Introduction to Information Retrieval
IIR 4: Index Construction

Mihai Surdeanu

(Based on slides by Hinrich Schütze at `informationretrieval.org`)

Fall 2015

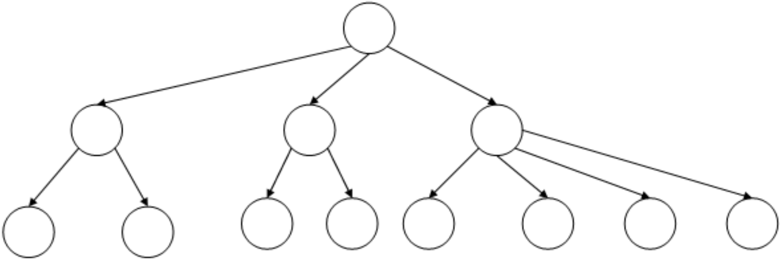
Overview

Outline

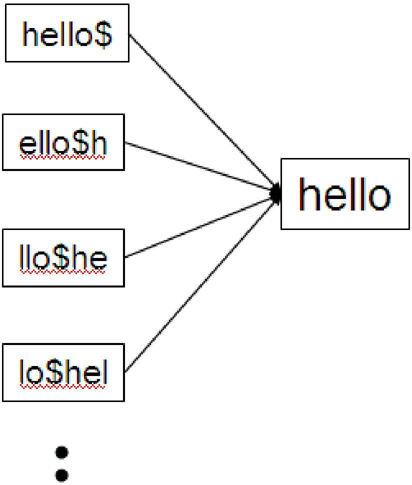
Dictionary as array of fixed-width entries

	term	document frequency	pointer to postings list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$
space needed:	20 bytes	4 bytes	4 bytes

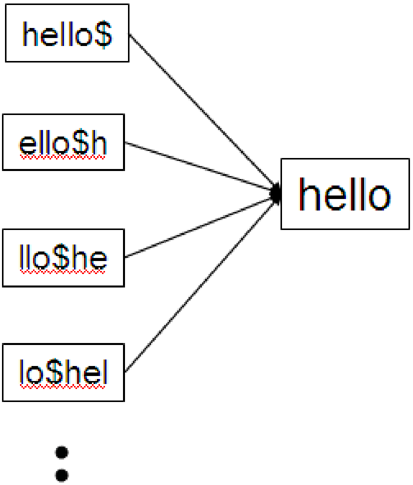
B-tree for looking up entries in array



Wildcard queries using a permuterm index



Wildcard queries using a permuterm index



Queries:

- For X, look up X$
- For X*, look up X*$
- For *X, look up X$*
- For *X*, look up X*
- For X*Y, look up Y$X*

k-gram indexes for spelling correction: *bordroom*

BO → aboard → about → boardroom → border

OR → border → lord → morbid → sordid

RD → aboard → ardent → boardroom → border

Levenshtein distance for spelling correction

LevenshteinDistance(s_1, s_2)

for $i \leftarrow 0$ **to** $|s_1|$
do $m[i,0] = i$
for $j \leftarrow 0$ **to** $|s_2|$
do $m[0,j] = j$
for $i \leftarrow 1$ **to** $|s_1|$
do for $j \leftarrow 1$ **to** $|s_2|$
    do if $s_1[i] = s_2[j]$
        then $m[i,j] = \min\{m[i-1,j]+1, m[i,j-1]+1, m[i-1,j-1]\}$
        else $m[i,j] = \min\{m[i-1,j]+1, m[i,j-1]+1, m[i-1,j-1]+1\}$
return $m[|s_1|,|s_2|]$

Operations: insert, delete, replace, copy

Take-away

- Two index construction algorithms: BSBI (simple) and SPIMI (more realistic)

- The material below is NOT covered in this course, and NOT required for any test!
- Distributed index construction: MapReduce
- Dynamic index construction: how to keep the index up-to-date as the collection changes

Outline

Hardware basics

- Many design decisions in information retrieval are based on hardware constraints.
- We begin by reviewing hardware basics that we'll need in this course.

