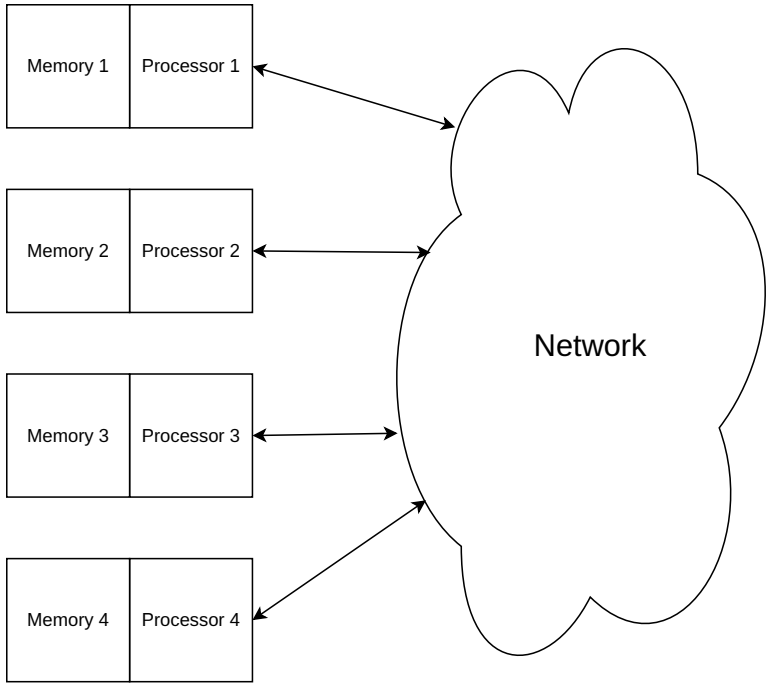
Collective Computation

Bulk Synchronous Parallel (BSP)

> Due to the rapidly decreasing cost of processing, memory, and communication, it has appeared inevitable for at least two decades that parallel machines will eventually displace sequential ones in computationally intensive domains.

Leslie G. Valiant. 1990. A bridging model for parallel computation.



Multiple processors with local memory.

Any processor can communicate with any other processor through the network.

Minimize communication!

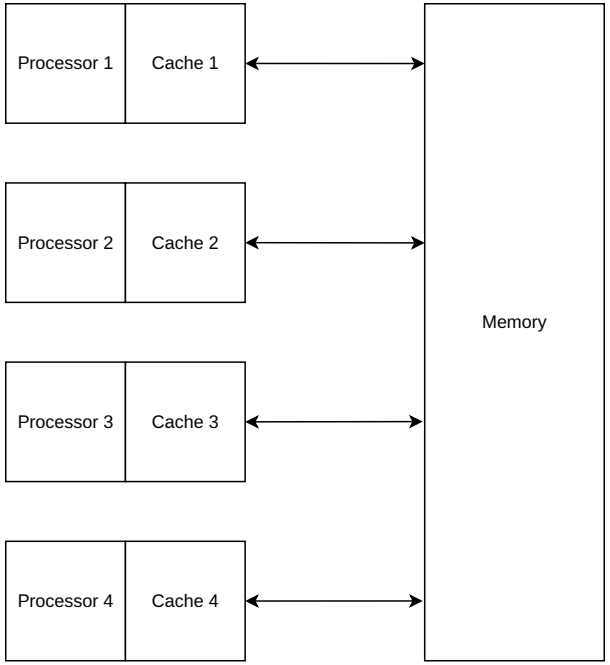
Not Distributed Shared Memory

Parallel Random Access Machine (PRAM)

Fine-grained writes to shared state

Does not scale:

- cache coherence makes it slow
- false sharing makes it slow
- fault tolerance makes it slow



Steven Fortune and James Wyllie. 1978. Parallelism in random access machines.

Distributed Tasks

> A Dryad application developer can specify an arbitrary directed acyclic graph to describe the application's communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared-memory FIFOs) between the computation vertices.

Michael Isard, et al. 2007. Dryad: distributed data-parallel programs from sequential building blocks.

Vertices are connected by channels, producing and consuming streams.

Matei Zaharia, et al. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing.

