PART A. BIG DATA TECHNOLOGY

3. DISTRIBUTED COMPUTING MODELS FOR SCALABLE BATCH COMPUTING

SECTION 2: IN-MEMORY CLUSTER COMPUTING

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

• PA2 description will be posted this week

• Weekly Reading List
  
## Topics of Today's Class

- RDD Actions and Persistence
- 3. Distributed Computing Models for Scalable Batch Computing
  - Spark cluster
  - RDD dependency
  - Job scheduling
  - Closure

---

**In-Memory Cluster Computing: Apache Spark**

- RDD: Transformations
- RDD: Actions
- RDD: Persistence
### Actions [1/2]

- Returns a final value to the driver program
  - Or writes data to an external storage system

- Log file analysis example is continued
  - `take()` retrieves a small number of elements in the RDD at the driver program
  - Iterates over them locally to print out information at the driver

```scala
println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

### Actions [2/2]

- `collect()`
  - Retrieves entire RDD to the driver
  - Entire dataset (RDD) should fit in memory on single machine
  - If the RDD is filtered down to a very small dataset, it is useful

- For very large RDD
  - Store them in the external storage (e.g. S3, or HDFS)
  - `saveAsTextFile()` action
reduce() 

- Takes a function that operates on two elements of the type in your RDD and returns a new element of the same type
- The function should be **commutative** and **associative** so that it can be computed correctly in parallel

```scala
val rdd1 = sc.parallelize(List(1, 2, 5))
val sum = rdd1.reduce{(x, y) => x + y}
//results: sum: Int = 8
```

reduce() vs. fold() 

- Similar to reduce() but it takes 'zero value'
  - initial value
- The function should be **commutative** and **associative** so that it can be computed correctly in parallel

```scala>
scala> val rdd1 = sc.parallelize(List( ("maths", 80), ("science", 90) ))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[8] at parallelize at :21
scala> rdd1.partitions.length
result: res8: Int = 8
scala> val additionalMarks = ("extra", 4)
additionalMarks: (String, Int) = (extra,4)
scala> val sum = rdd1.fold(additionalMarks){(acc, marks) => val sum = acc._2 + marks._2 ("total", sum)}
What will be the result(sum)?
reduce() vs. fold()  

- Similar to reduce() but it takes ‘zero value’ (initial value)  
- The function should be **commutative** and **associative** so that it can be computed correctly in parallel

```scala
scala> val rdd1 = sc.parallelize(List(("maths", 80), ("science", 90)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[8] at parallelize at :21
scala> rdd1.partitions.length
result: res8: Int = 8
scala> val additionalMarks = ("extra", 4)
additionalMarks: (String, Int) = (extra,4)
scala> val sum = rdd1.fold(additionalMarks){(acc, marks) =>
val sum = acc._2 + marks._2 ("total", sum)}
// result: sum: (String, Int) = (total,206)
// (4x8)+80+90 = 206
```

take(n)  

- returns n elements from the RDD and attempts to minimize the number of partitions it accesses  
  - It may represent a biased collection  
  - It does not return the elements in the order you might expect  
  - Useful for unit testing
In-Memory Cluster Computing: Apache Spark

RDD: Transformations
RDD: Actions
RDD: Persistence

Persistence

- Caches dataset across operations
  - Nodes store any partitions of results from previous operation(s) in memory reuse them in other actions

- An RDD to be persisted can be specified by `persist()` or `cache()`

- The persisted RDD can be stored using a different storage level
  - Using a `StorageLevel` object
  - Passing `StorageLevel` object to `persist()`
Persistence levels

<table>
<thead>
<tr>
<th>level</th>
<th>Space used</th>
<th>CPU time</th>
<th>In memory/On disk</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>High</td>
<td>Low</td>
<td>Y/N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Low</td>
<td>High</td>
<td>Y/N</td>
<td>Store RDD as <em>serialized</em> Java objects (one byte array per partition).</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>High</td>
<td>Medium</td>
<td>Some/Some</td>
<td>Spills to disk if there is too much data to fit in memory.</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>Low</td>
<td>High</td>
<td>Some/Some</td>
<td>Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory</td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>Low</td>
<td>High</td>
<td>N/Y</td>
<td></td>
</tr>
</tbody>
</table>
Spark cluster and resources

Driver program

SparkContext

Cluster Manager

Executor

Task

Cache

Executor

Task

Cache

Hadoop YARN
Mesos
Standalone

Spark cluster

• Each application gets its own executor processes
  • Must be up and running for the duration of the entire application
  • Run tasks in multiple threads
  • Isolate applications from each other
    • Scheduling side (each driver schedules its own tasks)
    • Executor side (tasks from different applications run in different JVMs)
  • Data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system
Spark cluster

- Spark is agnostic to the underlying cluster manager
  - As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN)

