CS 455: INTRODUCTION TO DISTRIBUTED SYSTEMS [SPARK]

Shrideep Pallickara
Computer Science
Colorado State University

March 28, 2017

Topics covered in this lecture

- Resilient Distributed Datasets
- Common Transformations and Actions

RESILIENT DISTRIBUTED DATASET [RDD]

Every Spark program and shell works as follows

1. **Create** some input RDD from external data
2. **Transform** them to define new RDDs using transformations like `filter()`
3. **Ask Spark to persist()** any intermediate RDDs that needs to be reused
4. **Launch actions** such as `count()`, etc. to kickoff a parallel computation
   - Computing is optimized and executed by Spark

A CLOSER LOOK AT RDD OPERATIONS

Frequently asked questions from the previous class survey

- Other DFS that Spark can run on?
- Using Spark for ML vs GPUs?
- How is this faster, if you are doing the same I/O?
  - Avoid repeated sweeps of the data
  - Pin only relevant portions in memory
- Driver program, Context, Executors

Computing is optimized and executed by Spark
RDDs support two types of operations

- **Transformations**
  - Operations that **return a new RDD**. E.g.: `filter()`

- **Actions**
  - Operations that **return a result** to the driver program or write to storage
  - Kicks off a computation. E.g.: `count()`

- **Distinguishing aspect?**
  - Transformations return RDDs
  - Actions return some other data type

In our previous example...

- **filter** does not **mutate** `inputRDD`
  - Returns a pointer to an entirely new RDD
  - `inputRDD` can still be reused later in the program

- We could use `inputRDD` to search for lines with the word “warning”
  - While we are at it, we will use another transformation, `union()`, to print number of lines that contained either
    - `errorsRDD = inputRDD.filter(lambda x: "error" in x)`
    - `warningsRDD = inputRDD.filter(lambda x: "warning" in x)`
    - `badlinesRDD = errorsRDD.union(warningsRDD)`

RDD Lineage graphs

- As new RDDs are derived from each other using transformations, Spark tracks dependencies
  - **Lineage graph**

- Uses lineage graph to
  - Compute each RDD on demand
  - Recover lost data if part of persistent RDD is lost

Transformations

- Many transformations are **element-wise**
  - Work on only one element at a time

- Some transformations are not element-wise
  - E.g.: We have a logfile, `log.txt`, with several messages, but we only want to select error messages
    - `inputRDD = sc.textFile("log.txt")`
    - `errorsRDD = inputRDD.filter(lambda x: "error" in x)`

In our previous example

- Note how `union()` is different from `filter()`
  - Operates on 2 RDDs instead of one

- Transformations can actually operate on **any number** of RDDs

RDD lineage graph for our example

```
inputRDD
  filter
  errorsRDD
  warningsRDD
  union
  badLinesRDD
```

Actions

- We can create RDDs from each other using transformations
- At some point, we need to actually do something with the dataset
  - Actions
- Forces evaluations of the transformations required for the RDD they were called on

Let’s try to print information about badlinesRDD

```
print "Input had " + badLinesRDD.count() + " concerning lines"
for line in badLinesRDD.take(10):
    print line
```

RDDS also have a collect to retrieve the entire RDD

- Useful if program filters RDD to a very small size and you want to deal locally
  - Your entire dataset must fit in memory on a single machine to use collect() on it
    - Should NOT be used on large datasets
- In most cases, RDDs cannot be collect()ed to the driver
- Common to write data out to a distributed storage system ... HDFS or S3

Lazy Evaluation

- Transformations on RDDs are lazily evaluated
  - Spark will not begin to execute until it sees an action
- Uses this to reduce the number of passes it has to take over data by grouping operations together
- What does this mean?
  - When you call a transformation on an RDD (for e.g. map) the operation is not immediately performed
  - Spark internally records metadata that operation is requested

How you should think of RDDs

- Rather than thinking of it as containing specific data
  - Best to think of it as containing instructions on how to compute the data that we build through transformations
- Loading data into a RDD is lazily evaluated just as transformations are

COMMON TRANSFORMATIONS AND ACTIONS
**Element-wise transformations: filter()**

- Takes in a function and returns an RDD that only has elements that pass the `filter()` function.

**Element-wise transformations: map()**

- Takes in a function and applies it to each element in the RDD.
- Result of the function is the new value of each element in the resulting RDD.

**Things that can be done with map()**

- Fetch website associated with each URL in collection to just squaring numbers.
- `map()`’s return type does not have to be the same as its input type.
- Multiple output elements for each input element?
  - Use `flatMap()`

**Difference between map and flatMap**

- `RDD1.map(tokenize)`
- `RDD1.flatMap(tokenize)`

**Psuedo set operations**

- RDDs support many of the operations of mathematical sets such as union, intersection, etc.
- Even when the RDDs themselves are not properly sets.

