PART 1. BATCH COMPUTING MODEL FOR BIG DATA ANALYTICS

3. IN-MEMORY CLUSTER COMPUTING

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

CS535 FAQs

• Do NOT consider external links
  - Your HITS calculates only within the wiki pages
• To create base set
  - Check if the title contains your keyword(s)
  - For example for "eclipse", your base set should include,
    - "Solar eclipse"
    - "Eclipse (disambiguation)"
    - "Lunar eclipse"
    - "Solar eclipse of August 21, 2017" etc.
• Minimum search unit is a word
  - Do not consider sub-string search ("eclipse" and "clip" do not match)
• Consider only exact match
• Ignore plurals

Today's topics

• Apache Spark
  - Closures
  - Pair RDD
  - Aggregators
  - Actions

3. 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.

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

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

```scala
def add(v: MyVector): Unit = { myVector.add(v) }
```

- 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 acc = sc.longAccumulator("My Accusulator")
acc: org.apache.spark.util.LongAccumulator = longAccumulator(id: 0, name: Some(My Accumulator), value: 0)
... 10/09/19 18:41:08 INFO SparkContext: Tasks finished in 0.01186 s
acc: acc.value seq2: Long = 10
```
Why Key/Value Pairs?

- Pair RDDs
- Spark provides special operations on RDDs containing key/value pairs
- Pair RDDs allow you to act on each key in parallel or regroup data across the network
- reduceByKey()
  - Aggregates data separately for each key
- join()
  - Merge two RDDs by grouping elements with the same key

Creating Pair RDDs

- Running map() function

```
val pairs = lines.map(x => (x.split(" ").(0), x))
```

Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduceByKey()</td>
<td>Combine values with the same key</td>
<td>rdd.reduceByKey((x, y) =&gt; x + y)</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>Group values with the same key</td>
<td>rdd.groupByKey()</td>
</tr>
<tr>
<td>combineByKey()</td>
<td>Combine values with the same key using a</td>
<td>We will revisit this function later.</td>
</tr>
<tr>
<td></td>
<td>different result type</td>
<td></td>
</tr>
<tr>
<td>mapValues(func)</td>
<td>Apply a function to each value of a pair</td>
<td>rdd.mapValues(x =&gt; x + 1)</td>
</tr>
<tr>
<td></td>
<td>RDD without changing the key</td>
<td>{(1, 3), (3, 5), (3, 7)}</td>
</tr>
<tr>
<td>flatMapValues(func)</td>
<td>Apply a function that returns an iterator</td>
<td>rdd.flatMapValues(x =&gt; (x to 5))</td>
</tr>
<tr>
<td></td>
<td>RDD (map to 5)</td>
<td>{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5), (3, 6), (3, 7)}</td>
</tr>
<tr>
<td>keys()</td>
<td>Return an RDD of just the keys</td>
<td>rdd.keys()</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{1, 3, 3}</td>
</tr>
<tr>
<td>values()</td>
<td>Return an RDD of just the values</td>
<td>rdd.values()</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{2, 4, 6}</td>
</tr>
<tr>
<td>sortByKey()</td>
<td>Return an RDD sorted by the key</td>
<td>rdd.sortByKey()</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{(1, 2), (3, 4), (3, 5)}</td>
</tr>
<tr>
<td>subtractByKey(other)</td>
<td>Remove elements with a key present in the</td>
<td>rdd.subtractByKey(o)</td>
</tr>
<tr>
<td></td>
<td>other RDD</td>
<td>{(0, 0), (2, 0), (3, 0)}</td>
</tr>
<tr>
<td>join(other)</td>
<td>Perform a join where the key must be present in the other RDD</td>
<td>rdd.join(other)</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{(1, 2), (2, 0), (3, 0), (3, 0), (3, 0), (3, 0)}</td>
</tr>
<tr>
<td>rightOuterJoin()</td>
<td>Perform a join where the key must be present in the other RDD</td>
<td>rdd.rightOuterJoin(o)</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{(1, None, Some(4)), (2, Some(4)), (3, None, Some(4))}</td>
</tr>
<tr>
<td>leftOuterJoin()</td>
<td>Perform a join where the key must be present in the first RDD</td>
<td>rdd.leftOuterJoin(o)</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{(1, (Some(2), None)), (1, (None, Some(4))), (2, Some(4)), (3, None, Some(4))}</td>
</tr>
<tr>
<td>cogroup</td>
<td>Group data from both RDDs sharing the same key</td>
<td>rdd.cogroup(o)</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>{(1, (Some(2), Some(5))), (2, (Some(4), Some(9))), (3, Some(4), Some(9))}</td>
</tr>
</tbody>
</table>
Pair RDDs are still RDDs
- Supports the same functions as RDDs

