Mapreduce and (in) Search

Amund Tveit (PhD)

amund@atbrox.com

http://atbrox.com
TOC

1. Brief about my background
2. Mapreduce Overview
3. Mapreduce in Search
4. Advanced Mapreduce Example
Brief about my background

- Now: Co-founder of Atbrox
  - Search & Mapreduce
- PhD in Computer Science
  - http://amundtveit.info/publications/
- Past: Googler for 4 years:
  - Cluster Infrastructure
  - Nordic Search (and Maps) Quality
  - Google News for iPhone
Details: http://no.linkedin.com/in/amundtveit

- Interested in how mapreduce is used (algorithmic patterns)
  - http://mapreducepatterns.org
  - Working on projects using mapreduce in search and other large-scale problems
- Passionate about search and search technology

- Less known trivia:
  - shared office with your professor back in 2000
Part 1

mapreduce
What is Mapreduce?

Mapreduce is a concept and method for typically batch-based large-scale parallelization. It is inspired by functional programming's `map()` and `reduce()` functions.

Nice features of mapreduce systems include:
- reliably processing job even though machines die
- parallization en-large, e.g. thousands of machines for terasort

Mapreduce was invented by the Google fellows below:

http://labs.google.com/papers/mapreduce.html

Jeff D.            Sanjay G.
Mapper

- Processes one key and value pair at the time, e.g.
- **word count**
  - map(key: uri, value: text):
    - for word in tokenize(value)
    - emit(word, 1) // found 1 occurrence of word
- **inverted index**
  - map(key: uri, value: text):
    - for word in tokenize(value)
    - emit(word, key)
Reducers processes one key and all values that belong to it, e.g.

- **word count**
  - reduce(key: word type, value: list of 1s):
    - emit(key, sum(value))
- **inverted index**
  - reduce(key: word type, value: list of URIs):
    - emit(key, value) // e.g. to a distr. hash table
Combiner

Combiner functions is the subset of reducer functions where reducer functions can be written as recurrence equations, e.g. 
\[ \text{sum}_{n+1} = \text{sum}_n + x_{n+1} \]

This property happens (surprisingly) often and can be used to speed up mapreduce jobs (dramatically) by putting the combiner function as an "afterburner" on the map functions tail.

But sometimes, e.g. for advanced machine learning algorithms reducers are more advanced and can't be used as combiners
Shuffler - the silent leader

When the mapper emits a key, value pair - the shuffler does the actual work in shipping it to the reducer, the addressing is a function of the key, e.g.

chosen reducer = hash(key) % num reducers

but could basically be any routing function.

Q. When is changing the shuffle function useful?
A. When data distributed is skewed, e.g. according to Zip's law. Examples of this are stop words in text (overly frequent) and skewness in sales numbers (where bestsellers massively outnumber items in the long tail)
Part 2

mapreduce in search
What is important in search?

1. **Precision**
   - results match query but primarily user intent

2. **Recall**
   - not missing important results

3. **Freshness**
   - timely recall, e.g. for news

4. **Responsiveness**
   - time is money, but search latency is minus-money

5. **Ease-of-use**
   - few input widgets and intelligent query understanding

6. **Safety**
   - not providing malware or web spam
Recap: What is Search?

1. Getting data
2. Processing data
3. Indexing data
4. Searching data
5. Search Quality (maintain and improve)
6. Ads Quality (which pays for the fun)

**But:**
- not necessarily in that order
- and with feedback loops

searching might lead to getting usage data used in both processing and indexing
2005 - Google and Yahoo competing with:
  - ~20 Billion \((20 \times 10^9)\) documents in their indices

2008 - Google seeing 1 trillion \((1 \times 10^{12})\) simultaneously existing unique urls
  - \(\Rightarrow\) PetaBytes (i.e. thousands of harddisks)

