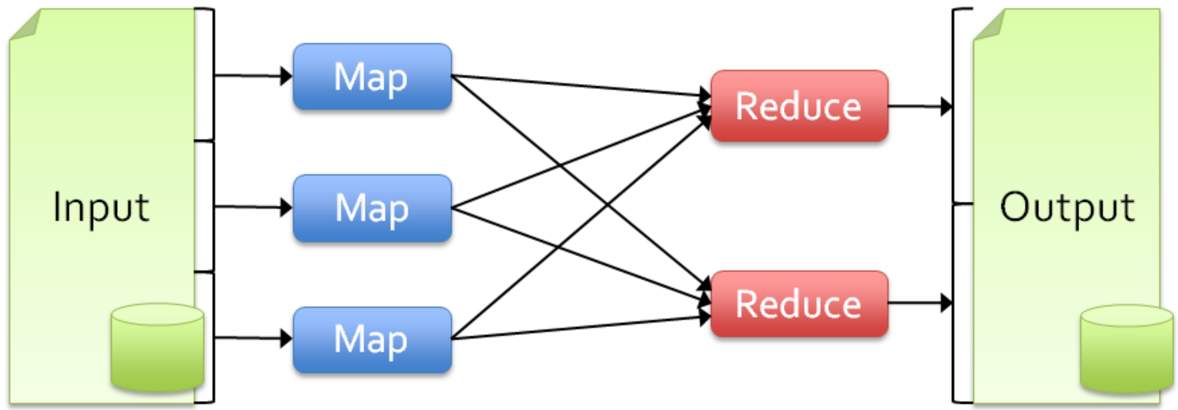
Apache Spark

COMP9313: Big Data Management

Motivation of Spark

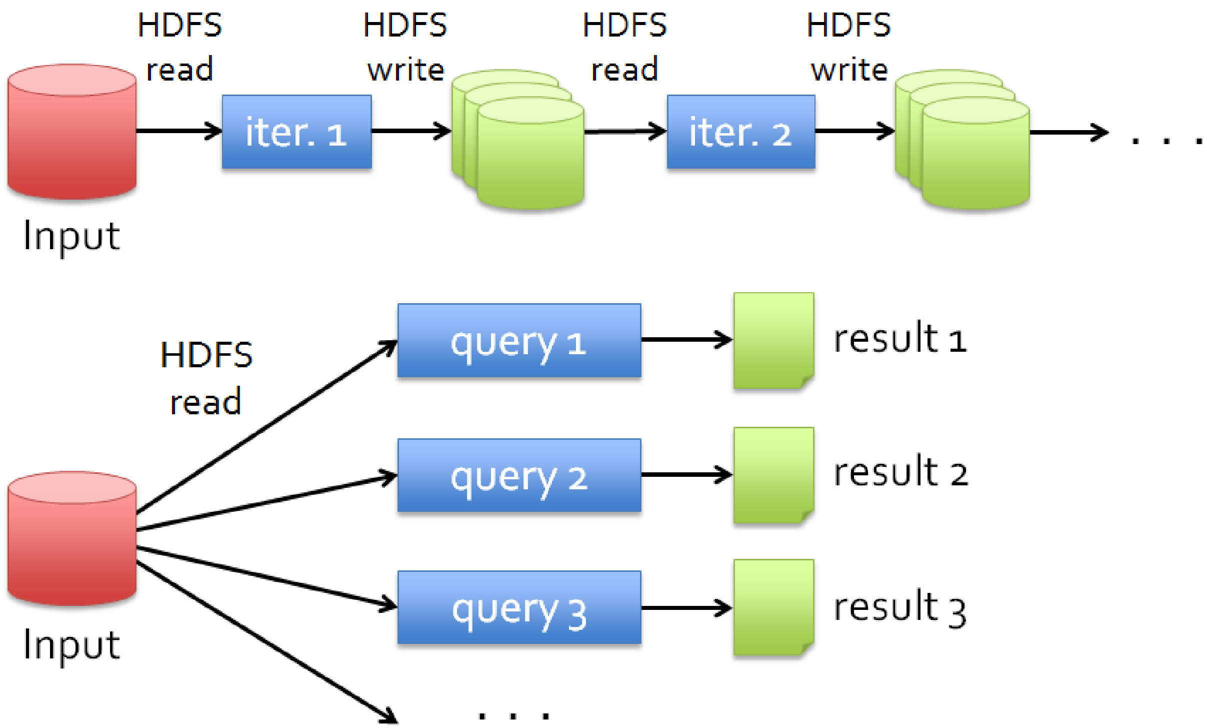
- MapReduce greatly simplified big data analysis on large, unreliable clusters. It is great at one-pass computation
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graphs)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All 3 need faster **data sharing** across parallel jobs
  - One reaction: specialized models for some of these apps, e.g.,
    - Pregel (graph processing)
    - Storm (stream processing)

