FAQs

• Term project: Proposal
  • 5:00PM October 23, 2018

Today's topics

• In-Memory cluster computing
  • Apache Spark

This material is built based on

• Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphhe McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)

• Holden Karau, Andy Komwinski, Patrick Wendell and Matei Zaharia, "Learning Spark", O'Reilly, 2015

• Spark Overview, https://spark.apache.org/docs/2.3.2/
• Spark programming guide
  • Job Scheduling
    • https://spark.apache.org/docs/2.3.2-preview/jvm-scheduling.html
Distributed processing with the Spark framework

- **Cluster Computing**
  - Spark standalone
  - YARN
  - Mesos
- **Storage**
  - HDFS/file system/
  - HBase/Cassandra, etc.

Inefficiencies for emerging applications:
(1) Data reuse
- Data reuse is common in many iterative machine learning and graph algorithms
  - PageRank, K-means clustering, and logistic regression

Inefficiencies for emerging applications:
(2) Interactive data analytics
- User runs multiple ad-hoc queries on the same subset of the data

Existing approaches
- **Hadoop**
  - Writing output to an external stable storage system
    - e.g. HDFS
  - Substantial overheads due to data replication, disk I/O, and serialization
- **Pregel**
  - Iterative graph computations
- **HaLoop**
  - Iterative MapReduce interface
- **Pregel/HaLoop** support specific computation patterns
  - e.g. looping a series of MapReduce steps

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
RDD (Resilient Distributed Dataset)
RDD (Resilient Distributed Dataset)

- Read-only, memory resident partitioned collection of records
  - A fault-tolerant collection of elements that can be operated on in parallel

- RDDs are the core unit of data in Spark
  - Most Spark programming involves performing operations on RDDs

Word Count Example

We use a few transformations to build a dataset of (String, Int) pairs called counts and then save it to a file.

```java
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts =
    textFile.flatMap(s -> Arrays.asList(s.split(" "))).iterator()
        .mapToPair(word -> new Tuple2<>(word, 1))
        .reduceByKey((a, b) -> a + b);

counts.saveAsTextFile("hdfs://...");
```

Overview of RDD

- **Lineage**
  - How it was derived from other dataset to compute its partitions from data in stable storage?
  - RDDs do not need to be materialized at all times

- **Persistence**
  - Users can indicate which RDDs they will reuse and the storage strategy

- **Partitioning**
  - Users can specify the partitioning method across machines based on a key in each record

Spark Programming Interface to RDD:

**Transformation** [1/3]

- "transformations"
  - Operations that create RDDs
    - Return pointers to new RDDs
      - e.g., map, filter, and join
  - RDDs can only be created through deterministic operations on either
    - Data in stable storage
    - Other RDDs

**Action** [2/3]

- "actions"
  - Operations that return a value to the application or export data to a storage system
    - e.g., count: returns the number of elements in the dataset
    - e.g., collect: returns the elements themselves
    - e.g., save: outputs the dataset to a storage system

**Persist** [3/3]

- "persist"
  - Indicates which RDDs they want to reuse in future operations
  - Spark keeps persistent RDDs in memory by default
  - If there is not enough RAM
    - It can spill them to disk
  - Users are allowed to:
    - Store the RDD only on disk
    - Replicate the RDD across machines
    - Specify a persistence priority on each RDD
Revisit: Word Count Example

we use a few transformations to build a dataset of (String, Int) pairs called counts and then save it to a file

```java
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Int> counts = textFile.flatMap(s -> Arrays.asList(s.split(" "))).iterator()
.reduceByKey((a, b) -> a + b);

counts.saveAsTextFile("hdfs://...");
```

Lazy Evaluation

- Transformations on RDDs are **lazy evaluated**
  - Spark will NOT begin to execute until it sees an action
  - Spark internally records metadata to indicate that this operation has been requested
- Loading data from files into an RDD is lazily evaluated
  - Reduces the number of passes it has to take over our data by grouping operations together

