Large-scale Data Mining: MapReduce and beyond
Part 1: Basics

Spiros Papadimitriou, IBM Research
Jimeng Sun, IBM Research
Rong Yan, Facebook
Data everywhere

- Flickr (3 billion photos)
- YouTube (83M videos, 15 hrs/min)
- Web (10B videos watched / mo.)
- Digital photos (500 billion / year)
- All broadcast (70,000TB / year)
- Yahoo! Webmap (3 trillion links, 300TB compressed, 5PB disk)
- Human genome (2-30TB uncomp.)

So what??

more is:
more …

more is:
different!
Data everywhere

- **Opportunities**
  - Real-time access to content
  - Richer context from users and hyperlinks
  - Abundant training examples
  - “Brute-force” methods may suffice

- **Challenges**
  - “Dirtier” data
  - Efficient algorithms
  - Scalability (with reasonable cost)
“The Google Way”

“All models are wrong, but some are useful”
– George Box

“All models are wrong, and increasingly you can succeed without them.” – Peter Norvig

- Google PageRank
- Shotgun gene sequencing
- Language translation
- …
Getting over the marketing hype…

Cloud Computing

= Internet

+ Commoditization/

‘It’s what I and many others have worked towards our entire careers. It’s just happening now.’

– Eric Schmidt
This tutorial

- Is **not** about cloud computing
- **But** about large scale data processing

Data + Algorithms
Tutorial overview

- **Part 1 (Spiros): Basic concepts & tools**
  - MapReduce & distributed storage
  - Hadoop / HBase / Pig / Cascading / Hive

- **Part 2 (Jimeng): Algorithms**
  - Clustering (canopy, k-means)
  - Classification (k-NN, naïve Bayes)
  - Graph algorithms

- **Part 3 (Rong): Applications**
  - Text processing
  - Data warehousing
  - Machine learning
Outline

- Introduction
- MapReduce & distributed storage
  - Hadoop
    - HBase
    - Pig
    - Cascading
    - Hive
- Summary
What is MapReduce?

- Programming model?
- Execution environment?
- Software package?

“MapReduce” (this talk) == Distributed computation + distributed storage + scheduling / fault tolerance

It’s all of those things, depending who you ask...
Example – Programming model

mapper

```python
def getName (line):
    return line.split(‘\t’)[1]
```

reducer

```python
def addCounts (hist, name):
    hist[name] = \n    hist.get(name,default=0) + 1
    return hist
```

```python
input = open(‘employees.txt’, ‘r’)
```

```python
intermediate = map(getName, input)
```

```python
result = reduce(addCounts, \nintermediate, {})
```

Q: “What is the frequency of each first name?”
Example – Programming model

```python
def getName(line):
    return (line.split('t')[1], 1)
def addCounts(hist, (name, c)):
    hist[name] = hist.get(name, default=0) + c
    return hist

input = open('employees.txt', 'r')
intermediate = map(getName, input)
result = reduce(addCounts, intermediate, {})
```

**employees.txt**

<table>
<thead>
<tr>
<th>#</th>
<th>LAST</th>
<th>FIRST</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td>2</td>
<td>Brown</td>
<td>David</td>
<td>$70,000</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>George</td>
<td>$95,000</td>
</tr>
<tr>
<td>4</td>
<td>Yates</td>
<td>John</td>
<td>$80,000</td>
</tr>
<tr>
<td>5</td>
<td>Miller</td>
<td>Bill</td>
<td>$65,000</td>
</tr>
<tr>
<td>6</td>
<td>Moore</td>
<td>Jack</td>
<td>$85,000</td>
</tr>
<tr>
<td>7</td>
<td>Taylor</td>
<td>Fred</td>
<td>$75,000</td>
</tr>
<tr>
<td>8</td>
<td>Smith</td>
<td>David</td>
<td>$80,000</td>
</tr>
<tr>
<td>9</td>
<td>Harris</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Q: “What is the frequency of each first name?”