Limitations of MapReduce



- As a general programming model:
  - It is more suitable for one-pass computation on a large dataset
  - Hard to compose and nest multiple operations
  - No means of expressing iterative operations
- As implemented in Hadoop
  - All datasets are read from disk, then stored back on to disk
  - All data is (usually) triple-replicated for reliability

Data Sharing in MapReduce



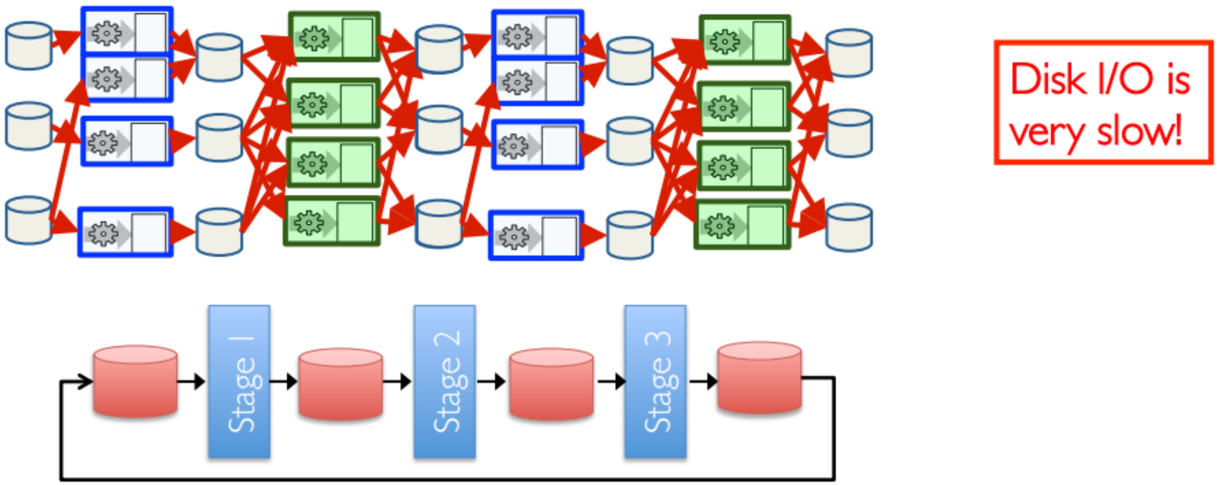
Slow due to replication, serialization, and disk IO

- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

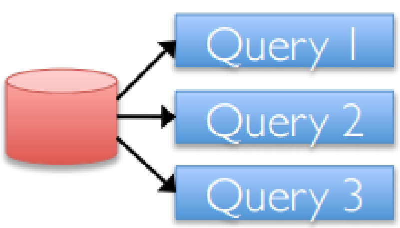
Efficient primitives for **data sharing**

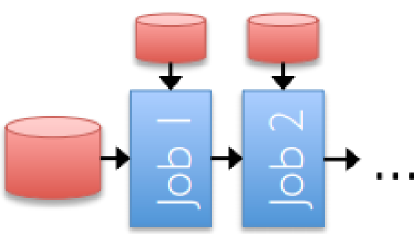
Data Sharing in MapReduce

- Iterative jobs involve a lot of disk I/O for each repetition



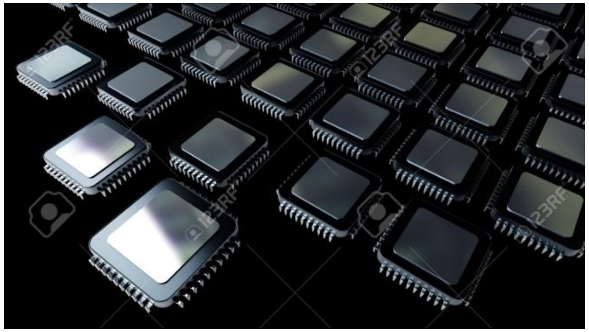
- Interactive queries and online processing involves lots of disk I/O





