MapReduce, Hadoop and Spark

Bompotas Agorakis
Big Data Processing

Most of the computations are conceptually straightforward on a single machine but the volume of data is HUGE

- Need to use many (1,000s) of computers together to get results in a reasonable amount of time
- Management of parallelization, data distribution, failures handling, etc. => much more complex than the computation itself
MapReduce

- Simplifying model for large-scale data processing
- Inspired by functional programming paradigm
- Adapted to embarrassingly parallel workloads
  - Lots of concurrent operations on separate parts of the data with little or no synchronization
- Runtime support for parallelization, data distribution, failures handling, etc.
The example of LISP

- Lists are a primitive data type
  - '(1 2 3 4 5)
  - '((a 1) (b 2) (c 3))

- Functions written in prefix notation
  - (+ 1 2) → 3
  - (sqrt (+ (* 3 3) (* 4 4))) → 5

- Functions = lambda expression bound to variables
  - (define foo
    (lambda (x y)
      (sqrt (+ (* x x) (* y y)))))))
Lisp → MapReduce

- But what does this have to do with MapReduce?
  - After all, Lisp is about processing lists

- Two important concepts (first class higher order functions) in functional programming
  - Map: do something to everything in a list
  - Fold: combine results of a list in some way
Map

- Map is a higher-order function
- How map works:
  - Function is applied to every element in a list
  - Result is a new list
- Note that each operation is independent and, due to referential transparency (no side effects of functions evaluation), applying $f$ on one element and re-applying it again will always give the same result
Fold

- Fold is also a higher-order function
- How fold works:
  - Accumulator set to initial value
  - Function applied to list element and the accumulator
  - Result stored in the accumulator
  - Repeated for every item in the list
  - Result is the final value in the accumulator
Lisp → MapReduce

- Let’s assume a long list of records: imagine if...
  - We can parallelize map operations
  - We have a mechanism for bringing map results back together in the fold operation
- That’s MapReduce!
- Observations:
  - No limit to map parallelization since maps are independent
  - We can reorder folding if the fold function is commutative and associative
MapReduce: Programmers’ View

- Programmers specify two functions:
  - map \((k, v) \rightarrow <k', v'>\)*
  - reduce \((k', v') \rightarrow <k'', v''>\)*

- All \(v'\) with the same \(k'\) are reduced together

- MapReduce jobs are submitted to a scheduler that allocates the machines and deals with scheduling, fault tolerance, etc.
MapReduce: Schema
Example 1: word count

- Count how many times each word appears in a text corpus

Map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_values:
    EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
  // key: a word, same for input and output
  // intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
Example 2: Inverted index

- Get all documents containing some particular keyword
  - Used by the search mechanisms of Google, Yahoo!, etc.
  - Second input for PageRank

- Map function
  - Parse each document and emit a set of pairs <word, documentID>

- Reduce function
  - Take all pairs for a given word
  - Sort the document IDs
  - Emit a final <word, list(document IDs)> pair
Example 2: Inverted index

To be, or not to be

map

<to, a>, <be, a>, <or, a>, <not, a>, <to, a>, <be, a>

<be, a>, <a, b>, <do, b>, <is, b>, <not, a>, <or, a>, <to, a, b>

reduce

To be is to do

map

<to, b>, <be, b>, <is, b>, <to, b>, <do, b>

<not, b>, <or, a>, <to, a, b, b>

<be, a>, <a, b>, <do, b>, <is, b>, <not, a>, <or, a>, <to, a, b>
Hadoop

- Hadoop is the most known open-source MapReduce implementation
  - Lots of contributions by Yahoo!, now an Apache foundation project
  - Written in Java
  - Uses the HDFS file system (amongst others)
  - Many extensions and optimizations over the original Google paper

- A MapReduce implementation of choice when using Amazon’s cloud services
  - EC2: rent computing power and temporary space
  - S3: rent long term storage space
HDFS: Hadoop Distrib. File System

