PART 1. BATCH COMPUTING MODELS FOR BIG DATA ANALYTICS

1. DISTRIBUTED MODEL FOR SCALABLE BATCH COMPUTING - MAPREDUCE

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

Objectives

• In-Memory Cluster Computing
  • Programming with Spark

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.

FAQs

• Questions about PA1
  - Send an email to cs535@cs.colostate.edu

• FAQ for PA1 page is available at:
  - http://www.cs.colostate.edu/~cs535/FAQ_PA1.html

Understanding closures

1: int counter = 0;
2: JavaRDD<Integer> rdd = sc.parallelize(data);
3: 4: rdd.foreach(x -> counter += x);
5: 6: println("Counter value: " + counter);

• counter (in line 4) 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?

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

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

```java
LongAccumulator accum = sc.sc().longAccumulator();
sc.parallelize(Arrays.asList(1, 2, 3, 4)).forEach(x ->
    accum.add(x));
// ...
// 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
accum.value(); // returns 10
```

Accumulators

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

```java
class VectorAccumulatorParam implements AccumulatorParam<Vector> {
    public Vector zero(Vector initialValue) {
        return Vector.zeros(initialValue.size());
    }
    public Vector addInPlace(Vector v1, Vector v2) {
        v1.addInPlace(v2); return v1;
    }
}

// Then, create an Accumulator of this type:
Accumulator<Vector> vecAccum = sc.accumulator(new Vector(...), new VectorAccumulatorParam());
```

Accumulators

- 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

LongAccumulator accum = sc.sc().longAccumulator();
data.map(x -> { accum.add(x); return f(x); });
// Here, accum is still 0 because no actions have caused the 'map' to be computed.
```
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
  - Returns key/value pairs

PairFunction

```java
PairFunction< String, String, String > keyData =
    new PairFunction< String, String, String >() {
        public Tuple2< String, String > call( String x) {
            return new Tuple2(x.split(" ") [0], x);
        }
    };

JavaPairRDD<String,String> pairs =
    lines.mapToPair(keyData);
```

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

Function | purpose | Example | Result
--- | --- | --- | ---
subtractByKey | Remove elements with a key present in the other RDD | rdd.subtractByKey(other) | {(1,2)}
join | Inner join | rdd.join(other) | {(3,4),(3,6)}
rightOuterJoin | Perform a join where the key must be present in the other RDD | rdd.rightOuterJoin(other) | {(3, some(4, 9)),(3, some(6, 9))}
leftOuterJoin | Perform a join where the key must be present in the first RDD | rdd.leftOuterJoin(other) | {{{(1,2, None)}}, (3, (4, some(9))), (3, (6, some(9)))}
cogroup | Group data from both RDDs sharing the same key | rdd.cogroup(other) | {{{{(1,2)},(3, (4,6)),(9)}}}
Pair RDDs are still RDDs
- Supports the same functions as RDDs

```java
Function< Tuple2 < String, String >, Boolean > longWordFilter =
new Function < Tuple2 < String, String >, Boolean>() {
    public Boolean call( Tuple2 < String, String > keyValue) {
        return (keyValue._2(). length() < 20);
    }
};
JavaPairRDD < String, String > result =
pairs.filter(longWordFilter);
```

Filter on Pair RDDs

Aggregations with Pair RDDs
- Aggregate statistics across all elements with the same key

```scala
reduceByKey()
- Similar to reduce()
- Takes a function and use it to combine values
- Runs several parallel reduce operations
  - One for each key in the dataset
  - Each operations combines values that have the same keys
- reduceByKey() is not implemented as an action that returns a value to
  the user program
  - It returns a new RDD consisting of each key and the reduced value for that key
```

Example
- Key-value pairs are represented using the scala.Tuple2 class

```scala
JavaRDD[String] lines = sc.textFile("data.txt");
JavaPairRDD[String, Integer] pairs =
lines.mapToPair(s -> new Tuple2(s, 1));
JavaPairRDD[String, Integer] counts =
pairs.reduceByKey((a, b) -> a + b);
```
- How many times does each line of text occur in a file?

Word count example