Spark cluster

- **Driver program** must listen for and accept incoming connections from its executors throughout its lifetime
  - Driver program must be network addressable from the worker nodes

- Driver program should run close to the worker nodes
  - On the same local area network
Cluster Manager Types

- **Standalone**
  - Simple cluster manager included with Spark

- **Mesos**
  - Fine-grained sharing option
    - Frequently shared objects for Interactive applications
    - Mesos master determines the machines that handle the tasks

- **Hadoop YARN**
  - Resource manager in Hadoop 2

Dynamic Resource Allocation

- Dynamically adjust the resources that the applications occupy
  - Based on the workload
  - Your application may give resources back to the cluster if they are no longer used

- Only available on coarse-grained cluster managers
  - Standalone mode, YARN mode, Mesos coarse grained mode
In-Memory Cluster Computing: Apache Spark

RDDs in Spark

User's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.
Representing RDDs

- A set of partitions
  - Atomic pieces of the dataset

- A set of dependencies on parent RDDs

- A function for computing the dataset based on its parents

- Metadata about its partitioning scheme

- Data placement

Interface used to represent RDDs in Spark

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions()</td>
<td>Return a list of Partition objects</td>
</tr>
<tr>
<td>preferredLocations(p)</td>
<td>List nodes where partition p can be accessed faster due to data locality</td>
</tr>
<tr>
<td>dependencies()</td>
<td>Return a list of dependencies</td>
</tr>
<tr>
<td>iterator(p, parentIters)</td>
<td>Compute the elements of partition p given iterators for its parent partitions</td>
</tr>
<tr>
<td>partitioner()</td>
<td>Return metadata specifying whether the RDD is hash/range partitioned</td>
</tr>
</tbody>
</table>
Lazy Evaluation

- Transformations on RDDs are **lazily evaluated**
  - Spark will **NOT** begin to execute until it sees an action
  - Spark internally records metadata to indicate that this operation has been requested

- Loading data from files into an RDD is lazily evaluated

- Reduces the number of passes it has to take over our data by grouping operations together
Example: Console Log Mining

Spark code

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
errors.filter(_.contains("HDFS"))
  .map(_.split('/t')(3))
  .collect()
```

Benefits of RDDs as a distributed memory abstraction

- RDDs can only be created ("written") through **coarse-grained transformations**
- Distributed shared memory (DSM) allows reads and writes to each memory location
  - Reads on RDDs can still be fine-grained
    - A large read-only lookup table
- Applications perform bulk writes
- More efficient fault tolerance
  - Lineage based bulk recovery
Benefits of RDDs as a distributed memory abstraction [2/3]

- RDDs’ immutable data
  - System can mitigate slow nodes (Stragglers)
  - Creates backup copies of slow tasks
    - without accessing the same memory
  - Spark distributes the data over different working nodes that run computations in parallel
    - Orchestrates communicating between nodes to Integrate intermediate results and combine them for the final result

Benefits of RDDs as a distributed memory abstraction [3/3]

- Runtime can schedule tasks based on data locality
  - To improve performance

- RDDs degrade gracefully when there is insufficient memory
  - Partitions that do not fit in the RAM are stored on disk
Applications not suitable for RDDs

- RDDs are best suited for batch applications that apply the same operations to all elements of a dataset
  - Steps are managed by lineage graph efficiently
  - Recovery is managed effectively

- RDDs would not be suitable for applications
  - Making asynchronous fine-grained updates to shared state
  - e.g. a storage system for a web application or an incremental web crawler

In-Memory Cluster Computing: Apache Spark

RDD Dependency in Spark
Dependency between RDDs [1/4]

Narrow Dependency

Wide Dependency

Dependency between RDDs [2/4]

- Narrow dependency
  - Each partition of the parent RDD is used by at most one partition of the child RDD

http://www.cs.colostate.edu/~cs535  Spring 2020 Colorado State University, page 17
Dependency between RDDs [3/4]

- **Wide** dependency
  - *Multiple child partitions* may depend on a single partition of parent RDD

![Diagram](groupByKey)

Join with inputs not co-partitioned

Dependency between RDDs [4/4]

- **Narrow** dependency
  - Pipelined execution on one cluster node
  - e.g. a map followed by a filter
  - Failure recovery is more straightforward

- **Wide** dependency
  - Requires data from all parent partitions to be available and to be shuffled across the nodes
  - Failure recovery could involve a large number of RDDs
    - Complete re-execution may be required
Dependency

- `filter` (Narrow/Wide)
- `leftOuterJoin` (Narrow/Wide)
- `distinct` (Narrow/Wide)
- `mapPartitions` (Narrow/Wide)
- `repartition` (Narrow/Wide)
- `reduceByKey` (Narrow/Wide)

**Dependency-Answers**

- `filter` (Narrow/Wide)
- `leftOuterJoin` (Narrow/Wide)
- `distinct` (Narrow/Wide)
- `mapPartitions` (Narrow/Wide)
- `repartition` (Narrow/Wide)
- `reduceByKey` (Narrow/Wide)
In-Memory Cluster Computing: Apache Spark
Scheduling

Jobs in Spark application

• “Job”
  • A Spark action (e.g. save, collect) and any tasks that need to run to evaluate that action

• Within a given Spark application, multiple parallel tasks can run simultaneously
  • If they were submitted from separate threads
Job scheduling

- **Stage** is a physical unit of execution
  - A set of parallel tasks

- User runs an **action** (e.g. count or save) on an RDD

- Scheduler examines that RDD’s lineage graph to build a DAG of **stages** to execute

- Each stage contains **as many pipelined transformations as possible**
  - With narrow dependencies

- The **boundaries of the stages** are the **shuffle operations**
  - For **wide dependencies**
  - For any already computed partitions that can short circuit the computation of a parent RDD

---

Example of Spark job stages

