Midterm

P1: Crawling
- (10 pt) circular for consistent hashing + replica for load balancing
- Hashing on hostname, reassign existing URLs upon machine removal, rate limiting per machine
- ".edu"!

P2: Segmentation
- (10 pt) applying statistical methods
- Pseudo-code, discussing batch and live evaluation

P3: Index Compression ➔ byte-aligned is decodable.

Grades are released: mean is 73, median is 75.

Outline

Big Data

MapReduce Techniques

Case Studies
- Document Inverted Indexing
- Query Log Cube Analysis

Fallacies of Big Data
Handling Large Scale Data is Necessary

Google: 20PB per day processed by MapReduce

Facebook: 2.5 PB user data, growing 15TB per day in 2009

Petabytes data are the norm for Internet companies.

Scientific studies generate similar amount of data

• Sloan Digital Sky Survey: 200GB per night
• Human Genome Project: 3GB of raw data per individual

Leveraging Large Scale Data is Powerful

Translation: analyzing parallel texts (sentences written in two languages in parallel) at a large scale drastically improves the accuracy of automatic translation.

Google Trends: analyzing search queries can predict flu spread more accurately and faster than CDC in 2009.

Big Data vs Cloud Computing

Big Data / Data Science: techniques for storing, analyzing, mining large scale, fast moving, heterogeneous data

• Volume, Velocity, Variety, Veracity

Cloud Computing: hosting various services for remote customers

• Infrastructure, Platform, Software, Data

IaaS: Virtual Machines

• Amazon EC2

PaaS: IaaS+OS+Servers+Runtime ➔ Developer facing

• Cloud Foundry, Google App Engine, Microsoft Azure

SaaS and DaaS: PaaS+Applications+Data ➔ User facing

• Google Apps

Big data techniques are usually integrated at the level of PaaS.
The MapReduce Paper

MapReduce: Simplified Data Processing on Large Clusters, by Jeff Dean and Sanjay Ghemawat, OSDI 2004.

Built upon decades of research in parallel and distributed computing
Demonstrated, to the large audience, the power of shared-nothing parallel processing architecture
Extremely valuable to data processing on the Web, essentially launched the Big Data era.

MapReduce now refers to both the Google implementation and the programming model with open and proprietary implementations
• Hadoop, Dryad
• Latest advancements: Spark (iterative), Kafka (streaming), TensorFlow

Intuition and Infrastructure

Divide and Conquer

• Divide the large amount of input data into smaller chunks
• Process each chunk separately with shared-nothing environment, and produce intermediate outputs from each chunk
• Aggregate those intermediate outputs into the final output

A MapReduce stack provides the infrastructure for:
• Retrieving input from and writing output to the distributed storage system
• Data processing:
  • Running user provided code for processing data chunks
  • Shuffling intermediate outputs to their destinations
  • Running user provided code for aggregating intermediate outputs to produce the final output
• A master for monitoring failures and restarting failed tasks
• A scheduling system for coordination among tasks and jobs

Key Ideas

Offline batch processing

Lots of cheap machines is better than a few high end machines
Failures are inevitable, as a consequence of the above
Moving data is more expensive than moving computation
As simple as possible for developers
But not suitable for all data processing tasks
• If possible, adjust the tough tasks to suit the MapReduce model
Basic Concepts

Mapper
- Workers that execute map tasks, each of which processes one individual input chunk
- The Map function is provided by the user
  - input record \( \rightarrow (\text{Key}, \text{Value}), (\text{Key}, \text{Value}), \ldots \)

Reducer
- Workers that execute reduce tasks, each of which aggregates intermediate output records of the same Key
- The Reduce function is provided by the user
  - \((\text{Key}, (\text{Value}_1, \text{Value}_2, \ldots)) \rightarrow (\text{Key}, \text{Result})\)

Document Word Count: Simple Flow

```
class MAPPER
  method MAP(docid a, doc d)
    for all term t \in doc d do
      Emit(term t, count 1)
  end method

class REDUCER
  method REDUCE(term t, counts [c_1, c_2, \ldots])
    sum = 0
    for all count c \in counts [c_1, c_2, \ldots] do
      sum \leftarrow sum + c
    Emit(term t, count sum)
  end method
```
The infrastructure provides the critical layer of shuffle/sort
• Performance differentiator between implementations

**Shuffle** (i.e. Partition)
• Partitions the intermediate output based on Key
• Assigns each partition to specific reducers

**Sort**, for each partition
• Sorts the records based on Key
• Groups values with the same Key together
• Each reducer can expect the records come in sorted by the Key
• There is no guarantee of order across the different reducers

