

# ΑΝΑΚΤΗΣΗ ΠΛΗΡΟΦΟΡΙΩΝ ΚΑΙ ΑΝΑΖΗΤΗΣΗ ΣΤΟΝ ΠΑΓΚΟΣΜΙΟ ΙΣΤΟ

Παροράματα από το Πανεπιστήμιο της Στουγκάρδης

# Introduction to Information Retrieval

<http://informationretrieval.org>

## IIR 4: Index Construction

Hinrich Schütze, Christina Lioma

Institute for Natural Language Processing, University of Stuttgart

2010-05-04

# Overview

- 1 Recap
- 2 Introduction
- 3 BSBI algorithm
- 4 SPIMI algorithm
- 5 Distributed indexing
- 6 Dynamic indexing

# Outline

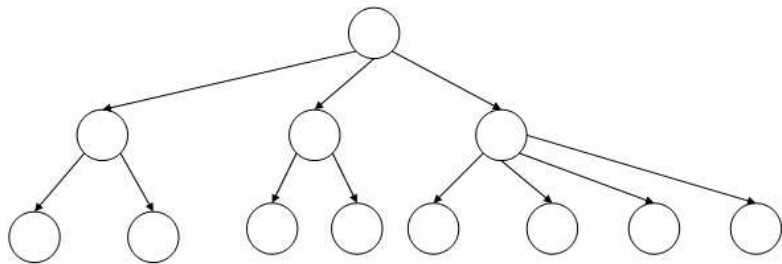
- 1 Recap
- 2 Introduction
- 3 BSBI algorithm
- 4 SPIMI algorithm
- 5 Distributed indexing
- 6 Dynamic indexing

## Dictionary as array of fixed-width entries

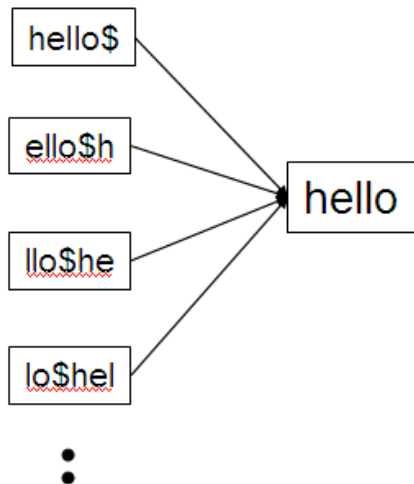
term	document frequency	pointer to postings list
a	656,265	→
aachen	65	→
...	...	...
zulu	221	→

space needed: 20 bytes 4 bytes 4 bytes

## B-tree for looking up entries in array



# Wildcard queries using a permuterm index

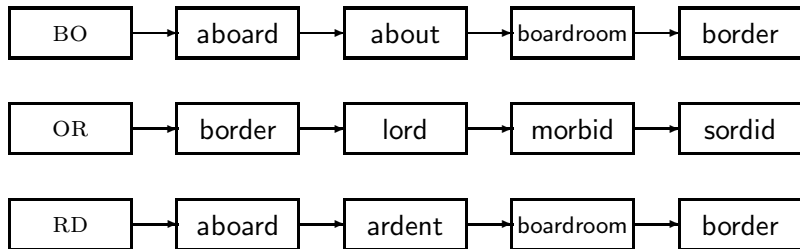


Queries:

- 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*



# Levenshtein distance for spelling correction

LEVENSHTEINDISTANCE( $s_1, s_2$ )

```
1  for  $i \leftarrow 0$  to  $|s_1|$ 
2  do  $m[i, 0] = i$ 
3  for  $j \leftarrow 0$  to  $|s_2|$ 
4  do  $m[0, j] = j$ 
5  for  $i \leftarrow 1$  to  $|s_1|$ 
6  do for  $j \leftarrow 1$  to  $|s_2|$ 
7     do if  $s_1[i] = s_2[j]$ 
8         then  $m[i, j] = \min\{m[i - 1, j] + 1, m[i, j - 1] + 1, m[i - 1, j - 1]\}$ 
9         else  $m[i, j] = \min\{m[i - 1, j] + 1, m[i, j - 1] + 1, m[i - 1, j - 1] + 1\}$ 
10 return  $m[|s_1|, |s_2|]$ 
```

