Behavioral Data Mining

Lecture 6
Hadoop and MapReduce
(slides originally by Matei Zaharia)
What is MapReduce?

- **Data-parallel** programming model for clusters of commodity machines
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by open-source Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …
What is MapReduce used for?

• At Google:
  – Index building for Google Search
  – Article clustering for Google News
  – Statistical machine translation

• At Yahoo!:
  – Index building for Yahoo! Search
  – Spam detection for Yahoo! Mail

• At Facebook:
  – Data mining
  – Ad optimization
  – Spam detection
Example: Facebook Lexicon

Search: party tonight, hangover

Suggestions: skiing, beach  |  hip hop, techno  |  happy birthday  |  eid

☑️ party tonight  ☑️ hangover

www.facebook.com/lexicon
Example: Facebook Lexicon

Search: hola, salut, ciao

Suggestions: vacation | xoxo, xoxoxo | midterm, final | party tonight, hangover

☑ hola ☑ salut ☑ ciao

www.facebook.com/lexicon
What is MapReduce used for?

- In research:
  - Analyzing Wikipedia (PARC)
  - Natural language processing (CMU)
  - Bioinformatics (Maryland)
  - Astronomical image analysis (Washington)
  - Ocean climate simulation (Washington)
  - <Your application here>
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• Getting started with Hadoop
• Hadoop for machine learning
MapReduce Design Goals

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 50 MB/s = 23 days
   - Scan on 1000-node cluster = 33 minutes

2. **Cost-efficiency**:
   - Commodity nodes (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)
Typical Hadoop Cluster

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo terasort):
  8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)
Challenges

• **Cheap nodes fail, especially if you have many**
  – Mean time between failures for 1 node = 3 years
  – MTBF for 1000 nodes = 1 day
  – **Solution:** Build fault-tolerance into system

• **Commodity network = low bandwidth**
  – **Solution:** Push computation to the data

• **Programming distributed systems is hard**
  – **Solution:** Data-parallel programming model: users write “map” and “reduce” functions, system handles work distribution and fault tolerance
Hadoop Components

• **Distributed file system (HDFS)**
  – Single namespace for entire cluster
  – Replicates data 3x for fault-tolerance

• **MapReduce implementation**
  – Executes user jobs specified as “map” and “reduce” functions
  – Manages work distribution & fault-tolerance
Hadoop Distributed File System

- Files split into 128MB blocks
- Blocks replicated across several datanodes (usually 3)
- Single namenode stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only
MapReduce Programming Model

• Data type: key-value records

• Map function:
  \((K_{\text{in}}, V_{\text{in}}) \Rightarrow \text{list}(K_{\text{inter}}, V_{\text{inter}})\)

• Reduce function:
  \((K_{\text{inter}}, \text{list}(V_{\text{inter}})) \Rightarrow \text{list}(K_{\text{out}}, V_{\text{out}})\)
MapReduce Programming Model

• To map tabular data to MapReduce, use keys which are row/column tuples:

• Row-based:
  \[ K = (K_{row}, K_{column}) \]

• Column-based:
  \[ K = (K_{column}, K_{row}) \]

• or lumped row-based:
  \[ K = K_{row}, V = list(v_{col1}, v_{col2}, v_{col3}, \ldots) \]
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
Word Count Execution

Input | Map | Shuffle & Sort | Reduce | Output
---|---|---|---|---
the quick brown fox | Map | | Reduce | brown, 2
the fox ate the mouse | Map | | Reduce | fox, 2
how now brown cow | Map | | Reduce | how, 1
cow, 1
Think:

SELECT red_fcn(cols) FROM
   SELECT map_fcn(cols) FROM data_source
GROUP BY key
MapReduce Execution Details

- Single *master* controls job execution on multiple *slaves* as well as user scheduling
- Mappers preferentially placed on same node or same rack as their *input block*
  - Push computation to data, minimize network use
- Mappers save outputs to local disk rather than pushing directly to reducers
  - Allows having more reducers than nodes
  - Allows recovery if a reducer crashes
An Optimization: The Combiner

- A **combiner** is a local aggregation function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases size of intermediate data
- Very often the same function as the reducer

- Example: local counting for Word Count:

```python
def combiner(key, values):
    output(key, sum(values))
```
Word Count with Combiner

Input  Map & Combine  Shuffle & Sort  Reduce  Output

the quick brown fox
the fox ate the mouse
how now brown cow

Map
the, 1
brown, 1
fox, 1

Map
the, 2
fox, 1

Map
how, 1
now, 1
brown, 1

Map
ate, 1
mouse, 1

Reduce
cow, 1

Reduce
brown, 2
fox, 2
how, 1
now, 1
the, 3

Reduce
ate, 1
cow, 1
mouse, 1
quick, 1
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• Getting started with Hadoop
• Hadoop for machine learning
Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     • Okay for a map because it had no dependencies
     • Okay for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block (user-controlled)

➢ Note: For this and the other fault tolerance features to work, your map and reduce tasks must be side-effect-free
2. If a node crashes:
   − Relaunch its current tasks on other nodes
   − Relaunch any maps the node previously ran
     • Necessary because their output files were lost along with the crashed node
Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and kill the other one

• On behavioral data (usually power law) you will often get structural stragglers – nodes whose random share of the data is very large. These require special treatment – truncation, splitting,…
Takeaways

• By providing a data-parallel programming model, MapReduce can control job execution in useful ways:
  – Automatic division of job into tasks
  – Automatic placement of computation near data
  – Automatic load balancing
  – Recovery from failures & stragglers

• User focuses on application, not on complexities of distributed computing
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• **Sample applications**
• Getting started with Hadoop
• Hadoop and machine learning
I. Search

