CSCI 5417
Information Retrieval Systems
Jim Martin

Lecture 9
9/20/2011

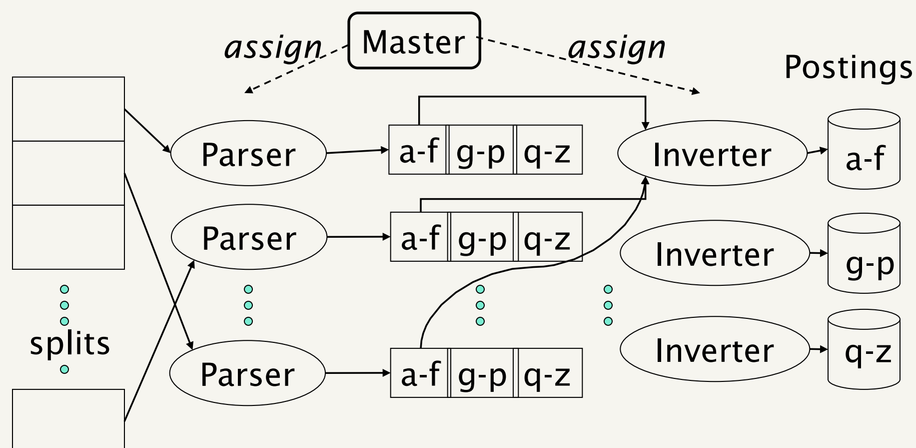
Today 9/20

- Where we are
- MapReduce/Hadoop
- Probabilistic IR
  - Language models
  - LM for ad hoc retrieval

Where we are...

- Basics of ad hoc retrieval
  - Indexing
  - Term weighting/scoring
    - Cosine
  - Evaluation
- Document classification
- Clustering
- Information extraction
- Sentiment/Opinion mining

But First: Back to Distributed Indexing



Huh?

- That was supposed to be an explanation of MapReduce (Hadoop)...
- Maybe not so much...
- Here's another try

MapReduce

- MapReduce is a distributed programming framework that is intended to facilitate applications that are
  - Data intensive
  - Parallelizable in a certain sense
  - In a commodity-cluster environment
- MapReduce is the original internal Google model
- Hadoop is the open source version

Inspirations

- MapReduce elegantly and efficiently combines inspirations from a variety of sources, including
  - Functional programming
  - Key/value association lists
  - Unix pipes

Functional Programming

- The focus is on side-effect free specifications of input/output mappings
- There are various idioms, but map and reduce are two central ones...
  - Mapping refers to applying an identical function to each of the elements of a list and constructing a list of the outputs
  - Reducing refers to receiving the elements of a list and aggregating the elements according to some function.

Python Map/Reduce

- Say you wanted to compute simple sum of squares of a list of numbers

$$\sum_{i=0}^{n} w_i^2$$

```
>>> z
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> z2 = map(lambda x: x**2, l)
>>> z2
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> reduce(lambda x,y: x+y, z2)
285
>>> reduce(lambda x,y: x+y, map(lambda x: x**2, z))
285
```

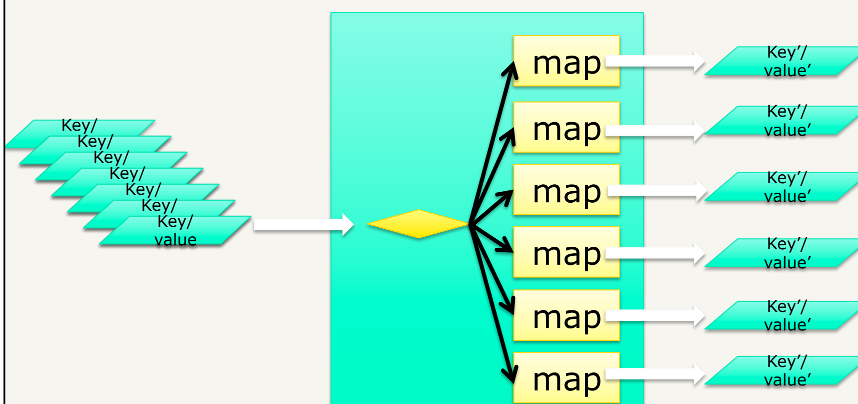
Association Lists (key/value)

- The notion of association lists goes way back to early lisp/ai programming. The basic idea is to try to view problems in terms of sets of key/value pairs.
  - Most major languages now provide first-class support for this notion (usually via hashes on keys)
- We've seen this a lot this semester
  - Tokens and term-ids
  - Terms and document ids
  - Terms and posting lists
  - Docids and tf/idf values
  - Etc.

