Index Construction

David Kauchak
cs458
Fall 2012
adapted from:
http://www.stanford.edu/class/cs276/handouts/handout4-indexconstruction.pdf

Administrative
- Homework 1?
- Homework 2 out soon
- Issues with assignment 1?
- Talks Thursday
- videos?

Chinese

Google trends

Play with it at some point...

Many terms peak:
- Mitt Romney
- Michael Jackson

Some terms are cyclical:
- sunscreen
- christmas
Stemming

Reduce terms to their “roots” before indexing

The term “stemming” is used since it is accomplished mostly by chopping off part of the suffix of the word

<table>
<thead>
<tr>
<th>automate</th>
<th>automates</th>
<th>automat</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>automation</td>
<td></td>
</tr>
<tr>
<td>run</td>
<td>runs</td>
<td>running</td>
</tr>
</tbody>
</table>

Stemming example

Taking a course in information retrieval is more exciting than most courses

Take a course in inform retriev is more excit than most cours

http://maya.cs.depaul.edu/~classes/ds575/porter.html
or use the class from assign1 to try some examples out

Porter’s algorithm (1980)

Most common algorithm for stemming English

- Results suggest it’s at least as good as other stemming options

Multiple sequential phases of reductions using rules, e.g.

- sses → ss
- les → l
- ational → ate
- tional → tion

http://tartarus.org/~martin/PorterStemmer/

Lemmatization

Reduce inflectional/variant forms to base form

Stemming is an approximation for lemmatization

Lemmatization implies doing “proper” reduction to dictionary headword form

e.g.,

- am, are, is → be
- car, cars, car’s, cars’ → car

the boy’s cars are different colors
the boy car be different color
What normalization techniques to use...

What is the size of the corpus?
- small corpora often require more normalization

Depends on the users and the queries

Query suggestion (i.e. “did you mean”) can often be used instead of normalization

Most major search engines do little to normalize data except lowercasing and removing punctuation (and not even these always)

---

Hardware basics

Many design decisions in information retrieval are based on the characteristics of hardware

<table>
<thead>
<tr>
<th>cpu</th>
<th>main memory</th>
<th>disk</th>
</tr>
</thead>
</table>

fast, particularly relative to hard-drive access times

- gigahertz processors
- multi-core

64-bit for larger workable address space

<table>
<thead>
<tr>
<th>cpu</th>
<th>main memory</th>
<th>disk</th>
</tr>
</thead>
</table>

- GBs to 100s of GBs for servers
- main memory buses run at hundreds of megahertz
- ~random access
Hardware basics

No data is transferred from disk while the disk head is being positioned.

Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.

Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).

Block sizes: 8KB to 256 KB.

A few TBs

- average seek time: 5 ms
- transfer time per byte: 0.02 µs

RCV1: Our corpus for this lecture

As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.

This is one year of Reuters newswire (part of 1995 and 1996).

Still only a moderately sized data set.

<table>
<thead>
<tr>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>documents</td>
<td>800K</td>
</tr>
<tr>
<td>avg. # tokens per doc</td>
<td>200</td>
</tr>
<tr>
<td>terms</td>
<td>400K</td>
</tr>
<tr>
<td>non-positional postings</td>
<td>100M</td>
</tr>
</tbody>
</table>
### Index construction

<table>
<thead>
<tr>
<th>Documents</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
</tbody>
</table>

**Input:** tokenized, normalized documents

**Output:** postings lists sorted by docID

---

### Index construction: collecting docIDs

**Doc 1**

I did enact Julius Caesar I was killed
I the Capitol; Brutus killed me.

**Doc 2**

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

**Running time?** $\Theta(\text{tokens})$

**Memory?** $\Theta(1)$

---

### Index construction: sort dictionary

**Running time?** $\Theta(\text{tokens})$

**Memory?** $\Theta(T)$

---

### Index construction: create postings list

**Create postings lists from identical entries**

**Running time?** $\Theta(\text{tokens})$

What does this imply about the sorting algorithm?
Scaling index construction

In-memory index construction does not scale!

What is the major limiting step?
- both the collecting document IDs and creating posting lists require little memory since it's just a linear traversal of the data
- sorting is memory intensive! Even in-place sorting algorithms still require $O(n)$ memory

For RCV1:

<table>
<thead>
<tr>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>documents</td>
<td>800K</td>
</tr>
<tr>
<td>avg. # tokens per doc</td>
<td>200</td>
</tr>
<tr>
<td>terms</td>
<td>400K</td>
</tr>
<tr>
<td>non-positional postings</td>
<td>100M</td>
</tr>
</tbody>
</table>

How much memory is required?

What about for 10 years of news data?

What about for 300 billion web pages?
On-disk sorting
What are our options?
- sort on-disk: keep all data on disk. When we need to access entries, access entries
  - Random access on disk is slow…….
- Break up list into chunks. Sort chunks, then merge chunks (e.g. unix "merge" function or mergesort)

On-disk sorting
Do this while processing
When we reach a particular size, start the sorting process

On-disk sorting: splitting

On-disk sorting: sorting chunks
Pick the chunk size so that we can sort the chunk in memory
Generally, pick as large a chunk as possible while still being able to sort in memory
On-disk sorting

How can we do this so that it is time and memory efficient?

merge chunks

? \rightarrow ?

Binary merges

Could do binary merges, with a merge tree

For $n$ chunks, how many levels will there be?
- $\log(n)$

n-way merge

More efficient to do an $n$-way merge, where you are reading from all blocks simultaneously

Providing you read decent-sized chunks of each block into memory, you're not killed by disk seeks

Only one level of merges!

Is it linear?

Another approach: SPIMI

Sorting can still be expensive
Is there any way to do the indexing without sorting?

- Accumulate posting lists as they occur
- When size gets too big, start a new chunk
- Merge chunks at the end
Another approach

Doc 1

I did enact Julius Caesar I was killed in the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

Another approach

When posting lists get too large to fit in memory, we write to disk and start another one.

The merge

Running time?

- linear in the sizes of the postings list being merged
- As with merging sorted dictionary entries we can either do pairwise binary tree type merging or do an n-way merge
**Distributed indexing**

For web-scale indexing we must use a distributed computing cluster.

Individual machines are fault-prone
  - Can unpredictably slow down or fail

How do we exploit such a pool of machines?

**Google data centers**

Google data centers mainly contain commodity machines.

Data centers are distributed around the world.

Estimates:
- 2011: a total of 1 million servers, 3 million processors
- Google says: In planning 1 million – 10 million machines
- Google installs 100,000 servers each quarter
  - Based on expenditures of 200–250 million dollars per year
  - This would be 10% of the computing capacity of the world?!
- 0.01% of the total worldwide electricity

**Fault tolerance**

Hardware failures

>30% chance of failure within 5 years


What happens when you have 1 million servers?
Hardware is always failing!