In-Memory Cluster Computing: Apache Spark

**RDD: Transformations**

**RDD: Actions**

**RDD: Persistence**

### Persistence levels

<table>
<thead>
<tr>
<th>level</th>
<th>Space</th>
<th>CPU</th>
<th>in memory/On disk</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY_HIGH</td>
<td>High</td>
<td>Low</td>
<td>Y/N</td>
<td>Store RDD as unserialized Java objects (one byte array per partition)</td>
</tr>
<tr>
<td>MEMORY_ONLY_LOW</td>
<td>High</td>
<td>Y/N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMORY_AND_DISK_H</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_L</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 is memory</td>
<td></td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>Low</td>
<td>High</td>
<td>N/Y</td>
<td></td>
</tr>
</tbody>
</table>
In-Memory Cluster Computing: Apache Spark

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 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 on a cluster manager that also supports other applications (e.g. Mesos/YARN)

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

http://www.cs.colostate.edu/~cs535
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

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

Lazy Evaluation

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>iterater(p, partitioned)</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

```
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 backup table
    - Applications perform bulk writes
    - More efficient fault tolerance
    - Lineage based bulk recovery

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

http://www.cs.colostate.edu/~cs535
In-Memory Cluster Computing: Apache Spark

RDD Dependency in Spark

Dependency between RDDs

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

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

- 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
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

Question: How many stages does this job have?

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
- 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

http://www.cs.colostate.edu/~cs535
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

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.

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

```java
1: var counter = 0
2: var add = sc.parallelize(data)
3: 
4: // Wrong: Don’t do this!
5: add.foreach(x => counter += x)
6: 
7: println("Counter value: " + counter)
```

- How many counters are in this example?

- 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
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
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))
```

```scala
accum.value res2: Long = 10
```

If accumulators are created with a name, they will be displayed in Spark’s UI

- 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

PageRank

- Goals
  - Providing effective summaries for the search results
  - Ordering/Ranking results

- Simulate random Web surfers
  - Pages that would have a large number of surfers were considered more “important” than pages that would rarely be visited

- The content of a page was judged not only by the terms appearing on that page
  - But by the terms used in or near the links to that page

http://www.cs.colostate.edu/~cs535
Definition of PageRank

- A function that assigns a real number to each page in the Web
- The higher the PageRank of a page, the more "important" it is
- There is NOT one fixed algorithm for assignment of PageRank

Example

- Page A has links to B, C and D
- Page B has links to A and D
- Page C has a link to A
- Page D has links to B and C

Example

- Suppose that a random surfer starts at page A
- Page B, C and D will be the next with probability 1/3
- 0 probability of being at A

What does this matrix mean?

- The probability \( x_i \) that a random surfer will be at node \( i \) at the next step

\[
x_i = \sum m_{ij} v_j
\]

- \( m_{ij} \) is the probability that a surfer at node \( j \) will move to node \( i \) at the next step
- \( v_j \) is the probability that the surfer was at node \( j \) at the previous step

\[
x' = \beta M v + (1 - \beta) e / n
\]

- Where \( \beta \) is a chosen constant
- Usually in the range 0.8 to 0.9

- \( e \) is a vector for all 1's with the appropriate number of components
- \( n \) is the number of nodes in the Web graph

http://www.cs.colostate.edu/~cs535
PageRank

1. Initialize each page’s rank to 1.0
2. On each iteration, have page $p$ send a contribution of $\frac{\text{rank}(p)}{\text{numNeighbors}(p)}$ to its neighbors (the pages it has link to)
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribution Received}$

PageRank in Spark (in Scala)

```scala
// Assume that our neighbor list was saved as a Spark objectFile val links = 
  sc.objectFile[(String, Seq[String])]("links")
  .partitionBy(new HashPartitioner(100))
  .persist()

// Initialize each page’s rank to 1.0; since we use
// mapValues, the resulting RDD will have the same
// partitioner as links
val links = // RDD of (url, neighbors) pairs
val ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks)
    .flatMap{
      case (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }
  ranks = contribs.reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

Questions?