More MapReduce

Jimmy Lin
The iSchool
University of Maryland
Tuesday, March 31, 2009

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details

Today’s Topics
- MapReduce algorithm design
  - Managing dependencies
  - Coordinating mappers and reducers
- Case study #1: statistical machine translation
- Case study #2: pairwise similarity comparison
- Systems integration
  - Back-end and front-end processes
  - RDBMS and Hadoop/HDFS
- Hadoop "nuts and bolts"

MapReduce Algorithm Design

Managing Dependencies
- Remember: Mappers run in isolation
  - You have no idea in what order the mappers run
  - You have no idea on what node the mappers run
  - You have no idea when each mapper finishes
- Tools for synchronization:
  - Ability to hold state in reducer across multiple key-value pairs
  - Sorting function for keys
  - Partitioner
  - Cleverly-constructed data structures

Motivating Example
- Term co-occurrence matrix for a text collection
  - $M = N \times N$ matrix ($N =$ vocabulary size)
  - $M_{ij}$: number of times $i$ and $j$ co-occur in some context
    (for concreteness, let say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

"You shall know a word by the company it keeps" (Firth, 1957)

MapReduce: Large Counting Problems
- Term co-occurrence matrix for a text collection
  - specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of events (the collection itself)
  - Goal: keep track of interesting statistics about the events
- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?
First Try: “Pairs”

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit \((a, b) \rightarrow count\)
- Reducers sum up counts associated with these pairs
- Use combiners!

Note: in all my slides, I denote a key-value pair as \(k \rightarrow v\)

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)

Another Try: “Stripes”

- Idea: group together pairs into an associative array
  - \((a, b) \rightarrow 1\)
  - \((a, c) \rightarrow 2\)
  - \((a, d) \rightarrow 5\)
  - \((a, e) \rightarrow 3\)
  - \((a, f) \rightarrow 2\)
  - \(a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}\)
- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit \(a \rightarrow \{ b: count_b, c: count_c, d: count_d, \ldots \}\)
- Reducers perform element-wise sum of associative arrays
  - \(a \rightarrow \{ b: 1, d: 5, e: 3 \}\)
  - \(a \rightarrow \{ b: 1, c: 2, d: 2, f: 2 \}\)
  - \(a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}\)

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object is more heavyweight
  - Fundamental limitation in terms of size of event space

Conditional Probabilities

- How do we compute conditional probabilities from counts?
  - \(P(B\mid A) = \frac{count(A,B)}{count(A)} = \sum_{B'} \frac{count(A,B')}{count(A)}\)
- Why do we want to do this?
- How do we do this with MapReduce?
P(B|A): “Pairs”

(a, *) → 32
(a, b1) → 3  (a, b1) → 3 / 32
(a, b2) → 12  (a, b2) → 12 / 32
(a, b3) → 7  (a, b3) → 7 / 32
(a, b4) → 1  (a, b4) → 1 / 32
...

Reducer holds this value in memory

For this to work:
- Must emit extra (a, *) for every b_i in mapper
- Must make sure all a’s get sent to same reducer (use partitioner)
- Must make sure (a, *) comes first (define sort order)
- Must hold state in reducer across different key-value pairs

P(B|A): “Stripes”

a → (b_1:3, b_2:12, b_3:7, b_4:1, ...)

Easy!
- One pass to compute (a, *)
- Another pass to directly compute P(B|A)

Synchronization in Hadoop

- Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
  - Illustrated by the “pairs” approach
- Approach 2: construct data structures that “bring the pieces together”
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the “stripes” approach

Issues and Tradeoffs

- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
  - De/serialization overhead
- Combiners make a big difference!
  - RAM vs. disk and network
  - Arrange data to maximize opportunities to aggregate partial results

Questions?

Case study #1:
statistical machine translation
Statistical Machine Translation
Chris Dyer (Ph.D. student, Linguistics)
Aaron Cordova (undergraduate, Computer Science)
Alex Mont (undergraduate, Computer Science)

- Conceptually simple:
  (translation from foreign \( f \) into English \( e \))

\[
\hat{e} = \arg \max_{e} P(f | e)P(e)
\]

- Difficult in practice!

- Phrase-Based Machine Translation (PBMT):
  - Break up source sentence into little pieces (phrases)
  - Translate each phrase individually

MT Architecture

The Data Bottleneck

HMM Alignment: Giza
What's the point?

- The optimally-parallelized version doesn’t exist!
- It’s all about the right level of abstraction
  - Goldilocks argument

Questions?

Case study #2: pairwise similarity comparison

Applications:
- Clustering
- Cross-document coreference resolution
- “more-like-that” queries
Problem Description

- Consider similarity functions of the form:
  \[ \text{sim}(d_i, d_j) = \sum_{w} W_{d_i} W_{d_j} \]
  But, actually...
  \[ \text{sim}(d_i, d_j) = \sum_{w} W_{d_i} W_{d_j} \]

- Two step solution in MapReduce:
  1. Build inverted index
  2. Compute pairwise similarity from postings

Building the Inverted Index

- Obama Clinton Obama
- Obama McCain
- Obama Bush Clinton
- Obama Clinton

Computing Pairwise Similarity

- Obama 1
- McCain 2
- Bush 1
- Clinton 1

Analysis

- Main idea: access postings once
  - O(df) pairs are generated
  - MapReduce automatically keeps track of partial weights

- Control effectiveness-efficiency tradeoff by dfCut
  - Drop terms with high document frequencies
  - Has large effect since terms follow Zipfian distribution

