Data Management in the Cloud

APACHE SPARK

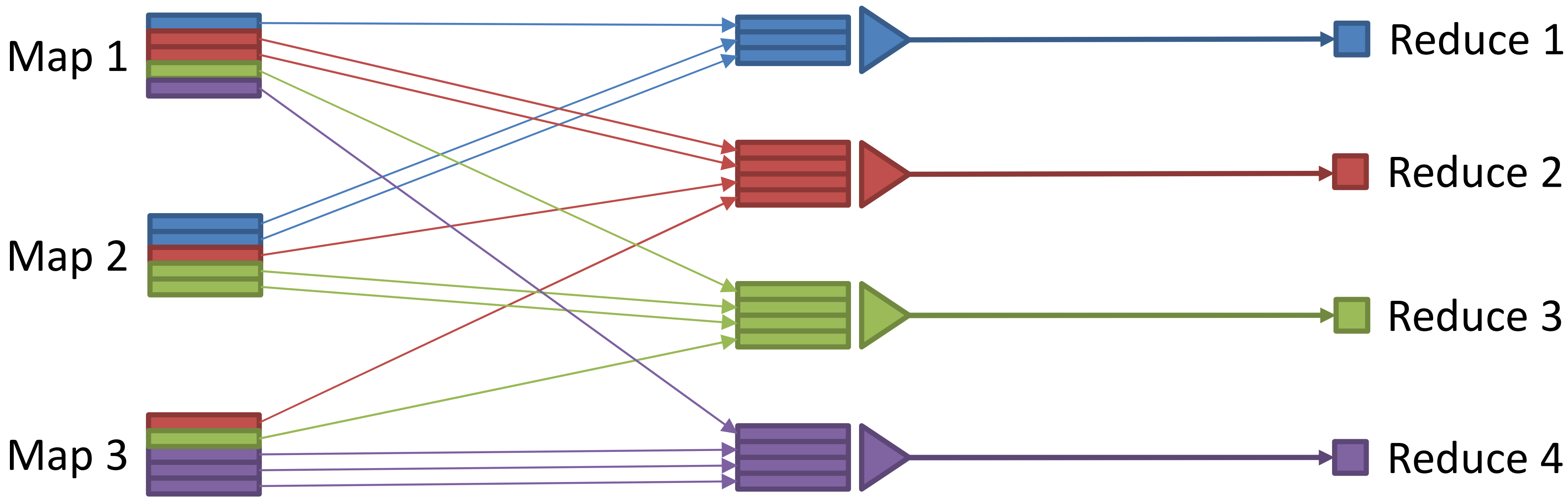
THANKS TO M. ZAHARIA

Map Reduce Overview

MapReduce is a programming model designed to process large-scale datasets in parallel over distributed clusters.

- Key Components:
  - Map: Processes input data to produce intermediate key-value pairs.
  - Reduce: Aggregates and processes these pairs to produce final results.
- Purpose:
  - Simplifies distributed data processing.
  - Handles data splitting, task scheduling, and fault tolerance automatically.