Bibliography:
- [http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html](http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html)
- [http://www.chato.cl/research/crawling_thesis](http://www.chato.cl/research/crawling_thesis)
Detect:
- exact duplicates
- near-duplicates
- scraped/combined content

with mapreduce:

Naively $O(N^2)$ problem - compare all $N$ documents with each other, but can be pruned by:
- comparing only document which share substrings
- compare only documents of similar size
mapreduce job 1:

map\( (\text{key: uri, value: text}) \)
create hashes (shingles) of substrings of content
foreach hash in hashes: emit(\(\text{key, hash}\))

reduce\( (\text{key: uri, values: list of hashes}) \)
size = number of hashes in values
outputvalue = concat(\(\text{key, size}\))
foreach hash in hashes: emit(\(\text{hash, outputvalue}\))

mapreduce job 2:

map\( (\text{key: hash, value: uri-numhashes}) \)
emit(\(\text{key, value}\))

reduce\( (\text{key: hash, value: list of uri-numhashes}) \)
emit all unique pairs of uri-numhash in value
// e.g. (uri0-4, uri1-7), (uri0-4, uri13-5), ..
mapreduce job 3:

map(key: uri-numhashes, value: uri-numhashes)
emit(key, value)

reduce(key: uri-numhashes, value: list of uri-numhashes)
// e.g. values for key uri0-4 could be
// uri1-7, uri1-7, uri13-5, uri33-7
for each unique uri-numhash h in values
    alpha = count of h occurrences in values
    doc_similarity = alpha/(key's numhashes + h's numhashes - alpha)
    output_key = key's uri concatenated with h's uri
emit(output_key, doc_similarity)

References:
E.g. a sequence of structurally similar `map(reduce)` steps where one updates a package with new fields:

- **Content Normalization** *(to text and remove boilerplate)*
  - `map(key: URI, value: rawcontent)`
    - create new `documentpackage`
    - `documentpackage.setRawContent(rawcontent)`
    - `documentpackage.setText(stripBoilerPlate(ToText(rawContent)))`
    - `emit(key, documentpackage)`

- **Entity Extraction**
  - `map(key: URI, value: documentpackage)`
    - names = `findPersonNameEntities(documentpackage.getText())`
    - `documentpackage.setNames(names)`
    - `emit(key, documentpackage)`

- **Sentiment Analysis**
  - `map(key: URI, value: documentpackage)`
    - `sentimentScore = calculateSentiment(documentpackage.getText())`
    - `documentpackage.setSentimentScore(sentimentScore)`
    - `emit(key, documentpackage)`
Creating a simple hypothetical ranking signal with mapreduce:
=> ratio of vowels per page

map(key: URI, value: documentpackage)
    numvowels, numnonvowels = 0
    for each character in documentpackage.getText():
        if isWowel(character)
            ++numvowels
        else
            ++numnonvowels
    vowelratio = numvowels / (numvowels+numnonvowels)
documentpackage.setVowelRatio( vowelratio)
emit(key, documentpackage)
Inverted index among the "classics" of mapreduce example, it resembles word count but outputs the URI instead of occurrence count

\textbf{map}(key: URI, value: text)  
for word in tokenize(value)  
emit(word, key)  

\textbf{reduce}(key: wordtype, value: list of URIs)  
output(key, value) // e.g. to a distributed hash
How about positional information?

map(key: URI, value: text)
    wordposition = 0
    foreach word in tokenize(value)
        outputvalue = concat(key, wordposition)
        ++wordposition
        emit(word, outputvalue)

reduce(key: wordtype, value: list of URI combined with word position)
    create positional index for the given word (key) in the format
    output(key, newvalue) // e.g. to a distributed hash
How about bigram or n-gram indices?