```scala
pairs.filter{(key, value) => value.length < 20}
```

Filter on value using `mapValues()`
- Access only the “value” part
  ```scala```
  ```scala
  mapValues(f)
  ```
  ```scala
  same as
  ```scala
  map{(x, y) => (x, f(y))}
  ```

Aggregations with Pair RDDs
- Aggregate statistics across all elements with the same key
  ```scala```
  ```scala
  reduceByKey()
  ```
  ```scala
  similar to `reduce()`
  ```scala
  ```scala
  runs several parallel reduce operations
  ```scala
  ```scala
  one for each key in the dataset
  ```scala
  ```scala
  each operation combines values that have the same keys
  ```scala
  ```scala
  reduceByKey() is not implemented as an action that returns a value to the user program
  ```scala
  ```scala
  it returns a new RDD consisting of each key and the reduced value for that key

Example
- Per-key average with `reduceByKey()` and `mapValues()`

```scala```
```scala
rdd.mapValues(x => (x, 1)).reduceByKey(x, y => (x._1 + y._1, x._2 + y._2))
```

Word count example using `flatMap`
```scala```
```scala
wo = data.map(lambda line:line.split(" "));
```scala
```scala
wo.assemble()
```scala
```scala
[u'Colorado state university', u'Ohio state university', u'Washington state university', u'Boston university']
```scala
```scala
Using map
```scala
```scala
fm = data.flatMap(lambda line:line.split(" "));
```scala
```scala
fm.assemble()
```scala
```scala
[u'Colorado', u'state', u'university', u'Ohio', u'state', u'university', u'Washington', u'state', u'university', u'Boston', u'university']
```
Word count example using `flatMap`

```
val input = sc.textFile("s3://...");
val words = input.flatMap(x => x.split(" "));
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y);
```

**combineByKey()**

- The most general of the per-key aggregation functions
- Allows the user to return values that are not the same type as the input data
- `createCombiner()`: If `combineByKey()` finds a new key, this happens the first time a key is found in each partition, rather than only the first time the key is found in the RDD
- `mergeValue()`: If it is not a new value in that partition
- `mergeCombiners()`: Merging the results from each partition

Per-key average using `combineByKey`

```
val result = input.combineByKey((v) => (v, 1),
(acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
(acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)).map{
  case (key, value) => key -> value._1 / value._2.toFloat
}
result.collectAsMap().map(_.toString)
```
3. In-Memory Cluster Computing: Apache Spark

Key-Value pairs

Actions available on Pair RDDs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>countByKey()</td>
<td>Count the number of elements for each key</td>
<td>rdd.countByKey(1)</td>
<td>{(1,1),(7,2)}</td>
</tr>
<tr>
<td>collectAsMap()</td>
<td>Collect the result as a map to provide easy lookup</td>
<td>rdd.collectAsMap()</td>
<td>Map{(1,2),(3,4),(3,6)}</td>
</tr>
<tr>
<td>lookup(key)</td>
<td>Return all values associated with the provided key</td>
<td>rdd.lookup(3)</td>
<td>[4,6]</td>
</tr>
</tbody>
</table>

Why partitioning?

- Consider an application that keeps a large table of user information in memory
- An RDD of (UserID,UserInfo) pairs
- The application periodically combines this table with a smaller file representing events that happened in the last five minutes

Using partitionBy()

- Transforms userData to hash-partitioned RDD

PageRank

1. Initialize each page’s rank to 1.0
2. On each iteration, have page p send a contribution of rank(p)/numNeighbors(p) to its neighbors (the pages it has link to)
3. Set each page’s rank to 0.15 + 0.85 * contributionReceived
PageRank in Spark (in Scala)