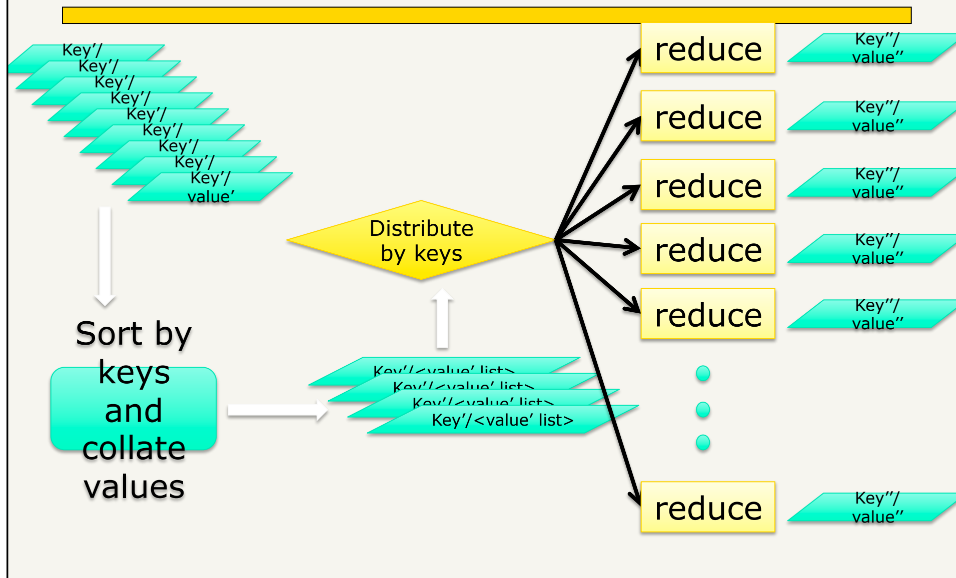
MapReduce

- MapReduce combines these ideas in the following way
- There are two phases of processing mapping and reducing. Each phase consists of multiple identical copies of map and reduce methods
  - Map methods take individual key/value pairs as input and return some function of those pairs to produce a new key/value pair
  - Reduce methods take key/<list of values> pairs as input, and return some aggregate function of the values as an answer.

Map Phase



Reduce Phase



Example

- Simple example used in all the tutorials
  - *Get the counts of each word type across a bunch of docs*
    - Let's assume each doc is a big long string

For map
Input: Filenames are keys; content string is values
Output: Term tokens are keys; values are 1's

For reduce
Input: Terms tokens are keys, 1's are values
Output: Term types are keys, summed counts are values

Dumbo Example

```
def map(docid, contents):
    for term in contents.split():
        yield term, 1

def reduce(term, counts):
    sum = 0
    for count in counts:
        sum = sum + count
    yield term, sum
```

Key
Value
Key
Value

Hidden Infrastructure

- Partitioning the incoming data
  - Hadoop has default methods
    - By file, given a bunch of files
      - <filename, contents>
    - By line, given a file full of lines
      - <line #, line>
- Sorting/collating the mapped key/values
- Moving the data among the nodes
  - Distributed file system
    - Don't move the data; just assign mappers/reducers to nodes

Example 2

- Given our normal postings
  - term -> list of (doc-id, tf) tuples
- Generate the vector length normalization for each document in the index

$$\sqrt{\sum_{t \in d} w_{t,d}^2}$$

- Map
  - Input: terms are keys, posting lists are values
  - Output: doc-ids are keys, squared weights are values
- Reduce
  - Input: doc-ids are keys, list of squared weights are values
  - Output: doc-ids are keys, square root of the summed weights are the values

Example 2

```
def map(term, postings):
    for post in postings:
        yield post.docID(), post.weight() ** 2

def reduce(docID, sqWeights):
    sum = 0
    for weight in sqWeights:
        sum = sum + weight
    yield docID, math.sqrt(sum)
```

Break

- Thursday we’ll start discussion of projects. So come to class ready to say something about projects.
- Part 2 of the HW is due next Thursday
  - Feel free to mail me updated results (R-precisions) as you get them...