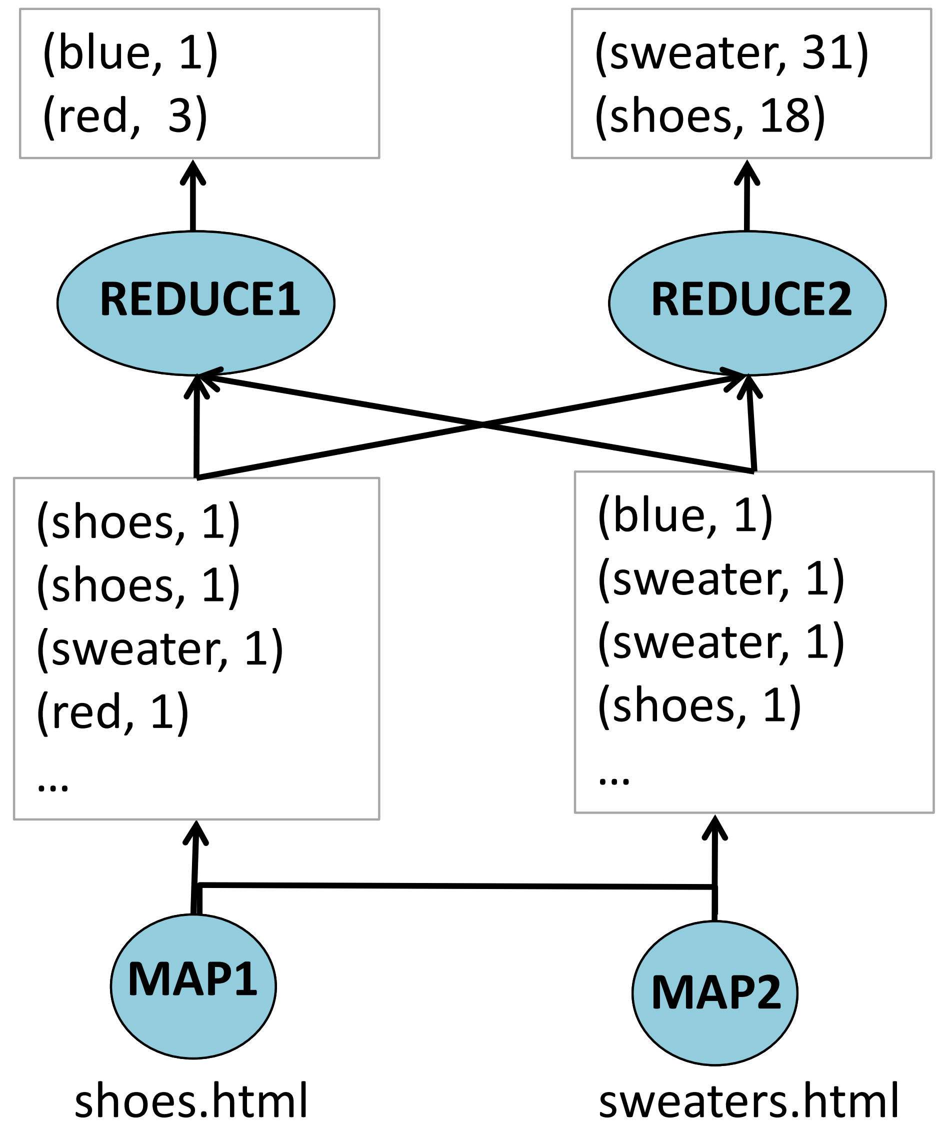
Map-to-Reduce Interaction



- Map functions create a user-defined “index” from source data
- Reduce functions compute grouped aggregates based on index
- Flexible framework
  - users can cast raw original data—in any model—that they need
  - wide range of tasks can be expressed in this simple framework

Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, “1”) for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: (word, 4) (sum of counts)



Map Reduce Pros and Cons

- Advantages:
  - Scalability: Easily scales with the size of data and number of machines.
  - Fault Tolerance: Automatically recovers from node failures by reprocessing lost tasks.
  - Simplicity: Abstracts the complexities of parallel and distributed programming.
- Limitations:
  - Inefficiency with Iterative Tasks: Not well-suited for iterative algorithms (e.g., many machine learning algorithms).
  - Disk I/O Overhead: Writes intermediate results to disk, which can slow down processing.
- Transition to Spark:

  Spark improves on MapReduce by keeping data in-memory for faster iterative processing and supporting more complex workflows.

Participation Activity

In this activity, your team will analyze temperature data using the MapReduce framework.

Steps:

- Setup (30 sec): Assign roles - Mapper(s), Shuffler, Reducer
- Map Phase (1.5 min): Categorize temperatures as Cold (<50°F), Mild (50-70°F), or Hot (>70°F)
- Shuffle Phase (1 min): Group locations by temperature category
- Reduce Phase (1.5 min):
    - Count how many locations fall into each category (Cold, Mild, Hot)
    - Calculate what percentage each category represents of the total number of locations
    - Record both the count and percentage for each category

https://docs.google.com/presentation/d/1pH-etePPV1LJI5zQSWbR20otW0meoM4BJa0Kaqb1Ep0/edit?usp=sharing