Hardware for Big Data

Lots of hard drives



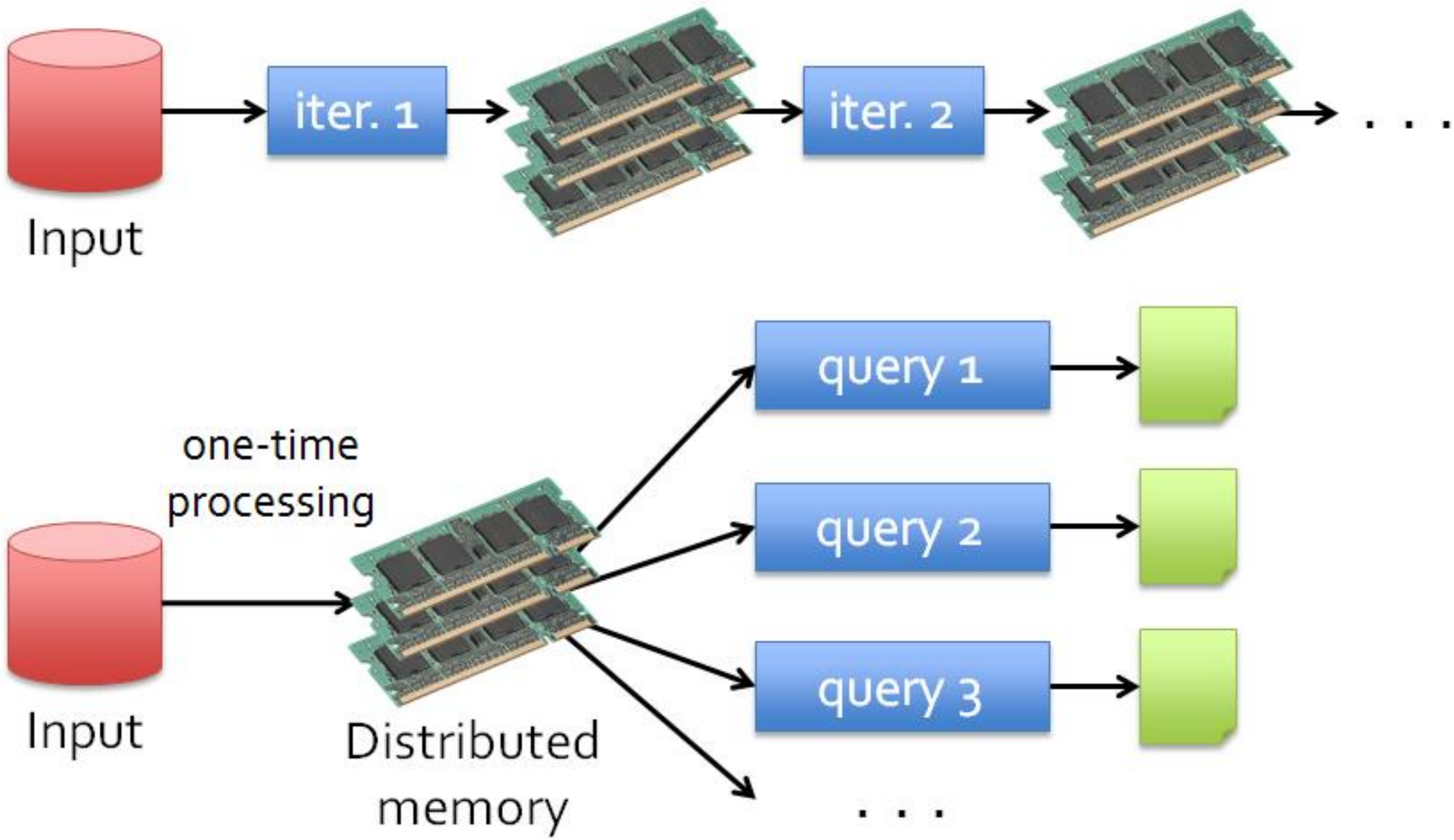
Lots of CPUs

And lots of memory!

Goals of Spark

- Keep more data in-memory to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability:
  - Introduce rich API libraries
  - More to be done with less Lines of Code

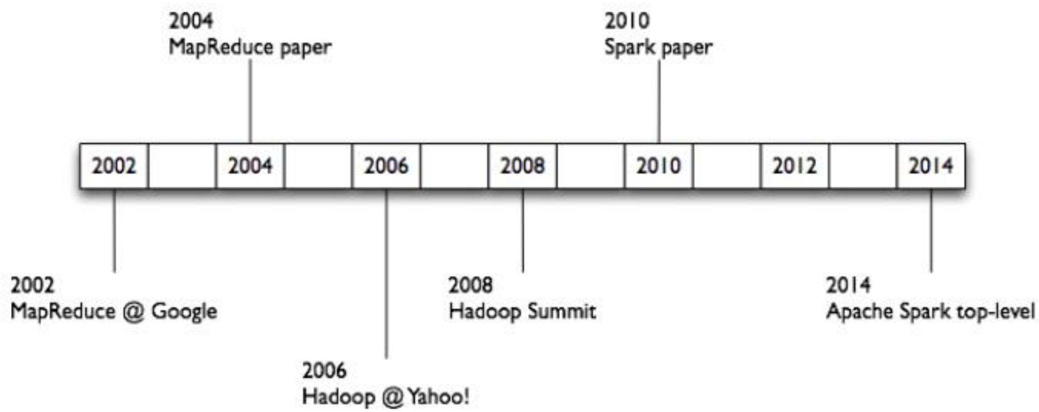
Data Sharing in Spark with RDDs (Resilient Distributed Dataset)



10-100× faster than network and disk

What is Spark?

