Apache Spark Internals

Pietro Michiardi

Eurecom

Acknowledgments & Sources

- Research papers:
  `https://spark.apache.org/research.html`
- Presentations:
- M. Zaharia, “Introduction to Spark Internals”
  `https://www.youtube.com/watch?v=49Hr5xZyTEA`
- A. Davidson, “A Deeper Understanding of Spark Internals”,
  `https://www.youtube.com/watch?v=dmL0N3qfSc8`

Introduction and Motivations

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

- **Software engineering point of view**
  - Hadoop code base is huge
  - Contributions/Extensions to Hadoop are cumbersome
  - Java-only hinders wide adoption, but Java support is fundamental

- **System/Framework point of view**
  - Unified pipeline
  - Simplified data flow
  - Faster processing speed

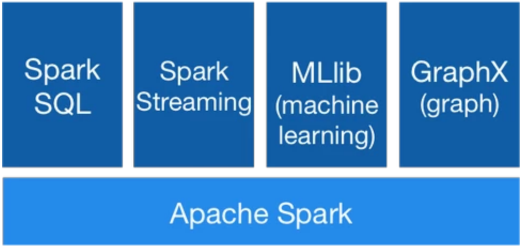
- **Data abstraction point of view**
  - New fundamental abstraction RDD
  - Easy to extend with new operators
  - More descriptive computing model

Hadoop: No Unified Vision

General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

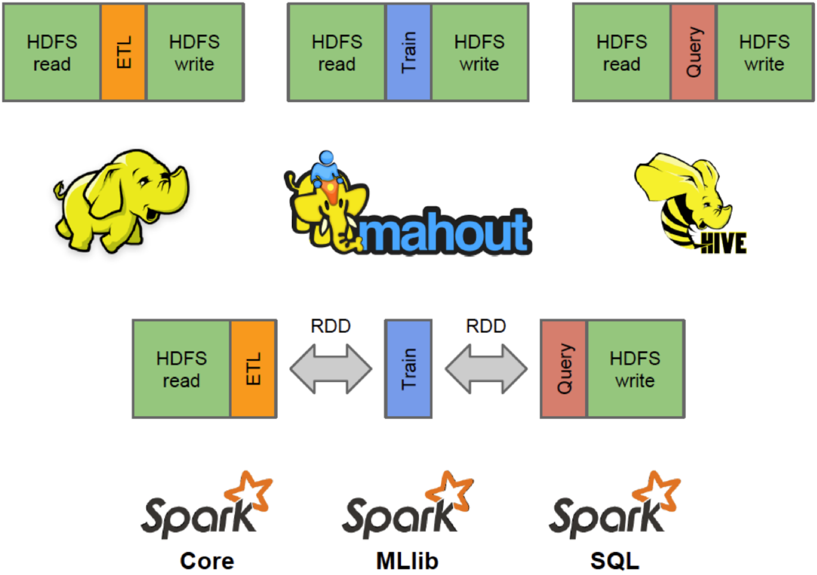
- Sparse modules
- Diversity of APIs
- Higher operational costs

SPARK: A Unified Pipeline



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

A Simplified Data Flow



Hadoop: Bloated Computing Model

```
public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }
```