- A **distributed, scalable** file system for M-R applications
  - Distributed: Runs in a cluster
  - Scalable: 10K nodes, 100K files, 10PB storage
  - Closed-source optimizations
  - HDFS provides a single file system view to the whole cluster

- Files are split up in **blocks**
  - Typically 128MB
  - Each block is replicated on multiple **DataNodes** (typically 3)
  - Block placement is rack-aware
MapReduce Architecture

- Master/Slave Architecture
- HDFS
  - A centralized **NameNode** controls multiple **DataNodes**
  - **NameNode**: keeps track of which DataNode stores which block
  - **DataNodes**: “dumb” servers storing raw file chunks
- MapReduce
  - A centralized **JobTracker** controls multiple **TaskTrackers**
- Placement
  - **NameNode** and **JobTracker** run on the master
  - **DataNode** and **TaskTracker** run on workers
  - Data locality is exploited
Hadoop Usecases

- **NY Times**
  - Large Scale Image Conversions
  - 100 Amazon EC2 Instances, 4TB raw TIFF data
  - 11 Million PDF in 24 hours and 240$

- **Facebook**
  - Internal log processing
  - Reporting, analytics and machine learning
  - Cluster of 1110 machines, 8800 cores and 12PB raw storage
  - Open source contributors (Hive)

- **Twitter**
  - Store and process tweets, logs, etc
  - Open source contributors (Hadoop-lzo)
Hadoop Use Cases

- **Yahoo**
  - 100,000 CPUs in 25,000 computers
  - Content/Ads Optimization, Search index
  - Machine learning (e.g. spam filtering)
  - Open source contributors (Pig)

- **Microsoft**
  - Natural language search (through Powerset)
  - 400 nodes in EC2, storage in S3
  - Open source contributors (!) to HBase

- **Amazon**
  - ElasticMapReduce service
  - On demand elastic Hadoop clusters for the Cloud
MapReduce: Conclusion

- MapReduce is a powerful simplifying abstraction for programming large-scale data processing
  - Naturally suited to embarrassingly parallel jobs
  - But is not adapted to all types of jobs (e.g., jobs with data interdependencies)
- Master = single point of failure
- Extensions
  - Process streams of data (StreamMine project, StreamMapReduce)
    - Real-Time support and complex event processing
  - Decentralize the master and use a collaborative scheme
    - Build the master using a DHT and replication for fault tolerance
  - Automatic MapReduce-ization
    - Some work already on automatic MR code generation from SQL queries (Prof. W. Zwaenepoel @ EPFL - EuroSys 2011)
Spark

- Fast and Expressive Cluster Computing System compatible with Apache Hadoop
- Efficient
  - General execution graphs
  - In-memory storage
- Usable
  - Rich APIs in Java, Scala, Python, R
  - Interactive shell
Key Concepts

- Write programs in terms of transformations on distributed datasets

- Resilient Distributed Datasets
  - Collections of objects spread across a cluster, stored in RAM or on Disk
  - Built through parallel transformations
  - Automatically rebuilt on failure

- Operations
  - Transformations (e.g. map, filter, groupBy)
  - Actions (e.g. count, collect, save)
Working With RDDs

```python
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
textFile = sc.textFile("SomeFile.txt")
linesWithSpark.count()
74
linesWithSpark.first()
# Apache Spark
```
Scaling Down

![Bar chart showing execution time vs. cache percentage](chart.png)

**Execution time (s)**
- Cache disabled: 69 s
- 25% cache: 58 s
- 50% cache: 41 s
- 75% cache: 30 s
- Fully cached: 12 s

% of working set in cache
Fault Recovery

- RDDs track lineage information that can be used to efficiently recompute lost data

```scala
msgs = textFile.filter(lambda s: s.startsWith("ERROR")) .map(lambda s: s.split("\t")[2])
```

Diagram:
- HDFS File
  - filter
    - (func = startsWith(...))
- Filtered RDD
  - map
    - (func = split(...))
- Mapped RDD
Language Support

- **Standalone Programs**
  - Python, Scala, Java, R

- **Interactive Shells**
  - Python & Scala

- **Performance**
  - Java & Scala are faster due to static typing
  - ... but Python is often fine
Interactive Shell

- The fastest way to learn Spark
- Available in Python and Scala
- Runs as an application on an existing cluster or can run locally
First thing that a Spark program does is create a SparkContext object, which tells Spark how to access a cluster.