```scala
// Assume that our neighbor list was saved as a Spark objectFile
val links = sc.objectFile
  (.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 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(...)
```

4. Distributed File Systems

Google File System

This material is built on:

- Andrew Fikes, Storage Architecture and Challenges, Faculty Summit, 2010
- Jeff Dean's SOCC keynote, Building Large-Scale Internet Services
- Erasure Coding: Backblaze Open sources Reed-Solomon
- An Introduction to Reed-Solomon codes

The Machinery
Google Cluster Software Environment

- Clusters contain 1000s of machines, typically one or handful of configurations
  - File system (GFS or Colossus) + cluster scheduling system are core services
- Typically 100s to 1000s of active jobs
  - mix of batch and low-latency, user-facing production jobs

MapReduce Usage statistics in Google

<table>
<thead>
<tr>
<th></th>
<th>Aug. 04</th>
<th>Mar. 06</th>
<th>Sep. 07</th>
<th>May. 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jobs</td>
<td>29K</td>
<td>171K</td>
<td>2,217K</td>
<td>4,474K</td>
</tr>
<tr>
<td>Average completion time (secs)</td>
<td>634</td>
<td>874</td>
<td>385</td>
<td>748</td>
</tr>
<tr>
<td>Machine years used</td>
<td>217</td>
<td>2,002</td>
<td>11,081</td>
<td>39,121</td>
</tr>
<tr>
<td>Input data read (TB)</td>
<td>3,388</td>
<td>52,254</td>
<td>403,152</td>
<td>946,460</td>
</tr>
<tr>
<td>Intermediate data (TB)</td>
<td>756</td>
<td>6,743</td>
<td>34,774</td>
<td>132,960</td>
</tr>
<tr>
<td>Output data written (TB)</td>
<td>193</td>
<td>2,970</td>
<td>14,018</td>
<td>45,720</td>
</tr>
<tr>
<td>Average worker machines</td>
<td>157</td>
<td>268</td>
<td>394</td>
<td>368</td>
</tr>
</tbody>
</table>

The Realistic View of a Data Center

- Typical first year for a new cluster:
  - ~1 network rewiring (rolling downtimes: ~5% of machines over 2-day span)
  - ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
  - ~3 router reloads (takes out DNS and external IPs for an hour)
  - ~dozens of minor 30-second blips for DNS
  - ~1000 individual machine failures
  - ~1000 individual machine failures
  - ~thousands of hard drive failures
  - slow disks, bad memory, misconfigured machines, flaky machines, etc.
  - Long distance links
  - Reliability/availability must come from software

Numbers we should know [1/2]

- Level 1 cache reference
  - 0.5 ns
- Branch misprediction
  - 5 ns
- Level 2 cache reference
  - 7 ns
- Mutex lock/unlock
  - 25 ns
- Main memory reference
  - 100 ns
- Compress 1KB with cheap compression algorithm
  - 3,000 ns

Numbers we should know [2/2]

- Read 1 MB sequentially from memory
  - 250,000 ns
- Round trip within the same datacenter
  - 500,000 ns
- Disk seek
  - 10,000,000 ns
- Read 1 MB sequentially from disk
  - 20,000,000 ns
- Send packet CA->Netherlands->CA
  - 150,000,000 ns

Back of the Envelope Calculation

- How long to generate an image results page (30 thumbnails)?
- Design 1: Read serially, thumbnail images (256KB) on the fly
  - 30 seek * 10 ms/seek + 30 * 256KB / 30 MB/s = 560 ms
- Design 2: Issue reads in parallel:
  - 10 ms/seek + 256KB read / 30 MB/s = 18 ms
- Lots of variations:
  - caching (single images? whole sets of thumbnails?)
  - pre-computing thumbnails
Storage Software: GFS
- Google's first cluster-level file system (2003)
  - Designed for batch applications with large files
  - Single master for metadata and chunk management
  - Chunks are typically replicated 3x for reliability

- Lessons
  - Scaled to approximately 50M files, and 10PB
  - Large files increased application complexity
  - Not appropriate for latency sensitive applications
  - Scaling limits added management overhead

Storage Software: Colossus (GFS2)
- Next-generation cluster-level file system
  - Automatically sharded metadata layer
    - Data typically written using Reed-Solomon (1.5x)
  - Client-driven replication, encoding and replication
  - Metadata space has enabled availability

- Why Reed-Solomon?
  - Cost. Especially with cross cluster replication
  - More flexible cost vs. availability choice

Storage Landscape
- Early Google:
  - US-centric traffic
    - Batch, latency-insensitive indexing processes
    - Document “snippets” serving (single seek)

- Current day:
  - World-wide traffic
    - Continuous crawl and indexing processes (Caffeine)
    - Seek-heavy, latency-sensitive apps (Gmail)
    - Person-to-person, person-to-group sharing (Docs)

Storage Landscape: Flash (SSDs)
- Important future directions:
  - More workloads that are increasingly seek heavy
    - 50-150x less expensive than disk per random read
  - Best usage is still being explored

- Concerns:
  - Availability of devices
  - 17-32x more expensive per GB than disk
  - Endurance not yet proven in the field