SPARK: Descriptive Computing Model

```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
   .map(word => (word,1))
   .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

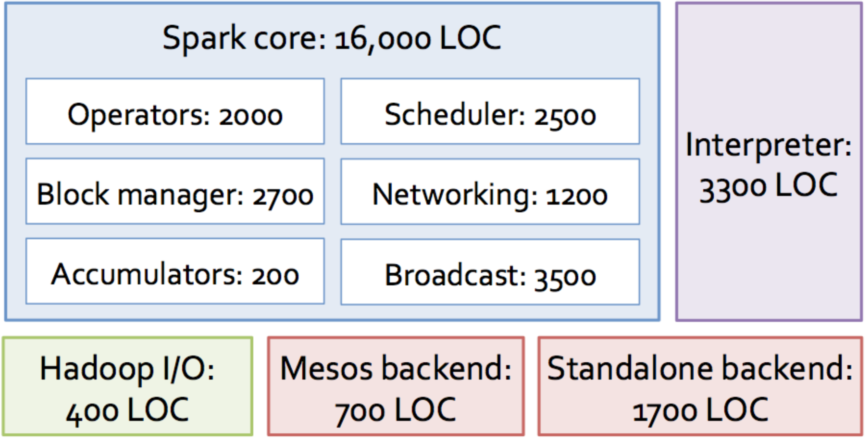
- Organize computation into multiple stages in a processing pipeline
  - **Transformations** apply user code to distributed data in parallel
  - **Actions** assemble final output of an algorithm, from distributed data

Faster Processing Speed

- **Let's focus on iterative algorithms**
  - Spark is faster thanks to the simplified data flow
  - We avoid materializing data on HDFS after each iteration

- **Example: k-means algorithm, 1 iteration**
  - `HDFS Read`
  - **`Map`**(*Assign sample to closest centroid*)
  - **`GroupBy`**(Centroid_ID)
  - `NETWORK Shuffle`
  - **`Reduce`**(*Compute new centroids*)
  - `HDFS Write`

Code Base (2012)



- 2012 (version 0.6.x): 20,000 lines of code
- 2014 (branch-1.0): 50,000 lines of code

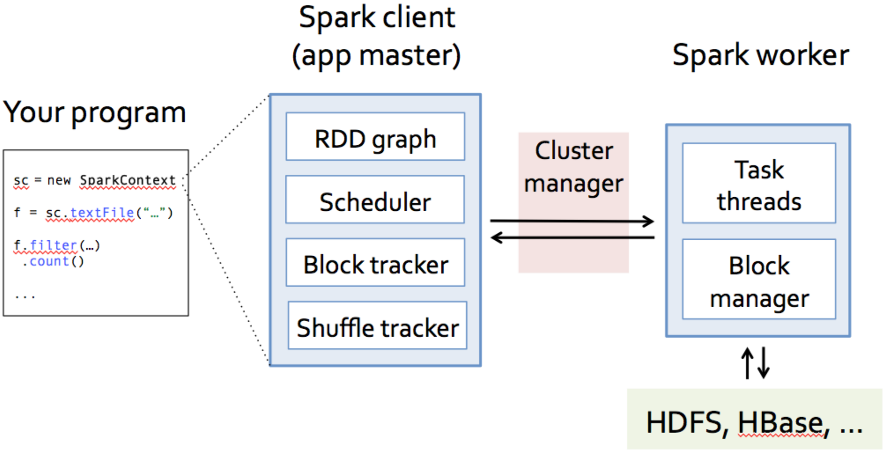
Anatomy of a Spark Application

A Very Simple Application Example

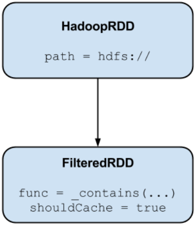
```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

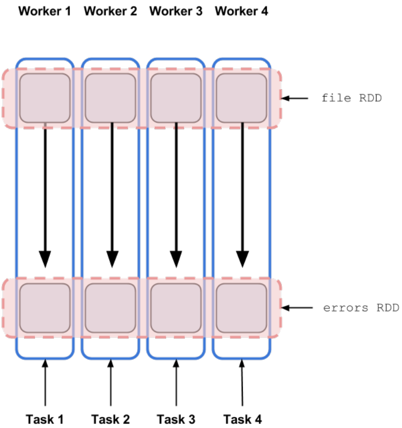
Apache Spark adopts the lazy evaluation model

Spark Components: details



The RDD graph: dataset vs. partition views



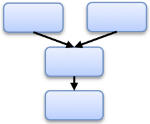


Data Locality

- **Data locality principle**
  - Same as for Hadoop MapReduce
  - Avoid network I/O, workers should manage local data
- **Data locality and caching**
  - First run: data not in cache, so use HadoopRDD's locality prefs (from HDFS)
  - Second run: FilteredRDD is in cache, so use its locations
  - If something falls out of cache, go back to HDFS