Key-value iterators
public class HistogramJob extends Configured implements Tool {

    public static class FieldMapper extends MapReduceBase implements 
        Mapper<LongWritable, Text, Text, LongWritable> {

        private static LongWritable ONE = new LongWritable(1);
        private static Text firstname = new Text();

        @Override
        public void map (LongWritable key, Text value, 
            OutputCollector<Text, LongWritable> out, Reporter r) {
            firstname.set(value.toString().split("\t")[1]);
            output.collect(firstname, ONE);
        }
    } // class FieldMapper

Example – Programming model
Sneak-peek: Hadoop
public static class LongSumReducer extends MapReduceBase implements Mapper<LongWritable, Text, Text, LongWritable> {

    private static LongWritable sum = new LongWritable();

    @Override
    public void reduce (Text key, Iterator<LongWritable> vals, OutputCollector<Text, LongWritable> out, Reporter r) {
        long s = 0;
        while (vals.hasNext())
            s += vals.next().get();
        sum.set(s);
        output.collect(key, sum);
    }
} // class LongSumReducer
Example – Programming model
Sneak-peek: Hadoop

```java
public int run (String[] args) throws Exception {
    JobConf job = new JobConf(getConf(), HistogramJob.class);
    job.setJobName("Histogram");
    job.setInputPath(args[0]);
    job.setMapperClass(FieldMapper.class);
    job.setCombinerClass(LongSumReducer.class);
    job.setReducerClass(LongSumReducer.class);
    JobClient.runJob(job);
    return 0;
} // run()

public static main (String[] args) throws Exception {
    ToolRunner.run(new Configuration(), new HistogramJob(), args);
} // main()
} // class HistogramJob
```

~ 30 lines = 25 boilerplate (Eclipse) + 5 actual code
MapReduce for…

- **Distributed clusters**
  - Google’s original
  - Hadoop (Apache Software Foundation)

- **Hardware**
  - SMP/CMP: Phoenix (Stanford)
  - Cell BE

- **Other**
  - Skynet (in Ruby/DRB)
  - QtConcurrent
  - BashReduce
  - …many more
Recap

<table>
<thead>
<tr>
<th>Quick-n-dirty script</th>
<th>vs</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>~5 lines of (non-boilerplate) code</td>
<td></td>
<td>Up to <em>thousands</em> of machines and drives</td>
</tr>
<tr>
<td>Single machine, local drive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is hidden to achieve this:

- Data partitioning, placement and replication
- Computation placement (and replication)
- Number of nodes (mappers / reducers)

As a programmer, you don’t *need* to know what I’m about to show you next...
Execution model: Flow

- Input file:
  - Smith John $90,000
  - Yates John $80,000
  - SPLIT 1
  - SPLIT 2
  - SPLIT 3

- Key/value iterators:
  - All-to-all, hash partitioning
  - Sort-merge
  - Sequential scan

- Output file:
  - PART 0
  - PART 1
Execution model: Placement

Computation co-located with data (as much as possible)
Execution model: Placement

HOST 0
- SPLIT 0: Replica 1/3
- SPLIT 1: Replica 2/3
- SPLIT 3: Replica 2/3
- Mapper: Replica 1/3

HOST 1
- SPLIT 0: Replica 2/3
- SPLIT 4: Replica 1/3
- Mapper: Replica 1/3
- Reducer: Replica 2/3

HOST 2
- SPLIT 3: Replica 3/3
- SPLIT 1: Replica 2/3
- SPLIT 4: Replica 2/3
- Mapper: Replica 2/3

HOST 3
- SPLIT 2: Replica 3/3
- SPLIT 1: Replica 1/3
- SPLIT 4: Replica 2/3
- Mapper: Replica 1/3

HOST 4
- SPLIT 2: Replica 3/3
- SPLIT 1: Replica 1/3

HOST 5
- SPLIT 0: Replica 3/3

HOST 6
- SPLIT 0: Replica 1/3
- SPLIT 1: Replica 2/3
- SPLIT 3: Replica 2/3
- Mapper: Replica 1/3
- Reducer: Replica 2/3

Rack/network-aware

COMBINER
MapReduce Summary

- Simple programming model
- Scalable, fault-tolerant
- Ideal for (pre-)processing large volumes of

‘However, if the data center is the computer, it leads to the even more intriguing question “What is the equivalent of the ADD instruction for a data center?” [...] If MapReduce is the first instruction of the “data center computer”, I can’t wait to see the rest of the instruction set, as well as the data center programming language, the data center operating system, the data center storage systems, and more.’

Outline

- Introduction
- MapReduce & distributed storage

- Hadoop
  - HBase
  - Pig
  - Cascading
  - Hive

- Summary
Hadoop’s stated mission (Doug Cutting interview):
Commoditize infrastructure for web-scale, data-intensive applications
Who uses Hadoop?