```scala
JavaRDD <String> input = sc.textFile("s3://...");
JavaRDD <String> words = input.flatMap(new FlatMapFunction <
String, String>() {
    public Iterable < String > call( String s) {
        return Arrays.asList (s.split( " "));
    }
});
JavaPairRDD <String, Integer> result = words
.mapToPair(new PairFunction <String, String, Integer>() {
    public Tuple2 <String, Integer> call(String x) {
        return new Tuple2(x, 1);
    }
});
.javaRDD[New Function< Integer, Integer, Integer>(),
public Integer call(Integer a, Integer b) {
    return a + b;
}];
```
combineByKey()

- The most general of the per-key aggregation functions
- Most of the other per-key combiners are implemented using it
- 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()

```java
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {
    total_ = total;
    num_ = num;
  }
  public int total() {
    return total_ / (float) num_;
  }
}
```

Function2 < Integer, AvgCount, AvgCount > combine =
new Function2 < Integer, AvgCount, AvgCount () {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total_ = b.total_;
    a.num_ = b.num_;
    return a;
  }
};

AvgCount initial = new AvgCount(0, 0);
JavaPairRDD < String, AvgCount > avgCounts =
nums.combineByKey(createAcc, addAccCount, combine);
Map < String, AvgCount > countMap = avgCounts.collectAsMap();
for (Entry < String, AvgCount > entry : countMap.entrySet()) {
  System.out.println(entry.getKey() + ', ' + entry.getValue().avg());
}
```

Tuning the level of parallelism

- When performing aggregations or grouping operations, we can ask Spark to use a specific number of partitions
- reduceByKey(( x, y) -> x + y, 10)
- repartition()
  - Shuffles the data across the network to create a new set of partitions
  - Expensive operation
  - Optimized version: coalesce()
  - Avoids data movement

joins

- Inner join
  - Only keys that are present in both pair RDDs are output
- leftOuterJoin(other) and rightOuterJoin(other)
  - One of the pair RDDs can be missing the key
- leftOuterJoin(other)
  - The resulting pair RDD has entries for each key in the source RDD
- rightOuterJoin(other)
  - The resulting pair RDD has entries for each key in the other RDD
Key-Value pairs

Actions available on Pair RDDs

### Actions 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()</td>
<td>(1,1), (3,2)</td>
</tr>
<tr>
<td>collectAsMap()</td>
<td>Collect the result as a map to provide easy lookup</td>
<td>rdd.collectAsMa(p())</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

- User data
- Event data
- Network communication

#### Using partitionBy()

- Transforms user data 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 \( \frac{\text{rank}(p)}{\text{numNeighbors}(p)} \) to its neighbors (the pages it has link to)
3. Set each page’s rank to \( 0.15 + 0.85 \times \text{contributionReceived} \)
PageRank in Spark (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 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(...)
```

Distributed File Systems

- GFS

Analytics in the Data Center

- 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

- Servers
  - CPUs
  - DRAMS
  - Disks
- Racks
  - 40-80 servers
  - Ethernet switch
- Cluster
  - >10,000 nodes

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>395</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>3,288</td>
<td>52,254</td>
<td>403,152</td>
<td>946,460</td>
</tr>
<tr>
<td>Intermediate data (TB)</td>
<td>758</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>288</td>
<td>394</td>
<td>388</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)
  - ~5 racks go wonky (40-80 machines see 50% packet loss)
  - ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
  - ~12 router reloads (takes out DNS and external IPs for a couple minutes)
  - ~3 router failures (have to immediately pull traffic for an hour)
  - ~dozens of minor 30-second blips for DNS
  - ~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 seeks * 10 ms/seek + 256K / 30 MB/s = 560 ms
  - Design 2: Issue reads in parallel:
    - 10 ms/seek + 256K 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