Lifetime of a Job in Spark

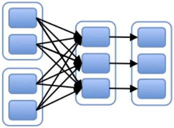
RDD Objects



```
 rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

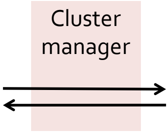
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

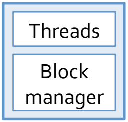
Task Scheduler



Launch tasks via Master

Retry failed and straggler tasks

Worker



Execute tasks

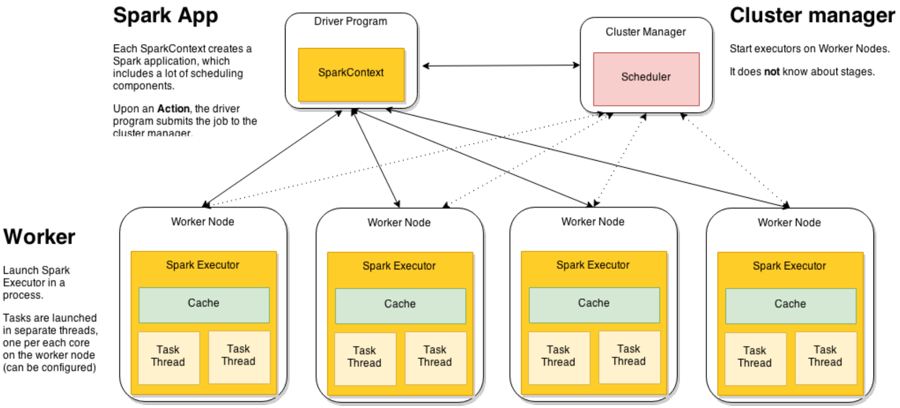
Store and serve blocks

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

Spark Deployments

Spark Components: System-level View

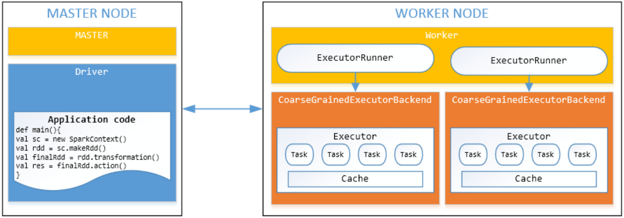


Spark Deployment Modes

- **The Spark Framework can adopt several cluster managers**
  - *Local Mode*
  - *Standalone mode*
  - *Apache Mesos*
  - *Hadoop YARN*

- **General "workflow"**
  - The Spark application creates a `SparkContext`, which initializes the `DriverProgram`
  - Registers to the `ClusterManager`
  - Ask resources to allocate Executors
  - Schedule Task execution

Worker Nodes and Executors



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

Comparison with Hadoop MapReduce

Hadoop MapReduce

- One Task per UNIX process (JVM), more if JVM reuse
- `MultiThreadedMapper`, advanced feature to have threads in Map Tasks

→ **Short-lived** Executor, with one **large Task**

Spark

- Tasks run in one or more Threads, within a single UNIX process (JVM)
- Executor process statically allocated to worker, even with no threads

→ **Long-lived** Executor, with many **small Tasks**

Benefits of the Spark Architecture

- **Isolation**
  - Applications are completely isolated
  - Task scheduling *per application*
- **Low-overhead**
  - Task setup cost is that of spawning a thread, not a process
  - 10-100 times faster
  - Small tasks $\rightarrow$ mitigate effects of data skew
- **Sharing data**
  - Applications cannot share data in memory natively
  - Use an external storage service like Tachyon
- **Resource allocation**
  - Static process provisioning for executors, even without active tasks
  - Dynamic provisioning under development

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

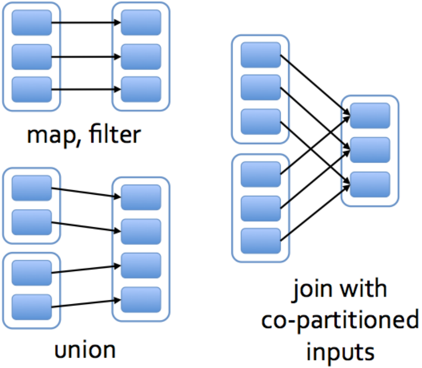
Joined RDD

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

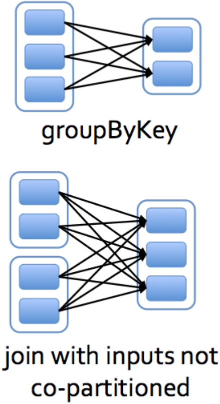
[1] Spark knows this data is hashed.

Dependency Types (1)

Narrow dependencies



Wide dependencies

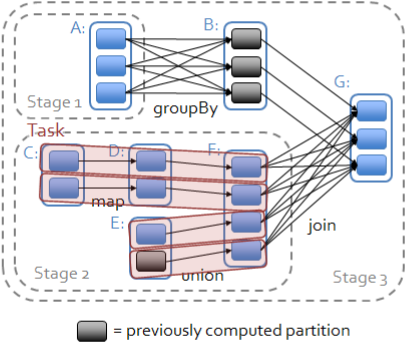


Dependency Types (2)

- **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. (E.g. `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** (E.g. `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:
    - **Pipelining** narrow dependencies within a Stage
    - **Join plan selection** based on partitioning
    - **Cache reuse**



Operations on RDDs: Transformations

- **Transformations**
  - Set of operations on an 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 allows optimizations to take place before execution

- **Examples** (not exhaustive)
  - `map(func), flatMap(func), filter(func)`
  - `groupByKey()`
  - `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 $\rightarrow$ transformation
  - Return type: built-in scala/java types such as `int`, `long`, `List<Object>`, `Array<Object>` $\rightarrow$ action
- **Caching is a transformation**
  - Hints to keep RDD in memory after its first evaluation
- **Transformations depend on RDD "flavor"**
  - `PairRDD`
  - `SchemaRDD`

Detailed Example: Word Count

Spark Word Count: the driver

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

val sc = new SparkContext("spark://...", "MyJob",
   "spark home", "additional jars")
```