Questions?
**Systems Integration**

**Issue #1: Front-end and back-end processes**

**Front-end**
- Real-time
- Customer-facing
- Well-defined workflow

**Back-end**
- Batch
- Internal
- Ad hoc analytics

---

**Typical “Scale Out” Strategies**

- LAMP stack as standard building block
- Lots of each (load balanced, possibly virtualized):
  - Web servers
  - Application servers
  - Cache servers
  - RDBMS
- Reliability achieved through replication
- Most workloads are easily partitioned
  - Partition by user
  - Partition by geography
  - ...
Database layer: 800 eight-core Linux servers running MySQL (40 TB user data)

Caching servers: 15 million requests per second, 95% handled by memcache (15 TB of RAM)

**BigTable: Google does databases**

- **What is it?** A sparse, distributed, persistent multidimensional sorted map
- **Why not a traditional RDBMS?**
  - Map is indexed by a row key, column key, and a timestamp
  - Each value in the map is an uninterpreted array of bytes
    
    \[
    \text{(row:}\text{string, column:}\text{string, time:}\text{int64}) \to \text{string}
    \]
  - Layout:
    - Rows sorted lexicographically
    - Columns grouped by “column family”
  - Atomic read/writes on rows (only)

**Typical Workflows**

- Large volume reads/writes
- Scans over subset of rows
- Random reads/writes

```
<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>“com.cnnsi.com”</td>
<td>“anchor:cnnsi.com”</td>
</tr>
<tr>
<td>“com.cnnsi.com”</td>
<td>“anchor:mylook.ca”</td>
</tr>
<tr>
<td>“com.cnnsi.com”</td>
<td>“anchor:mylook.ca”</td>
</tr>
<tr>
<td>“com.cnnsi.com”</td>
<td>“anchor:mylook.ca”</td>
</tr>
</tbody>
</table>
```
Explicit Choices

- Does not support a relational model
  - No table-wide integrity constraints
  - No multi-row transactions
  - No joins (!)
- Simple data model
  - Let client applications control data layout
  - Take advantage of locality:
    - rows (key sort order) and column (column family)

The Bad News…

Rows vs. Columns

The $19$ billion question. 

*Size of the DBMS market in 2008, as estimated by IDC.
It's all about data flows!

MapReduce

What if you need...

Join, Union  Split  Chains

... and filter, projection, aggregates, sorting, distinct, etc.

Current solution?

Example: Find the top 10 most visited pages in each category

Visits

<table>
<thead>
<tr>
<th>User</th>
<th>Url</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>cnn.com</td>
<td>8:00</td>
</tr>
<tr>
<td>Amy</td>
<td>bbc.com</td>
<td>10:00</td>
</tr>
<tr>
<td>Amy</td>
<td>flickr.com</td>
<td>10:05</td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com</td>
<td>12:00</td>
</tr>
</tbody>
</table>

Url Info

<table>
<thead>
<tr>
<th>Url</th>
<th>Category</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnn.com</td>
<td>News</td>
<td>0.9</td>
</tr>
<tr>
<td>bbc.com</td>
<td>News</td>
<td>0.8</td>
</tr>
<tr>
<td>flickr.com</td>
<td>Photos</td>
<td>0.7</td>
</tr>
<tr>
<td>espn.com</td>
<td>Sports</td>
<td>0.9</td>
</tr>
</tbody>
</table>

visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts, 10);
store topUrls into '/data/topUrls';
Load Visits

Group by url

Foreach url generate count

Load Url Info

Map1

Reduce1

Map2

Reduce2

Reduce3

Join on url

Group by category

Foreach category generate top10(urls)

Reduce2

Map3

Reduce3

Pig Slides adapted from Olston et al. (SIGMOD 2008)

Relationship to SQL?

SQL

Pig

MapReduce

Hadoop “nuts and bolts”

Hadoop Zen

- Don’t get frustrated (take a deep breath)...
  - Remember this when you experience those W3*#T@F! moments
- This is bleeding edge technology:
  - Lots of bugs
  - Stability issues
  - Even lost data
  - To upgrade or not to upgrade (damned either way)?
  - Poor documentation (or none)
- But… Hadoop is the path to data nirvana

Source: http://davidzinger.wordpress.com/2007/05/page/2/

Hadoop Zen

- Don’t get frustrated (take a deep breath)...
  - Remember this when you experience those W3*#T@F! moments
- This is bleeding edge technology:
  - Lots of bugs
  - Stability issues
  - Even lost data
  - To upgrade or not to upgrade (damned either way)?
  - Poor documentation (or none)
- But… Hadoop is the path to data nirvana

Source: http://davidzinger.wordpress.com/2007/05/page/2/
Data Types in Hadoop

- **Writable**: Defines a de/serialization protocol. Every data type in Hadoop is a Writable.
- **WritableComparable**: Defines a sort order. All keys must be of this type (but not values).
- **IntWritable**: Concrete classes for different data types.
- **LongWritable**: ...
- **Text**: ...

Complex Data Types in Hadoop

- **How do you implement complex data types?**
  - **The easiest way**:
    - Encoded it as Text, e.g., (a, b) = “a:b”
    - Use regular expressions to parse and extract data
    - Works, but pretty hack-ish
  - **The hard way**:
    - Define a custom implementation of WritableComparable
    - Must implement: readFields, write, compareTo
    - Computationally efficient, but slow for rapid prototyping
  - **Alternatives**:
    - Cloud9 offers two other choices: Tuple and JSON

Questions?