Scalable Data Analysis (CIS 602-02)

General Cluster Computing

Dr. David Koop
MapReduce Overview

- Pre-loaded local input data
- Intermediate data from mappers
- Values exchanged by shuffle process
- Reducing process generates outputs
- Outputs stored locally

[Diagram of MapReduce process]

[Yahoo! Hadoop Tutorial]
Word Count Example

map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));

[Dean and Ghemawat, 2004]
Word Count Example

The overall MapReduce word count process

Input

- Deer Bear River
- Car Car River
- Deer Car Bear

Splitting

- Deer Bear River
- Car Car River
- Deer Car Bear

Mapping

- Deer, 1 River, 1
- Bear, 1 Car, 1
- Car, 1 River, 1
- Deer, 1 Car, 1
- Deer, 1 Bear, 1
- River, 1 Bear, 1

Shuffling

- Bear, 1 Bear, 1
- Car, 1 Car, 1 Car, 1
- Deer, 1 Deer, 1
- River, 1 River, 1

Reducing

- Bear, 2
- Car, 3
- Deer, 2
- River, 2

Final result

- Bear, 2
- Car, 3
- Deer, 2
- River, 2
Dewitt and Stonebraker on MapReduce (2008)

"We are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications.

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on"

[via Bill Howe]
## Parallel Databases vs. MapReduce

<table>
<thead>
<tr>
<th></th>
<th>Parallel DBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema Support</td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td>Indexing</td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td>Programming Model</td>
<td>Declarative (SQL)</td>
<td>Imperative (C/C++, Java, …) Extensions through Pig and Hive</td>
</tr>
<tr>
<td>Optimizations (Compression, Query Optimization)</td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Not out of the box</td>
<td>✓</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Coarse grained techniques</td>
<td>✓</td>
</tr>
</tbody>
</table>

[via J. Freire, from Pavlo et al., 2009 & Stonebraker et al., 2010]
Comparing Load Times

Figure 1: Load Times – Grep Task Data Set (535MB/node)

Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

Figure 1: Load Times – Grep Task Data Set (535MB/node)

Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

[Pavlo et al., 2009]
Comparing Execution Times

Figure 4: Grep Task Results – 535MB/node Data Set

Figure 5: Grep Task Results – 1TB/cluster Data Set

[Source: Pavlo et al., 2009]
Dean and Ghemawat's Response

• Heterogeneous systems:
  - There is data is lots of different storage systems: files, Google Bigtable, column stores
  - Can write your own storage backend to combine data
  - Parallel databases require all data to be loaded

• Indices:
  - Database indexing techniques can be used for MapReduce
  - Push data processing to the database
  - Or, use partitioning of data

[via J. Freire, Dean and Ghemawat, 2010]
Dean and Ghemawat's Response (continued)

• Complex functions:
  - MapReduce was designed for complex tasks that do not fit well into the relational paradigm
  - RDBMS have user-defined functions but buggy/missing

• Structured data and schemas:
  - Google's MapReduce supports Protocol Buffer that allows an optimized binary representation
  - https://developers.google.com/protocol-buffers/

[via J. Freire, Dean and Ghemawat, 2010]
Quiz
Project Example

- Jake VanderPlas's report on Seattle's bike sharing service:
  - [http://nbviewer.jupyter.org/github/jakevdp/ProntoData/blob/master/ProntoData.ipynb](http://nbviewer.jupyter.org/github/jakevdp/ProntoData/blob/master/ProntoData.ipynb)
Spark: Focus on data reads/writes

- MapReduce is very useful, but data is written to disk
- What about multiple operations?
- What about different types of operations?
Spark: Cluster Computing with Working Sets

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica
Comparison with Distributed Shared Memory

<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Coarse- or fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Coarse-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using backup tasks</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (runtimes aim for transparency)</td>
</tr>
<tr>
<td>Behavior if not enough RAM</td>
<td>Similar to existing data flow systems</td>
<td>Poor performance (swapping?)</td>
</tr>
</tbody>
</table>

[Zaharia et al., 2012]
## Spark Transformations and Actions

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td>RDD[T] ⇒ RDD[T]</td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td>RDD[T] ⇒ RDD[T] (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD[T], RDD[T]) ⇒ RDD[T]</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>RDD[(K, V)], RDD[(K, W)] ⇒ RDD[(K, Seq[V], Seq[W])]</td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD[T] ⇒ Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD[T] ⇒ Seq[T]</td>
</tr>
<tr>
<td><code>reduce(f : (T, T) ⇒ T)</code></td>
<td>RDD[T] ⇒ T</td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td>RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)</td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

---

[Zaharia et al., 2012]
PageRank in Spark

// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
    // Build an RDD of (targetURL, float) pairs
    // with the contributions sent by each page
    val contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) => x+y)
        .mapValues(sum => a/N + (1-a)*sum)
    val contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) => x+y)
        .mapValues(sum => a/N + (1-a)*sum)
    ...}

[Zaharia et al., 2012]
Narrow vs. Wide Dependencies

Spark stores datasets in a graph-based data representation where *partitions* are atomic.

[Zaharia et al., 2012]
Spark Job Stages

Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1’s output RDD is already in RAM, so we run stage 2 and then 3. [Zaharia et al., 2012]
Evaluation

Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

[Zaharia et al., 2012]
Evaluation

Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

[Zaharia et al., 2012]
Performance on PageRank

![Graph: Performance of PageRank on Hadoop and Spark]

Figure 10: Performance of PageRank on Hadoop and Spark.

[Zaharia et al., 2012]
Memory Scaling

Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

[Zaharia et al., 2012]
Node Scaling

(a) Traffic modeling

(b) Spam classification

[Zaharia et al., 2012]
Conclusions

• Spark focuses on dealing with the data first and writing operations that interact with the data in lazy fashion

• More flexibility but also more complexity
Demo
Next…

• Streaming data…
• More on using Spark with EC2
• Continue to work on projects