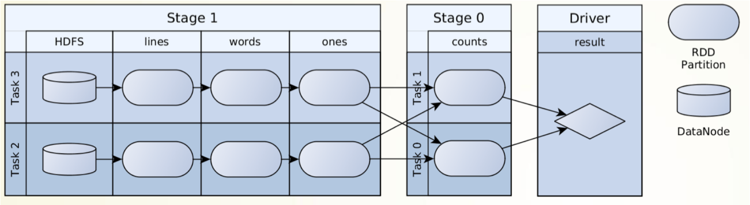
- **Driver and `SparkContext`**
  - A SparkContext initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors
  - Then, the driver takes full control of the Spark job

Spark Word Count: the code

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(w => (w, 1))
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

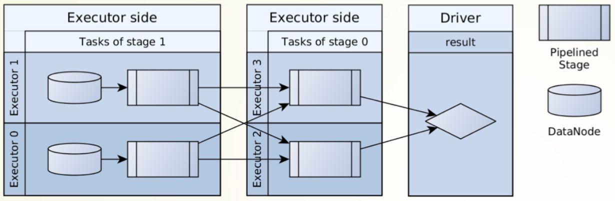
- **RDD lineage DAG is built on driver side with**
  - Data source RDD(s)
  - Transformation RDD(s), which are created by transformations
- **Job submission**
  - An *action* triggers the DAG scheduler to submit a job

Spark Word Count: the DAG



- **Directed Acyclic Graph**
  - Built from the RDD lineage
- **DAG scheduler**
  - Transforms the DAG into stages and turns each partition of a stage into a single task
  - Decides what to run

Spark Word Count: the execution plan



- **Spark Tasks**
  - Serialized RDD lineage DAG + closures of transformations
  - Run by Spark executors

- **Task scheduling**
  - The driver side task scheduler launches tasks on executors according to resource and locality constraints
  - The task scheduler decides where to run tasks

Spark Word Count: the Shuffle phase

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(w => (w, 1))
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- **reduceByKey transformation**
  - Induces the shuffle phase
  - In particular, we have a *wide dependency*
  - Like in Hadoop MapReduce, intermediate <key,value> pairs are stored on the local file system

- **Automatic combiners!**
  - The `reduceByKey` transformation implements map-side combiners to pre-aggregate data