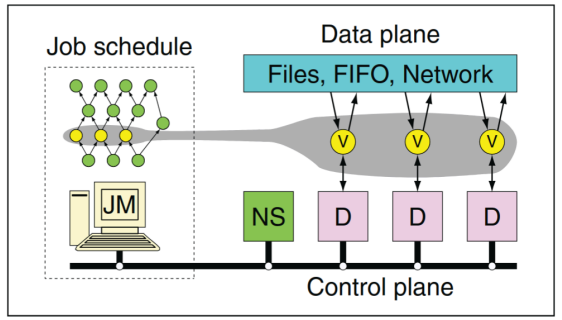


Figure 1: The Dryad system organization. The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

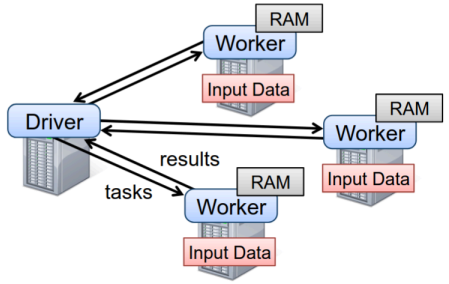


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Spark

https://spark.apache.org/

```
>>> textFile.filter(textFile.value.contains("Spark")).count()
15
```

> We show that Spark is up to 20× faster than Hadoop for iterative applications, speeds up a real-world data analytics report by 40×, and can be used interactively to scan a 1 TB dataset with 5–7s latency.

Matei Zaharia, et al. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing.

Scala

Compiled, statically-typed programming language for the JVM.

Functional, imperative, and object-oriented.

```
val fruits =
  List("apple", "banana", "avocado", "papaya")

val countsToFruits =
  fruits.groupBy(fruit => fruit.count(_ == 'a'))

for (count, fruits) <- countsToFruits do
  println(s"with 'a' × $count = $fruits")
```

Created by Martin Odersky in 2004.

demo/Basics.scala

Resilient Distributed Dataset (RDD)

> 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.

Matei Zaharia, et al. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing.

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]

union(): (RDD[T], RDD[T]) => RDD[T]
```

Actions:

```
count(): RDD[T] => Long

collect(): RDD[T] => Seq[T]

reduce(f: (T, T) => T): RDD[T] => T

save(path: String) // Outputs to a storage system
```

demo/Laziness.scala

Partitioning

Distribution of data determines distribution of computation.

Hash partitioning: assign data by a hash modulo the number of processors.

Example with four processors:

```
("apple", 1)  →  hash("apple") % 4 = 2  →  Processor 2
("banana", 2) →  hash("banana") % 4 = 1 →  Processor 1
("apple", 3)  →  hash("apple") % 4 = 2  →  Processor 2
("cherry", 4) →  hash("cherry") % 4 = 0 →  Processor 0
```

Related to horizontal sharding in databases.

Partitioning in Spark

Different partitioning strategies:

```
sc.parallelize[T](seq: Seq[T]): RDD[T] // split into ranges

sc.textFile(path: String): RDD[String] // split into blocks

repartition(numPartitions: Int): RDD[V] => RDD[V] // hash of value
```

For key value pairs partitioning is based on the key:

```
mapValues(f: V => W): RDD[(K, V)] => RDD[(K, W)] // preserves partitioning

groupByKey(): RDD[(K, V)] => RDD[(K, Seq[V])] // hash of key

reduceByKey(f: (V, V) => V): RDD[(K, V)] => RDD[(K, V)] // hash of key
```

demo/Grouping.scala

Map Reduce

> Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the reduce function

> The reduce function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges these values together to form a possibly smaller set of values.

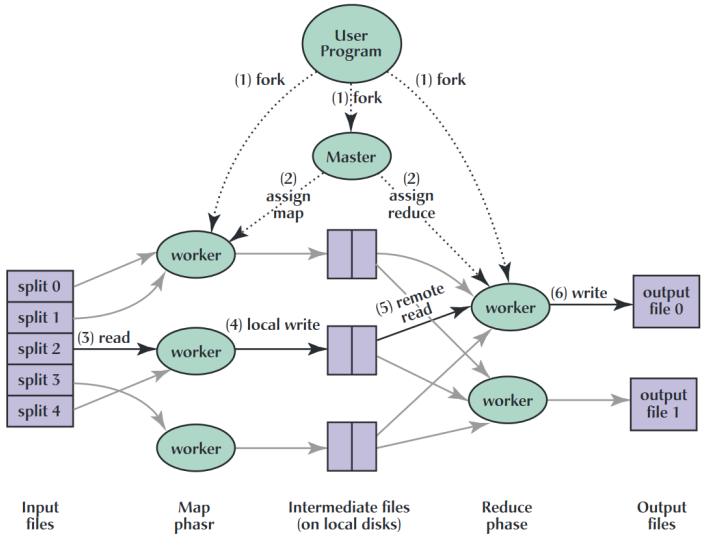


Fig. 1. Execution overview.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters.

Shuffle

Implementation of `groupByKey` and `reduceByKey`:

- Map workers write intermediate data partitioned by hash to local disk
- Map workers notify master of output location
- Master keeps track of these locations
- Master tells reduce workers where to find their partitions
- Reduce workers pull their data over the network from those locations

Potentially M mappers times R reducers network messages.

demo/MapReduce.scala

Stream Fusion

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness

Avoid materializing intermediate data structures in streaming pipelines.

