CS535 BIG DATA

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
- Use the posted configuration file with 2GB memory for the worker node

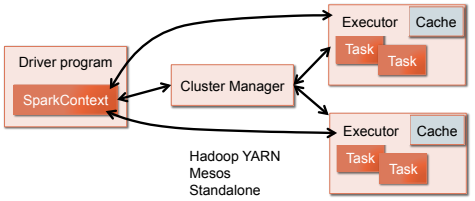
Objectives

- In-Memory Cluster Computing
  - Spark cluster
  - Scheduling
  - Programming with Spark

In-Memory Cluster Computing: Apache Spark

Spark Cluster

Spark cluster and resources



Spark cluster [1/3]

- 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 [2/3]

- 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 [3/3]

- 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

Scheduling

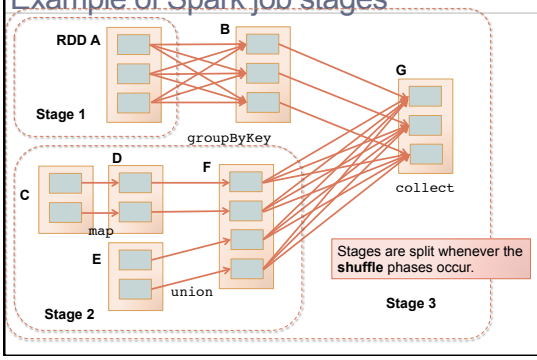
Jobs in Spark application

- "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

- 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



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

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
Programming with RDD

Linking with Spark --Java

- Spark 2.0.0 works with Java 7 and higher
  - If you are using Java 8, Spark supports lambda expressions
  - You can use the org.apache.spark.api.java.function package
- Add a dependency in Spark

```
groupID = org.apach.spark
artifactId = spark-core_2.11
Version = 2.0.0
```

- Add an HDFS cluster

```
groupID = org.apach.hadoop
artifactId = hadoop-client
Version = <your-hdfs-version>
)
```


Linking with Spark --Java

- Import some Spark classes into your program

```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf
```


Initializing Spark

- Create a JavaSparkContext object
  - Tells Spark how to access a cluster

Your application name that shows in the cluster UI

```
SparkConf conf = new SparkConf()
                    .setAppName(appName)
                    .setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

Spark, Mesos, or YARN cluster URL

Parallelized Collection

- Parallelized collections are created by calling `JavaSparkContext`'s `parallelize` method on the existing `collection` in your driver program
- The elements of the collection are copied to form a distributed dataset that can be operated on in parallel
- Creating a parallelized collection holding the numbers 1 to 5:

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
JavaRDD<Integer> distData = sc.parallelize(data, 10);
```


Simple example

```
Line 1: JavaRDD<String> lines = sc.textFile("data.txt");
Line 2: JavaRDD<Integer> lineLengths = lines.map(s ->
s.length());
Line 3: int totalLength = lineLengths.reduce((a, b) -> a +
b);
Line 4: lineLengths.persist(StorageLevel.MEMORY_ONLY());
```

- Line 1
  - `lines` is a pointer to the file
  - No loading or action is performed
- Line 2
  - Defines `lineLengths` as the result of a map transformation.
- Line 3
  - We run reduce, which is an action
  - Returns only its answer to the driver program
- Line 4
  - If we also wanted to use `lineLengths` again later

Reading files

- HDFS
  - Spark and HDFS can be collocated on the same machine
  - Spark can take advantage of this **data locality to avoid network overhead**
  - `hdfs://master:port/path`
- Amazon S3
  - Set the **`AWS_ACCESS_KEY_ID`** and **`AWS_SECRET_ACCESS_KEY`** environment variables to your S3 credential
  - Pass a path starting with `s3n://` to Spark's file input methods, of the form `s3n://bucket/path-within-bucket`
  - Wildcard paths for S3, such as `s3n:// bucket/ my-files/*.txt` is allowed
- Regular file system, SQL, Apache Hive, Cassandra, Elasticsearch
- JDBC (MySQL/Postgres)

Passing Function to Spark

- Spark's API relies heavily on passing **functions** in the driver program to run on the cluster
- In Java, functions are represented as objects that implement `org.apache.spark.api.java.function`
  - Implement the Function interfaces in your own class, either as an anonymous inner class or a named one, and pass an instance of it to Spark.
  - In Java 8, use lambda expression
- While we use lambda syntax for conciseness, it is easy to use all the same APIs in long-form

Function Classes

Name	Method to implement	Usage
`Function<T, R>`	`R call(T)`	`One input/one output map(), filter()`
`Function2<T1, T2, R>`	`R call(T1, T2)`	`Two input/one output aggregate(), fold()`
`FlatMapFunction<T, R>`	`Iterable<R> call(T)`	`One input / 0 or more outputs flatMap()`