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks are "idle" time: No data is transferred from disk while the disk head is being positioned.
- To optimize transfer time from disk to memory: one large chunk is faster than 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
- Servers used in IR systems typically have many GBs of main memory and TBs of disk space.

RCV1 collection

- Shakespeare's collected works are not large enough for demonstrating many of the points in this course.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- English newswire articles sent over the wire in 1995 and 1996 (one year).

A Reuters RCV1 document

REUTERS

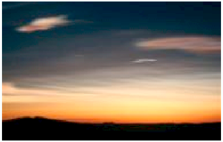
You are here: Home > News > Science > **Article**

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints



[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian

Reuters RCV1 statistics

N	documents	800,000
L	tokens per document	200
M	terms (= word types)	400,000
	bytes per token (incl. spaces/punct.)	6
	bytes per token (without spaces/punct.)	4.5
	bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Reuters RCV1 statistics

N	documents	800,000
L	tokens per document	200
M	terms (= word types)	400,000
	bytes per token (incl. spaces/punct.)	6
	bytes per token (without spaces/punct.)	4.5
	bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Exercise: Average frequency of a term (how many tokens)? 4.5 bytes per word token vs. 7.5 bytes per word type: why the difference? How many positional postings?

Outline

Goal: construct the inverted index

BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	...
CALPURNIA	→	2	31	54	101					
⋮										

dictionary **postings**

Index construction in IIR 1: Sort postings in memory

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Sort-based index construction

- As we build the index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- Thus: We need to store intermediate results on disk.

Same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?

Same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting very large sets of records on disk is too slow – too many disk seeks.
- We need an external sorting algorithm. That is, generate postings in smaller blocks that we can keep in memory. Then sort them to obtain the global order.

"External" sorting algorithm (using few disk seeks)

- We must sort $T = 100{,}000{,}000$ non-positional postings.
  - Each posting has size 8 bytes (4+4: termID, docID).
  - Note that we termID instead of term here! Why?
- Define a block to consist of 10,000,000 such postings
  - We can easily fit that many postings into memory.
  - We will have 10 such blocks for RCV1.
- Basic idea of algorithm:
  - For each block: (i) accumulate postings, (ii) sort in memory, (iii) write to disk
  - Then merge the blocks into one long sorted order.
  - We need a unique dictionary for the merge!

Merging two blocks

postings
to be merged

Block 1	
brutus	d3
caesar	d4
noble	d3
with	d4

Block 2	
brutus	d2
caesar	d1
julius	d1
killed	d2

brutus	d2
brutus	d3
caesar	d1
caesar	d4
julius	d1
killed	d2
noble	d3
with	d4

merged
postings

disk

Blocked Sort-Based Indexing

```
BSBIndexConstruction()
n ← 0
while (all documents have not been processed)
do n ← n + 1
    block ← ParseNextBlock()
    BSBI-Invert(block) // produces an inverted index for this block
    WriteBlockToDisk(block, f_n)
MergeBlocks(f_1, ..., f_n; f_merged) // needs global dictionary
```

Outline

The problem with the sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, during the merging of blocks, we could work with term,docID postings instead of termID,docID postings . . .
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

Single-pass in-memory indexing

- Abbreviation: SPIMI
- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur. This means there is a hash lookup for each addition to a postings list.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI-Invert

```
SPIMI-Invert(token_stream)
output_file ← NewFile()
dictionary ← NewHash()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list ← AddToDictionary(dictionary, term(token))
      else postings_list ← GetPostingsList(dictionary, term(token))
    if full(postings_list)
      then postings_list ← DoublePostingsList(dictionary, term(token)
    AddToPostingsList(postings_list, docID(token))
sorted_terms ← SortTerms(dictionary) // in preparation for the merge
WriteBlockToDisk(sorted_terms, dictionary, output_file)
return output_file
```

SPIMI-Invert

```
SPIMI-Invert(token_stream)
output_file ← NewFile()
dictionary ← NewHash()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
       then postings_list ← AddToDictionary(dictionary,term(token))
       else postings_list ← GetPostingsList(dictionary,term(token))
    if full(postings_list)
       then postings_list ← DoublePostingsList(dictionary,term(token)
    AddToPostingsList(postings_list,docID(token))
sorted_terms ← SortTerms(dictionary) // in preparation for the merge
WriteBlockToDisk(sorted_terms,dictionary,output_file)
return output_file
```

Merging of blocks is analogous to BSBI, but using alphabetical ordering rather than by termID.

SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
  - See next lecture

Stop here

- We won't cover the remaining material in this lecture beyond this point.
- Distributed indexing and dynamic indexing are NOT required for any exam!

Outline

Distributed indexing

- For web-scale indexing (don't try this at home!): must use a distributed computer cluster
- Individual machines are fault-prone.
  - Can unpredictably slow down or fail.
- How do we exploit such a pool of machines?

Google data centers (2007 estimates; Gartner)

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- 1 million servers, 3 million processors/cores
- 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!
- Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?

Google data centers (2007 estimates; Gartner)

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- 1 million servers, 3 million processors/cores
- 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!
- Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?
- Answer: less than two minutes: $(3*365*24*60)/1000000 = 1.5768$

Distributed indexing

- Maintain a master machine directing the indexing job – considered "safe"
- Break up indexing into sets of parallel tasks
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- We will define two sets of parallel tasks and deploy two types of machines to solve them:
  - Parsers
  - Inverters
- Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
- Each split is a subset of documents.

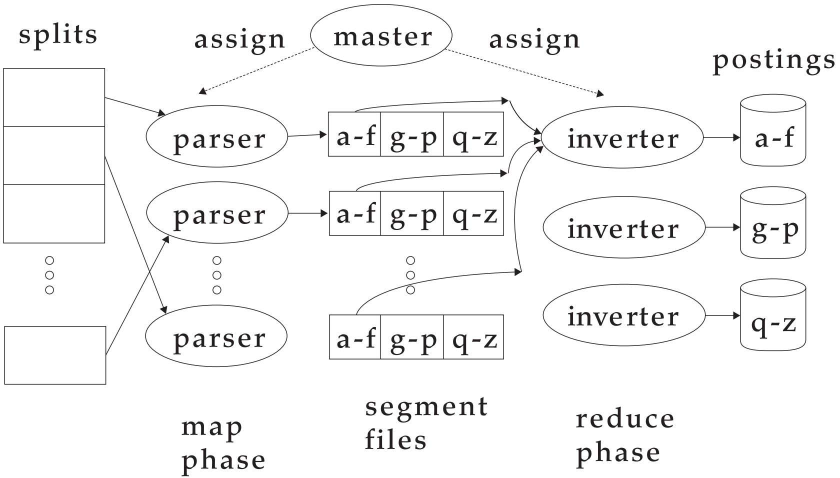
Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,docID)-pairs.
- Parser writes pairs into j term-partitions.
- Each for a range of terms' first letters
  - E.g., a-f, g-p, q-z (here: $j = 3$)

Inverters

- An inverter collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f).
- Sorts and writes to postings lists

Data flow


splits
assign
master
assign
postings
parser
a-f g-p q-z
inverter
a-f
parser
a-f g-p q-z
inverter
g-p
parser
a-f g-p q-z
inverter
q-z
map phase
segment files
reduce phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce is a robust and conceptually simple framework for distributed computing . . .
- . . . without having to write code for the distribution part.
- The Google indexing system (ca. 2002) consisted of a number of phases, each implemented in MapReduce.
- Index construction was just one phase.
- Another phase: transform term-partitioned into document-partitioned index.

Index construction in MapReduce

Schema of map and reduce functions

map:	input	$\rightarrow$ list(k, v)
reduce:	(k,list(v))	$\rightarrow$ output

Instantiation of the schema for index construction

map:	web collection	$\rightarrow$ list(termID, docID)
reduce:	$(\langle \text{termID}_1, \text{list(docID)}\rangle, \langle \text{termID}_2, \text{list(docID)}\rangle, \dots)$	$\rightarrow (\text{postings\_list}_1, \text{postings\_list}_2, \dots)$