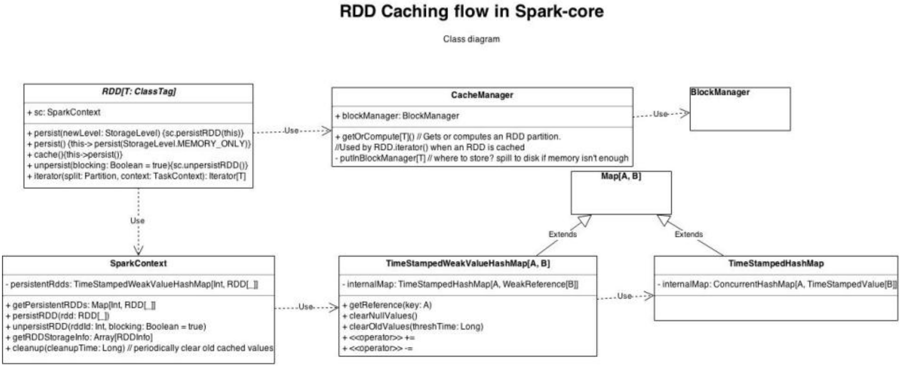
Advanced Topics

Caching and Storage

Spark's Storage Module

- **The storage module**
  - Access (I/O) "external" data sources: HDFS, Local Disk, RAM, remote data access through the network
  - Caches RDDs using a variety of "storage levels"
- **Main components**
  - The Cache Manager: uses the Block Manager to perform caching
  - The Block Manager: distributed key/value store

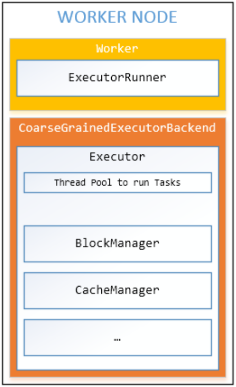
Class Diagram of the Caching Component



How Caching Works

- **Frequently used RDD can be stored in memory**
  - Deciding which RDD to cache is an art!
  - One method, one short-cut: `persist()`, `cache()`
- **`SparkContext` keeps track of cached RDD**
  - Uses a data-structed called `persistentRDD`
  - Maintains references to cached RDD, and eventually call the garbage collector
  - Time-stamp based invalidation using `TimeStampedWeakValueHashMap[A, B]`

How Caching Works




```
- rdd.iterator(split:Partition)
-- cacheManager.getOrCompute(rdd, partition)
--- val key = RDDBlockId(rdd.id, partition.index)
--- switch(blockManager.get(key))
---       case Some: return
---       case None:
                  val computedValue = rdd.computeOrReadCheckPoint(partition)
                  if isRunningLocally
                          return computedValue
                  else
                          cachedValue = putInBlockManager(key, computedValue, storageLevel)
```

BlockManager.scala

```
// write-once key-value
// serves both cachedRdds and shuffle data
// tracks storage level of each block
// can drop data on disk if RAM mem is low
// can replicate data across nodes
def get(blockId){
        tryGetLocal()
        tryGetRemote()
        if no, return None
}
```

The Block Manager

- **"Write-once" key-value store**
  - One node per worker
  - No updates, data is immutable
- **Main tasks**
  - Serves shuffle data (local or remote connections) and cached RDDs
  - Tracks the "Storage Level" (RAM, disk) for each block
  - Spills data to disk if memory is insufficient
  - Handles data replication, if required

Storage Levels

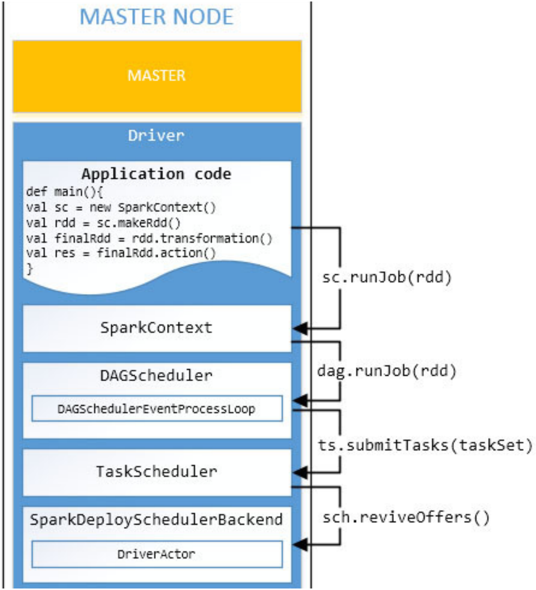
- **The Block Manager can hold data in various storage tiers**
  - `org.apache.spark.storage.StorageLevel` contains flags to indicate which tier to use
  - Manual configuration, in the application
  - Deciding the storage level to use for RDDs is not trivial
- **Available storage tiers**
  - RAM (default option): if the the RDD doesn't fit in memory, some partitions will not be cached (will be re-computed when needed)
  - Tachyon (off java heap): reduces garbage collection overhead, the crash of an executor no longer leads to cached data loss
  - Disk
- **Data format**
  - Serialized or as Java objects
  - Replicated partitions

