PART 1. BATCH COMPUTING MODELS FOR BIG DATA ANALYTICS

1. DISTRIBUTED MODEL FOR SCALABLE BATCH COMPUTING - MAPREDUCE

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

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

Objectives

- In-Memory Cluster Computing
  - Programming with Spark

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: 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 [1/4]

- 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

```
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 [2/4]

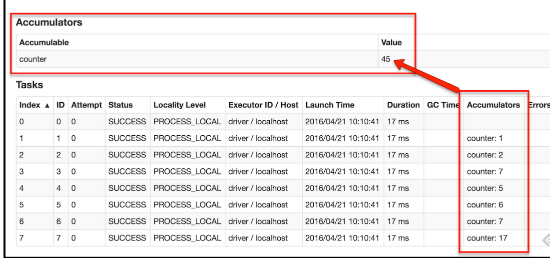
- Spark natively supports accumulators of numeric types, and programmers can add support for new types

```
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 [3/4]

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



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Accumulators [4/4]

- 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.
```


In-Memory Cluster Computing: Apache Spark

Key-Value pairs

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 < 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);
```


In-Memory Cluster Computing: Apache Spark
Key-Value pairs
Transformations on Pair RDDs

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

- Pair RDDs are allowed to use all the transformations available to standard RDDs.

Function	purpose	Example	Result
`reduceByKey()`	Combine values with the same key	`rdd.reduceByKey( (x, y) = > x + y)`	{(1,2),(3,10)}
`groupByKey()`	Group values with the same key	`rdd.groupByKey()`	{(1,[2]), (3,[4,6])}
`combineByKey( createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type		

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

Function	purpose	Example	Result
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key	`rdd.mapValues(x=> x+1)`	`{(1,3), (3,5),(3,7)}`
`flatMapValues(func)`	Apply a function that returns an iterator	`rdd.flatMapValues (x=>(x to 5)`	`{(1,2), (1,3),(1,4), (1,5),(3,4), (3,5)}`
`keys()`	Return an RDD of just the keys	`rdd.keys()`	`{1,3,3}`
`values()`	Return an RDD of just the values	`rdd.values()`	`{2,4,6}`
`sortByKey()`	Return an RDD sorted by the key	`rdd.sortByKey()`	`{(1,2), (3,4),(3,5)}`

Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)}) other={(3,9)}

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,(4,9)),(3,(6,9))}`
`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]))}`

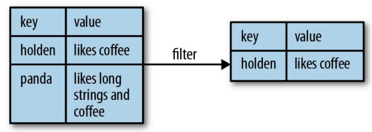
Pair RDDs are still RDDs

- Supports the same functions as RDDs

```
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



In-Memory Cluster Computing: Apache Spark

Key-Value pairs

Aggregations

Aggregations with Pair RDDs

- Aggregate statistics across all elements with the same key

- **`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

```
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

```
JavaRDD <String> input = sc.textFile("s3://...")
JavaRDD <String> words = input.flatMap(new FlatMapFunction
< String, String >() {
   public Iterable < String > call( String x) {
      return Arrays.asList( x.split(" "));
   }
});
JavaPairRDD <String,Integer> result = words
    .mapToPair(new PairFunction<String,String,Integer>() {
       public Tuple2 <String,Integer> call(String x) {
         return new Tuple2(x, 1);
      }
     })
    .reduceByKey(new Function2 <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() [1/2]

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

Function < Integer, AvgCount > createAcc = new
Function < Integer, AvgCount >() {
   public AvgCount call( Integer x) {
     return new AvgCount( x, 1); }
   };

Function2 < AvgCount, Integer, AvgCount > addAndCount = new
   Function2 < AvgCount, Integer, AvgCount >() {
      public AvgCount call( AvgCount a, Integer x) {
          a.total_ + = x;
          a.num_ + = 1;
```


Per-key average using combineByKey() [2/2]

```
       return a; }
};
Function2 < AvgCount, AvgCount, AvgCount > combine =
   new Function2 < AvgCount, 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, addAndCount, 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

groupByKey()

- Group our data using the key in our RDD
- On an RDD consisting of keys of type `K` and values of type `V`
  - Results will be RDD of type `[K, Iterable[V]]`
- `cogroup()`
  - Grouping data from multiple RDDs
  - Over two RDDs sharing the same key type, `K`, with the respective value types `V` and `W` gives us back `RDD[(K, (Iterable[V], Iterable[W]))]`

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

In-Memory Cluster Computing: Apache Spark
Key-Value pairs
Actions available on Pair RDDs

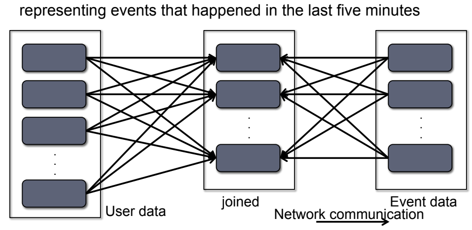
Actions on pair RDDs(example({(1,2),(3,4),(3,6)}))

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

In-Memory Cluster Computing: Apache Spark
Data Partitioning

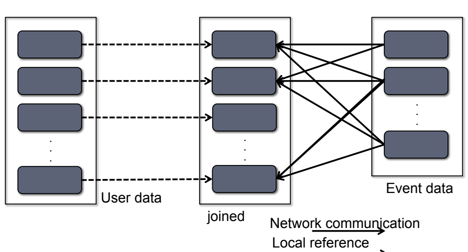
Why partitioning?

- 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 pate 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 (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
var 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

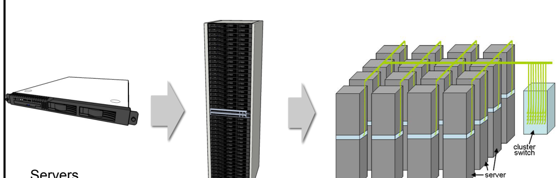
GFS

This material is built based on

- Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. Proceedings of SOSP 2003: 29-43
- Andrew Fikes, Storage Architecture and Challenges, Faculty Summit, 2010
  - http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.reverse-proxy.org/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf
- Jeff Dean’s SOCC keynote, Building Large-Scale Internet Services
  - http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf
- http://sysmagazine.com/posts/206986/
- Erasure Coding: Backblaze Open sources Reed-Solomon
  - https://www.backblaze.com/blog/reed-solomon/
- An introduction to Reed-Solomon codes
  - http://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

Analytics in the Data Center

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

	Aug. 04	Mar. 06	Sep.07	May. 10
Number of jobs	29K	171K	2,217K	4,474K
Average completion time (secs)	634	874	395	748
Machine years used	217	2,002	11,081	39,121
Input data read (TB)	3,288	52,254	403,152	946,460
Intermediate data (TB)	758	6,743	34,774	132,960
Output data written (TB)	193	2,970	14,018	45,720
Average worker machines	157	268	394	368

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
  - ~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 seeks * 10 ms/seek + 30 * 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