```
RDD A -> B -> C -> D -> E -> F -> G
```

- `groupByKey`
- `map`
- `union`
- `collect`

Stages are split whenever the **shuffle phases** occur.

**Question:** How many stages does this job have?
Example of Spark job stages

- RDD A
- B
- C
- D
- E
- F
- G

Stage 1

Stage 2

Stages are split whenever the shuffle phases occur.

Stage 3

Default FIFO scheduler

- By default, Spark's scheduler runs jobs in FIFO fashion

- First job gets the first priority on all available resources
  - Then the second job gets the priority, etc.
  - As long as the resource is available, jobs in the queue will start right away
Fair Scheduler

- Assigns tasks between jobs in a “round robin” fashion
  - All jobs get a roughly equal share of cluster resources
  - Fair Schedule Pool

- Short jobs that were submitted when a long job is running can start receiving resources right away
  - Good response times, without waiting for the long job to finish

- Best for multi-user settings

Fair Scheduler Pools

- Supports grouping jobs into pools
  - With different options (e.g. weights)
  - “high-priority” pool for more important jobs

- This approach is modeled after the Hadoop Fair Scheduler

- Default behavior of pools
  - Each pool gets an equal share of the cluster
  - Inside each pool, jobs run in FIFO order
  - If the Spark cluster creates one pool per user
    - Each user will get an equal share of the cluster
    - Each user’s queries will run in order
In-Memory Cluster Computing: Apache Spark
Closures

Understanding Closures

- To execute jobs, Spark breaks up the processing of RDD operations into tasks to be executed by an executor.

- Prior to execution, Spark computes the task’s closure.

- The closure is those variables and methods that must be visible for the executor to perform its computations on the RDD.

- This closure is serialized and sent to each executor.
Understanding Closures

1: var counter = 0
2: var rdd = sc.parallelize(data)
3: // Wrong: Don't do this!!
4: rdd.foreach(x => counter += x)
5: println("Counter value: " + counter)

- How many different counters are in this example code?

Counter (in line 5) is referenced within the foreach function, it's no longer the counter (in line 1) on the driver node
- Counter (in line 1) will still be zero
- In local mode, in some circumstances the foreach function will actually execute within the same JVM as the driver
  - Counter may be actually updated
Solutions?

- Closures (e.g. loops or locally defined methods) should not be used to mutate some global state
  - Spark does not define or guarantee the behavior of mutations to objects referenced from outside the closures

- **Accumulator** provides a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster

Accumulators

- Variables that are only “added” to through an associative and commutative operation
  - Efficiently supported in parallel
  - Used to implement counters (as in MapReduce) or sums

```scala
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator{id: 0, name: Some(My Accumulator), value: 0}
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
...  
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value res2: Long = 10
```
Accumulators

- Spark natively supports accumulators of type Long, and programmers can add support for new types.

```scala
// Suppose that we have MyVector class representing mathematical vectors
class VectorAccumulatorV2 extends AccumulatorV2[MyVector, MyVector] {
  private val myVector: MyVector = MyVector.createZeroVector
  def reset(): Unit = {
    myVector.reset()
  }
  def add(v: MyVector): Unit = {
    myVector.add(v)
  }
  ...
}

// Then, create an Accumulator of this type:
val myVectorAcc = new VectorAccumulatorV2
// Then, register it into spark context:
sc.register(myVectorAcc, "MyVectorAcc1")
```

- If accumulators are created with a name, they will be displayed in Spark's UI.
Accumulators

- Accumulator updates performed inside **actions only**
  - Spark guarantees that each task’s update to the accumulator will only be applied once
  - Restarted tasks will not update the value

```scala
val accum = sc.longAccumulator
data.map { x => accum.add(x); x }
// Here, **accum is still 0** because no actions have caused the map operation to be computed.
```