Resource Allocation: Spark Schedulers

Spark Schedulers

- **Two main scheduler components, executed by the driver**
  - The DAG scheduler
  - The Task scheduler
- **Objectives**
  - Gain a broad understanding of how Spark submits Applications
  - Understand how *Stages* and *Tasks* are built, and their optimization
  - Understand interaction among various other Spark components

Submitting a Spark Application: A Walk Through



Submitting a Spark Application: Details

```
- finalRdd.action()
-- sc.runJob()
--- dagScheduler.runJob()
---- dagScheduler.submitJob()
----- DAGSchedulerEventProcessLoop.post(JobSubmitted)
```

```
- DAGSchedulerEventProcessLoop.onRecieve(JobSubmitted)
-- dagScheduler.handleJobSubmitted()
--- finalStage = new Stage()
--- submitStage(finalStage)
---- missingStages = getMissingParentStages(stage).sortById()
---- if missingStages != Nil
     foreach parent <- missingStages: submitStage(parent)
---- else
     dagScheduler.submitMissingTasks(stage)
```

```
def getMissingParentStages(stage){
     case ShuffleDependency: missing += new Stage()
     case NarrowDependency: waiting4Visit.push()
}
```

```
def submitMissingTasks(stage){
     val loc = getPreferredLocs(stage.rdd)
     if stage.isShuffleMap
             tasks:Seq[Task] = partitions.foreach(yield new ShuffleMapTask(p, loc))
     else
             tasks:Seq[Task] = partitions.foreach(yield new ResultTask())

     taskScheduler.submitTasks(new TaskSet(tasks))
}
```

TaskSchedulerimpl.scala

```
def submitTasks(taskSet){
     schedulableBuilder.addTaskSetManager(new TaskSetManager(taskSet))
     if (!runInLocal && !hasReceivedTask){
             starvationTimer.scheduleAtFixedRate(new TimerTask(){…, STARVA
     }
     schedulerBackend.reviveOffers()
}
```

```
def sparkDeploySchedulerBackend.reviveOffers(){
     driverActor ! ReviveOffers
}
```

CoarseGrainedSchedulerBackend.scala

```
def receiveWithLogging(){
     case ReviveOffers => makeOffers(){
             launchTasks(tasks){
                     foreach task <- tasks:
                             executorActor(task.executorId) ! LaunchTask(t
             }
     }
}
```

The DAG Scheduler

- **Stage-oriented scheduling**
  - Computes a DAG of stages for each job in the application
    `Lines 10-14`, details in `Lines 15-27`
  - Keeps track of which RDD and stage output are materialized
  - Determines an optimal schedule, minimizing stages
  - Submit stages as sets of Tasks (`TaskSets`) to the Task scheduler
    `Line 26`
- **Data locality principle**
  - Uses “preferred location” information (optionally) attached to each RDD
    `Line 20`
  - Package this information into Tasks and send it to the Task scheduler
- **Manages Stage failures**
  - Failure type: (intermediate) data loss of shuffle output files
  - Failed stages will be resubmitted
  - NOTE: Task failures are handled by the Task scheduler, which simply resubmit them if they can be computed with no dependency on previous output

The DAG Scheduler: Implementation Details

- **Implemented as an event queue**
  - Uses a daemon thread to handle various kinds of events
    `Line 6`
  - `JobSubmitted`, `JobCancelled`, `CompletionEvent`
  - The thread “swipes” the queue, and routes event to the corresponding handlers

- **What happens when a job is submitted to the DAGScheduler?**
  - `JobWaiter` object is created
  - `JobSubmitted` event is fired
  - The daemon thread blocks and waits for a job result
    `Lines 3,4`

The DAG Scheduler: Implementation Details (2)

- **Who handles the `JobSubmitted` event?**
  - Specific handler called `handleJobSubmitted`
    `Line 6`
- **Walk-through to the Job Submitted handler**
  - Create a new job, called `ActiveJob`
  - New job starts with only 1 stage, corresponding to the last stage of the job upon which an action is called
    `Lines 8-9`
  - Use the dependency information to produce additional stages
    - ★ Shuffle Dependency: create a new map stage
      `Line 16`
    - ★ Narrow Dependency: pipes them into a single stage
      `getMissingParentStages`