- One popular answer to "What's beyond MapReduce?"
- Open-source engine for large-scale data processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java, Python and R
- Brief history:
  - Developed at UC Berkeley AMPLab in 2009
  - Open-sourced in 2010
  - Became top-level Apache project in February 2014
  - Commercial support provided by DataBricks



What is Spark?

- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - **In-memory** computing primitives
  - General computation graphs

→ Up to 100 × faster (10 × on disk)

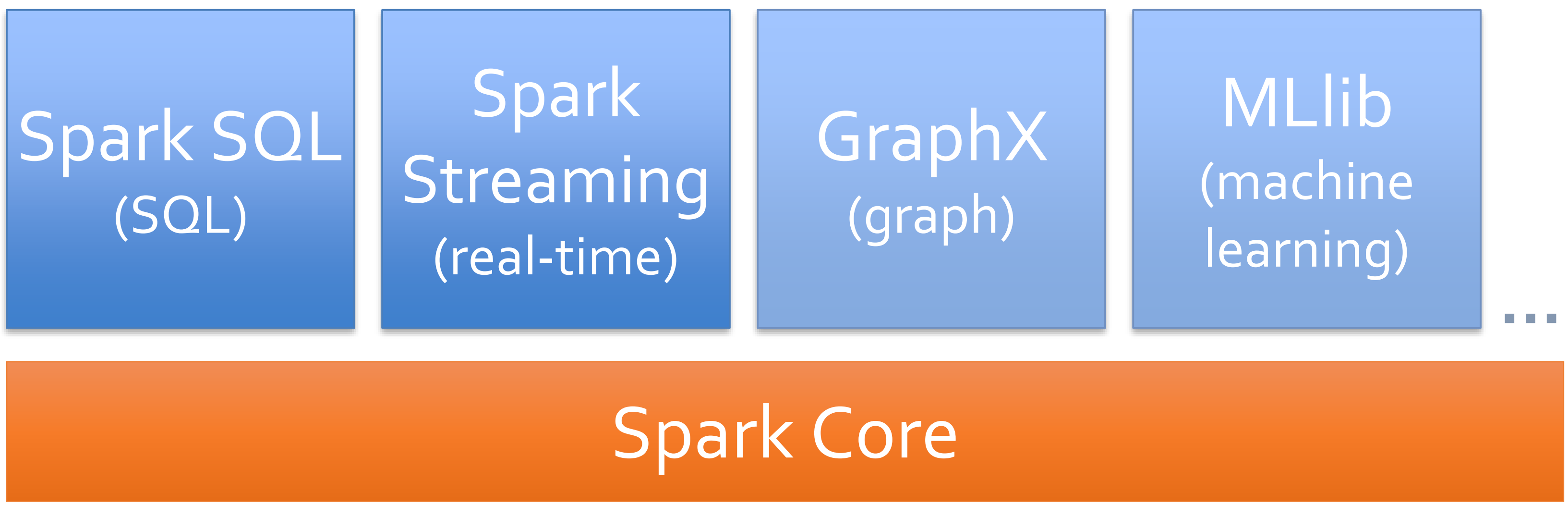
- Improves usability through:
  - Rich APIs in Scala, Java, Python, R
  - Interactive shell

→ Often 5 × less code

- **Spark is not**
  - a modified version of Hadoop
  - dependent on Hadoop because it has its own cluster management (Spark can use Hadoop YARN and HDFS)

What is Spark?

- Spark is the basis of a wide set of projects in the Berkeley Data Analytics Stack (BDAS)



- ➢ Spark SQL (SQL on Spark)
- ➢ Spark Streaming (stream processing)
- ➢ GraphX (graph processing)
- ➢ MLlib (machine learning library)

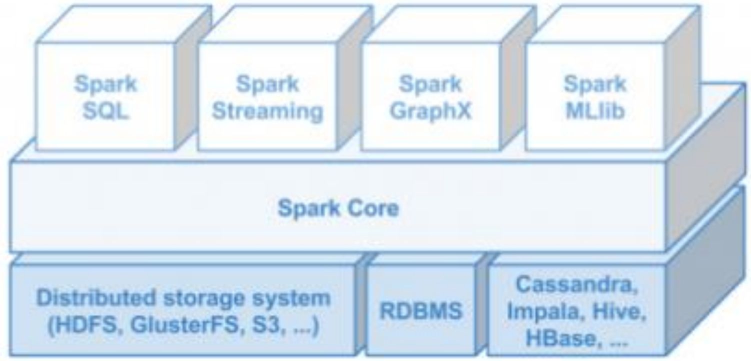
Spark Running Modes

You can run Spark using:

- Its standalone cluster mode
- On Hadoop YARN
- On Apache Mesos
- On Kubernetes
- Or on the Cloud (e.g., DataBricks).

