Analytics in Spark

Yanlei Diao
Tim Hunter

Slides Courtesy of Ion Stoica, Matei Zaharia and Brooke Wenig
Outline

1. A brief history of Big Data and Spark
2. Technical summary of Spark
3. Unified analytics
4. What does unification bring?
5. Structured programming with DataFrames
6. Machine Learning with MLlib
2009: State-of-the-art in Big Data

Apache Hadoop
- Open Source: HDFS, Hbase, MapReduce, Hive
- Large scale, flexible data processing engine
- Batch computation (e.g., 10s minutes to hours)

New requirements emerging
- Iterative computations, e.g., machine learning
- Interactive computations, e.g., ad-hoc analytics
Prior MapReduce Model

MapReduce model for clusters transforms data flowing from stable storage to stable storage, e.g., Hadoop:
How to Support ML Apps Better?

Iterative and interactive applications

Spark core
(RDD API)
How to Support Iterative Apps

Iterative and interactive applications

Spark core
(RDD API)
How to Support Interactive Apps

Iterative and interactive applications

Cache data in memory

Spark core
(RDD API)
Motivation for Spark

Acyclic data flow (MapReduce) is a powerful abstraction, but not efficient for apps that repeatedly reuse a working set of data:
  • iterative algorithms (many in machine learning)
  • interactive data mining tools (R, Excel, Python)

Spark makes working sets a first-class concept to efficiently support these apps
2. Technical Summary of Spark
What is Apache Spark?

1) Parallel execution engine for big data
   • Implements BSP (Bulk Synchronous Processing) model

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

3) Automatic recovery based on lineage of bulk transformations
1) Bulk Synchronous Processing (BSP) Model

In this classic model for designing parallel algorithms, computation proceeds in a series of **supersteps**:

- **Concurrent computation**: parallel processes perform local computation
- **Communication**: processes exchange data
- **Barrier synchronization**: when a process reaches the barrier, it waits until all other processes have reached the same barrier
Spark, as a BSP System

![Diagram showing the relationship between RDD tasks, Shuffle, and stage (super-step) in Spark.]

- RDD (processors)
- tasks
- Shuffle
- RDD (processors)

Stage (super-step)
Spark, as a BSP System

- All tasks in same stage run same operation,
- single-threaded, \textit{deterministic} execution

- \textbf{Immutable} dataset

- \textbf{Barrier} \textit{implicit} by data dependency such as group data by key
2) Programming Model

Resilient distributed datasets (RDDs)

- *Immutable* collections *partitioned* across cluster that can be rebuilt if a partition is lost
- 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>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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 Iteration<*> => Iteration<*> 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**
- **...**
3) RDD Fault Tolerance

RDDs maintain *lineage* (like *logical logging* in Aries) that can be used to reconstruct lost partitions

Ex: $\text{cachedMsgs} = \text{textFile}(\ldots).\text{filter}(\_\_.\text{contains}(\text{“error”}))$

$\quad .\text{map}(\_\_.\text{split}(‘\text{\t}’)(2))$

$\quad .\text{cache}()$
Benefits of RDD Model

Consistency is easy due to immutability (no updates)

Fault tolerance is inexpensive (log lineage rather than replicating/checkpointing data)

Locality-aware scheduling of tasks on partitions

Despite being restricted, model seems applicable to a broad variety of applications
3. Apache Spark’s Path to Unification
Apache Spark’s Path to Unification

Unified engine across data workloads and data sources

Streaming  SQL  ML  Graph  Batch  ...

Apache Spark™
DataFrame API

*DataFrame* logically equivalent to a relational table

Operators mostly relational with additional ones for statistical analysis, e.g., quantile, std, skew

Popularized by R and Python/pandas, languages of choice for Data Scientists
`pdata.map(lambda x: (x.dept, [x.age, 1]))`.reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]).map(lambda x: [x[0], x[1][0] / x[1][1]]).collect()
DataFrames, a Unifying Abstraction

Make DataFrame declarative, unify DataFrame and SQL

DataFrames and SQL share the same
- query optimizer, and
- execution engine

Tightly integrated with the rest of Spark
- ML library takes DataFrames as input & output
- Easily convert RDDs ↔ DataFrames

Every optimization automatically applies to SQL, and Scala, Python and R DataFrames
Today’s Apache Spark

Iterative, interactive, batch, and streaming applications

<table>
<thead>
<tr>
<th>SQL</th>
<th>Structured Streams</th>
<th>MLlib</th>
<th>Graph Frames</th>
<th>SparkR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SparkSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Spark core**
(DataFrames API, Catalyst, RDD API)