**map**(key: URI, value: text)
- bigramposition = 0
- foreach bigram in bigramtokenizer(value)
  - outputvalue = concat(key, wordposition)
  - ++bigramposition
  - emit(bigram, outputvalue)

**reduce**(key: bigramtype, value: list of URI combined with bigram position)
create positional bigram index for the given bigram (key) in the format
output(key, newvalue) // e.g. to a distributed hash
How about indexing of stems or synonyms?

\[
\text{map}(\text{key}: \text{URI}, \text{value}: \text{text}) \\
\quad \text{foreach word in tokenizer(value)} \\
\quad \quad \text{synonyms} = \text{findSynonyms}(\text{word}) \\
\quad \quad \text{stem} = \text{createStem}(\text{word}) \\
\quad \quad \text{emit}(\text{word}, \text{key}) \\
\quad \quad \text{emit}(\text{stem}, \text{key}) \\
\quad \quad \text{foreach synonym in synonyms} \\
\quad \quad \quad \text{emit}(\text{synonym}, \text{key})
\]

\[
\text{reduce}(\text{key}: \text{wordtype}, \text{value}: \text{list of URI}) \\
\quad \text{output}(\text{key}, \text{value}) // \text{e.g. to a distributed hash}
\]
A lot of things typically happens at search time, the query is "dissected", classified, interpreted and perhaps rewritten before querying the index. This can generate clickthrough log data where we can find most clicked-on uris:

**mapreduce job 1**

map\( (\text{key}: \text{query}, \text{value}: \text{URI clicked on}) \)

emit\( (\text{value}, \text{key}) \)

reduce\( (\text{key}: \text{URI clicked on}, \text{value}: \text{list of queries}) \)

emit\( (\text{count(\text{value})}, \text{key}) \) // (count, query)

**mapreduce job 2**

map\( (\text{key}: \text{uri count}, \text{value}: \text{uri}) \)

emit\( (\text{key}, \text{value}) \) // (count, query)

reduce\( (\text{key}: \text{uri count}, \text{value}: \text{list of uris}) \)

emit\( (\text{key}, \text{value}) \)

// generate list of uris per clickthrough count
MR+Search: Searching data II

Or we can find which the number of clicks per query (*note: queries with relatively few clicks probably might have issues with ranking*)

**mapreduce job 1**

**map** (key: query, value: URI clicked on)
emit(value, key)

**reduce** (key: URI clicked on, value: list of queries)
foreach unique query q in value // get all clicks per query for this uri
outputvalue = count number of occurrences (clicks) of q in value
emit(q, outputvalue)

**mapreduce job 2**

**map** (key: query, value: count)
outputvalue = sum(value)
emit(key, value)

**reduce** (key: query, value: list of counts)
emit(sum(value), key)

// get all clickthroughs per query for all urls
MR+Search: Quality

Changing how processing and indexing mapreduce jobs work is likely to effect search quality (e.g. precision and recall), this can be evaluated with mapreduce, e.g. comparing the old and new set of indices by running querying both set of indices with the same (large) query log and compare differences in results.

Task: how will you do that with mapreduce?

References:
Ads are commercial search results, they should have similar requirements to relevancy as "organic" results, but have less text "to rank with" themselves (~twitter tweet size or less), but fortunately a lot of metadata (about advertiser, landing pages, keywords bid for etc.) that can be used to measure, improve and predict their relevancy.

References:

- [http://web2py.iiit.ac.in/publications/default/download/techreport.pdf.a373bbf4a5b76063_4164436c69636b5468726f7567685261746549494954485265706f72742e706466.pdf](http://web2py.iiit.ac.in/publications/default/download/techreport.pdf.a373bbf4a5b76063_4164436c69636b5468726f7567685261746549494954485265706f72742e706466.pdf)
- [http://pages.stern.nyu.edu/~narchak/wfp0828-archak.pdf](http://pages.stern.nyu.edu/~narchak/wfp0828-archak.pdf)
MR+Search: Other Topics

Plenty of mapreduce+search related topics haven't been covered, but here are some to consider looking at:

- machine translation
- clustering
- graph algorithms
- spam/malware detection
- personalization

