Introduction to Apache Spark

Pietro Michiardi

Eurecom

Overview

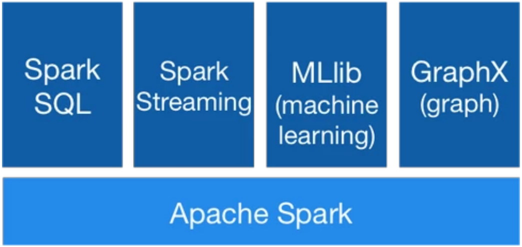
What is Apache Spark

- **Project goals**
  - Generality: diverse workloads, operators, job sizes
  - Low latency: sub-second
  - Fault tolerance: faults are the norm, not the exception
  - Simplicity: often comes from generality

Motivations

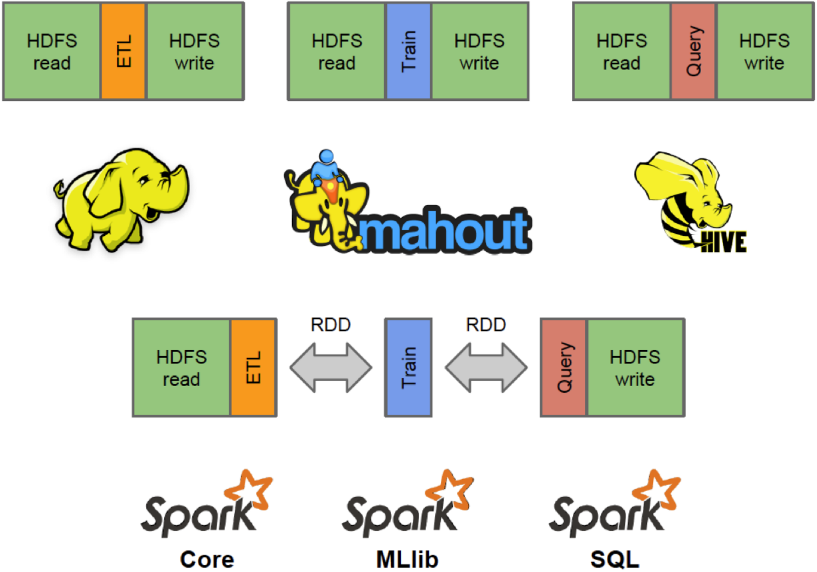
- **System/Framework point of view**
  - Unified pipeline
  - Simplified data flow
  - Faster processing speed
- **Data abstraction point of view**
  - New fundamental abstraction RDD
  - Easy to extend with new operators
  - More descriptive computing model

