Analytics in Spark

Yanlei Diao

Slides Courtesy of Ion Stoica, Matei Zaharia, Brooke Wenig, Tim Hunter
What is Apache Spark?

1) Data abstraction: **Resilient Distributed Datasets** (RDDs)
   - Sets of objects partitioned & distributed across a cluster
   - Stored in RAM or on Disk

2) Parallel execution engine for big data
   - Implements **BSP** (Bulk Synchronous Processing) model

3) Automatic recovery based on **lineage** of bulk transformations

4) Unified analytics for SQL, ML, Graph, Statistical analysis…
1) Programming Model

Resilient distributed datasets (RDDs)

- Immutable collections partitioned across cluster that can be rebuilt if a partition is lost (fault tolerant)
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
- Can be cached across parallel operations

Restricted shared variables
  - Accumulators, broadcast variables
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
```

**Result:** full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)
## RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Actions (return a result to driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
<tr>
<td>reduceByKey</td>
<td></td>
</tr>
<tr>
<td>join</td>
<td></td>
</tr>
<tr>
<td>cache</td>
<td></td>
</tr>
</tbody>
</table>

http://spark.apache.org/docs/latest/programming-guide.html
Transformations (define a new RDD)

**map**(func): Return a new distributed dataset formed by passing each element of the source through a function `func`.

**filter**(func): Return a new dataset formed by selecting those elements of the source on which `func` returns true.

**flatMap**(func): Similar to map, but each input item can be mapped to 0 or more output items (so `func` should return a Seq rather than a single item).

**mapPartitions**(func): Similar to map, but runs separately on each partition (block) of the RDD, so `func` must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

**sample**: Sample a fraction `fraction` of the data, with or without replacement, using a given random number generator seed.

**union**(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.

**intersection**(otherDataset): Return a new RDD that is the intersection of elements in the source dataset and the argument.

**distinct**: Return a new dataset that contains the distinct elements of the source dataset.

**groupByKey**: When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note**: To perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.

**reduceByKey**(func): When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function `func`, which must be of type (V,V) => V.

**sort**([ascending]): When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order.

**join**(otherDataset): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.

**cogroup**(otherDataset): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples.
Actions (return a result to driver)

`count()`: Return the number of elements in the dataset.

`collect()`: Return all the elements of the dataset as an array at the driver program.

`reduce(func)`: Aggregate the elements of the dataset using a function `func` (which takes two arguments and returns one). The function should be **commutative** and **associative** so that it can be computed correctly in parallel.

`take(n)`: Return an array with the first `n` elements of the dataset.

`takeSample(n)`: Return an array with a random sample of `num` elements of the dataset

`saveAsTextFile(path)`: Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Call `toString` on each element to convert it to a line of text in the file.

`saveAsObjectFile(path)`: Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.

`lookupKey`
Multi-language Programming Interface

- Standalone programs can be written in any, but console is only Python & Scala
- **Python developers:** can stay with Python for both
- **Java developers:** consider using Scala for console (to learn the API)

Performance: Java / Scala will be faster (statically typed), but Python can do well for numerical work with NumPy
First Stop: SparkContext

Main entry point to Spark functionality
Created for you in Spark shells as variable `sc`
In standalone programs, you’d make your own

http://spark.apache.org/docs/latest/programming-guide.html
Creating RDDs

# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from storage systems such as local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)

Spark can read/write to any storage system / format that has a plugin for Hadoop!
- Examples: HDFS, S3, HBase, Cassandra, Avro, SequenceFile
- Reuses Hadoop’s InputFormat and OutputFormat APIs

APIs like SparkContext.textFile support filesystems, while SparkContext.hadoopRDD allows passing any Hadoop JobConf to configure an input source
Basic Transformations (Python)

```python
ums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)  # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0)  # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, ..., x-1)
Basic Actions (Python)

```python
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect()  # => [1, 2, 3]

# Return first K elements
nums.take(2)    # => [1, 2]

# Count number of elements
nums.count()    # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```
Spark’s “distributed reduce” transformations act on RDDs of *key-value pairs*

**Python:**
```python
pair = (a, b)
    pair[0] # => a
    pair[1] # => b
```

**Scala:**
```scala
val pair = (a, b)
    pair._1 // => a
    pair._2 // => b
```

**Java:**
```java
Tuple2 pair = new Tuple2(a, b); // class scala.Tuple2
    pair._1 // => a
    pair._2 // => b
```
Some Key-Value Operations (Python)

```python
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}
pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1})
pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements *combiners* on the map side
Example: Word Count (Python)

```python
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda x, y: x + y)
```

- "to be or"  
  - "to"  
  - "be"  
  - "or"  
  - (to, 1)  
  - (be, 1)  
  - (or, 1)  
  - (be, 2)

- "not to be"  
  - "not"  
  - "to"  
  - "be"  
  - (not, 1)  
  - (to, 1)  
  - (be, 1)  
  - (or, 1)

- (to, 2)  
- (not, 1)
Multiple Datasets

visits = sc.parallelize([("index.html", "1.2.3.4"),
                         ("about.html", "3.4.5.6"),
                         ("index.html", "1.3.3.1")])

pageNames = sc.parallelize([("index.html", "Home"),
                             ("about.html", "About")])

visits.join(pageNames)
  # ("index.html", ("1.2.3.4", "Home"))
  # ("index.html", ("1.3.3.1", "Home"))
  # ("about.html", ("3.4.5.6", "About"))

visits.cogroup(pageNames)
  # ("index.html", (Seq("1.2.3.4", "1.3.3.1"), Seq("Home")))
  # ("about.html", (Seq("3.4.5.6"), Seq("About")))
Controlling the Level of Parallelism (Python)

All the pair RDD operations take an optional second parameter for number of tasks

```python
words.reduceByKey(lambda x, y: x + y, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```