Advanced Programming

Lesson 5

Hortonworks

Learning Objectives

- After you complete this lesson you should be able to:
  - Explain the core components of a Spark Application
  - Understand partitions in Spark
  - Understand the shuffle
  - Explain Job/Stage/Task and creation of the DAG
  - Describe the difference between cache and persist

Hortonworks

Core Components of a Spark application

- Storage System (HDFS/S3/etc)
- YARN or Spark Standalone Resource Manager
- Driver
- SparkContext
- Executor

Hortonworks

HDFS

- Hadoop distributed file system
- Provides fault tolerant storage for data
- Scales incredibly well to accommodate more data
- Creates blocks of data that in turn can be processed by other frameworks

Hortonworks

YARN – Yet Another Resource Manager

- The “HR Team”
- Democratizes hadoop to allow multiple execution/ processing frameworks to use hadoop
- Provides resources for applications to run in the form of a container
- Allows jobs to be run on a Kerberized cluster

Hortonworks

Spark Executor

- The “workers”
- Comparable, but not equivalent to mappers and reducers
- Do all the processing for the application
- If we lose an executor, anything assigned to the current one, along with any lost data will be reassigned and recomputed on another executor
- Configuring correctly can greatly increase performance

Hortonworks

Spark Context

- The “boss”
- Contains the code/objects required to process data in the cluster
- Works with YARN to get resources for the application
- Coordinates the processing, following the DAG schedule
- Schedules the tasks to be done on the executors
- Checks in with executors to report on work being done
  - We can see this in the web ui, covered in the future

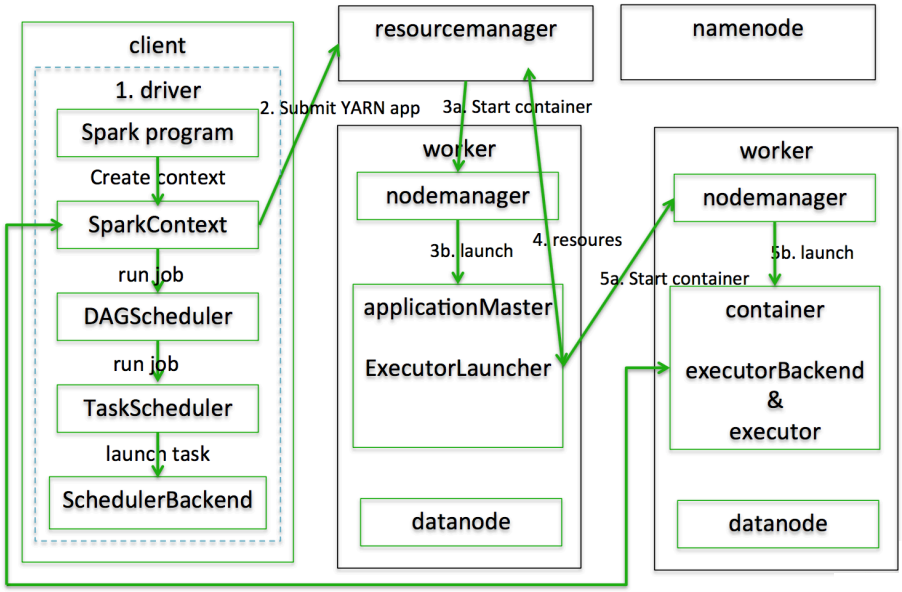
Hortonworks

Spark Driver

- The “owner”
- Responsible for containing the Spark Context
- Holds resources for the Spark Context to communicate with the cluster
- Responsible for writing logs from the context
- Essentially the most important part
  - If this goes down, the job fails

Hortonworks

How do they all fit together?



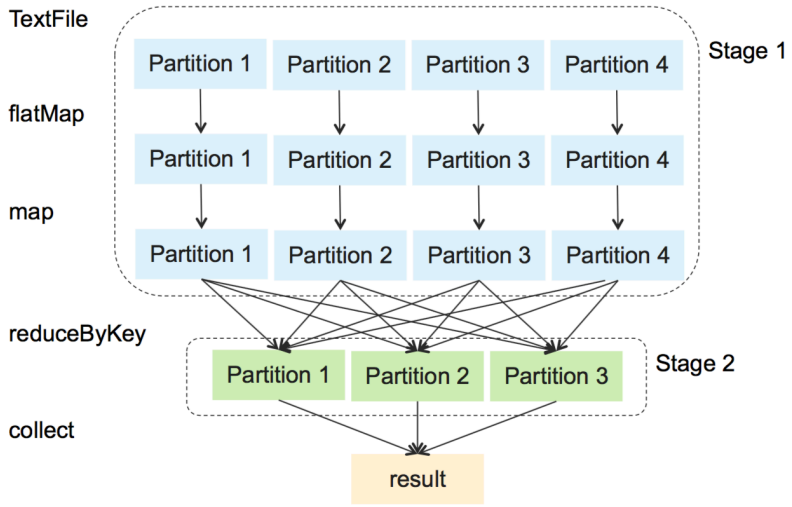
Hortonworks

Spark Partitioning/Parallelism

- White Board
- Notes:
  - Defaults to number of blocks a file is on HDFS
  - Can be set with spark.default.parallelism
    - Default is num of cores on local, or total cores on all executors
    - This setting is for operations with no parent RDD's
  - Shuffle based operations (join, reduceByKey, etc) take the largest number in the parent
  - Shuffle based operations take an optional param for partitions
    - rdd.reduceByKey(lamda a, b: a+b, **20**) will have 20 partitions

Hortonworks

Example: word-count



Hortonworks

Tune Data Parallelism

- Spark works with partitions as the core mechanism for data parallelization
- Use repartition() or coalesce() to control parallelism when needed
  - Use coalesce when reducing partitions, repartition to increase

  rdd.repartition(500)

  rdd.coalesce(20)
- Many operations include numPartitions as parameter that does this automatically