Example: Console Log Mining

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop file system (HDFS) to find the cause
- The user loads the error messages from the logs into the RAM across a set of nodes and query them interactively

```scala
lines = spark.textFile("hdfs://...");
errors = lines.filter(_.startsWith("ERROR"));
errors.persist();
errors.filter(_.contains("HDFS")).map(_.split(" ")(3)).collect();
```

Benefits of RDDs as a distributed memory abstraction

- RDDs can only be created ("written") through **coarse-grained transformations**
  - Distributed shared memory (DSM) allows reads and writes to each memory location
  - Reads on RDDs can still be fine-grained
    - A large read-only lookup table
  - Applications perform bulk writes
    - More efficient fault tolerance
      - Lineage based bulk recovery
Benefits of RDDs as a distributed memory abstraction [2/3]

- Advantage of using RDDs’ immutable data
  - System can mitigate slow nodes (Stragglers)
    - Creates backup copies of slow tasks
    - Without accessing the same memory
  - Spark distributes the data over different working nodes that run computations in parallel
  - Orchestrates communicating between nodes to integrate intermediate results and combine them for the final result

Benefits of RDDs as a distributed memory abstraction [3/3]

- Runtime can schedule tasks based on data locality
  - To improve performance
- RDDs degrade gracefully when there is insufficient memory
  - Partitions that do not fit in the RAM are stored on disk

Applications not suitable for RDDs

- RDDs are best suited for batch applications that apply the same operations to all elements of a dataset
- Steps are managed by lineage graph efficiently
- Recovery is managed effectively
- RDDs would not be suitable for applications
  - Making asynchronous fine-grained updates to shared state
  - E.g. a storage system for a web application or an incremental web crawler

Large Scale Data Analytics

In-Memory Cluster Computing: Apache Spark

Key-Value pairs: Transformations on Pair RDDs

Transformations on one pair RDD (example: rdd={(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>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduceByKey()</td>
<td>Combine values with the same key</td>
<td>(1,2), (3,6)</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>Group values with the same key</td>
<td>(1,2), (3,4)</td>
</tr>
<tr>
<td>combineByKey()</td>
<td>Combine values using a different result type</td>
<td>See note: 34-40</td>
</tr>
</tbody>
</table>
## Transformations on one pair RDD (example: rdd={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(fuc)</td>
<td>Apply a function to each value of a pair RDD without changing the key</td>
<td><code>rdd.mapValues(x =&gt; x + 1)</code></td>
<td><code>{(1,3), (3,5), (3,7)}</code></td>
</tr>
<tr>
<td>flatMapValues(fuc)</td>
<td>Apply a function that returns an iterator</td>
<td><code>rdd.flatMapValues(x =&gt; x + 1)</code></td>
<td><code>{(1,2), (1,3), (1,4), (1,5), (3,4), (3,5)}</code></td>
</tr>
<tr>
<td>keys()</td>
<td>Return an RDD of just the keys</td>
<td><code>rdd.keys()</code></td>
<td><code>{1, 3, 3}</code></td>
</tr>
<tr>
<td>values()</td>
<td>Return an RDD of just the values</td>
<td><code>rdd.values()</code></td>
<td><code>{2, 4, 6}</code></td>
</tr>
<tr>
<td>sortByKey()</td>
<td>Return an RDD sorted by the key</td>
<td><code>rdd.sortByKey()</code></td>
<td><code>{(1,2), (3,4), (3,5)}</code></td>
</tr>
</tbody>
</table>