Data Sources

- Local Files
  - file:///opt/httpd/logs/access_log
- Amazon S3
- Hadoop Distributed Filesystem
  - Regular files, sequence files, any other Hadoop InputFormat
- HBase, Cassandra, etc.

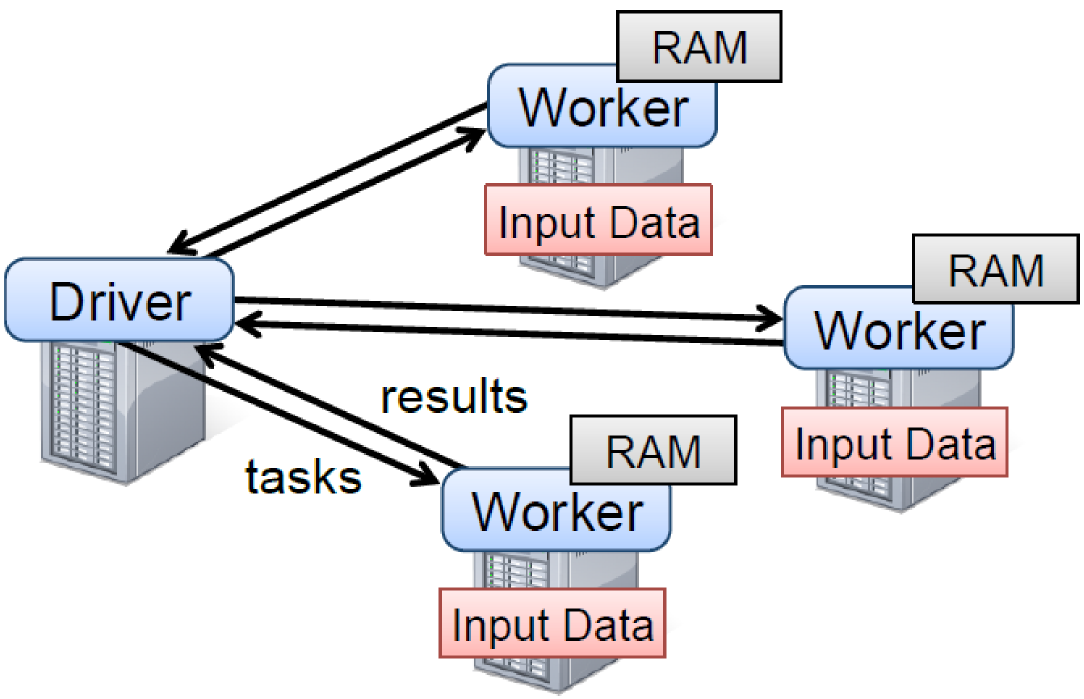


Spark Ideas

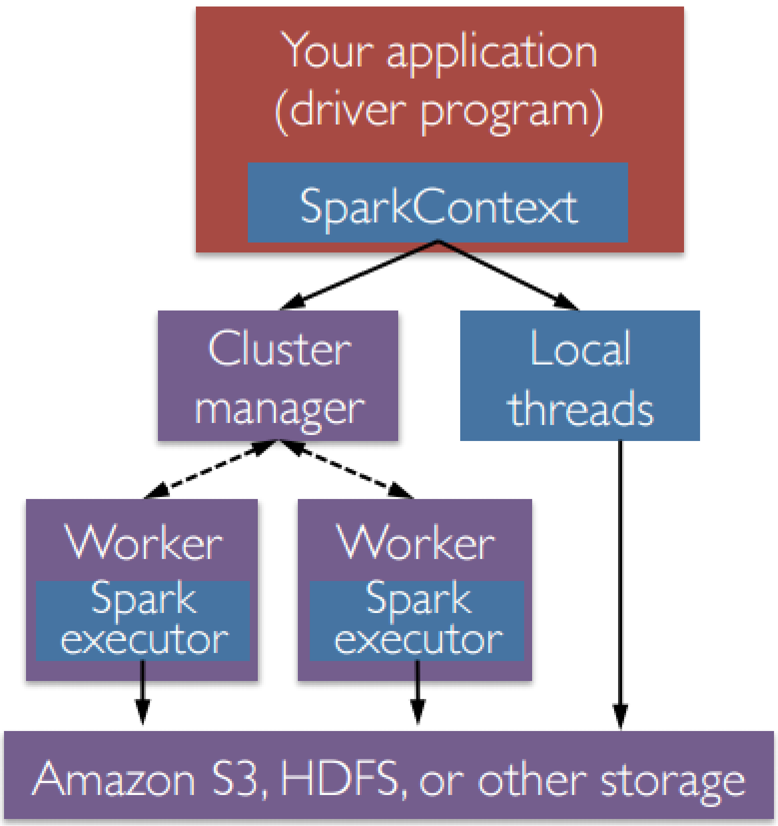
- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an in-memory system on top of Hadoop
- Achieve fault-tolerance by re-execution instead of replication

