Parallel Processing
(4/4)

Introduction to
MapReduce/HADOOP
What is MapReduce?

• A programming model (& its associated implementation)
• For processing large data set
• Exploits large set of commodity computers
• Executes process in distributed manner
• Offers high degree of transparencies
• In other words:
  – simple and maybe suitable for your tasks !!!
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
What is MapReduce used for?

- In research:
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Bioinformatics (Maryland)
  - Astronomical image analysis (Washington)
  - Ocean climate simulation (Washington)
  - <Your application here>
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:
  \[(K_{\text{inter}}, \text{list}(V_{\text{inter}})) \rightarrow \text{list}(K_{\text{out}}, V_{\text{out}})\]
Functional Programming Review
Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter
fun foo(l: int list) = sum(l) + mul(l) + length(l)

• Order of sum() and mul(), etc does not matter
• They do not modify l
Functional Updates Do Not Modify Structures

- fun append(x, lst) =
  let lst' = reverse lst in
  reverse ( x :: lst' )

- The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.

- But it never modifies lst!
Functions Can Be Used As Arguments

- fun DoDouble(f, x) = f (f x)

- It does not matter what f does to its argument; DoDouble() will do it twice.

- What is the type of this function?
Map

- map f a [] = f(a)
  map f(a:as) = list(f(a), map(f,as))
- Creates a new list by applying f to each element of the input list; returns output in order.
Fold

- fold f x_0 lst: ('a*'b->'b)->'b->('a list)->'b
- Moves across a list, applying f to each element plus an accumulator. f returns the next accumulator value, which is combined with the next element of the list
fold left vs. fold right

- Order of list elements can be significant
- Fold left moves left-to-right across the list
- Fold right moves from right-to-left

- SML Implementation:

  - fun foldl f a [] = a
    | foldl f a (x::xs) = foldl f (f(x, a)) xs

  - fun foldr f a [] = a
    | foldr f a (x::xs) = f(x, (foldr f a xs))
Example

- **fun** foo(l: int list)
  
  = sum(l) + mul(l) + length(l)

- How can we implement this?
Example (Solved)

• fun foo(l: int list)
  = sum(l) + mul(l) + length(l)

• fun sum(lst) = foldl (fn (x,a)=\>x+a) 0 lst
• fun mul(lst) = foldl (fn (x,a)=\>x*a) 1 lst
• fun length(lst) = foldl (fn (x,a)=\>1+a) 0 lst
A More Complicated Fold Problem

• Given a list of numbers, how can we generate a list of partial sums?

• e.g.: [1, 4, 8, 3, 7, 9]
  =>  [0, 1, 5, 13, 16, 23, 32]
A More Complicated Fold Problem

• Given a list of numbers, how can we generate a list of partial sums?

• e.g.: [1, 4, 8, 3, 7, 9]
  => [0, 1, 5, 13, 16, 23, 32]

fun partialsum(lst)
  = foldl(fn(x,a) => list(a (last(a) +x))) 0 lst
A More Complicated Map Problem

• Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?

• [“my”, “happy”, “cat” ]
  -> [“tac”, “yppha”, “ym”]
A More Complicated Map Problem

• Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?

  • [“my”, “happy”, “cat” ]
    -> [“tac”, “yppha”, “ym”]

• fun reverse2(lst) =
  foldr(fn(x,a) => list(a, reverseword(x)) [] lst
map Implementation

- 
  fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)

- This implementation moves left-to-right across the list, mapping elements one at a time

- \[ \] But does it need to?
Implicit Parallelism In map

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements.
- If order of application of f to elements in list is commutative, we can reorder or parallelize execution.
- This is the secret (insight) that MapReduce exploits.
Motivation: Large Scale Data Processing

- Want to process lots of data ( > 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- Want to make this easy
  - Hide the details of parallelism, machine management, fault tolerance, etc.
MapReduce

• Automatic parallelization & distribution
• Fault-tolerant
• Provides status and monitoring tools
• Clean abstraction for programmers
Programming Model

• Borrows from functional programming
• Users implement interface of two functions:
  • map (in_key, in_value) ->
    (out_key, intermediate_value) list
  • reduce (out_key, intermediate_value list) ->
    out_value list
• Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).