- Yahoo!
- Facebook
- Last.fm
- Rackspace
- Digg
- Apache Nutch

... more in part 3
Filesystems and I/O:
- Abstraction APIs
- RPC / Persistence
Cross-language serialization:
- RPC / persistence
- ~ Google ProtoBuf / FB Thrift

Hadoop

- Core
- Avro
Hadoop

Distributed execution (batch)
- Programming model
- Scalability / fault-tolerance

MapReduce
HDFS
ZooKeeper
Core
Avro
Hadoop

Distributed storage (read-opt.)
- Replication / scalability
- ~ Google filesystem (GFS)
Hadoop

- Coordination service
  - Locking / configuration
  - ~ Google Chubby

- HBase
- MapReduce
- HDFS
- Core
- Avro
- ZooKeeper
Hadoop

- HBase
- Pig
- Hive
- Chukwa

Column-oriented, sparse store
- Batch & random access
- ~ Google BigTable
Hadoop

Data flow language
- Procedural SQL-inspired lang.
- Execution environment
Hadoop

- HBase
- Pig
- Hive
- Chukwa
- Avro
- Core
- ZooKeeper

Distributed data warehouse
- SQL-like query language
- Data mgmt / query execution
Hadoop

- HBase
- Pig
- Hive
- Chukwa
- MapReduce
- HDFS
- ZooKeeper
- Core
- Avro

... ... more
MapReduce

- **Mapper**: \((k_1, v_1) \rightarrow (k_2, v_2)[\] \)
  - E.g., \((\text{void}, \text{textline} : \text{string}) \)
    \(\rightarrow (\text{first} : \text{string}, \text{count} : \text{int})\)

- **Reducer**: \((k_2, v_2[]) \rightarrow (k_3, v_3)[\] \)
  - E.g., \((\text{first} : \text{string}, \text{counts} : \text{int[]} \)
    \(\rightarrow (\text{first} : \text{string}, \text{total} : \text{int})\)

- **Combiner**: \((k_2, v_2[]) \rightarrow (k_2, v_2)[\] \)

- **Partition**: \((k_2, v_2) \rightarrow \text{int}\)
Mapper interface

```java
interface Mapper<K1, V1, K2, V2> {
    1  void configure (JobConf conf);
    2  void map (K1 key, V1 value,
                  OutputCollector<K2, V2> out,
                  Reporter reporter);
    3  void close();
}
```

- Initialize in `configure()`
- Clean-up in `close()`
- Emit via `out.collect(key,val)` anywhere
Reducer interface

```java
interface Reducer<K2, V2, K3, V3> {
  void configure (JobConf conf);
  void reduce (K2 key, Iterator<V2> values, OutputCollector<K3, V3> out, Reporter reporter);
  void close();
}
```

- Initialize in `configure()`
- Clean-up in `close()`
- Emit via `out.collect(key, val)` anywhere
Some canonical examples

- Histogram-type jobs:
  - Graph construction (bucket = edge)
  - K-means et al. (bucket = cluster center)

- Inverted index:
  - Text indices
  - Matrix transpose

- Sorting

- Equi-join

- More details in part 2
Equi-joins
“Reduce-side”

(Smith, 7)
(Jones, 7)
(Brown, 7)
(Davis, 3)
(Dukes, 5)
(Black, 3)
(Gruhl, 7)

(Sales, 3)
(Devel, 7)
(Acct., 5)

7: (□, (Smith))
-OR-
(7,□): (Smith)

7: (□, (Devel))
-OR-
(7,□): (Devel)
Equi-joins

“Reduce-side”

(Smith, 7)
(Jones, 7)
(Brown, 7)
(Davis, 3)
(Dukes, 5)
(Black, 3)
(Gruhl, 7)

(Sales, 3)
(Devel, 7)
(Acct., 5)

7: (Smith, 7)
7: (Jones, 7)
7: (Brown, 7)
7: (Gruhl, 7)

7: (Smith, Devel),
(Jones, Devel),
(Brown, Devel),
(Gruhl, Devel)
HDFS & MapReduce processes
Hadoop Streaming & Pipes

Don’t have to use Java for MapReduce

Hadoop Streaming:
- Use stdin/stdout & text format
- Any language (C/C++, Perl, Python, shell, etc)

Hadoop Pipes:
- Use sockets & binary format (more efficient)
- C++ library required
Outline

- Introduction
- MapReduce & distributed storage
- Hadoop
  - HBase
  - Pig
  - Cascading
  - Hive
- Summary
HBase introduction

- MapReduce canonical example:
  - Inverted index (more in Part 2)

- Batch computations on large datasets:
  - Build static index on crawl snapshot

- However, in reality crawled pages are:
  - Updated by crawler
  - Augmented by other parsers/analytics
  - Retrieved by cache search
  - Etc…
HBase introduction

- MapReduce & HDFS:
  - Distributed storage + computation
  - Good for batch processing
  - But: no facilities for accessing or updating individual items

- HBase:
  - Adds random-access read / write operations
  - Originally developed at Powerset
  - Based on Google’s Bigtable
**HBase data model**