Spark Cluster

- To use Spark, developers write a driver program that connects to a cluster of worker



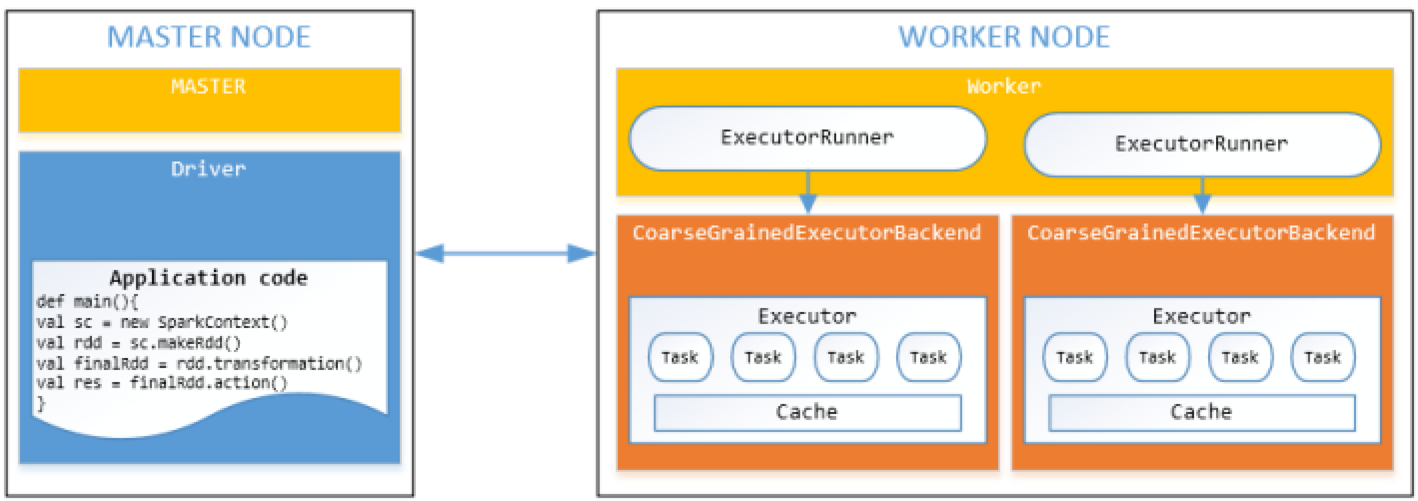
Spark Workflow



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Connects to several types of cluster managers (e.g. YARN, Mesos, or its own manager)
- Cluster manager:
  - Allocates resources across applications
- Spark executor:
  - Runs computations
  - Accesses data storage

Workers Nodes and Executors

- Worker nodes are machines that run executors
  - Host one or multiple Workers
  - One JVM (1 process) per Worker
  - Each Worker can spawn one or more Executors
- Executors run tasks
  - Run in child JVM (1 process)
  - Execute one or more task using threads in a ThreadPool



Introduction to RDDs

Challenge

- Existing Systems
  - Existing in-memory storage systems have interfaces based on fine-grained updates
    - Reads and writes to cells in a table
    - E.g., databases, key-value stores, distributed memory
  - Requires replicating data or logs across nodes for fault tolerance

    -> expensive!
    - 10-100x slower than memory write

- How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

Solution: Resilient Distributed Datasets

- *Resilient Distributed Datasets (RDDs)*
  - Distributed collections of objects that can be cached in memory across cluster
  - Manipulated through parallel operators
  - Automatically recomputed on failure based on lineage
- RDDs can express many parallel algorithms, and capture many current programming models
  - Data flow models: MapReduce, SQL, …
  - Specialized models for iterative apps: Pregel, …

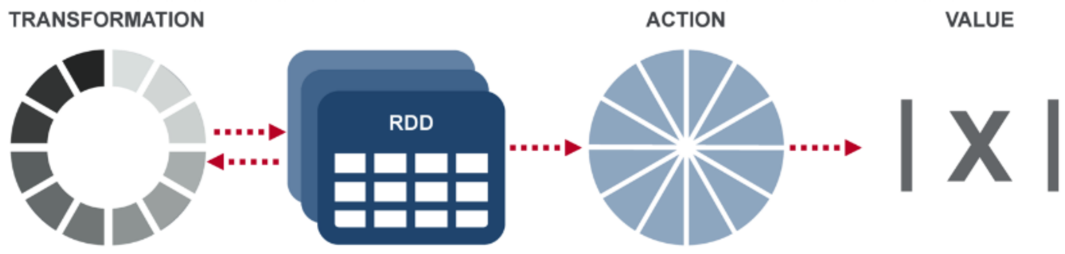
What is RDD?