References:
- [https://cwiki.apache.org/confluence/display/MAHOUT/Algorithms](https://cwiki.apache.org/confluence/display/MAHOUT/Algorithms)
  - [http://www.youtube.com/watch?v=1ZDybXI212Q](http://www.youtube.com/watch?v=1ZDybXI212Q) (clustering)
  - [http://www.youtube.com/watch?v=BT-piFBP4fE](http://www.youtube.com/watch?v=BT-piFBP4fE) (graphs)
Part 3

advanced mapreduce example
This gives an example of how to port a machine learning classifier to mapreduce with discussion about optimizations.

**The Classifier:** Tikhonov Regularization with a square loss function (*this family includes:* proximal support vector machines, ridge regression, shrinkage regression and regularized least-squares classification)

\[(\omega, \gamma) = (\frac{1}{\mu} + E^T E)^{-1} (E^T D e)\]

- **D** - a matrix of training classes, e.g. \([-1.0, 1.0, 1.0, .. ]\)
- **A** - a matrix with feature vectors, e.g. \([2.9, 3.3, 11.1, 2.4], .. ]\)
- **e** - a vector filled with ones, e.g \([1.0, 1.0, .., 1.0]\)
- **E = [A -e]**
- **mu** = scalar constant # used to tune classifier
- **D** - a diagonal matrix with -1.0 or +1.0 values (depending on the class)
How to classify an incoming feature vector $\mathbf{x}$

Class = $\mathbf{x}^T \omega - \gamma$

The classifier expression in Python (numerical python):

$$(\omega, \gamma) = \left( \frac{I}{\mu} + \mathbf{E}_T \mathbf{E} \right) \mathbf{I} \left( \mathbf{E}_T \mathbf{D} \mathbf{e} \right)$$

The expression have a (nice) additive property such that:

$$(\omega, \gamma) = \left( \frac{I}{\mu} + \mathbf{E}_1^T \mathbf{E}_1 + \mathbf{E}_2^T \mathbf{E}_2 \right) \mathbf{I} \left( \mathbf{E}_1^T \mathbf{D}_1 \mathbf{e} + \mathbf{E}_2^T \mathbf{D}_2 \mathbf{e} \right)$$

With induction this becomes:

$$(\omega, \gamma) = \left( \frac{I}{\mu} + \mathbf{E}_1^T \mathbf{E}_1 + \ldots + \mathbf{E}_i^T \mathbf{E}_i \right) \mathbf{I} \left( \mathbf{E}_1^T \mathbf{D}_1 \mathbf{e} + \ldots + \mathbf{E}_i^T \mathbf{D}_i \mathbf{e} \right)$$
Brief Recap

D - a matrix of training classes, e.g.
\[
\begin{bmatrix}
-1.0, 1.0, 1.0, & ..
\end{bmatrix}
\]
A - a matrix with feature vectors, e.g.
\[
\begin{bmatrix}
2.9, 3.3, 11.1, 2.4, & ..
\end{bmatrix}
\]
e - a vector filled with ones, e.g [1.0, 1.0, .., 1.0]
E = [A -e]

mu = scalar constant # used to tune classifier

A and D represent distributed training data, e.g. spread out on many machines or on distributed file system. Given the additive nature of the expression we can parallelize the calculation of E.T*E and E.T*De
Adv. Mapreduce Example IV

\[ (\omega, \gamma) = \frac{1}{\mu} + E.T^*E).I^*(E.T^*D^*e) \]
def map(key, value):
    # input key= class for one training example, e.g. "-1.0"
    classes = [float(item) for item in key.split(',')]  # e.g. [-1.0]
    D = numpy.diag(classes)

    # input value = feature vector for one training example, e.g. "3.0, 7.0, 2.0"
    featurematrix = [float(item) for item in value.split(',')]  
    A = numpy.matrix(featurematrix)