User can provide customized partition and sort functions to control the shuffle/sort behavior

**In-Class Exercise**
Write a MapReduce program to compute the *average per document term frequency* over the documents they appear in for all terms
Notable Issues

Reduce workers can not start until all map workers are done:
- Speculative execution can speed things up in anticipation of mapper failures: i.e., run duplicate copies of same map task if it is running slowly

Straggling reducers:
- When data distribution is skewed, a single Key may have lots of Values
- The reducer being assigned that task becomes the job bottleneck

Failure handling:
- Master polls each worker periodically and considers one has failed if no response within certain amount of time
- All tasks (even those completed b/c master doesn’t know) executed by that worker will be reset and executed by a new worker.
- All reduce workers will be notified to read from the new worker

Outline

Big Data

MapReduce Techniques

Case Studies
- Document Inverted Indexing
- Query Log Cube Analysis

Fallacies of Big Data
Local Aggregation

In the most basic MapReduce model, aggregations only happen on the reducers

- All the intermediate data must be shipped (often over the wire) from the mapper to the remote reducer

Sometimes, it is much more efficient to execute the reduce tasks partially on the mapper to avoid the extra communication cost

- Helps alleviate the straggling reducer problem, although does not eliminate it

Local Aggregation within Mapper

```
1: class Mapper
2:     method INITIALIZE
3:         H ← new AssociativeArray
4:     method MAP(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H(t) ← H(t) + 1
7:     method CLOSE
8:         for all term t ∈ H do
9:             Emit(term t, count H(t))
```

Mapper can easily run out of memory, important to flush the data in blocks

Local Aggregation through Combiner

Important notes on combiners

- Not guaranteed to be executed and may get executed multiple times
- Must process the intermediate output in the same way as reducers
- Must produce the intermediate output that can be fed into the reducers
Algebraic Measure and Combiner

Let \( A \) and \( B \) be two different sets of numbers

\[
\text{AVERAGE}(A \cup B) \neq \text{AVERAGE}(\text{AVERAGE}(A), \text{AVERAGE}(B))
\]

- \( \text{AVERAGE}([1, 2, 3, 4, 5]) = 3 \)
- \( \text{AVERAGE}([1, 2], \text{AVERAGE}([3, 4, 5])) = 2.75 \)

\[
\text{SUM(SUM}(A), \text{SUM}(B)) / \text{SUM(COUNT}(A) + \text{COUNT}(B))
\]

- \( \text{SUM(SUM}([1, 2], \text{SUM}([3, 4, 5])) = 15 \)
- \( \text{SUM(COUNT}([1, 2] + \text{COUNT}([3, 4, 5])) = 5 \)

Measures like sum, count, average are algebraic measures that lend themselves to optimization through combiner

Using a Combiner

```java
class COMBINER {
  method COMBINE(string t, integers \{ r_1, r_2, \ldots \}) {
    sum = 0
    end = 0
    for all integer r in integers \{ r_1, r_2, \ldots \} do
      sum = sum + r
    end
    Emit(string t, integer end)
  }
}
```

Using a Combiner, Correctly

```java
class MAPPER {
  method MAP(docid a, doc d) {
    H = new AssociativeArray
    for all term t \in doc d do
      H\{t\} = H\{t\} + 1
    end
    for all term t \in H do
      Emit(term t, count(H\{t\}))
    end
  }
}
```
Thought Exercise

How to write a MapReduce program that computes the number of times each pair of words co-occurs in the same sentence?

• I.e., "new york university", "new york city"
• {"new", "york"} = 2, {"new", "university"} = 1, etc.

Pairs and Stripes

There is a fundamental tension in most MapReduce algorithms on two aspects

• Number of intermediate output records
• Size of intermediate output records

Pair-based approaches aim to reduce the latter

Stripe-based approaches aim to reduce the former

The Pair Approach

```java
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term w ∈ doc d do
4:       for all term u ∈ NEIGHBORS(w) do
5:         Emit(pair (w, u), count 1)
1: class REDUCER
2:   method REDUCE(pair p, counts [c1, c2, . . .])
3:     s ← 0
4:     for all count c ∈ counts [c1, c2, . . .] do
5:       s ← s + c
6:     Emit(pair p, count s)
```
Pair versus Stripe

There is no right way, it all depends on the problem

The Pair approach
• Each intermediate output record is small
• There are a lot of them ⇒ all possible word pair combinations