SPARK: A Unified Pipeline



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

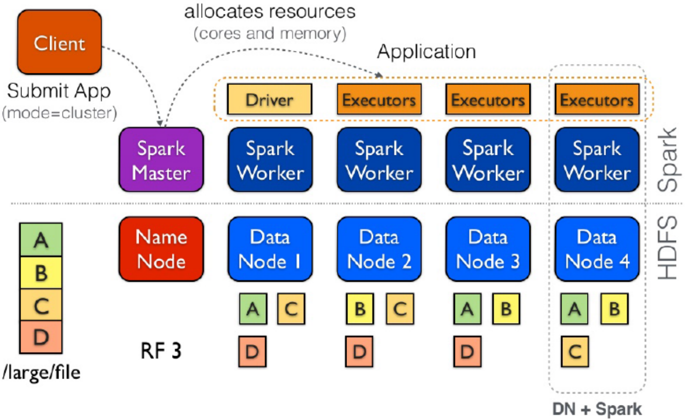
A Simplified Data Flow



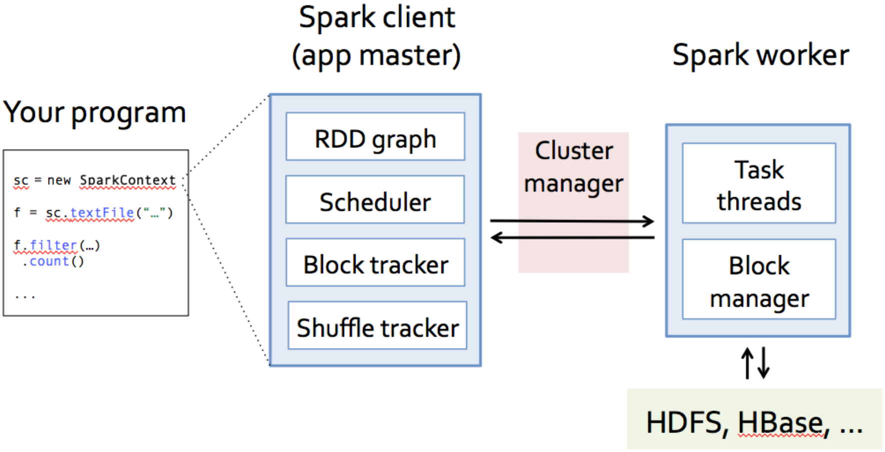
Anatomy of a Spark Application

Spark Applications: The Big Picture

- There are two ways to manipulate data in Spark
  - Use the interactive shell, *i.e.*, the REPL
  - Write standalone applications, *i.e.*, driver programs



Spark Components: details



In Summary

- **Our example Application**: a **jar** file
  - Creates a `SparkContext`, which is the core component of the driver
  - Creates an input RDD, from a file in HDFS
  - Manipulates the input RDD by applying a `filter(f: T => Boolean)` transformation
  - Invokes the action `count()` on the transformed RDD
- **The DAG Scheduler**
  - Gets: RDDs, functions to run on each partition and a listener for results
  - Builds *Stages* of *Tasks* objects (code + preferred location)
  - Submits Tasks to the **Task Scheduler** as ready
  - Resubmits failed *Stages*
- **The Task Scheduler**
  - Launches *Tasks* on executors
  - Relaunches failed *Tasks*
  - Reports to the DAG Scheduler

Resilient Distributed Datasets

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica.
Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,
USENIX Symposium on Networked Systems Design and Implementation, 2012

What is an RDD

- **RDD are partitioned, locality aware, distributed collections**
  - RDD are *immutable*
- **RDD are data structures that:**
  - Either point to a direct data source (e.g. HDFS)
  - Apply some transformations to its parent RDD(s) to generate new data elements
- **Computations on RDDs**
  - Represented by *lazily evaluated* lineage *DAGs* composed by chained RDDs

RDD Abstraction

- **Overall objective**
  - Support a wide array of operators (more than just `Map` and `Reduce`)
  - Allow arbitrary composition of such operators
- **Simplify scheduling**
  - Avoid to modify the scheduler for each operator

→ The question is: *How to capture dependencies in a general way?*

RDD Interfaces

- **Set of partitions ("splits")**
  - Much like in Hadoop MapReduce, each RDD is associated to (input) partitions
- **List of dependencies on parent RDDs**
  - This is completely new w.r.t. Hadoop MapReduce
- **Function to compute a partition given parents**
  - This is actually the "user-defined code" we referred to when discussing about the `Mapper` and `Reducer` classes in Hadoop
- **Optional preferred locations**
  - This is to enforce data locality
- **Optional partitioning info (Partitioner)**
  - This really helps in some "advanced" scenarios in which you want to pay attention to the behavior of the shuffle mechanism

Hadoop RDD

- partitions = one per HDFS block
- dependencies = none
- compute(partition) = read corresponding block
- preferredLocations(part) = HDFS block location
- partitioner = none

Filtered RDD

- partitions = same as parent RDD
- dependencies = *one-to-one* on parent
- compute(partition) = compute parent and filter it
- preferredLocations(part) = none (*ask parent*)
- partitioner = none

