PART 1. LARGE SCALE DATA ANALYTICS
IN-MEMORY CLUSTER COMPUTING
Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

FAQs
Today’s topics

• In-Memory cluster computing
  • Apache Spark

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
RDD Dependency in Spark
Dependency between RDDs

- **Narrow** dependency
- **Wide** dependency
Dependency between RDDs [3/4]

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

![Diagram](groupId)

Join with inputs not co-partitioned

Dependency between RDDs [4/4]

- **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
Interface used to represent RDDs in Spark

• **partitions()**
  - Returns a list of partition objects

• **preferredLocations(p)**
  - List nodes where partition p can be accessed faster due to data locality

• **dependencies()**
  - Return a list of dependencies

• **iterator(p, parentIters)**
  - Compute the elements of partition p given iterators for its parent partitions

• **partitioner()**
  - Return metadata specifying whether the RDD is hash/range partitioned

---

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
Spark Cluster
Spark cluster and resources

- 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 even 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

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
Large Scale Data Analytics
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

Stages are split whenever the shuffle phases occur.
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

Large Scale Data Analytics
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.

```java
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?

- 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
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

```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 [3/4]

• If accumulators are created **with a name**, they will be displayed in Spark's UI
Accumulators

- **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
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.
```

Large Scale Data Analytics

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 (using the first word as the key)**

• **Running map() function**
  - A transformation that passes each dataset element through a function and returns a new RDD representing the results
  - Returns **key/value pairs**

```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);
```

**PairFunction**: A function that returns key-value pairs (Tuple2<K, V>), and can be used to construct PairRDDs

**Tuple2**: A function that creates a tuple
Large Scale Data Analytics

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

<table>
<thead>
<tr>
<th>Function</th>
<th>purpose</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduceByKey()</td>
<td>Combine values with the same key</td>
<td>rdd.reduceByKey((x, y) = x + y)</td>
<td>{(1,2),(3,10)}</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>Group values with the same key</td>
<td>rdd.groupByKey()</td>
<td>{(1,[2]),(3,[4,6])}</td>
</tr>
<tr>
<td>combineByKey(...)</td>
<td>Combine values with the same key using a different result type</td>
<td>Slides: 44</td>
<td></td>
</tr>
</tbody>
</table>
Transformations on one pair RDD (example: \{(1, 2), (3, 4), (3, 6)\})

<table>
<thead>
<tr>
<th>Function</th>
<th>purpose</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapValues(func)</td>
<td>Apply a function to each value of a pair RDD without changing the key</td>
<td>rdd.mapValues(x=&gt;x+1)</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>
<td>{(1,2), (1,3), (1,4), (1,5), (3,4), (3,5)}</td>
</tr>
<tr>
<td>keys()</td>
<td>Return an RDD of just the keys</td>
<td>rdd.keys()</td>
<td>{1, 3, 3}</td>
</tr>
<tr>
<td>values()</td>
<td>Return an RDD of just the values</td>
<td>rdd.values()</td>
<td>{2, 4, 6}</td>
</tr>
<tr>
<td>sortByKey()</td>
<td>Return an RDD sorted by the key</td>
<td>rdd.sortByKey()</td>
<td>{(1,2), (3,4), (3,5)}</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>purpose</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtractByKey</td>
<td>Remove elements with a key present in the other RDD</td>
<td>rdd.subtractByKey(other)</td>
<td>{(1,2)}</td>
</tr>
<tr>
<td>join</td>
<td>Inner join</td>
<td>rdd.join(other)</td>
<td>{(3,(4,9)), (3,(6,9))}</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td>Perform a join where the key must be present in the other RDD</td>
<td>rdd.rightOuterJoin(other)</td>
<td>{(3,(Some(4),9)), (3, (Some(6),9))}</td>
</tr>
<tr>
<td>leftOuterJoin()</td>
<td>Perform a join where the key must be present in the first RDD</td>
<td>rdd.leftOuterJoin(other)</td>
<td>{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}</td>
</tr>
<tr>
<td>coGroup</td>
<td>Group data from both RDDs sharing the same key</td>
<td>rdd.cogroup(other)</td>
<td>{((1,[]), (3,[[4,6],[9]]))}</td>
</tr>
</tbody>
</table>
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
Large Scale Data Analytics
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 operation combines values that have the same keys

  - `reduceByKey()` is **not implemented as an action** that returns a value to the user program
    - There can be a large number of keys
    - It **returns a new RDD** consisting of each key and the reduced value for that key
      - Therefore this is a transformation
Example

• Key-value pairs are represented using the Tuple2 class

```java
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);
```

Word count example

```java
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()`  

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

Function2 `<AvgCount, Integer, AvgCount>` **addAndCount** = new
```java
Function2 < AvgCount, Integer, AvgCount >() {
    public AvgCount call(AvgCount a, Integer x) {
        a.total_ += x;
        a.num_ += 1;
    }
};
```
Per-key average using `combineByKey()` [2/2]

```java
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()`
    - Reduces data movement
    - A subset of current nodes will be kept
**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 \([K, \text{Iterable}[V]]\)

**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\([(K, (\text{Iterable}[V], \text{Iterable}[W]))]\)

**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
Large Scale Data Analytics
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()</td>
<td>(1,1), (3,2)</td>
</tr>
<tr>
<td>collectAsMap()</td>
<td>Collect the result as a map to provide easy lookup at the driver</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>
Large Scale Data Analytics

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

![Diagram](image)

Questions?