Introduction to Information Retrieval

http://informationretrieval.org

IIR 4: Index Construction

Hinrich Schütze

Institute for Natural Language Processing, Universität Stuttgart

2008.05.05

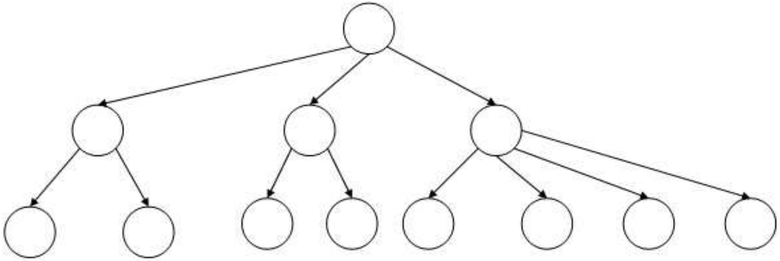
Overview

Outline

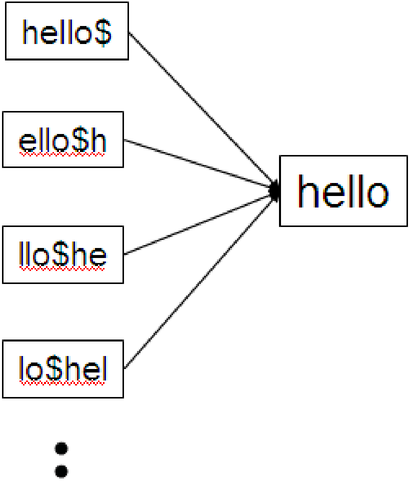
Dictionary as array of fixed-width entries

	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

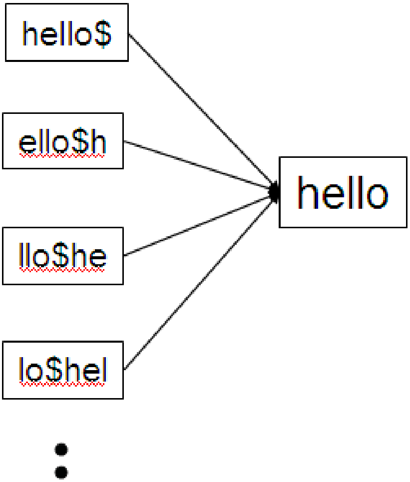
B-tree for looking up entries in array



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*

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

Peter Norvig's spell corrector

```
import re, collections

def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))

alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
    n = len(word)
    return set([word[0:i]+word[i+1:] for i in range(n)] +                     # deletion
               [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] + # transposition
               [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] + # alteration
               [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])  # insertion

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=lambda w: NWORDS[w])
```

Outline

Hardware basics

- Many design decisions in information retrieval are based on hardware constraints.

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)

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks: No data is transferred from disk while the disk head is being positioned.

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: 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

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: 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
- Servers used in IR systems typically have several GB of main memory, sometimes tens of GB. Available disk space is several orders of magnitude larger.

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: 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
- Servers used in IR systems typically have several GB of main memory, sometimes tens of GB. Available disk space is several orders of magnitude larger.
- Fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine.

Hardware basics: Summary

symbol	statistic	value
s	average seek time	5 ms $= 5 \times 10^{-3}$ s
b	transfer time per byte	0.02 μs $= 2 \times 10^{-8}$ s
	processor's clock rate	10^9 s^{-1}
p	lowlevel operation (e.g., compare & swap a word)	0.01 μs $= 10^{-8}$ s
	size of main memory	several GB
	size of disk space	1 TB or more

RCV1 collection

- Shakespeare's collected works are not large enough for demonstrating many of the points in this course.

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.

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 meteorological base at Mawson Station on July 25.

Reuters RCV1 statistics

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

Reuters RCV1 statistics

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

4.5 bytes per word token vs. 7.5 bytes per word type: why?

Outline

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

Scaling index construction

- In-memory index construction does not scale.

Scaling index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?

Scaling index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .

Scaling index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed etc.

Sort-based index construction

- As we build index, we parse docs one at a time.

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- At 10–12 bytes per postings entry, demands a lot of space for large collections.

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- At 10–12 bytes per postings entry, demands a lot of space for large collections.
- $T = 100{,}000{,}000$ in the case of RCV1

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- At 10–12 bytes per postings entry, demands a lot of space for large collections.
- $T = 100{,}000{,}000$ in the case of RCV1
- Actually, we can do 100,000,000 in memory, but typical collections are much larger than RCV1.

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- At 10–12 bytes per postings entry, demands a lot of space for large collections.
- $T = 100{,}000{,}000$ in the case of RCV1
- Actually, we can do 100,000,000 in memory, but typical collections are much larger than RCV1.
- 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 $T = 100{,}000{,}000$ records on disk is too slow – too many disk seeks.