Probabilistic pproaches.

- The following is a mix of chapters 12 and 13.
- Only the material from 12 will be on the quiz

An Alternative

- Basic vector space model uses a geometric metaphor/framework for the ad hoc retrieval problem
  - One dimension for each word in the vocab
  - Weights are usually tf-idf based
- An alternative is to use a probabilistic approach
  - So we'll take a short detour into probabilistic language modeling

Using Language Models for ad hoc Retrieval

- Each document is treated as (the basis for) a language model.
- Given a query q
- Rank documents based on $P(d|q)$ via

$$P(d|q) = \frac{P(q|d)P(d)}{P(q)}$$

- $P(q)$ is the same for all documents, so ignore
- $P(d)$ is the prior – often treated as the same for all d
  - But we can give a higher prior to "high-quality" documents
    - PageRank, click through, social tags, etc.
- $P(q|d)$ is the probability of q given d.
  - So to rank documents according to relevance to q, rank according to $P(q|d)$

Where we are

- In the LM approach to IR, we attempt to model the query generation process.
  - Think of a query as being generated from a model derived from a document (or documents)
- Then we rank documents by the probability that a query would be observed as a random sample from the respective document model.
- That is, we rank according to $P(q|d)$.
- Next: how do we compute $P(q|d)$?

Stochastic Language Models

- Models *probability* of generating strings (each word in turn) in the language (commonly all strings over Σ). E.g., unigram model

Model M

0.2	the
0.1	a
0.01	man
0.01	woman
0.03	said
0.02	likes
…	

the	man	likes	the	woman
0.2	0.01	0.02	0.2	0.01

multiply

$P(s \mid M) = 0.00000008$

Stochastic Language Models

- Model *probability* of generating any string (for example, a query)

Model M1	
0.2	the
0.01	class
0.0001	sayst
0.0001	pleaseth
0.0001	yon
0.0005	maiden
0.01	woman

Model M2	
0.2	the
0.0001	class
0.03	sayst
0.02	pleaseth
0.1	yon
0.01	maiden
0.0001	woman

the	class	pleaseth	yon	maiden
0.2	0.01	0.0001	0.0001	0.0005
0.2	0.0001	0.02	0.1	0.01

P(s|M2) > P(s|M1)

How to compute $P(q|d)$

- This kind of conditional independence assumption is often called a Markov model

$$P(q|M_d) = P(\langle t_1, \ldots, t_{|q|}\rangle|M_d) = \prod_{1 \leq k \leq |q|} P(t_k|M_d)$$

($|q|$: length ofr q; t_k : the token occurring at position k in q)

- This is equivalent to:

$$P(q|M_d) = \prod_{\text{distinct term } t \text{ in } q} P(t|M_d)^{\mathrm{tf}_{t,q}}$$

- $\mathrm{tf}_{t,q}$: term frequency (# occurrences) of t in q

So…. LMs for ad hoc Retrieval

- Use each individual document as the corpus for a language model
- For a given query, assess P(q|d) for each document in the collection
- Return docs in the ranked order of P(q|d)

- Think about how scoring works for this model…
  - Term at a time?
  - Doc at a time?

Unigram and higher-order models

P (🔴 🟡 🔴 🔵)

- **= P (🔴) P (🟡 | 🔴) P (🔴 | 🔴 🟡) P (🔵 | 🔴 🟡 🔴)**
- Unigram Language Models

  P (🔴) P (🟡) P (🔴) P (🔵)

  Easy. Effective!
- Bigram (generally, *n*-gram) Language Models

  P (🔴) P (🟡 | 🔴) P (🔴 | 🟡) P (🔵 | 🔴)
- Other Language Models
  - Grammar-based models (PCFGs), etc.
    - Probably not the first thing to try in IR

Next time

- Lots of practical issues
  - Smoothing