Passing Function to Spark

```
JavaRDD<String> lines = sc.textFile("data.txt");

JavaRDD<Integer> lineLengths = lines.map(new
Function<String, Integer>() {
  public Integer call(String s) {
      return s.length();
}});

int totalLength = lineLengths.reduce(new Function2<Integer,
Integer, Integer>() {
  public Integer call(Integer a, Integer b) {
      return a + b;
}});
```

http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/function/package-summary.html

Passing Function to Spark

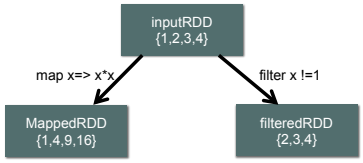
- OR

```
class GetLength implements Function<String, Integer> {
  public Integer call(String s) { return s.length(); }
}

class Sum implements Function2<Integer, Integer, Integer> {
  public Integer call(Integer a, Integer b) { return a +
b; }
}

JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new GetLength());
int totalLength = lineLengths.reduce(new Sum());
```


In-Memory Cluster Computing: Apache Spark
Transformations

`map()` vs. `filter()` [1/2]

- The **`map()`** transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD
- The **`filter()`** transformation takes in a function and returns an RDD that only has elements that pass the filter() function



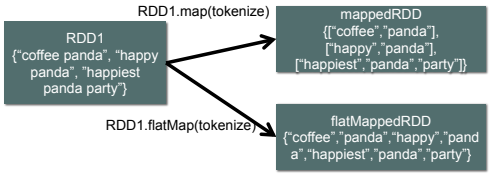
map() vs. filter() [2/2]

- map() that squares all of the numbers in an RDD

```
JavaRDD <Integer> rdd = sc.parallelize(Arrays.asList(1, 2,
3, 4));
JavaRDD <Integer> result = rdd.map(new Function < Integer,
Integer >() {
   public Integer call(Integer x) {
         return x*x;
   }
});


System.out.println(StringUtils.join(result.collect(),","));
```


map() vs. flatMap() [1/2]

- As results of **flatMap()**, we have an RDD of the elements
  - Instead of RDD of lists of elements



map() vs. flatMap() [2/2]

- Using flatMap() that splits lines to multiple words

```
JavaRDD < String > lines = sc.parallelize( Arrays.asList("
hello world", "hi"));

JavaRDD < String > words =
  lines.flatMap(new FlatMapFunction < String, String >() {
     public Iterable < String > call( String line) {
       return Arrays.asList(line.split(" "));
     }
  });

words.first();
// returns "hello"
```


Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

name	purpose	results
union()	union	{1.2.3.3.4.5}
intersection()	intersection	{3}
subtract()	Remove the contents of one RDD (e.g. remove training data)	{1,2}
cartesian()	Cartesian product	{(1,3),(1,4),(1,5), (2,3), (2,4), (2,5),(3,4),(3,5), (3,6)}

In-Memory Cluster Computing: Apache Spark
Actions

reduce()

- Takes a function that operates on two elements of the type in your RDD and returns a new element of the same type

- fold()
  - Same as reduce() but with the provided zero value of the type we want to return

```
Integer sum = rdd.reduce(new Function2 < Integer, Integer,
Integer >() {
      public Integer call(Integer x, Integer y) {
          return x + y;
} });
```

aggregate()

- With `aggregate()`, like `fold()`, we supply an initial zero value of the type we want to return

```
class AvgCount implements Serializable {
   public AvgCount( int total, int num) {
      this.total = total; this.num = num; }

   public int total;
   public int num;
   public double avg() {
     return total / (double) num;
   }
}
```

*Code will be continued in the next slide

aggregate() -continued

```
Function2 < AvgCount, Integer, AvgCount > addAndCount =
  new Function2 < AvgCount, Integer, AvgCount >() {
    public AvgCount call( AvgCount a, Integer x) {
       a.total + = x;
       a.num + = 1;
       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);
AvgCount result = rdd.aggregate(initial, addAndCount,
combine);
System.out.println(result.avg());
```


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

Persistence

Persistence levels

level	Space used	CPU time	In memory/ On disk	Comment
`MEMORY_ONLY`	High	Low	Y/N	
`MEMORY_ONLY_SER`	Low	High	Y/N	Store RDD as *serialized* Java objects (one byte array per partition).
`MEMORY_AND_DISK`	High	Medium	Some/Some	Spills to disk if there is too much data to fit in memory
`MEMORY_AND_DISK_SER`	Low	High	Some/Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory
`DISK_ONLY`	Low	High	N/Y	