Same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting $T = 100{,}000{,}000$ records on disk is too slow – too many disk seeks.
- We need an external sorting algorithm.

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID
- Define a block to consist of 10,000,000 such postings

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID
- Define a block to consist of 10,000,000 such postings
  - We can easily fit that many postings into memory.

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID
- 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.

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID
- 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:

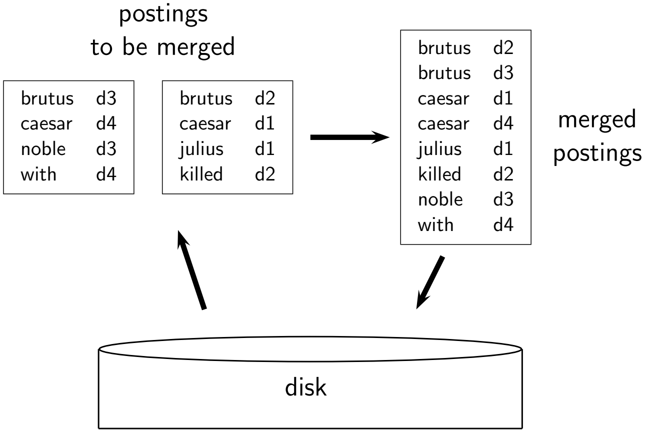
"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID
- 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:
  - Accumulate postings for each block, sort, write to disk.

"External" sorting algorithm (using few disk seeks)

- 12-byte (4+4+4) postings (termID, docID, document frequency)
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by termID
- 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:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.

Merging two blocks



Blocked Sort-Based Indexing

BSBIndexConstruction()

$n \leftarrow 0$
while (all documents have not been processed)
do $n \leftarrow n + 1$
    block $\leftarrow$ ParseNextBlock()
    BSBI-Invert(*block*)
    WriteBlockToDisk(*block*, f_n)
MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)

Blocked Sort-Based Indexing

```
BSBIndexConstruction()
n ← 0
while (all documents have not been processed)
do n ← n + 1
    block ← ParseNextBlock()
    BSBI-Invert(block)
    WriteBlockToDisk(block, f_n)
MergeBlocks(f_1, ..., f_n; f_merged)
```

- Key decision: What is the size of one block?

Outline

Problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.

Problem with 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.

Problem with 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, we could work with term,docID postings instead of termID,docID postings . . .

Problem with 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, 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

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.

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.

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.
- With these two ideas we can generate a complete inverted index for each block.

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.
- 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)
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)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

Merging of blocks is analogous to BSBI.

SPIMI: Compression

- Compression makes SPIMI even more efficient.

SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms

SPIMI: Compression

- Compression makes SPIMI even more efficient.
    - Compression of terms
    - Compression of postings

SPIMI: Compression

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

Outline

Distributed indexing

- For web-scale indexing (dont' try this at home!): must use a distributed computer cluster

Distributed indexing

- For web-scale indexing (dont' try this at home!): must use a distributed computer cluster
- Individual machines are fault-prone.

Distributed indexing

- For web-scale indexing (dont' try this at home!): must use a distributed computer cluster
- Individual machines are fault-prone.
  - Can unpredictably slow down or fail.

Distributed indexing

- For web-scale indexing (dont' 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

- Google data centers mainly contain commodity machines.

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
- Based on expenditures of 200–250 million dollars per year

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: 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!

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: 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!
- If in a non-fault-tolerant systemn with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: 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!
- If in a non-fault-tolerant systemn with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?
- Answer: 63%

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: 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!
- If in a non-fault-tolerant systemn with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?
- Answer: 63%
- Calculate the number of servers failing per minute for an installation of 1 million servers.

Distributed indexing

- Maintain a master machine directing the indexing job – considered "safe"

Distributed indexing

- Maintain a master machine directing the indexing job – considered "safe"
- Break up indexing into sets of parallel tasks

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:

Parallel tasks

- We will define two sets of parallel tasks and deploy two types of machines to solve them:
  - Parsers

Parallel tasks

- We will define two sets of parallel tasks and deploy two types of machines to solve them:
  - Parsers
  - Inverters

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)

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.

Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,doc) pairs.

Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,doc) pairs.
- Parser writes pairs into j term-partitions.

Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,doc) pairs.
- Parser writes pairs into j term-partitions.
- Each for a range of terms' first letters

Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,doc) 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: $j = 3$)

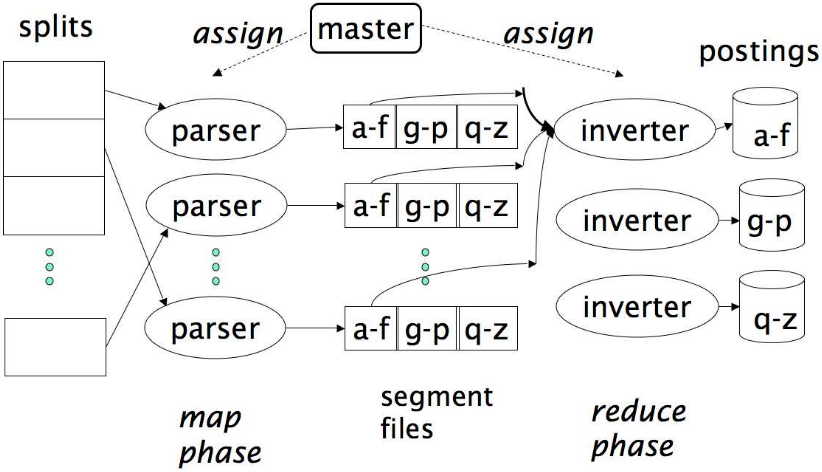
Inverters

- An inverter collects all (term,doc) pairs (= postings) for one term-partition.

Inverters

- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- 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
inverter
q-z
parser
a-f
g-p
q-z
map
phase
segment
files
reduce
phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce is a robust and conceptually simple framework for distributed computing . . .

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.

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.

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.

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.

MapReduce schema

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:	$(\langle \text{termID}_1, \text{list(docID)}\rangle, \langle \text{termID}_2, \text{list(docID)}\rangle, \ldots)$	$\rightarrow$ $(\text{postings\_list}_1, \text{postings\_list}_2, \ldots)$

Example for index construction

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

Outline

Dynamic indexing

- Up to now, we have assumed that collections are static.

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are.

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are.
- Documents are inserted, deleted and modified.

Dynamic indexing

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

Simplest approach

- Maintain "big" main index on disk

Simplest approach

- Maintain "big" main index on disk
- New docs go into "small" auxiliary index in memory.

Simplest approach

- Maintain “big” main index on disk
- New docs go into “small” auxiliary index in memory.
- Search across both, merge results

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 one main index

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 one main index
- Deletions:

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 one main index
- Deletions:
  - Invalidation bit-vector for deleted docs

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 one main index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this invalidation bit-vector; only return "valid" docs to user

Issue with auxiliary and main index

- Frequent merges

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge
- Actually:

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient.

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient.
- Assumption for the rest of the lecture: The index is one big file.

Issue with auxiliary and main index

- Frequent merges
- Poor performance during merge
- Actually:
    - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
    - Merge is the same as a simple append.
    - But then we would need a lot of files – inefficient.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (J_0) in memory

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (J_0) in memory
- Larger ones ($I_0, I_1, \ldots$) on disk

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (J_0) in memory
- Larger ones (I_0, I_1, ...) on disk
- If J_0 gets too big ($> n$), write to disk as I_0

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (J_0) in memory
- Larger ones (I_0, I_1, ...) on disk
- If J_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.

$\text{LMERGEADDTOKEN}(\textit{indexes}, Z_0, \textit{token})$

```
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 ← ∅
```

$\text{LOGARITHMICMERGE}()$

```
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

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

- 0001
- 0010

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

- 0001
- 0010
- 0011

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

- 0001
- 0010
- 0011
- 0100

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

- 0001
- 0010
- 0011
- 0100
- 0101

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

- 0001
- 0010
- 0011
- 0100
- 0101
- 0110

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

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

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

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

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

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

- Number of indexes bounded by $O(\log T)$ (T is total number postings read so far)

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes
- Time complexity of index construction: Each posting is merged $O(\log T)$ times.

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes
- Time complexity of index construction: Each posting is merged $O(\log T)$ times.
- Auxiliary index: index construction time is $O(T^2)$ as each posting is touched in each merge.

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes
- Time complexity of index construction: Each posting is merged $O(\log T)$ times.
- Auxiliary index: index construction time is $O(T^2)$ as each posting is touched in each merge.
- So logarithming merging is an order of magnitude more efficient.

Dynamic indexing at large search engines

- Often a combination

Dynamic indexing at large search engines

- Often a combination
  - Frequent incremental changes

Dynamic indexing at large search engines

- Often a combination
  - Frequent incremental changes
  - Occasional complete rebuild

Building positional indexes

Building positional indexes

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

Resources

- Chapter 4 of IIR

Resources

- Chapter 4 of IIR
- Resources at `http://ifnlp.org/ir`

Resources

- Chapter 4 of IIR
- Resources at `http://ifnlp.org/ir`
- Original publication on MapReduce by Dean and Ghemawat (2004)

Resources

- Chapter 4 of IIR
- Resources at `http://ifnlp.org/ir`
- Original publication on MapReduce by Dean and Ghemawat (2004)
- Original publication on SPIMI by Heinz and Zobel (2003)