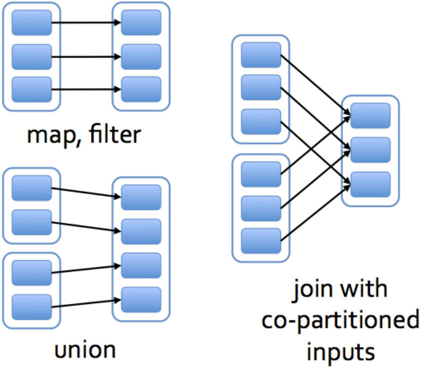
Joined RDD

- partitions = one per reduce task
- dependencies = *shuffle* on *each* parent
- compute(partition) = read and join shuffled data
- preferredLocations(part) = none
- partitioner = HashPartitioner(numTask)[1]

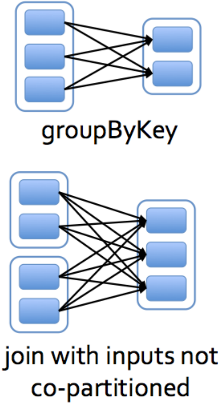
[1] Spark knows this data is hashed.

Dependency Types (1)

Narrow dependencies



Wide dependencies

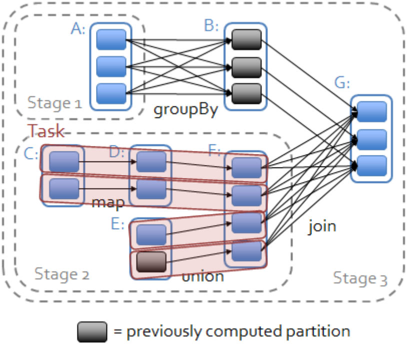


Dependency Types (2)

- **Narrow dependencies**
  - Each partition of the parent RDD is used by at most one partition of the child RDD
  - Task can be executed locally and we don't have to shuffle. (Eg: `map`, `flatMap`, `filter`, `sample`)
- **Wide Dependencies**
  - Multiple child partitions may depend on one partition of the parent RDD
  - This means we have to shuffle data **unless the parents are hash-partitioned** (Eg: `sortByKey`, `reduceByKey`, `groupByKey`, `cogroupByKey`, `join`, `cartesian`)

Dependency Types: Optimizations

- **Benefits of Lazy evaluation:** The DAG Scheduler optimizes *Stages* and *Tasks* before submitting them to the Task Scheduler
  - Examples:
    - **Piplining** narrow dependencies within a Stage
    - **Join plan selection** based on partitioning
    - **Cache reuse**



Operations on RDDs: Transformations

- **Transformations**
  - Set of operations on a RDD that define how they should be transformed
  - As in relational algebra, the application of a transformation to an RDD yields a new RDD (because RDDs are *immutable*)
  - Transformations are lazily evaluated, which allow for optimizations to take place before execution

- **Examples** (not exhaustive)
  - `map(func)`, `flatMap(func)`, `filter(func)`
  - `grouByKey()`
  - `reduceByKey(func)`, `mapValues(func)`, `distinct()`, `sortByKey(func)`
  - `join(other)`, `union(other)`
  - `sample()`

Operations on RDDs: Actions

- **Actions**
  - Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
  - Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver

- **Examples** (not exhaustive)
  - `reduce(func)`
  - `collect()`, `first()`, `take()`, `foreach(func)`
  - `count()`, `countByKey()`
  - `saveAsTextFile()`

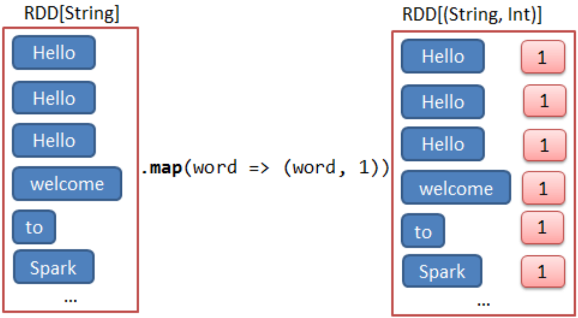
Operations on RDDs: Final Notes

- **Look at return types!**
  - Return type: RDD → transformation
  - Return type: built-in scala/java types such as `int`, `long`, `List<Object>`, `Array<Object>` → action
- **Caching is a transformation**
  - Hints to keep RDD in memory after its first evaluation
- **Transformations depend on RDD "flavor"**
  - `PairRDD`
  - `SchemaRDD`