## SubtractByKey (example: rdd={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><code>rdd.subtractByKey(other)</code></td>
<td><code>{(1, 2)}</code></td>
</tr>
<tr>
<td>join()</td>
<td>Inner join</td>
<td><code>rdd.join(other)</code></td>
<td><code>{(3, (4, 9)), (3, (6, 9))}</code></td>
</tr>
<tr>
<td>rightOuterJoin()</td>
<td>Perform a join where the key must be present in the other RDD</td>
<td><code>rdd.rightOuterJoin(other)</code></td>
<td><code>{(3, Some(4), 9)), (3, Some(6), 9))}</code></td>
</tr>
<tr>
<td>leftOuterJoin()</td>
<td>Perform a join where the key must be present in the first RDD</td>
<td><code>rdd.leftOuterJoin(other)</code></td>
<td><code>{(1, (2, None)), (3, (4, Some(9))), (3, (6, Some(9))))</code></td>
</tr>
<tr>
<td>cogroup()</td>
<td>Group data from both RDDs sharing the same key</td>
<td><code>rdd.cogroup(other)</code></td>
<td><code>{(1, ([2], []), (3, ([4, 6], [9])))</code></td>
</tr>
</tbody>
</table>

## Filter on Pair RDDs

```java
Function < Tuple3 < String, String, Boolean > longWordFilter = new Function < Tuple3 < String, String, Boolean >() { public Boolean call( Tuple3 < String, String, Boolean > keyValue) { return (keyValue._2().length() < 20); } }); JavaPairRDD < String, String > result = pairs.filter(longWordFilter);```

## Aggregations with Pair RDDs

### reduceByKey()

- 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
Aggregations with Pair RDDs

**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
- 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

**Example**

```java
JavaPairRDD lines = sc.textFile("data.txt")
JavaPairRDD lines.mapToPair(x -> new Tuple2<>(x, 1))
JavaPairRDD counts = countMap.Lines.reduceByKey(a, b) -> a + b;
```

**Per-key average using combineByKey()**

```java
public static class AvgCount implements Serializable {
  public Integer num;
  public Integer total;
  public float avg = 0;
  ...
}
```

```java
public float call(Integer x) {
  num_ = num_ + 1;
  total_ = total_ + x;
  return total_ / (float) num_;
}
```

```java
public int createAcc() {
  return 0;
}
```

```java
public int addAndCount(AvgCount a, AvgCount b) {
  a.num_ = a.num_ + b.num_;
  a.total_ = a.total_ + b.total_;
  return a;
}
```

```java
public int combine(AvgCount a, AvgCount b) {
  return a;
}
```

```java
public static class AvgCount implements Serializable {
  public Integer num;
  public Integer total;
  public float avg = 0;
  ....
}
```

```java
public float call(Integer x) {
  num_ = num_ + 1;
  total_ = total_ + x;
  return total_ / (float) num_;
}
```

```java
public int createAcc() {
  return 0;
}
```

```java
public int addAndCount(AvgCount a, AvgCount b) {
  a.num_ = a.num_ + b.num_;
  a.total_ = a.total_ + b.total_;
  return a;
}
```

```java
public int combine(AvgCount a, AvgCount b) {
  return a;
}
```

**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
- `coalesce()`
  - Reduces data movement
Aggregations with Pair RDDs:

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

---

Join operations:

- **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>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>countByKey()</td>
<td>Count the number of elements for each key</td>
<td>rdd.countByKey()</td>
<td></td>
</tr>
<tr>
<td>collectAsMap()</td>
<td>Collect the result as a map to provide fast lookup at the driver</td>
<td>rdd.collectAsMap()</td>
<td></td>
</tr>
<tr>
<td>lookup(key)</td>
<td>Return values associated with the provided key</td>
<td>rdd.lookup()</td>
<td></td>
</tr>
</tbody>
</table>

---

Example

```
(rdd={(1,2),(3,4),(3,6)})
```
Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
RDD in Spark

RDDs in Spark: The Runtime

- User's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Representing RDDs

- A set of partitions
  - Atomic pieces of the dataset
- A set of dependencies on parent RDDs
- A function for computing the dataset based on its parents
- Metadata about its partitioning scheme
- Data placement