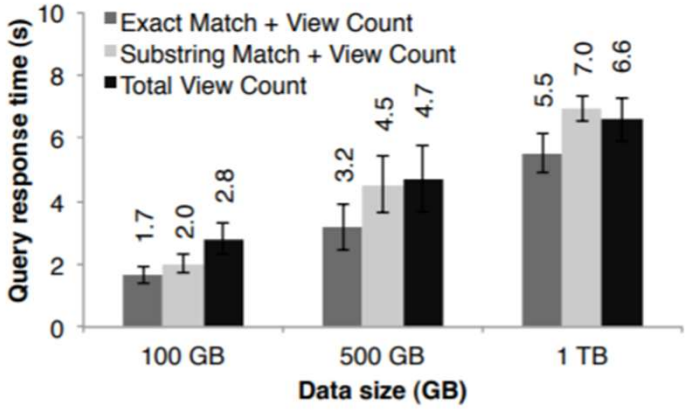


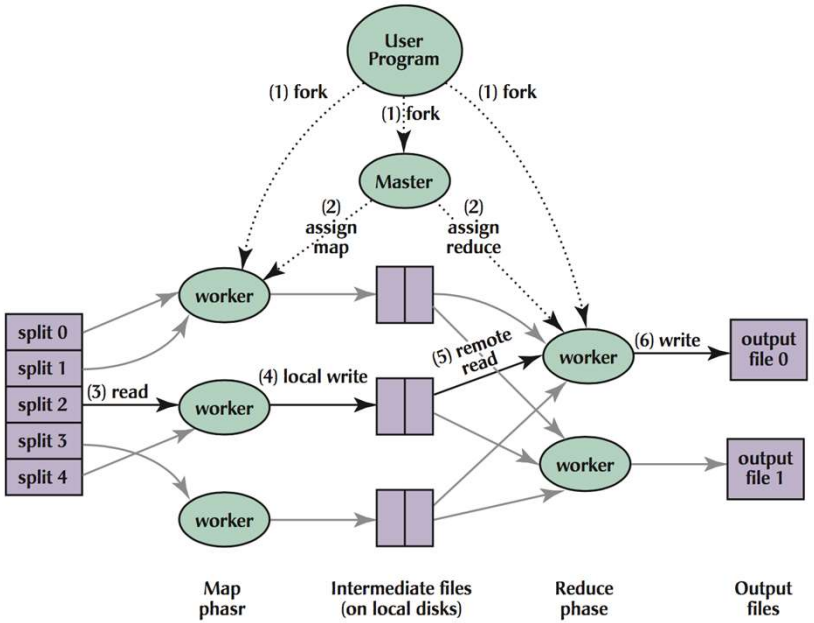
Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

Thank You

Q & A

Appendix

MapReduce Shuffle



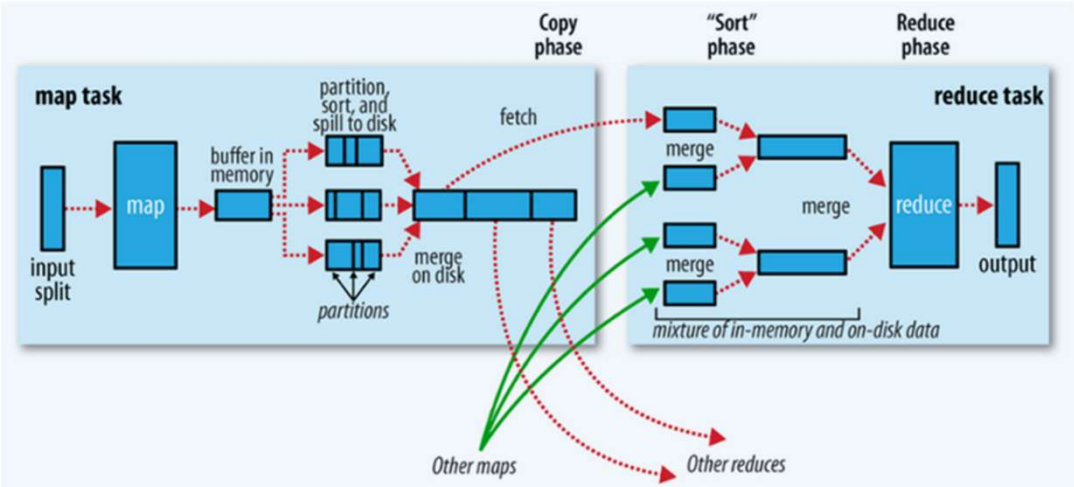


Figure 6-4 Shuffle and sort in MapReduce