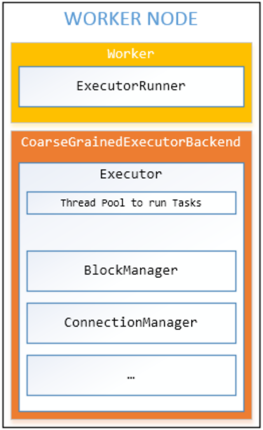
More About Stages

- **What is a DAG**
  - Directed acyclic graph of stages
  - Stage boundaries determined by the shuffle phase
  - Stages are run in *topological order*
- **Definition of a Stage**
  - Set of *independent* tasks
  - All tasks of a stage apply the same function
  - All tasks of a stage have the same dependency type
  - All tasks in a stage belong to a `TaskSet`
- **Stage types**
  - Shuffle Map Stage: stage tasks results are inputs for another stage
  - Result Stage: tasks compute the final action that initiated a job (e.g., `count()`, `save()`, etc.)

The Task Scheduler

- **Task oriented scheduling**
  - Schedules tasks for a *single* `SparkContext`
  - Submits tasks sets produced by the DAG Scheduler
  - Retries failed tasks
  - Takes care of *stragglers* with speculative execution
  - Produces events for the DAG Scheduler
- **Implementation details**
  - The Task scheduler creates a `TaskSetManager` to wrap the `TaskSet` from the DAG scheduler
    `Line 28`
  - The `TaskSetManager` class operates as follows:
    - Keeps track of each task status
    - Retries failed tasks
    - Imposes data locality using *delayed scheduling*
      `Lines 29,30`
  - Message passing implemented using *Actors*, and precisely using the *Akka framework*

Running Tasks on Executors



Executor.scala

```
def launchTask(serializedTask){
      val tr = new TaskRunner(serializedTask)
      threadPool.execute(tr)
}
```

TaskRunner.scala

```
def run(){
      executorBackend.statusUpdate(RUNNING)
      val task = serializer.deserialize(serializedTask)
      val value = task.run()
      val res = new DirectTaskResult(serializer.serialize(value))
      executorBackend.statusUpdate(FINISHED)
      if res.size > akkaFrameSize
              blockManager.putBytes(taskId, res)
      else
              return res
}
```

```
task.run()
if task is ResultTask
      val (rdd, func) = ser.deserialize(RDD, taskContext)
      return result = func(rdd.iterator(partition))

if task is ShuffleMapTask
      val (rdd, dep) = ser.deserialize(RDD, ShuffleDependency, taskContext)
      shuffleWriter = shuffleManager.getWriter(dep.shuffleHandler, partitionId)
      shuffleWriter.write(rdd.iterator(partition))
      return shuffleWriter.stop().get()
```

ShuffleRDD.scala / CoGroupedRDD.scala

```
def compute(split, taskContext){
      shuffleManager.getReader(dep.shuffleHandler, split.index, taskContext).read()
}
```

Running Tasks on Executors

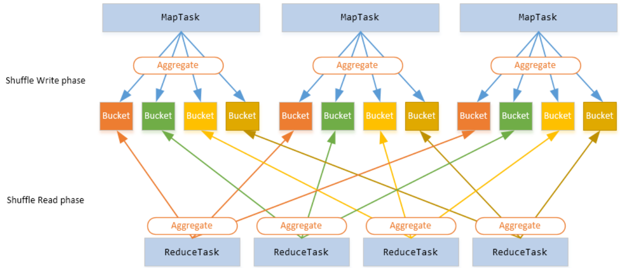
- **Executors run two kinds of tasks**
  - `ResultTask`: apply the action on the RDD, once it has been computed, alongside all its dependencies
    `Line 19`
  - `ShuffleTask`: use the Block Manager to store shuffle output using the `ShuffleWriter`
    `Lines 23,24`
  - The `ShuffleRead` component depends on the type of the RDD, which is determined by the compute function and the transformation applied to it