Common Transformations

`map(f: T => U)`

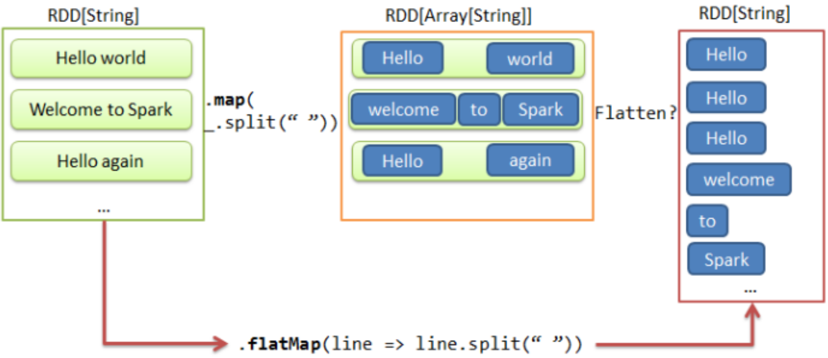
Returns a `MappedRDD[U]` by applying *f* to each element



Common Transformations

`flatMap(f: T => TraversableOnce[U])`

Returns a `FlatMappedRDD[U]` by first applying *f* to each element, then flattening the results



Advanced Topics

Accumulators (1)

- **Underlying idea**
  - Simple mechanism and syntax for aggregating values from worker nodes back to the driver program
- **Common uses**
  - Count events that occur during job execution for debugging purposes
  - Sharing state with the driver program

Accumulators (2)

- **Why another mechanism when we have `reduce()` or `collectAs()` calls?**
  - Accumulators are a simple way to aggregate values that are generated at different scale or granularity than that of an RDD
  - Workers cannot read the value of accumulators, they can only write to it → they are *write-only* variables from the workers' perspective
- **What about failures?**
  - Accumulators used in actions: Spark applies each task's update to each accumulator only once → we must put them inside an action like `foreach()` to achieve correctness.
  - Accumulators used in transformations: there are no guarantees, hence an accumulator update within a transformation can occur more than once

Accumulators (3)

Basic work-flow to use accumulators:

- Create them in the driver by calling the `SparkContext.accumulator(initial Value)` method, which produces an accumulator holding an initial value. The return type is an `org.apache.spark.Accumulator[T]` object, where T is the type of initialValue
- Worker code in Spark closures can add to the accumulator with its $+=$ method
- The driver program can call the value property on the accumulator to access its value

Broadcast Variables (1)

- **Underlying idea**
  - Shared variable allowing the driver to efficiently send a large, *read-only* value to all the worker nodes
- **Common uses**
  - Send a large, read-only lookup table to all the workers, or even a large feature vector in a machine learning algorithm
  - It is an enhanced version of Hadoop MapReduce distributed cache
- **A note of precaution:**
  - For large values to be broadcast, the time to send them over the network can quickly become a bottleneck
  - It is important to choose a data serialization format that is both fast and compact

Broadcast Variables (2)

Basic work-flow to use broadcast variables:

- Create a `Broadcast[T]` by calling `SparkContext.broadcast` on an object of type T. Any type works as long as it is also `Serializable`
- Access its value with the value property
- The variable will be sent to each node only once, and should be treated as *read-only* (updates will not be propagated to other nodes)

MLLib

- **Set of machine learning algorithms written on top of Spark**
  - High-quality implementations of standard algorithms
  - Special data types to manipulate Vectors, Matrices, ...
- **Examples of problems that can be addressed with MLLib**
  - Classification, regression
  - Clustering
  - Collaborative filtering
  - Dimensionality reduction
- **Machine Learning Pipelines**
  - Higher-level API built on top of DataFrames
  - Although it is an important topic, we will not use them in this course