The Stripe approach
• The number of intermediate output records is limited: # of distinct words
  • Each record can be very large ⇒ an array of many words and their counts
  • Can cause straggling reducer problem

If applicable, the Stripe approach is slightly preferred
• Less burden on shuffle and sort due to the lesser number of records
• Also moves slightly smaller amount of data due to savings on the Keys

Neither solves the straggling reducer problem

Thought Exercise

Write a MapReduce program that computes Pr(w_j|w_i)
• How likely w_j will appear if w_i appears (say in a sentence):

\[
f(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w' \in W} N(w_i, w')}
\]

Straight-forward using the Stripe approach
• In the Reduce function, the total count is readily available

How to do that using the Pair approach?
Customized Partition and Sort

A Customized Partition function on the key can be applied to guide how the shuffle phase distributes intermediate records
- All records with the same partition ID go to the same reducer

A Customized Sort function on the Key can be applied to determine the order of intermediate records being processed by the reducer

Computing \(P_r(w_1 | w_2)\) using the Pair Approach

In addition to word pairs, for each word in a sentence, output

\((w, \_) \Rightarrow \#\text{ of all words}\)

Customized Partition function to hash on only the first word

Customized Sort function to sort \("\) before any real word
- Using stateful Reduce function

\[
\begin{array}{|c|c|}
\hline
\text{key} & \text{values} \\
\hline
dog, \_ & (8927, 8314, \ldots) \\
\text{dog, wordbook} & (2, 1) \\
\text{dog, worddef} & (1) \\
\hline
\end{array}
\]

Design Principles for MapReduce Algorithms

Carefully design the Key and Value data structures to work with Partition and Sort functions

Apply Combiner whenever possible

Be aware of potential data skews and the straggling reducer problem

Chain multiple MapReduce jobs together to accomplish complex tasks
- Check out Flume, Pig, Dryad
Outline

Big Data

MapReduce Techniques

Case Studies

- Document Inverted Indexing (very simple)
- Query Log Cube Analysis (advanced)

Fallacies of Big Data

MapReduce and Web Search Engines

MapReduce was designed to handle two important tasks that are at the heart of Web Search Engines

- Document processing and indexing at Web Scale
- Mining search query logs

Big data techniques became necessary because of the enormous amount of data

- It’s not a coincident that MapReduce was developed at Google

Inverted Indexing via MapReduce: Basic

```java
1. class MAPPER
2. procedure MAP(dcid n, doc d)
3. H = new ASSOCIATIVE ARRAY
4. for all term t \in\ doc \ do
5. H(t) = H(t) + 1
6. for all term t \in\ H \ do
7. EMT(t, t, posting (n, H(t)))

1. class REDUCER
2. procedure REDUCE(term t, postings [(n1, f1), (n2, f2) ...])
3. P = new LIST
4. for all posting (n, f) \in\ postings [(n1, f1), (n2, f2) ...] \ do
5. APPEND P, (n, f)
6. SORT(P)
7. EMT(t, postings P)
```
Inverted Indexing using MapReduce: Basic

Any Issue?

The in-memory sort in the Reduce function can be costly

Apply the techniques we learned earlier

- Push docid to be part of the Key, i.e., <term, docid>
- Customize the Partition function to partition on term only
- Customize the Sort function to sort term first and docid second

Then the MapReduce execution framework will automatically perform the much more scalable distributed sort

This value-to-key strategy is often used to leverage the inherent sorting facility within MapReduce

Inverted Indexing via MapReduce: No User Sort

Inverted Indexing using MapReduce: Basic
Take-Home Exercise
Compute PageRank via MapReduce

Outline
Big Data
MapReduce Techniques
Case Studies
• Document Inverted Indexing (very simple)
• Query Log Cube Analysis (advanced)
Fallacies of Big Data

Cube Analysis of Query Logs
Search engines accumulate huge query logs
• Each query event has a lot of information
  • Location (inferred from IP)
  • Demographics (inferred from past behavior)
  • The query itself (mapped to semantic categories)

Used to identify interesting facts across all combinations of dimensions
• E.g. “Women in 40s from Michigan search a lot about knitting.”

Over an extended period, there can be a lot of query events
Distributed Cube Materialization

Distributed Cube Materialization on Holistic Measures, by Arnab Nandi, Cong Yu, Philip Bohannon, Raghu Ramakrishnan. ICDE, 2011.

A brief look at an advanced usage of MapReduce

Techniques proposed in the paper are incorporated into the official Hadoop implementation.