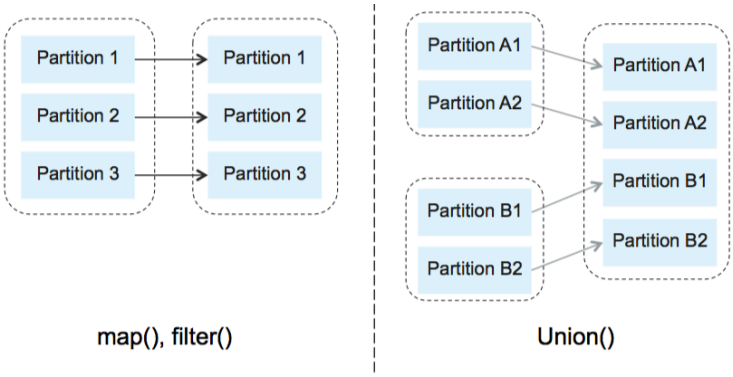
Hortonworks

Wide vs Narrow Operations

- Wide operations require a shuffling of data (normally)
  - reduceByKey
  - groupByKey
  - repartition
  - join
- Narrow operations can be executed locally
  - map
  - filter
  - flatMap

Hortonworks

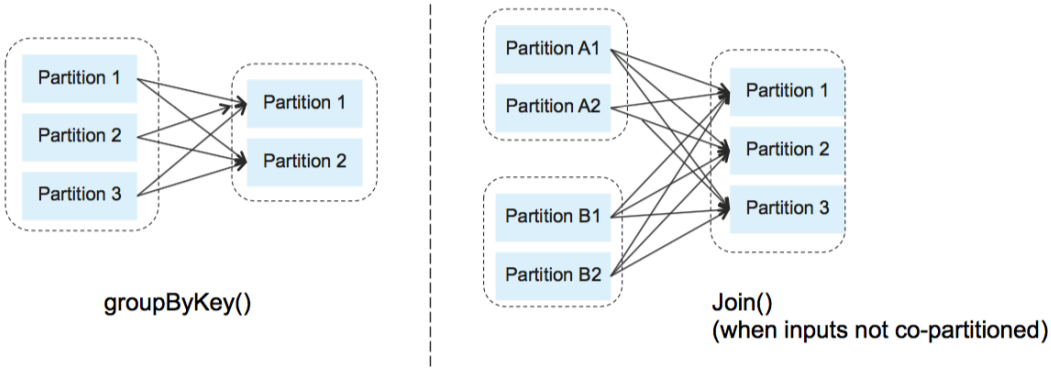
Narrow Dependencies/Operations



Narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD

Hortonworks

Wide Dependencies



Wide dependencies, where multiple child partitions may depend on a single partition

Hortonworks

Understanding the Shuffle

- White board how a shuffle works
- Extra Notes/Gotchas
  - Lots of parallelism + lots of reducers = lots of small files
    - Not a big deal, except when you open more then 32k files on a system
    - spark.shuffle.consolidateFiles=False
  - Shuffle the minimal amount of data
    - Avoid groupByKeys
    - Filter/Distinct Early!
  - Embrace the Shuffle
    - Not necessarily bad

Hortonworks

Spark Task/Stage/Job

- Task is a unit of work (pipeline of narrow operations)
- Stage is a group of tasks separated by a wide operation
- A job is a grouping of stages

- The next stage cannot start before all the tasks in the previous stage have finished

Hortonworks

Caching/Persisting Data

- Spark allows the developer to “put” data into memory
  - Very useful (and incredibly fast) for iterative applications
  - Useful when an RDD is going to be used more then once

- Spark offers several ways to “put” data into memory
  - persist()
  - cache()

Hortonworks

Use RDD persistence to avoid re-computation

- RDD can be persisted in memory across operations
  - Each node stores any partitions in the TDD computed by that node
  - Cached data is reused from memory for other transformations/ actions down the processing DAG
- Usage: rdd.persist(storageLevel)
  - cache() == persist(MEMORY_ONLY_SER)
- Must import library to use it:

  scala -> import org.apache.spark.storageLevel._

  python -> from pyspark import StorageLevel

Hortonworks

Spark RDD Persist options

Storage Level	Where?	Storage format	Comments
MEMORY_ONLY	RAM	Deserialized	Default level
MEMORY_AND_DISK	RAM and DISK	Deserialized	Disk is backup for partitions that don't fit in memory
MEMORY_ONLY_SER	RAM	Serialized	Reduced RAM but more CPU intensive
MEMORY_AND_DISK_SER	RAM AND DISK	Serialized	Reduced RAM but more CPU intensive
DISK_ONLY	DISK	Deserialized	
MEMORY_ONLY_2	RAM	Deserialized	Stores each partition on two cluster nodes
MEMORY_AND_DISK_2	RAM AND DISK	Deserialized	Stores each partition on two cluster nodes
OFF_HEAP	TACHYON	Serialized	Experimental

Hortonworks

Which storage level to choose?

- If your RDD fits in memory, use the default MEMORY_ONLY
- If RDDs don't fit in memory, try MEMORY_ONLY_SER
  - This uses more CPU, so use efficient serialization like Kryo
  - It may be faster to read and RDD from disk then recomputing then use MEMORY_AND_DISK
  - Set spark.rdd.compress to true, this saves space but cost CPU
- Replicated storage is good for fast fault recovery
  - Usually this is overkill, and not a good idea if you're using a lot of data relative to total memory

Hortonworks

Lab 3

- We're going to redo Lab 1, but use the cache to show performance improvements.

Hortonworks

Conclusion and Key Points