- **Row**: (billions; sorted)
- **Column**: (millions)
- **Column family**: (hundreds; fixed)

| Keys and cell values are arbitrary byte arrays |
| Can use any underlying data store (local, HDFS, S3, etc) |
| Partitioned over many nodes (thousands) |
Data model example

<table>
<thead>
<tr>
<th>empId</th>
<th>profile:last</th>
<th>profile:first</th>
<th>profile:salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>John</td>
<td>$90,000</td>
<td></td>
</tr>
</tbody>
</table>
**Data model example**

<table>
<thead>
<tr>
<th>profile: last</th>
<th>profile: first</th>
<th>profile: salary</th>
<th>bm: url1</th>
<th>bm: urlN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>John</td>
<td>$90,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Always access via primary key.

**Diagram:**

- **Profile:** family
- **Bookmarks:** (bookmarks) family

**Primary Key:** empId
HBase vs. RDBMS

- Different solution, similar problems
- RDBMSes:
  - Row-oriented
  - Fixed-schema
  - ACID
- HBase et al.:
  - Designed from ground-up to scale out, by adding commodity machines
  - Simple consistency scheme: atomic row writes
  - Fault tolerance
  - Batch processing
  - No (real) indexes
Outline

- Introduction
- MapReduce & distributed storage
- Hadoop
  - HBase
  - Pig
  - Cascading
  - Hive
- Summary
Pig introduction

“ ~5 lines of *non-boilerplate* code ”

Writing a single MapReduce job requires significant gruntwork

- Boilerplates (mapper/reducer, create job, etc)
- Input / output formats

Many tasks require more than one MapReduce job
Pig main features

- Data structures (multi-valued, nested)
- Pig-latin: data flow language
  - SQL-inspired, but imperative (not declarative)
Pig example

```pig
records = LOAD filename AS (last: chararray, first: chararray, salary: int);
grouped = GROUP records BY first;
counts = FOREACH grouped GENERATE group, COUNT(records.first);
DUMP counts;
```

Q: “What is the frequency of each first name?”

<table>
<thead>
<tr>
<th>#</th>
<th>LAST</th>
<th>FIRST</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td>2</td>
<td>Brown</td>
<td>David</td>
<td>$70,000</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>George</td>
<td>$95,000</td>
</tr>
<tr>
<td>4</td>
<td>Yates</td>
<td>John</td>
<td>$80,000</td>
</tr>
<tr>
<td>5</td>
<td>Miller</td>
<td>Bill</td>
<td>$65,000</td>
</tr>
<tr>
<td>6</td>
<td>Moore</td>
<td>Jack</td>
<td>$85,000</td>
</tr>
<tr>
<td>7</td>
<td>Taylor</td>
<td>Fred</td>
<td>$75,000</td>
</tr>
<tr>
<td>8</td>
<td>Smith</td>
<td>David</td>
<td>$80,000</td>
</tr>
<tr>
<td>9</td>
<td>Harris</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Pig schemas

- Schema = tuple data type

- Schemas are optional!
  - Data-loading step is not required
  - "Unknown" schema: similar to AWK ($0, $1, ..)

- Support for most common datatypes
- Support for nesting
Pig Latin feature summary

- Data loading / storing
  - LOAD / STORE / DUMP

- Filtering
  - FILTER / DISTINCT / FOREACH / STREAM

- Group-by
  - GROUP

- Join & co-group
  - JOIN / COGROUP / CROSS

- Sorting
  - ORDER / LIMIT

- Combining / splitting
  - UNION / SPLIT
Outline

- Introduction
- MapReduce & distributed storage
- Hadoop
  - HBase
  - Pig
  - Cascading
  - Hive
- Summary
Cascading introduction

- Provides higher-level abstraction
  - Fields, Tuples
  - Pipes
  - Operations
  - Taps, Schemes, Flows
- Ease composition of multi-job flows

Very flexible library, not a new language
Cascading example

Scheme srcScheme = new TextLine();
Tap source = new Hfs(srcScheme, "filename");
Scheme dstScheme = new TextLine();
Tap sink = new Hfs(dstScheme, "filename", REPLACE);

Pipe assembly = new Pipe("lastnames");

Function splitter = new RegexSplitter(
    new Fields("last", "first", "salary"), "\t");
assembly = new Each(assembly, new Fields("line"), splitter);

assembly = new GroupBy(assembly, new Fields("first"));

Aggregator count = new Count(new Fields("count"));
assembly = new Every(assembly, count);

FlowConnector flowConnector = new FlowConnector();
Flow flow = flowConnector.connect("last-names",
    source, sink, assembly);
flow.complete();

Q: “What is the frequency of each first name?”