Holistic Measures

Algebraic measures
• Value for parent can be computed easily from values of children.
• \( \text{COUNT}(A \cup B) = \text{SUM}(\text{COUNT}(A), \text{COUNT}(B)) \)

Holistic Measures
• Value for parent can not be computed from children.
• \( \text{COUNT}(\text{DISTINCT}(A \cup B)) \)
  • \( \text{distinct}(A) = \{a, b, b, c\} = 3; \text{distinct}(B) = \{a, c\} = 2; \)
  • \( \text{distinct}(A \cup B) = 3 \) can not be computed any other way

Real Example Task

Given the location and topic hierarchies, compute volume and reach (number of distinct users) of all cube groups whose reach is at least 5.

<table>
<thead>
<tr>
<th>User Query (Event)</th>
<th>Time</th>
<th>Topic</th>
<th>Location</th>
<th>Country</th>
<th>State</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1 Nikon d40</td>
<td>20090601</td>
<td>SLR</td>
<td>Ann Arbor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u2 Canon sd100</td>
<td>20090601</td>
<td>Point &amp; Shoot</td>
<td>Detroit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u1 Apple iPhone</td>
<td>20090602</td>
<td>Smartphone</td>
<td>Ann Arbor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u3 Nikon d40</td>
<td>20090604</td>
<td>SLR</td>
<td>New York</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u4 Nokia 1100</td>
<td>20090605</td>
<td>Basic Phone</td>
<td>Sunnyvale</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In particular, the function and distributes them for computation using the MapReduce framework. Intuitively, this naive algorithm divides the full cubing task into many small tasks. Let $R$ represents this combined cube computation. The ideal case is to combine all the queries into a single MapReduce job. However, this is beyond the scope of this work. We provide some discussion on how to optimize the naive algorithm to reduce the time to finish. The measure function is applied to all cube groups. Highlight those cube groups that share less than $5$ of the input values.

### Naive Algorithm

#### Algorithm 1 Naive Algorithm

**MAP**($e$)
1. # $e$ is a tuple in the data
2. let $C$ be the Cube Lattice
3. for each Region $R$ in $C$
   4. do $k = R(e)$;
   5. Emit $k \Rightarrow e$

**REDUCE**($k$, $(e_1, e_2, ...)\}$)
1. let $M$ be the measure function
2. Emit $k \Rightarrow M((e_1, e_2, ...))$

### Naive Algorithm

<table>
<thead>
<tr>
<th>User Query</th>
<th>Event Time</th>
<th>Topic</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>Nikon d40</td>
<td>20090601 SLR</td>
<td>Ann Arbor</td>
</tr>
<tr>
<td>u2</td>
<td>Canon sx100</td>
<td>20090601 Point &amp; Shoot</td>
<td>Detroit</td>
</tr>
<tr>
<td>u1</td>
<td>Apple iPhone</td>
<td>20090602 Smartphone</td>
<td>Ann Arbor</td>
</tr>
<tr>
<td>u3</td>
<td>Nikon d40</td>
<td>20090604 SLR</td>
<td>New York</td>
</tr>
<tr>
<td>u4</td>
<td>Nokia 1100</td>
<td>20090605 Basic Phone</td>
<td>Sunnyvale</td>
</tr>
</tbody>
</table>

Example 1 -

Example 2 -

Typically, the query optimizer would then select the measure function. Measure all frequent queries for each region in the cube, including the smallest and the broadest group.

The schemata are shown in Fig. 4 and Fig. 5. The style specification corresponding to the cubing task for this example is presented in Table 6.

### Coverage Analysis

** cubing task for this example is presented in Table 6.**

### A. Size of Intermediate Data

For algebraic measures, this challenge can be addressed by using a different set of data. We note again that cube computation is only part of both naive and topic hierarchies. Compute and hierarchical cube lattice are shown in Fig. 4 and Fig. 5.

### Coverage Analysis

** Cubing task for this example is presented in Table 6.**

### Naive Algorithm

**MAP**($e$)
1. # $e$ is a tuple in the data
2. let $C$ be the Cube Lattice
3. for each Region $R$ in $C$
   4. do $k = R(e)$;
   5. Emit $k \Rightarrow e$

**REDUCE**($k$, $(e_1, e_2, ...)\}$)
1. let $M$ be the measure function
2. Emit $k \Rightarrow M((e_1, e_2, ...))$

### Coverage Analysis

** Cubing task for this example is presented in Table 6.**

### Naive Algorithm

**MAP**($e$)
1. # $e$ is a tuple in the data
2. let $C$ be the Cube Lattice
3. for each Region $R$ in $C$
   4. do $k = R(e)$;
   5. Emit $k \Rightarrow e$

**REDUCE**($k$, $(e_1, e_2, ...)\}$)
1. let $M$ be the measure function
2. Emit $k \Rightarrow M((e_1, e_2, ...))$
Challenges

**Map phase:** many intermediate keys
- Number of Keys = |C| x |D|, where |C| is the number of regions in the lattice, and |D| is the input size.
- As the number of dimension increases, this can cause problems for the map/shuffle phase.