```
trait Stream[A] { def next(): Option[A] }
```

Example pipeline:

```
xs.map(f).filter(p).map(g)
```

Fuse into a single object:

```
new Stream[C] {
  def next() = xs.next() match {
    case Some(x) =>
      val y = f(x)
      if (p(y)) Some(g(y))
      else next()
    case None => None
  }
}
```

Fusion in Spark

Narrow dependencies: Each partition of the parent RDD is used by at most one partition of the child RDD.

- Examples: `map`, `filter`, `flatMap`, `mapPartitions`
- No shuffle needed: each task processes one partition independently
- Can fuse operations together in a single stage, cheap!

Wide dependencies: Each partition of the parent RDD may be used by multiple partitions of the child RDD.

- Examples: `groupByKey`, `reduceByKey`, `join`, `repartition`
- Requires shuffle: data must be redistributed across the network
- Forces a stage boundary, expensive!

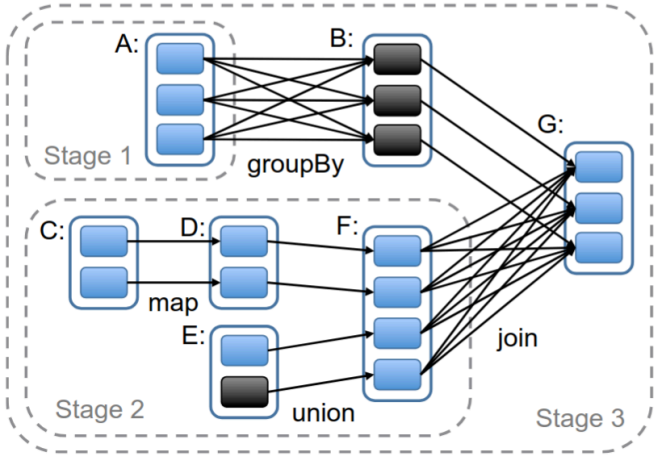


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

demo/Fusion.scala

Lineage

> If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters.

> [...] an RDD has enough information about how it was derived from other datasets (its *lineage*) to *compute* its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.

Matei Zaharia, et al. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing.

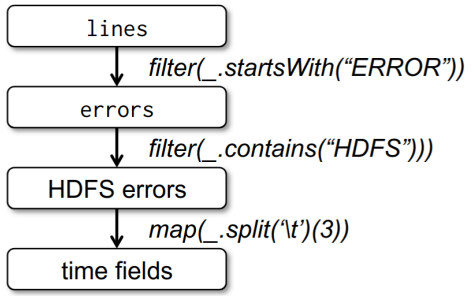


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Page Rank

Google's original algorithm for ranking search results.

Iterative algorithm simulating a random surfer.

When on a page, follows a random link, or jumps to a completely different page.

Calculates for each page the probably that the random surfer ends up on it.

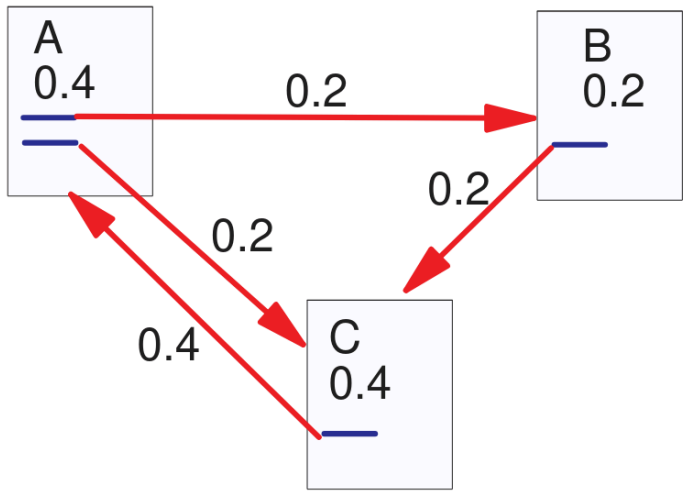


Figure 3: Simplified PageRank Calculation

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web.

demo/PageRank.scala

Summary and Outlook

Map Shuffle Reduce just works for many tasks.

Next week: Orchestration.