    # create matrix E and vector e
    e = numpy.matrix(numpy.ones(len(A)).reshape(len(A),1))
    E = numpy.matrix(numpy.append(A,-e,axis=1))

    # create a tuple with the values to be used by reducer
    # and encode it with base64 to avoid potential trouble with '\t' and '\n' used
    # as default separators in Hadoop Streaming
    producedvalue = base64.b64encode(pickle.dumps( (E.T*E, E.T*D*e) )

    # note: a single constant key "producedkey" sends to only one reducer
    # somewhat "atypical" due to low degree of parallelism on reducer side
    print "producedkey\t%s" % (producedvalue)
def reduce(key, values, mu=0.1):
    sumETE = None
    sumETDe = None

    # key isn't used, so ignoring it with _ (underscore).
    for _, value in values:
        # unpickle values
        ETE, ETDe = pickle.loads(base64.b64decode(value))
        if sumETE == None:
            # create the I/mu with correct dimensions
            sumETE = numpy.matrix(numpy.eye(ETE.shape[1])/mu)
            sumETE += ETE
        if sumETDe == None:
            # create sumETDe with correct dimensions
            sumETDe = ETDe
        else:
            sumETDe += ETDe

    if sumETDe == None:
        # create sumETDe with correct dimensions
        sumETDe = ETDe
    else:
        sumETDe += ETDe

    # note: omega = result[:-1] and gamma = result[-1]
    # but printing entire vector as output
    result = sumETE.I*sumETDe
    print "%s\t%s" % (key, str(result.tolist()))
Mapper Increment Size really makes a difference

E.T*E and E.T*D*e given to reducer are independent of number of training data(!)

==> put as much training data in each E.T*E and E.T*D*e package

Example:
- Assume 1000 mappers with 1 billion training examples each (web scale :) and 100 features per training example
- If putting all billion examples into one E.T*E and E.T*D*e package
  - reducer needs to summarize 1000 101x101 matrices (not bad)
- Or if sending 1 example per E.T*E and E.T*D*e package
  - reducer needs to summarize 1 trillion (10^12) 101x101 matrices (intractable)
Adv. Mapreduce Example VIII

Avoid stressing the reducer

\[
\begin{align*}
A_0, D_0 & \quad \cdots \quad A_{n-1}, D_{n-1} \\
E_0.T^*E_0, E_0.T^*D_0e_0 & \quad \cdots \quad E_{n-1}.T^*E_{n-1}, E_{n-1}.T^*D_{n-1}e_{n-1} \\
\text{(partial matrix sums)} & \quad \cdots \quad \text{(partial matrix sums)} \\
1 & \quad \text{Reducer} \\
\text{(omega, gamma)} = (I/\mu + E.T^*E).I^*(E.T^*D^*e) & \\
\end{align*}
\]
Part 4

landing
Mapreduce Patterns

Map() and Reduce() methods typically follow patterns, a recommended way of grouping code with such patterns is:

extracting and generalize fingerprints based on:

- **loops**: e.g. "do-while", "while", "for", "repeat-until" => "loop"
- **conditions**: e.g. "if" and "switch" => "condition"
- **emits**
- **emit data types** (if available)

the map() method for both word count and index would be:

```java
map(key, value)
  loop
    emit(key:string, value:string)
```
How to start with mapreduce?

Either download hadoop (comes with mapreduce):

- [http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html](http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html)

Or use Amazon Elastic Mapreduce (cloud service):

Summary

- Have given a brief introduction to:
  - mapreduce
  - mapreduce in search
  - in-depth mapreduce example (classification)

- Any questions, Comments?
- Further reading:
  - [http://mapreducepatterns.org](http://mapreducepatterns.org)
    - algorithms with mapreduce
  - [http://atbrox.com/about/](http://atbrox.com/about/)
    - misc. mapreduce blog posts (by me)
- Wanting to work with mapreduce?
  - amund@atbrox.com