Example for index construction

map:	d_2 : C DIED. d_1 : C CAME, C C'ED.	$\rightarrow (\langle \text{C}, d_2\rangle, \langle \text{DIED}, d_2\rangle, \langle \text{C}, d_1\rangle, \langle \text{CAME}, d_1\rangle, \langle \text{C}, d_1\rangle, \langle \text{C'ED}, d_1\rangle)$
reduce:	$(\langle \text{C},(d_2,d_1,d_1)\rangle, \langle \text{DIED},(d_2)\rangle, \langle \text{CAME},(d_1)\rangle, \langle \text{C'ED},(d_1)\rangle)$	$\rightarrow (\langle \text{C},(d_1{:}2,d_2{:}1)\rangle, \langle \text{DIED},(d_2{:}1)\rangle, \langle \text{CAME},(d_1{:}1)\rangle, \langle \text{C'ED},(d_1{:}1)\rangle)$

Exercise

- What information does the task description contain that the master gives to a parser?
- What information does the parser report back to the master upon completion of the task?
- What information does the task description contain that the master gives to an inverter?
- What information does the inverter report back to the master upon completion of the task?

Hadoop

- Google's MapReduce framework is not public.
- But this is: `http://hadoop.apache.org/`

Outline

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are: Documents are inserted, deleted and modified. Think Twitter, Facebook, etc.
- This means that the dictionary and postings lists have to be dynamically modified.

Dynamic indexing: Simplest approach

- Maintain big main index on disk
- New docs go into small auxiliary index in memory.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this bit-vector

Issue with auxiliary and main index

- Frequent merges
- Poor search performance during index merge

Logarithmic merge

- Logarithmic merging amortizes the cost of merging indexes over time.
  - $\rightarrow$ Users see smaller effect on response times.
- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (Z_0) in memory
- Larger ones (I_0, I_1, ...) on disk
- If Z_0 gets too big ($> n$), write to disk as I_0
- ... or merge with I_0 (if I_0 already exists) and write merger to I_1 etc.

```
LMergeAddToken(indexes, Z_0, token)
Z_0 ← Merge(Z_0, {token})
if |Z_0| = n
  then for i ← 0 to ∞
         do if I_i ∈ indexes
               then Z_{i+1} ← Merge(I_i, Z_i)
                    (Z_{i+1} is a temporary index on disk.)
                    indexes ← indexes − {I_i}
               else I_i ← Z_i    (Z_i becomes the permanent index I_i.)
                    indexes ← indexes ∪ {I_i}
                    Break
         Z_0 ← ∅
```

```
LogarithmicMerge()
Z_0 ← ∅    (Z_0 is the in-memory index.)
indexes ← ∅
while true
do LMergeAddToken(indexes, Z_0, getNextToken())
```

Binary numbers: $I_3 I_2 I_1 I_0 = 2^3 2^2 2^1 2^0$

- 0001
- 0010
- 0011
- 0100
- 0101
- 0110
- 0111
- 1000
- 1001
- 1010
- 1011
- 1100

Logarithmic merge

- Auxiliary index: index construction time is $O(T^2/n)$ as (in the worst case) each posting in the big index is touched in each merge.
  - T is total number of postings read
  - n size of in-memory auxiliary index
  - Each of the T postings is touched O(T/n) times
- With logarithmic indexing:
  - Number of indexes bounded by $O(\log T/n)$
  - Query processing requires the merging of $O(\log T/n)$ indexes (slightly slower)
  - Index construction complexity is $O(T \log T/n)$ (much faster!)
    - ...because each of T postings is merged $O(\log T)$ times.
- So logarithmic merging is an order of magnitude more efficient.

Dynamic indexing at large search engines

- Often a combination
  - Frequent incremental changes
  - Rotation of large parts of the index that can then be swapped in
  - Occasional complete rebuild (becomes harder with increasing size – not clear if Google can do a complete rebuild)

Building positional indexes

- Basically the same problem except that the intermediate data structures are large.

Take-away

- Two index construction algorithms: BSBI (simple) and SPIMI (more realistic)

- The material below is NOT covered in this course, and NOT required for any test!
- Distributed index construction: MapReduce
- Dynamic index construction: how to keep the index up-to-date as the collection changes

Resources

- Chapter 4 of IIR
- Youtube video: Google data centers, `https://www.youtube.com/watch?v=wSwWaC_IOpg`