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12 (paper)
  - RDD is a **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures
- **Distributed**
  - Data residing on multiple nodes in a cluster
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with)
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel

RDD Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed)
- **Parallel**, i.e. process data in parallel
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)]
- **Partitioned**, i.e. the data inside an RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node)

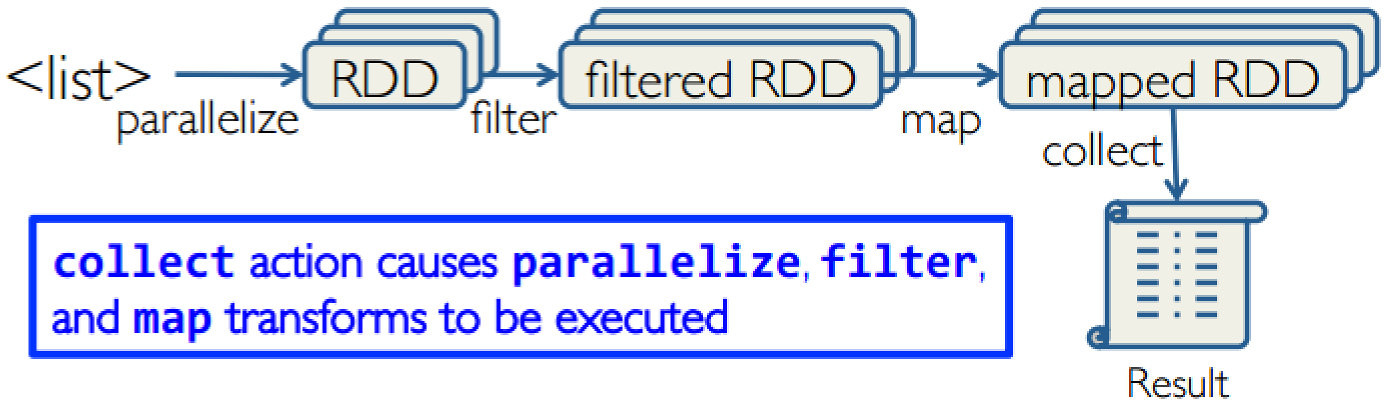
RDD Operations



- **Transformation:** returns a new RDD
  - ➢ Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD
  - ➢ Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, filter, join, etc.*
- **Action:** evaluates and returns a new value
  - ➢ When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned
  - ➢ Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

Working with RDDs

- Create an RDD from a data source
  - ➢ by parallelizing existing collections (lists or arrays)
  - ➢ by transforming an existing RDDs
  - ➢ from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., map, filter
- Apply actions to an RDD: e.g., collect, count



- Users can control two other aspects:
  - ➢ Persistence
  - ➢ Partitioning

Creating RDDs

- From HDFS, text files, Amazon S3, Apache HBase, SequenceFiles, any other Hadoop InputFormat
- Creating an RDD from a File
  - RDD distributed in 4 partitions
  - Elements are lines of input

```
JavaRDD<String> distFile = sc.textFile("data.txt“,4);
```

- Turn a collection into an RDD

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Spark Transformations

- Create new datasets from an existing one
- Use lazy evaluation: Results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - Spark optimizes the required calculations
  - Spark recovers from failures
- Some transformation functions