In the shell for either Scala or Python, this is the `sc` variable, which is created automatically.

Other programs must use a constructor to instantiate a new SparkContext.

Then in turn SparkContext gets used to create other variables.
Spark Essentials: SparkContext

- Scala:
  ```scala
  scala> sc
  res: spark.SparkContext = spark.SparkContext@470d1f30
  ```

- Python:
  ```python
  >>> sc
  <pyspark.context.SparkContext object at 0x7f7570783350>
  ```
Spark Essentials: **Master**

- The **master** parameter for a SparkContext determines which cluster to use.

<table>
<thead>
<tr>
<th>master</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>local</strong></td>
<td>run Spark locally with one worker thread (no parallelism)</td>
</tr>
<tr>
<td><strong>local[K]</strong></td>
<td>run Spark locally with K worker threads (ideally set to # cores)</td>
</tr>
<tr>
<td><strong>spark://HOST:PORT</strong></td>
<td>connect to a Spark standalone cluster; PORT depends on config (7077 by default)</td>
</tr>
<tr>
<td><strong>mesos://HOST:PORT</strong></td>
<td>connect to a Mesos cluster; PORT depends on config (5050 by default)</td>
</tr>
</tbody>
</table>
1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends app code to the *executors*
4. sends *tasks* for the executors to run
Resilient Distributed Datasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel.

There are currently two types:

- *parallelized collections* – take an existing Scala collection and run functions on it in parallel

- *Hadoop datasets* – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop
Spark Essentials: **RDD**

- two types of operations on RDDs: *transformations* and *actions*

- transformations are lazy (not computed immediately)

- the transformed RDD gets recomputed when an action is run on it (default)

- however, an RDD can be *persisted* into storage in memory or disk
Spark Essentials: RDD

- Scala:

```scala
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

- Python:

```python
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```
Spark Essentials: RDD

- Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

- Spark supports text files, SequenceFiles, and any other Hadoop InputFormat, and can also take a directory or a glob (e.g. /data/201404*)
Spark Essentials: RDD

- **Scala:**
  ```scala
  scala> val distFile = sc.textFile("README.md")
  distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
  ```

- **Python:**
  ```python
  >>> distFile = sc.textFile("README.md")
  14/04/19 23:42:40 INFO storage.MemoryStore: ensureFreeSpace(36827) called with curMem=0, maxMem=318111744
  14/04/19 23:42:40 INFO storage.MemoryStore: Block broadcast_0 stored as values to memory (estimated size 36.0 KB, free 303.3 MB)
  >>> distFile
  MappedRDD[2] at textFile at NativeMethodAccessorImpl.java:-2
  ```
Spark Essentials:
Transformations

- Transformations create a new dataset from an existing one

- All transformations in Spark are lazy: they do not compute their results right away – instead they remember the transformations applied to some base dataset
  - optimize the required calculations
  - recover from lost data partitions
## Spark Essentials: Transformations

<table>
<thead>
<tr>
<th>transformation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(func)</code></td>
<td>return a new distributed dataset formed by passing each element of the source through a function <code>func</code></td>
</tr>
<tr>
<td><code>filter(func)</code></td>
<td>return a new dataset formed by selecting those elements of the source on which <code>func</code> returns true</td>
</tr>
<tr>
<td><code>flatMap(func)</code></td>
<td>similar to <code>map</code>, but each input item can be mapped to 0 or more output items (so <code>func</code> should return a <code>Seq</code> rather than a single item)</td>
</tr>
<tr>
<td><code>sample(withReplacement, fraction, seed)</code></td>
<td>sample a fraction <code>fraction</code> of the data, with or without replacement, using a given random number generator <code>seed</code></td>
</tr>
<tr>
<td><code>union(otherDataset)</code></td>
<td>return a new dataset that contains the union of the elements in the source dataset and the argument</td>
</tr>
<tr>
<td><code>distinct([numTasks]))</code></td>
<td>return a new dataset that contains the distinct elements of the source dataset</td>
</tr>
</tbody>
</table>
# Spark Essentials: Transformations