- **Input**: (lineNumber, line) records
- **Output**: lines matching a given pattern

- **Map**: 
  ```
  if(line matches pattern):
    output(line)
  ```

- **Reduce**: identify function
  - Alternative: no reducer (map-only job)
2. Sort

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick partitioning function $h$ such that $k_1 < k_2 \Rightarrow h(k_1) < h(k_2)$
3. Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

- **Map:**
  
  ```python
  foreach word in text.split():
      output(word, filename)
  ```

- **Combine:** uniquify filenames for each word

- **Reduce:**
  
  ```python
  def reduce(word, filenames):
      output(word, sort(filenames))
  ```
Inverted Index Example

hamlet.txt
  to be or not to be

12th.txt
  be not afraid of greatness

afraid, (12th.txt)
  be, (12th.txt, hamlet.txt)
  greatness, (12th.txt)
  not, (12th.txt, hamlet.txt)
  of, (12th.txt)
  or, (hamlet.txt)
  to, (hamlet.txt)

be, (12th.txt, hamlet.txt)
  or, hamlet.txt

not, hamlet.txt

of, 12th.txt

greatness, 12th.txt
4. Most Popular Words

- **Input:** (filename, text) records
- **Output:** the 100 words occurring in most files

- **Two-stage solution:**
  - **Job 1:**
    - Create inverted index, giving (word, list(file)) records
  - **Job 2:**
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job

- **Optimizations:**
  - Map to (word, 1) instead of (word, file) in Job 1
  - Estimate count distribution in advance by sampling
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• **Getting started with Hadoop**
• Hadoop and machine learning
Getting Started with Hadoop

• Run jobs on icluster1.eecs.berkeley.edu
OR
• Download from hadoop.apache.org
• To install locally, unzip and set JAVA_HOME
• Details: hadoop.apache.org/core/docs/current/quickstart.html

• Three ½ ways to write jobs:
  – Java API
  – Hadoop Streaming (for Python, Perl, etc)
  – Pipes API (C++)
  – Write in Scala and run a jar with the runtime
public static class MapClass extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable ONE = new IntWritable(1);

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            output.collect(new Text(itr.nextToken()), ONE);
        }
    }
}
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
Word Count in Java

```java
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    FileInputFormat.setInputPaths(conf, args[0]);
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setOutputKeyClass(Text.class); // out keys are words (strings)
    conf.setOutputValueClass(IntWritable.class); // values are counts

    JobClient.runJob(conf);
}
```
Word Count in Python with Hadoop Streaming

**Mapper.py:**
```python
import sys
for line in sys.stdin:
    for word in line.split():
        print(word.lower() + "\t" + 1)
```

**Reducer.py:**
```python
import sys
counts = {}
for line in sys.stdin:
    word, count = line.split("\t")
    dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
    print(word.lower() + "\t" + 1)
```
The Good, the Bad, the Ugly

- MapReduce is extremely popular, especially since the release of Hadoop.
- Availability in cloud services (like EC2) makes it usable by just about anyone.
- While you can’t do everything in Hadoop, its task coverage seems to be very high.
- Hash partitioning, the basic sort/group model, and its error recovery make it relatively simple to use as a programming model.
- Leverages Java serialization, reflection,…
- Fits well with higher-level interfaces like Hive, Pig etc.
The Good, **the Bad, the Ugly**

- By itself, Hadoop/MR is too low-level for large-scale programming.
- It is procedural, and doesn’t support query optimization.
- Job turnaround time is high because of rigidity in the design.
- On a cluster, you don’t have to do everything “well”, and its easy to compensate for a weak implementation with extra cycles.
The Good, the Bad, the Ugly

- Mapper/reducer code has poor structure compared to columnar design (e.g. SQL functions).

- Hard memory partitioning in Hadoop make it a poor match for machine learning algorithms with large models.
- “stateless” mappers and reducers are also inefficient for ML algorithms that work incrementally.
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• Getting started with Hadoop
• Hadoop and machine learning
Data Descent
Data Descent

10 Mops
Text data
log data
event data
(strings)

100:1 shrink
Hadoop Cluster

Filtering
Tokenizing
Bag-of-features

10-100 Gflops
Multicore CPU (i7)

Effective
Performance
Gain = 100,000
Data Descent

• Simple optimizations can offset many cores.
• Should *always* do these optimizations, whether the target is a single CPU or a cluster.
• We should use the most efficient data representation i.e. blocked binary matrix data.
Blocked Matrix Files

Element-wise K-V representation

Blocked K-V representation

Tune block size for best perf.
Blocked Matrix rep

Blocked matrix representation achieves two things:

• Blocks are sized to optimize matrix operations (usually sparse matrix multiply)

• Expensive Hadoop steps (key extraction, comparison, reflection) occur only once per many data items. I/O is dominated by binary reads.
Conclusions

• MapReduce’s data-parallel programming model hides complexity of distribution and fault tolerance

• Principal philosophies:
  – *Make it scale*, so you can throw hardware at problems
  – *Make it cheap*, saving hardware, programmer and administration costs (but requiring fault tolerance)

• MapReduce is not suitable for all problems, but when it works, it may save you a lot of time

• Even with many cores, data representation dominates performance. Use efficient designs with one or many CPUs.
Resources

• Hadoop: http://hadoop.apache.org/core/
• Hadoop docs: http://hadoop.apache.org/core/docs/current/
• Pig: http://hadoop.apache.org/pig
• Hive: http://hadoop.apache.org/hive
• Hadoop video tutorials from Cloudera: http://www.cloudera.com/hadoop-training