Data Shuffling

The Spark Shuffle Mechanism

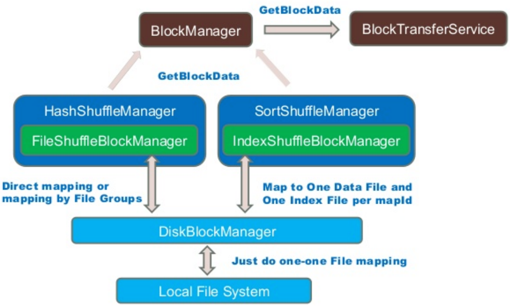
- **Same concept as for Hadoop MapReduce, involving:**
  - Storage of “intermediate” results on the local file-system
  - Partitioning of “intermediate” data
  - Serialization / De-serialization
  - Pulling data over the network
- **Transformations requiring a shuffle phase**
  - `groupByKey()`, `reduceByKey()`, `sortByKey()`, `distinct()`
- **Various types of Shuffle**
  - *Hash Shuffle*
  - *Consolidate Hash Shuffle*
  - *Sort-based Shuffle*

The Spark Shuffle Mechanism: an Illustration



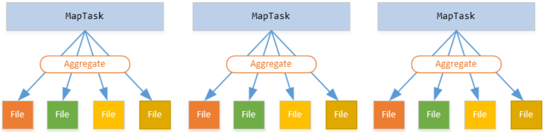
- **Data Aggregation**
  - Defined on `ShuffleMapTask`
  - Two methods available:
    - `AppendOnlyMap`: in-memory hash table combiner
    - `ExternalAppendOnlyMap`: memory + disk hash table combiner
- **Batching disk writes to increase throughput**

The Spark Shuffle Mechanism: Implementation Details



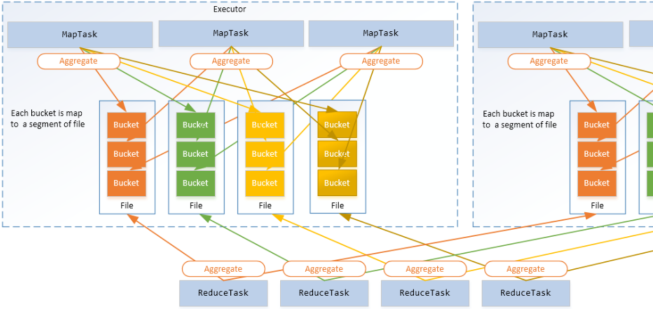
- **Pluggable component**
  - *Shuffle Manager*: components registered to `SparkEnv`, configured through `SparkConf`
  - *Shuffle Writer*: tracks "intermediate data" for the `MapOutputTracker`
  - *Shuffle Reader*: pull-based mechanism used by the `ShuffleRDD`
  - *Shuffle Block Manager*: mapping between logical partitioning and the physical layout of data

The Hash Shuffle Mechanism



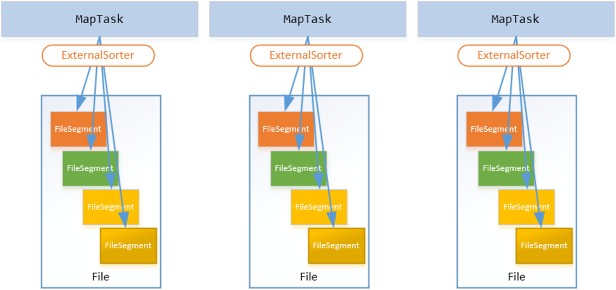
- **Map Tasks write output to multiple files**
  - Assume: m map tasks and r reduce tasks
  - Then: $m \times r$ shuffle files as well as in-memory buffers (for batching writes)
- **Be careful on storage space requirements!**
  - Buffer size must not be too big with many tasks
  - Buffer size must not be too small, for otherwise throughput decreases

The Consolidate Hash Shuffle Mechanism



- **Addresses buffer size problems**
  - Executor view vs. Task view
  - Buckets are consolidated in a single file
  - Hence: $F = C \times r$ files and buffers, where C is the number of Task threads within an Executor

The Sort-based Shuffle Mechanism



- **Implements the Hadoop Shuffle mechanism**
  - Single shuffle file, plus an index file to find "buckets"
  - Very beneficial for write throughput, as more disk writes can be batched
- **Sorting mechanism**
  - Pluggable external sorter
  - Degenerates to Hash Shuffle if no sorting is required

Data Transfer: Implementation Details

- **BlockTransfer Service**
  - General interface for `ShuffleFetcher`
  - Uses `BlockDataManager` to get local data
- **Shuffle Client**
  - Manages and wraps the "client-side", setting up the `TransportContext` and `TransportClient`
- **Transport Context**: manages the transport layer
- **Transport Server**: streaming server
- **Transport Client**: fetches consecutive chunks

Data Transfer: an Illustration