<table>
<thead>
<tr>
<th>transformation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>groupByKey([numTasks])</code></td>
<td>when called on a dataset of ((K, V)) pairs, returns a dataset of ((K, Seq[V])) pairs</td>
</tr>
<tr>
<td><code>reduceByKey(func, [numTasks])</code></td>
<td>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</td>
</tr>
<tr>
<td><code>sortByKey([ascending], [numTasks])</code></td>
<td>when called on a dataset of ((K, V)) pairs where (K) implements <code>Ordered</code>, returns a dataset of ((K, V)) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument</td>
</tr>
<tr>
<td><code>join(otherDataset, [numTasks])</code></td>
<td>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</td>
</tr>
<tr>
<td><code>cogroup(otherDataset, [numTasks])</code></td>
<td>when called on datasets of type ((K, V)) and ((K, W)), returns a dataset of ((K, Seq[V], Seq[W])) tuples – also called groupwith</td>
</tr>
<tr>
<td><code>cartesian(otherDataset)</code></td>
<td>when called on datasets of types (T) and (U), returns a dataset of ((T, U)) pairs (all pairs of elements)</td>
</tr>
</tbody>
</table>
Spark Essentials: Transformations

- Scala:
  ```scala
def readMD(): String = {
    val distFile = sc.textFile("README.md")
    distFile.map(l => l.split(" ")).collect()
    distFile.flatMap(l => l.split(" ")).collect()
  }
```

- Python:
  ```python
def readMD(): str = {
    distFile = sc.textFile("README.md")
    distFile.map(lambda x: x.split(' ')).collect()
    distFile.flatMap(lambda x: x.split(' ')).collect()
  }
```
## Spark Essentials: Actions

<table>
<thead>
<tr>
<th>action</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduce(func)</code></td>
<td>aggregate the elements of the dataset using a function <code>func</code> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>return the number of elements in the dataset</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>return the first element of the dataset – similar to <code>take(1)</code></td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>return an array with the first <code>n</code> elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements</td>
</tr>
<tr>
<td><code>takeSample(withReplacement, fraction, seed)</code></td>
<td>return an array with a random sample of <code>num</code> elements of the dataset, with or without replacement, using the given random number generator seed</td>
</tr>
<tr>
<td>action</td>
<td>description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>saveAsTextFile(path)</code></td>
<td>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. Spark will call <code>toString</code> on each element to convert it to a line of text in the file</td>
</tr>
<tr>
<td><code>saveAsSequenceFile(path)</code></td>
<td>write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).</td>
</tr>
<tr>
<td><code>countByKey()</code></td>
<td>only available on RDDs of type $(k, v)$. Returns a <code>Map</code> of $(k, Int)` pairs with the count of each key</td>
</tr>
<tr>
<td><code>foreach(func)</code></td>
<td>run a function <code>func</code> on each element of the dataset — usually done for side effects such as updating an accumulator variable or interacting with external storage systems</td>
</tr>
</tbody>
</table>
Spark Essentials: Actions

Scala:

```scala
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _).collect.foreach(println)
```

Python:

```python
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add).collect()
```
Spark Example: Word Count

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

1. Start each page at a rank of 1
2. On each iteration, have page p contribute \( \text{rank}_p / |\text{neighbors}_p| \) to its neighbors
3. Set each page’s rank to \( 0.15 + 0.85 \times \text{contribs} \)
Spark Example: PageRank
Spark Example: PageRank
Spark Example: PageRank
Spark Example: PageRank

val links = // RDD of (url, neighbors) pairs
val ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (neighbors, rank)) =>
      neighbors.map(x => (x, rank/neighbors.size))
  }
  ranks = contribs
    .reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}

ranks.saveAsTextFile("mypagerank.txt")
References

