FAQs

• Term project
  • 5:00PM March 29, 2018

• Programming Assignment 3 has been posted
  • We will discuss the assignment on 4/5 in class

• Recitations
  • Apache Spark tutorial 1 and 2
    • 3/30, and 4/6

Today’s topics

• In-Memory cluster computing
  • Apache Spark

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 (directed acyclic graph) 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 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
- 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

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: rdd.foreach(x -> counter += x);
4: 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));
// returns 10
```

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

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

Accumulators

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

```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.
```
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 re-group 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);

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, 3), (3, 10)</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>Group values with the same key</td>
<td>rdd.groupByKey()</td>
<td>(1, [2], [3, 10])</td>
</tr>
<tr>
<td>combineByKey()</td>
<td>Combine values with the same key using a different result type</td>
<td>rdd.combineByKey(dict, mergeValue, mergeCombiners, partitioner)</td>
<td>(1, [2, [3, 10]])</td>
</tr>
<tr>
<td>mapValues()</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()</td>
<td>Apply a function that returns an iterator</td>
<td>rdd.flatMapValues(x =&gt; x.to(5))</td>
<td>(1, [2, 3, 4, 5])</td>
</tr>
<tr>
<td>keys()</td>
<td>Return an RDD of just the keys</td>
<td>rdd.keys()</td>
<td>(1, 3, 4)</td>
</tr>
<tr>
<td>values()</td>
<td>Return an RDD of just the values</td>
<td>rdd.values()</td>
<td>(2, 3, 5)</td>
</tr>
<tr>
<td>sortByKey()</td>
<td>Return an RDD sorted by the key</td>
<td>rdd.sortByKey()</td>
<td>(1, 2, 3, 4, 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>(\text{rdd.subtractByKey(ot)})</td>
<td>((1,2))</td>
</tr>
<tr>
<td>join</td>
<td>Inner join</td>
<td>(\text{rdd.join(ot)})</td>
<td>((3,{4,8}), (3,{6,8}))</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td>Perform a join where the key must be present in the other RDD</td>
<td>(\text{rdd.rightOuterJoin(ot)})</td>
<td>((3,{4,8}), (3,{6,9}))</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>Perform a join where the key must be present in the first RDD</td>
<td>(\text{rdd.leftOuterJoin(ot)})</td>
<td>((1,{2,\text{None}}), (2,{4,\text{Some(9)}}), (3,{6,\text{Some(9)}}))</td>
</tr>
<tr>
<td>cogroup</td>
<td>Group data from both RDDs sharing the same key</td>
<td>(\text{rdd.cogroup(ot)})</td>
<td>((1,{2},{})), (3,{4,6},{9}))</td>
</tr>
</tbody>
</table>

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

Aggregations with Pair RDDs

- Aggregate statistics across all elements with the same key
  - `reduceByKey()`
    - Similar to `reduce()`
    - Takes a function and uses 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
  - `reduceByValue()` 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.
  - `reduceByKey()` is a transformation

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
Key-Value pairs : Aggregations

Example

- Key-value pairs are represented using the `Tuple2` class

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs =
    lines.mapToPair(x -> new Tuple2(x, 1));
JavaPairRDD<String, Integer> counts =
    pairs.reduceByKey((a, b) -> a + b);
```
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

---

Word count example

```java
JavaPairRDD<String, Integer> result = words
  .mapToPair(new Function2<String, Integer>() { public Integer call(String x) { return x.length(); } })
  .groupByKey() //<k, list<V>>
  .mapValues(new Function<List<Integer>, Integer>() { public Integer call(List<Integer> v) { return v.stream().mapToInt(Integer::intValue).sum(); } })
  .sortByKey();
```

---

combineByKey() [1/2]

- The most general of the per-key aggregation functions
  - Most of the other per-key combinators 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

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

  public int addAndCount(int value) {
    this.total_ += value;
    this.count_++;
    return this.total_ / this.count_;  
  }
}
```

---

groupByKey() [2/2]

- 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 of type `<K, Iterable<V>, Iterable<W>>`

```
JavaPairRDD<K, V> left = ...;
JavaPairRDD<K, W> right = ...;
JavaPairRDD<K, Tuple2<Iterable<V>, Iterable<W>>> cogrouped = left.cogroup(right);
```
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

Actions on pair RDDs(example([[(1,2),(3,4),(3,6)]]))

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