Accumulators & Broadcast Variables

Databricks Training
July 2015
Cluster Side Read and Write

• Imagine a Spark Function which needs to write data or read data for its work (on the cluster side).
• Remember a Function is instantiated on each Worker:
  • It resides next to the Partition with which it works…
  • If it writes, to where is it writing exactly?
  • If it reads, from where is reading?
Cluster Side Read and Write

• What are the major issues?
  • Each worker node can know:
    - How to write to a variable (or object) …
    • … but that object is not propagated around the nodes.
    • And how are they synchronized?
    - How to read from a variable (or object) …
    • … but that data is not aggregated from the other nodes.
    - And all that would be local to a single node, right... Hmmm?

• For this we have Accumulators & Broadcast Variables
Accumulator (1 of 2)

The **Accumulator** is a class which allows multiple workers to write to a single location for aggregation at the driver later:

- Use a Scala or Java implementation: `org.apache.spark.Accumulator<T>`
- Or write a custom accumulator:
  - implement `AccumulatorParam<T>`
- In Python use: `Accumulator[T]`
Accumulator (2 of 2)

• One Accumulator is instantiated for your Function on each worker node alongside a Partition.\(^1\)

• Your Function will:
  - work with Accumulator directly.\(^2\)
  - Add to the Accumulator value (think “write only”).\(^3\)

• Pass your Function to an appropriate action.

• Driver will read final result from Accumulator (representing all workers).

• Let’s see …
Sample **Accumulator :: Scala**

```scala
val file = sc.textFile("someBigFile.txt")
val blankLines = sc.accumulator(0)
val allLines = file.map(line => {
    if (line == "") {
        blankLines += 1
        // For debugging only
        // Found one so add it
    }
    line // Just return the line
})

allLines.count // Everything above is lazy
println("Number of blank lines: " + blankLines.value)
```

SparkContext Factory Method
Sample **Accumulator** :: Python

```python
file = sc.textFile("someBigFile.txt")
blankLines = sc.accumulator(0)

def showAccumulation(line):
    global blankLines  # use outer blankLines above
    if (line == ""):
        blankLines += 1  # for debug only
    return line

allLines = file.map(showAccumulation)
allLines.count()
print "Blank lines: \%d" \% blankLines.value
```
Beyond Blank Lines – DNA Data

• What if we are working with “real data” such as DNA Nucleotide Pairs?
  • For example, an incoming dataset of a stream of Fruit Fly base-pair data:
    “AAAACCACAACCTATTTTTAAAGTGGACGTTATAAAATTGAATTTTTTTGCTG”
  • 50 characters on each line, many-many lines…
  • Each character is only one side of the DNA helix:
    - A becomes AT
    - C becomes CG
    - G becomes GC
    - T becomes TA, because of rules, etc.¹
Sample Accumulator :: Java

SparkExample sw = new SparkExample(); // Encapsulate
// Load DNA dataset of nucleotide pairs as lines
JavaRDD<String> DNARDD1 = sw.loadData(DNApath1);
Accumulator<Integer> taAccumulator =
    sw.getContext().accumulator(0);
// Nucleotide pairs (K,V) comes out of lines
JavaRDD<Tuple2<String, String>> pairs =
    DNARDD1.flatMap(new DNAFlatMapFunction("na_armX.dmel.RELEASE5"));

// Pass Accumulator into Function, Function into action
pairs.foreach(new AccumulatorFunction(taAccumulator));

System.out.println("\n*** TA Accumulator :: Found: " +
    taAccumulator.value());
public class AccumulatorFunction implements VoidFunction<Tuple2<String, String>> {
    // Count the number of TA pairs
    private Accumulator taAccumulator = null;

    public AccumulatorFunction(Accumulator taAccumulator) {
        this.taAccumulator = taAccumulator;
    }

    // Key is String 1, Nucleotide pair value is String 2
    public void call(Tuple2<String, String> pair) throws Exception {
        String value = pair._2;
        if (value.equals("TA")) {
            taAccumulator.add(1);
        }
    }
}
Broadcast Variables (1 of 2)

• Understand what Spark does normally:
  • Sends *all Function (closure) variables* to the worker nodes for their use.
  • *Resends* the same variable(s) if used again in another operation (*grossly inefficient*).

• Broadcast Variables are sent:
  • To the cluster *only once (shared)*.
  • Using a *different protocol*.
  • Much *more efficiently*.
  • Big win! Let’s look more closely…
Broadcast Variables (2 of 2)

• Broadcast Variables:
  • Are broadcast(...) by the SparkContext.
  • Must be Serializable.¹

• Are “read only” because:
  - Writes (updates) are local to the worker node (never propagated around cluster).
  - Attempts to change the value work, but only locally (don’t do it)

• After broadcast takes place:
  - pass the broadcast variable to your Function

• Let’s look at the code...
theVar = sc.broadcast("... is Amazing!"")

namesRDD = sc.parallelize(['Andy', 'Rachel', 'Sameer', 'Donnita', 'Vanessa', 'Heinrich', 'Petra', 'Don', 'Lorenzo', 'Maria'])

def showBroadcast(name):
    return name + theVar.value

someNames = namesRDD.filter(lambda name: 'Don' in name)
print someNames.map(showBroadcast).collect()
public Broadcast<String> showBroadcast(Serializable bv) {

    JavaRDD<String> wordsRDD =
        sc.parallelize(Arrays.asList("Andy", "Mehera", "Sameer", "Don", "Vanessa", "Donnita"));

    String bvs = (String) bv; //What we are broadcasting

    //How we broadcast the variable to the Cluster
    final Broadcast<String> theVar =
        sc.broadcast(bvs);

    wordsRDD.foreach( new ConcatFunction(theVar));

    return theVar;
}
public class ConcatFunction implements VoidFunction<String> {

    private Broadcast<String> theVar = null;

    public ConcatFunction(final Broadcast<String> var) {
        theVar = var;
    }

    public void call(String word) throws Exception {
        System.out.println("\n*** Show Broadcast: " +
                         (word + " : " + theVar.value()));
    }
}

Read-only At worker
Thank you.

JavaRDD&lt;String&gt; answersRDD =
sc.parallelize(Arrays.asList("Any", "Questions?"));