**Some simple set operations**

- `RDD1.distinct()`: `RDD1.union(RDD2)`
- `RDD1.intersection(RDD2)`
- `RDD1.subtract(RDD2)`: `RDD1.intersect(RDD2)`
Cartesian product between two RDDs

```
RDD1.cartesian(RDD2) = 
{(User1, Venue("Betabrand")),
 (User1, Venue("Asha Tree House")),
 (User1, Venue("Ritual")),
 (User2, Venue("Betabrand")),
 (User2, Venue("Asha Tree House")),
 (User2, Venue("Ritual")),
 (User3, Venue("Betabrand")),
 (User3, Venue("Asha Tree House")),
 (User3, Venue("Ritual"))}
```

**COMMON ACTIONS**

- **reduce()**
  - Takes a function that operates on two elements in the RDD; returns an element of the same type
  - E.g. of such an operation: `+` sums the RDD
    
    ```
    sum = rdd.reduce((x,y) => x + y)
    ```

- **fold()** takes a function with the same signature as `reduce()`, but also takes a "zero value" for initial call
  - "Zero value" is the identity element for initial call
  - E.g., `0` for `+`, `1` for `*`, empty list for concatenation

Both `fold()` and `reduce()` require return type of same type as the RDD elements

**EXAMPLES: BASIC ACTIONS ON RDDs**

- **collect()**
  - Return all elements from the RDD
  - Invocation: `rdd.collect()`
  - Result: `{1, 2, 3, 3}`
Examples: Basic actions on RDDs [2/7]

- **count()**
  - Number of elements in the RDD
  - Invocation: `rdd.count()`
  - Result: 4

Examples: Basic actions on RDDs [3/7]

- **countByValue()**
  - Number of times each element occurs in the RDD
  - Invocation: `rdd.countByValue()`
  - Result: `{ (1,1), (2,1), (3,2) }`

Examples: Basic actions on RDDs [4/7]

- **take(num)**
  - Return `num` elements from the RDD
  - Invocation: `rdd.take(2)`
  - Result: `{1, 2}`

Examples: Basic actions on RDDs [5/7]

- **reduce(func)**
  - Combine the elements of the RDD together in parallel
  - Invocation: `rdd.reduce( (x,y) => x + y )`
  - Result: 9

Examples: Basic actions on RDDs [6/7]

- **aggregate(zeroValue)(seqOp, combOp)**
  - Similar to reduce() but used to return a different type
  - Invocation: `rdd.aggregate((0,0)) ((x,y) => (x._1 + y, x._2 + 1), (x,y) => (x._1 + y._1, x._2 + y._2))`
  - Result: (9, 4)

Examples: Basic actions on RDDs [7/7]

- **foreach(func)**
  - Apply the provided function to each element of the RDD
  - Invocation: `rdd.foreach(func)`
  - Result: Nothing
Why persistence?

- Spark RDDs are lazily evaluated, and we may sometimes wish to use the same RDD multiple times.
- Naively, Spark will recompute RDD and all of its dependencies each time we call an action on the RDD. Super expensive for iterative algorithms.
- To avoid recomputing RDD multiple times?
  - Ask Spark to persist the data.
  - The nodes that compute the RDD, store the partitions.
  - E.g.: `result.persist(StorageLevel.DISK_ONLY)`

Coping with failures

- If a node that has data persisted on it fails?
  - Spark recomputes lost partitions of data when needed.
- Also, replicate data on multiple nodes.
  - To handle node failures without slowdowns.

Persistence Levels for Spark

<table>
<thead>
<tr>
<th>Level</th>
<th>Space Used</th>
<th>CPU time</th>
<th>In Memory</th>
<th>On disk</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>High</td>
<td>Low</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Low</td>
<td>High</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>High</td>
<td>Medium</td>
<td>Some</td>
<td>Some</td>
<td>Spills to disk if there is too much data to fit in memory.</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>Low</td>
<td>High</td>
<td>Some</td>
<td>Some</td>
<td>Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.</td>
</tr>
<tr>
<td>DISKONLY</td>
<td>Low</td>
<td>High</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

What if you attempt to cache too much data that does not fit in memory?

- Spark will evict old partitions using a Least Recently Used Cache policy.
  - For memory only storage partitions, it will be recomputed the next time they are accessed.
  - For memory_and_disk ones? Write them out to disk.
- RDDs also come with a method, unpersist().
  - Manually remove data elements from the cache.

Working with Key/Value Pairs
RDDs of key/value pairs

- Key/value RDDs are commonly used to perform aggregations
- Might have to do ETL (Extract, Transform, and Load) to get data into key/value formats
- Advanced feature to control layout of pair RDDs across nodes
  - Partitioning

RDDs containing key/value pairs

- Are called pair RDDs
- Useful building block in many programs
  - Expose operations that allow actions on each key in parallel or regroup data across network
  - reduceByKey() to aggregate data separately for each key
  - join() to merge two RDDs together by grouping elements of the same key

Pair RDDs

- RDDs that contain key/value pairs
- Expose partitions that allow you to act on each key in parallel or regroup data across the network

Creating Pair RDDs

- `pairs=lines.map(lambda x: (x.split(" "))[0], x)`
  - Creates a pairRDD using the first word as the key
- Java does not have a built-in tuple type
  - `scala.Tuple2` class
    - `new Tuple2(elem1, elem2)`

The contents of this slide-set are based on the following references