<table>
<thead>
<tr>
<th>#</th>
<th>LAST</th>
<th>FIRST</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Smith</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td></td>
<td>Brown</td>
<td>David</td>
<td>$70,000</td>
</tr>
</tbody>
</table>
Cascading feature summary

- **Pipes**
  - Each
  - GroupBy / CoGroup
  - Every
  - SubAssembly

- **Operations**
  - Function
  - Filter
  - Aggregator / Buffer
Outline

- Introduction
- MapReduce & distributed storage
- Hadoop
  - HBase
  - Pig
  - Cascading
  - Hive
- Summary
Hive introduction

- Originally developed at Facebook
  - Now a Hadoop sub-project

- Data warehouse infrastructure
  - Execution: MapReduce
  - Storage: HDFS files

- **Large** datasets, e.g. Facebook daily logs
  - 30GB (Jan’08), 200GB (Mar’08), 15+TB (2009)

- Hive QL: SQL-like query language
Hive example

CREATE EXTERNAL TABLE records
  (last STRING, first STRING, salary INT)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION filename;

SELECT records.first, COUNT(1)
FROM records
GROUP BY records.first;

employees.txt

<table>
<thead>
<tr>
<th>#</th>
<th>LAST</th>
<th>FIRST</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td>2</td>
<td>Brown</td>
<td>David</td>
<td>$70,000</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>George</td>
<td>$95,000</td>
</tr>
<tr>
<td>4</td>
<td>Yates</td>
<td>John</td>
<td>$80,000</td>
</tr>
<tr>
<td>5</td>
<td>Miller</td>
<td>Bill</td>
<td>$65,000</td>
</tr>
<tr>
<td>6</td>
<td>Moore</td>
<td>Jack</td>
<td>$85,000</td>
</tr>
<tr>
<td>7</td>
<td>Taylor</td>
<td>Fred</td>
<td>$75,000</td>
</tr>
<tr>
<td>8</td>
<td>Smith</td>
<td>David</td>
<td>$80,000</td>
</tr>
<tr>
<td>9</td>
<td>Harris</td>
<td>John</td>
<td>$90,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Q: “What is the frequency of each first name?”
Hive schemas

- Data should belong to tables
  - But can also use pre-existing data
  - Data loading optional (like Pig) but encouraged

- Partitioning columns:
  - Mapped to HDFS directories
  - E.g., (date, time) → datadir/2009-03-12/18_30_00

- Data columns (the rest):
  - Stored in HDFS files

- Support for most common data types
- Support for pluggable serialization
Hive QL feature summary

- **Basic SQL**
  - FROM subqueries
  - JOIN (only equi-joins)
  - Multi GROUP BY
  - Multi-table insert
  - Sampling

- **Extensibility**
  - Pluggable MapReduce scripts
  - User Defined Functions
  - User Defined Types
  - SerDe (serializer / deserializer)
Outline

- Introduction
- MapReduce & distributed storage
- Hadoop
  - HBase
  - Pig
  - Cascading
  - Hive
- Summary
Recap

- Scalable: all
- High(-er) level: all except MR
- Existing language: MR, Cascading
- “Schemas”: HBase, Pig, Hive, (Casc.)
  - Pluggable data types: all
- Easy transition: Hive, (Pig)
Related projects

Higher level—computation:
- Dryad & DryadLINQ (Microsoft) [EuroSys 2007]
- Sawzall (Google) [Sci Prog Journal 2005]

Higher level—storage:
- Bigtable [OSDI 2006] / Hypertable

Lower level:
- Kosmos Filesystem (Kosmix)
- VSN (Parascale)
- EC2 / S3 (Amazon)
- Ceph / Lustre / PanFS

- Sector / Sphere ([http://sector.sf.net/](http://sector.sf.net/))
- Map-Reduce-Merge
- …
Summary

MapReduce:
- Simplified parallel programming model

Hadoop:
- Built from ground-up for:
  - Scalability
  - Fault-tolerance
  - Clusters of commodity hardware
- Growing collection of components and extensions (HBase, Pig, Hive, etc)
Tutorial overview

- Part 1 (Spiros): Basic concepts & tools
  - MapReduce & distributed storage
  - Hadoop / HBase / Pig / Cascading / Hive

- Part 2 (Jimeng): Algorithms
  - Clustering (canopy, k-means)
  - Classification (k-NN, naïve Bayes)
  - Graph algorithms

- Part 3 (Rong): Applications
  - Text processing
  - Data warehousing
  - Machine learning