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
- Some action functions

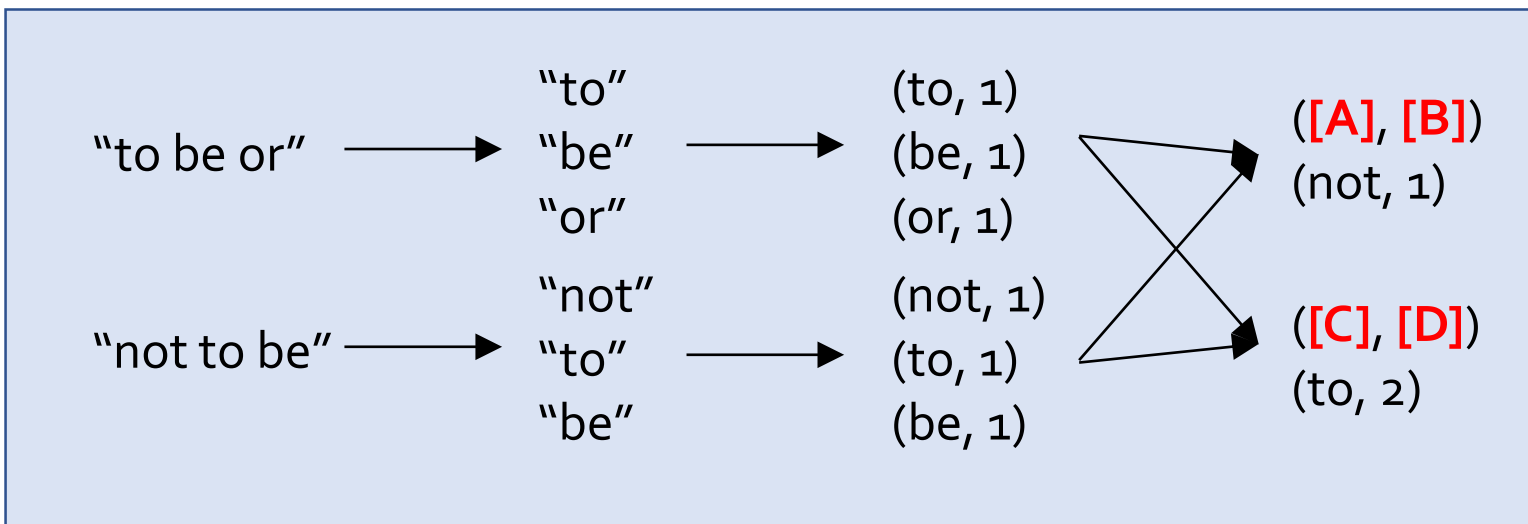
Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.

- Example: counts.saveAsTextFile("hdfs://...");

Word Count in Spark

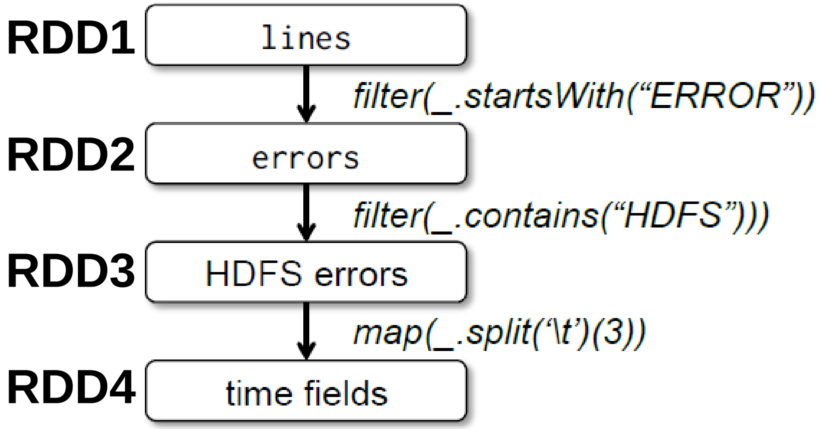
In this example we can use a few transformations to build a dataset of (String, Int) pairs called counts and then save it to a file

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```



Lineage Graph

- RDD is lazy in nature. It means a series of transformations are performed on an RDD, which is not even evaluated immediately
- RDDs keep track of lineage
- RDD has enough information about how it was derived from to compute its partitions from data in stable storage



- Example:
  - If a partition of errors is lost, Spark rebuilds it by applying a filter on only the corresponding partition of lines
  - Partitions can be recomputed in parallel on different nodes, without having to roll back the whole program

SparkContext

- SparkContext is the entry point to Spark for a Spark application
- Once a SparkContext instance is created you can use it to
  - Create RDDs
  - Create accumulators
  - Create broadcast variables
  - Access Spark services and run jobs
- A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application*
- The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster
- In the Spark shell, a special interpreter-aware SparkContext is already created for you

RDD Persistence: Cache/Persist

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations
- When you persist an RDD, each node stores any partitions of it. You can reuse it in other actions on that dataset
- Each persisted RDD can be stored using a different *storage level,* e.g.
  - MEMORY_ONLY:
    - Store RDD as deserialized Java objects in the JVM
    - If the RDD does not fit in memory, some partitions will not be cached and will be recomputed when they're needed
    - This is the default level
  - MEMORY_AND_DISK:
    - If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed
- cache() = persist(StorageLevel.MEMORY_ONLY)

Why Persisting RDD?

- Persist will tell Spark to cache the data in memory, to reduce the data loading cost for further actions on the same data
- errors.persist() will do nothing. It is a lazy operation. But now the RDD says "read this file and then cache the contents". The action will trigger computation and data caching

Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a *Pair RDD* is a pair tuple
- Some Key-Value transformation functions:

groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

Questions?