• map() produces one or more "intermediate" values along with an output key from the input.
reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more final values for that same output key
- (in practice, usually only one final value per key)
Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed independently
- Bottleneck: reduce phase cannot start until map phase is completely finished.
Distributed Grep

Very big data

Split data → grep → matches
Split data → grep → matches
Split data → grep → matches
Split data → grep → matches

→ cat → All matches
Distributed Word Count

Very big data

\[ \text{Split data} \rightarrow \text{count} \rightarrow \text{count} \rightarrow \text{merge} \rightarrow \text{merged count} \]
Map Reduce

• Map:
  – Accepts *input* key/value pair
  – Emits *intermediate* key/value pair

• Reduce:
  – Accepts *intermediate* key/value* pair
  – Emits *output* key/value pair

Very big data

Result
Partitioning Function

Input

Intermediate

Group by Key

Grouped

Output
Partitioning Function (2)

- **Default**: \( \text{hash(key)} \mod R \)
- **Guarantee**: 
  - Relatively well-balanced partitions
  - Ordering guarantee within partition
- **Distributed Sort**
  - **Map**:
    - \text{emit(key,value)}
  - **Reduce (with R=1)**:
    - \text{emit(key,value)}
MapReduce

• Distributed Grep
  – Map:
    
    \[
    \text{if match(value, pattern) emit(value, 1)}
    \]
  – Reduce:
    
    \[
    \text{emit(key, sum(value*)})
    \]

• Distributed Word Count
  – Map:
    
    \[
    \text{for all } w \text{ in value do emit(w, 1)}
    \]
  – Reduce:
    
    \[
    \text{emit(key, sum(value*)})
    \]
Example: Count word occurrences

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

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += parseInt(v);
    Emit(AsString(result));
```
Example Word Count (1)

- Map

```java
public static class MapClass extends MapReduceBase
    implements Mapper {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(WritableComparable key, Writable value,
                    OutputCollector output, Reporter reporter)
        throws IOException {
        String line = ((Text)value).toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken);
            output.collect(word, one);
        }
    }
}
```
Example Word Count (2)

• **Reduce**

```java
public static class Reduce extends MapReduceBase implements Reducer {
    public void reduce(WritableComparable key, Iterator values, OutputCollector output, Reporter reporter)
        throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += ((IntWritable) values.next()).get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```
Example Word Count (3)

• Main

```java
public static void main(String[] args) throws IOException {
    // checking goes here
    JobConf conf = new JobConf();

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

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

    conf.setInputPath(new Path(args[0]));
    conf.setOutputPath(new Path(args[1]));

    JobClient.runJob(conf);
}
```
Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library (Java in Hadoop)
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)
Locality

• Master program divvies up tasks based on location of data: tries to have `map()` tasks on same machine as physical file data, or at least same rack
• `map()` task inputs are divided into 64 MB blocks: same size as Google File System chunks
Fault Tolerance

• Master detects worker failures
  – Re-executes completed & in-progress map() tasks
  – Re-executes in-progress reduce() tasks
• Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  – Effect: Can work around bugs in third-party libraries!
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
Fault Tolerance in MapReduce

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

• Critical for performance in large clusters: stragglers occur frequently due to failing hardware, bugs, misconfiguration, etc
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
Optimizations

• No reduce can start until map is complete:
  – A single slow disk controller can rate-limit the whole process
• Master redundantly executes *slow-moving* map tasks; uses results of first copy to finish

• *Why is it safe to redundantly execute map tasks? Wouldn’t this mess up the total computation?*
Optimizations

• **Combiner** functions can run on same machine as a mapper
• Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

• *Under what conditions is it sound to use a combiner?*
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

- 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  Map  Map  Map

the, 1 brown, 1 fox, 1

the, 2 fox, 1

how, 1 now, 1 brown, 1

Map  Map

 Reduce  Reduce

ate, 1 mouse, 1
cow, 1

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

ate, 1 cow, 1 mouse, 1 quick, 1
Suitable for your task if

- Have a cluster
- Working with large dataset
- Working with independent data (or assumed)
- Can be cast into *map* and *reduce*
MapReduce outside Google

- Hadoop (Java)
  - Emulates MapReduce and GFS
- The architecture of Hadoop MapReduce and DFS is master/slave

<table>
<thead>
<tr>
<th></th>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>jobtracker</td>
<td>tasktracker</td>
</tr>
<tr>
<td>DFS</td>
<td>namenode</td>
<td>datanode</td>
</tr>
</tbody>
</table>
One time setup

- set `hadoop-site.xml` and `slaves`
- Initiate namenode
- Run Hadoop MapReduce and DFS
- Upload your data to DFS
- Run your process...
- Download your data from DFS
MapReduce Conclusions

• MapReduce has proven to be a useful abstraction
• Greatly simplifies large-scale computations at Google
• Functional programming paradigm can be applied to large-scale applications
• Fun to use: focus on problem, let library deal with messy details
References

- Original OSDI’04 paper (http://labs.google.com/papers/mapreduce.html)
- On Wikipedia (http://en.wikipedia.org/wiki/MapReduce)
- Hadoop – MapReduce in Java (http://lucene.apache.org/hadoop/)
- Starfish – MapReduce in Ruby (http://rufy.com/starfish/)
Some more Examples
1. 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:**
  ```
  foreach word in text.split():
    output(word, filename)
  ```