**Reduce phase:** extremely large reduce groups
- Some reduce shards can be very large and cause the reducer dealing with those groups to run out of memory.
- In fact, one group is guaranteed to have this problem: <*, *>

Handle Straggling Reducer

Inspired by a common method to compute the unique number of users in a query log
- The usual practice involves two steps: 1) MapReduce to get the set of unique user IDs; 2) calculate the size of result to produce the final answer

Holistic measure computation can not be distributed, but
- What if the input is split into disjoint sets with regard to the attribute of the measure (in the above example, user ID)?

Example

<table>
<thead>
<tr>
<th>User</th>
<th>Query (Event)</th>
<th>Time</th>
<th>Topic</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>Nikon d40</td>
<td>20090601</td>
<td>SLR</td>
<td>Ann Arbor</td>
</tr>
<tr>
<td>u2</td>
<td>Canon sd100</td>
<td>20090601</td>
<td>Point &amp; Shoot</td>
<td>Detroit</td>
</tr>
<tr>
<td>u3</td>
<td>Apple iPhone</td>
<td>20090603</td>
<td>Smartphone</td>
<td>Ann Arbor</td>
</tr>
<tr>
<td>u4</td>
<td>Nikon d40</td>
<td>20090603</td>
<td>SLR</td>
<td>New York</td>
</tr>
<tr>
<td>u5</td>
<td>Nokia 1100</td>
<td>20090603</td>
<td>Basic Phone</td>
<td>Sunnyvale</td>
</tr>
</tbody>
</table>

Reduce phase

- <*, *, uid> ➔ {u1}
- <*, *, uid> ➔ {u2}
- ...
- <*, *, uid> ➔ {u5}

no more large groups!
Detecting Large Cube Groups

Run a small MapReduce job over a sample of the data using the naïve algorithm

• Goal is to detect any cube group that might be reducer-unfriendly

With a sample size of 2M tuples, we can obtain accurate enough estimation for 20B tuples

• Proof based on Chernoff Bounds

Slightly Less Naïve Approach

<*,*,*,uid>
<*,*,topic,uid>
<*,*,cat,uid>
<*,country,*,uid>
<*,country,topic>
<*,state,*,uid>
<*,state,topic>
<*,state,cat>
<*,city,*,uid>
<*,city,topic>
<*,city,cat>
<*,country,subcat>
<*,*,subcat>

producing even more intermediate records!

Incorporating Partition Factor
Experimental Setup

Yahoo! Hadoop Cluster
- Hadoop 0.20, 2x4core 2.5GHz Xeon/8GB RAM/4x1TB
- 4GB reserved for the workers
- Python + Hadoop Streaming

Datasets:
- A sample of real click log: 516M tuples, 3 dims, 6 levels

Baselines:
- Naive
- MR-BPP and MR-PT, adapted from earlier works by Ng et. al., You et. al., Sergey et. al.
Outline
Big Data
MapReduce Techniques
Case Studies
  • Document Inverted Indexing (very simple)
  • Query Log Cube Analysis (advanced)

Fallacies of Big Data

Why Big Data Analysis Can be Wrong
Errors in traditional statistical analysis
  • Sampling error
  • Sampling bias

Sampling error: a randomly chosen sample may not reflect the total population
  • Big Data addresses this issue \(\rightarrow\) larger sample, less error

Sampling bias: the samples are not randomly chosen
  • Depending on the analysis, Big Data may make this worse!
  • Only solution is to gather all data points, which may be impossible

E.g., Facebook/twitter population is very different from real life, see US Election 2016!

The Best Practice
Be well versed in statistical techniques
Be aware of sampling bias and avoid drawing conclusions purely based on ‘found’ data
Be more hypothesis driven and test ideas with experiments
  • With the large volume of data, exploratory analysis inevitably lead to some interesting patterns that are simply there by chance
  • Cross-validation is critical
Review of Today’s Material

- Introduction to Big Data, Cloud Computing
- Various MapReduce techniques
- Two case studies relevant to core tasks of search engines
- What to pay attention to when doing big data analysis