Spark – Motivation

- MapReduce simplified big data analysis: could program with *data flows*
- But some uses stretched its capabilities
  - Multi-stage applications: Several map-reduce jobs in a row (such as iterative algorithms and machine learning)
  - Ad-hoc queries: Try different things on the same dataset
- Issues:
  - MR stages communicate via GFS or HDFS
  - Two queries on the same data set read it twice
- Need lighter-weight data sharing

Key Idea: Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs)

- Immutable collections, usually partitioned
- Defined via transformations (map, filter, join, …)
- Fault tolerant – can be reconstructed on failure

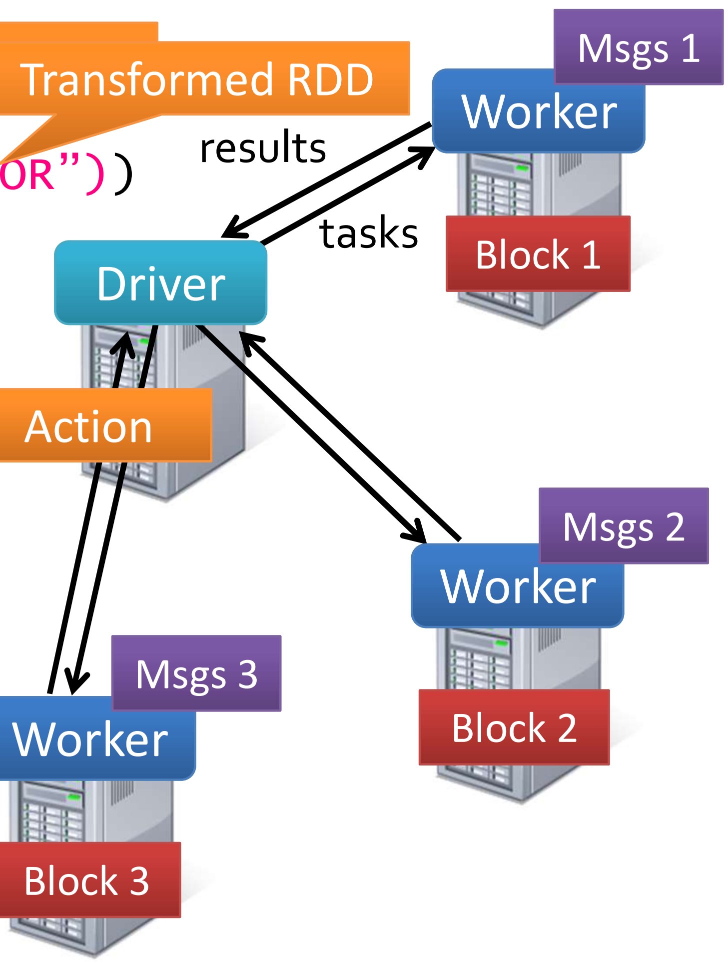
  Or checkpointed
- Can persist, in memory or on disk
  - Connect sequences of transformations
  - Use same dataset for multiple tasks

Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(x => x.startsWith("ERROR"))
messages = errors.map(y => y.split('\t')(2))
messages.persist()

messages.filter(_.contains("PHP")).count
messages.filter(_.contains("SQL")).count
. . .
```



Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Transformations vs. Actions

- Transformations are used to construct one RDD from other RDDs
  - Examples: `map, filter, join, union, groupBy, reduce, sample`
  - RDDs evaluated lazily

Why does Spark evaluate RDDs lazily?

- Actions cause RDDs to be evaluated, results returned or stored.
  - Examples: `count, collect, save`

List of Transformations and Actions

Transformations	$map(f: \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f: \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f: \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction: \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f: (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$\mathrm{(RDD[T], RDD[T])} \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$\mathrm{(RDD[(K, V)], RDD[(K, W)])} \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$\mathrm{(RDD[(K, V)], RDD[(K, W)])} \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$\mathrm{(RDD[T], RDD[U])} \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f: \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c: \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p: \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f: (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k: \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path: \mathrm{String})$	:	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.

From M. Zaharia et al., ***Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing***, NSDI 2012