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

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

- **hamlet.txt**
  - to be or not to be
  - to, hamlet.txt
  - be, hamlet.txt
  - or, hamlet.txt
  - not, hamlet.txt
- **12th.txt**
  - be not afraid of greatness
  - be, 12th.txt
  - not, 12th.txt
  - afraid, 12th.txt
  - of, 12th.txt
  - greatness, 12th.txt
- **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)
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
Getting started with Hadoop
Getting Started with Hadoop

• Download from hadoop.apache.org
• To install locally, unzip and set JAVA_HOME
• Details: 
  hadoop.apache.org/core/docs/current/quickstart.html

• Three ways to write jobs:
  – Java API
  – Hadoop Streaming (for Python, Perl, etc)
  – Pipes API (C++)
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);
        }
    }
}
Word Count in Java

```java
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));
    }
}
```
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)
```
Higher-level languages on top of Hadoop: Pig and Hive
Motivation

• MapReduce is great, as many algorithms can be expressed by a series of MR jobs

• But it’s low-level: must think about keys, values, partitioning, etc

• Can we capture common “job patterns”? 
Pig

• Started at Yahoo! Research
• Now runs about 30% of Yahoo!’s jobs
• Features:
  – Expresses sequences of MapReduce jobs
  – Data model: nested “bags” of items
  – Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  – Easy to plug in Java functions
  – Pig Pen dev. env. for Eclipse
An Example Problem

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited pages by users aged 18 - 25.

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
In MapReduce

In Pig Latin

Users = load ‘users’ as (name, age);
Filtered = filter Users by
    age >= 18 and age <= 25;
Pages = load ‘pages’ as (user, url);
Joined = join Filtered by name, Pages by user;
Grouped = group Joined by url;
Summed = foreach Grouped generate group,
    count(Joined) as clicks;
Sorted = order Summed by clicks desc;
Top5 = limit Sorted 5;

store Top5 into ‘top5sites’;

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
Ease of Translation

Notice how naturally the components of the job translate into Pig Latin.

- **Load Users**
- **Load Pages**
- **Filter by age**
- **Join on name**
- **Group on url**
- **Count clicks**
- **Order by clicks**
- **Take top 5**

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
Ease of Translation

Notice how naturally the components of the job translate into Pig Latin.

Job 1
- Load Users
  - Filter by age
    - Join on name
      - Users = load ...

Job 2
- Load Pages
  - Joined = join ...
    - Grouped = group ...
      - Summed = ... count()...

Job 3
- Count clicks
  - Sorted = order ...
    - Top5 = limit ...

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
Hive

• Developed at Facebook
• Used for majority of Facebook jobs
• “Relational database” built on Hadoop
  – Maintains list of table schemas
  – SQL-like query language (HQL)
  – Can call Hadoop Streaming scripts from HQL
  – Supports table partitioning, clustering, complex data types, some optimizations
Creating a Hive Table

CREATE TABLE page_views(viewTime INT, userid BIGINT, page_url STRING, referrer_url STRING, ip STRING COMMENT 'User IP address')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
STORED AS SEQUENCEFILE;

• Partitioning breaks table into separate files for each (dt, country) pair

   Ex: /hive/page_view/dt=2008-06-08,country=US
       /hive/page_view/dt=2008-06-08,country=CA
Simple Query

• Find all page views coming from xyz.com on March 31st:

```sql
SELECT page_views.*
FROM page_views
WHERE page_views.date >= '2008-03-01'
AND page_views.date <= '2008-03-31'
AND page_views.referrer_url like '%xyz.com';
```

• Hive only reads partition 2008-03-01,* instead of scanning entire table
Aggregation and Joins

• Count users who visited each page by gender:

```sql
SELECT pv.page_url, u.gender, COUNT(DISTINCT u.id)
FROM page_views pv JOIN user u ON (pv.userid = u.id)
GROUP BY pv.page_url, u.gender
WHERE pv.date = '2008-03-03';
```

• Sample output:

<table>
<thead>
<tr>
<th>page_url</th>
<th>gender</th>
<th>count(userid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home.php</td>
<td>MALE</td>
<td>12,141,412</td>
</tr>
<tr>
<td>home.php</td>
<td>FEMALE</td>
<td>15,431,579</td>
</tr>
<tr>
<td>photo.php</td>
<td>MALE</td>
<td>23,941,451</td>
</tr>
<tr>
<td>photo.php</td>
<td>FEMALE</td>
<td>21,231,314</td>
</tr>
</tbody>
</table>
Using a Hadoop Streaming Mapper Script

SELECT TRANSFORM(page_views.userid,
                   page_views.date)
USING 'map_script.py'
AS dt, uid CLUSTER BY dt
FROM page_views;
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)

• Hive and Pig further simplify programming

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