Operations: insert, delete, replace, copy

## Exercise: Understand Peter Norvig's spelling 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):
    splits      = [(word[:i], word[i:]) for i in range(len(word) + 1)]
    deletes     = [a + b[1:] for a, b in splits if b]
    transposes  = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b) > 1]
    replaces    = [a + c + b[1:] for a, b in splits for c in alphabet if b]
    inserts     = [a + c + b      for a, b in splits for c in alphabet]
    return set(deletes + transposes + replaces + inserts)
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=NWORDS.get)
```

# Take-away

- Two index construction algorithms: **BSBI** (simple) and **SPIMI** (more realistic)
- **Distributed** index construction: MapReduce
- **Dynamic** index construction: how to keep the index up-to-date as the collection changes

# Outline

- 1 Recap
- 2 Introduction**
- 3 BSBI algorithm
- 4 SPIMI algorithm
- 5 Distributed indexing
- 6 Dynamic indexing

# 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 **several GB of main memory**, sometimes tens of GB, and **TBs or 100s of GB of disk space**.
- **Fault tolerance is expensive**: It's cheaper to use many regular machines than one fault tolerant machine.

## Some stats (ca. 2008)

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



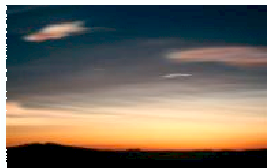
# RCV1 collection

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

## Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

[Email This Article](#) | [Print This Article](#) | [Reprint](#)



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

[\[-\] Text \[+\]](#)

# 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

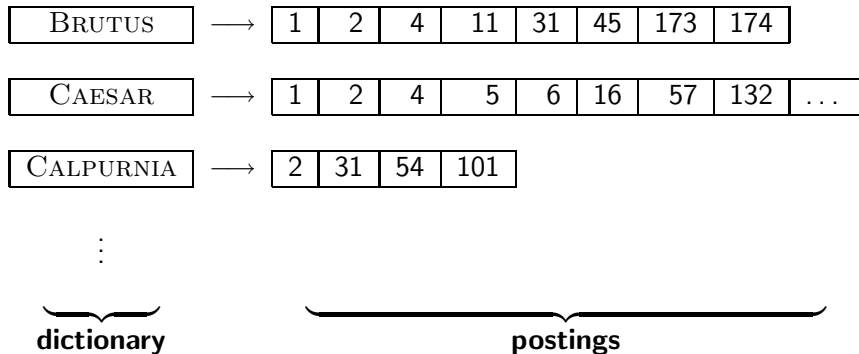
Exercise: Average frequency of a term (how many tokens)? 4.5

bytes per word token vs. 7.5 bytes per word type: why the difference? How many positional postings?

# Outline

- 1 Recap
- 2 Introduction
- 3 BSBI algorithm**
- 4 SPIMI algorithm
- 5 Distributed indexing
- 6 Dynamic indexing

# Goal: construct the inverted index



# Index construction in IIR 1: Sort postings in memory

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

# 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.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- $T = 100,000,000$  in the case of RCV1: we can do this in memory on a typical machine in 2010.
- But in-memory index construction does not scale 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?
- 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)

- We must sort  $T = 100,000,000$  non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, document frequency).
- 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.

# Merging two blocks

postings  
to be merged

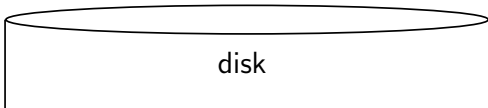
Block 1	
brutus	d3
caesar	d4
noble	d3
with	d4

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



# Blocked Sort-Based Indexing

BSBINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$ 
2  while (all documents have not been processed)
3  do  $n \leftarrow n + 1$ 
4      $block \leftarrow \text{PARSENEXTBLOCK}()$ 
5      $\text{BSBI-INVERT}(block)$ 
6      $\text{WRITEBLOCKTODISK}(block, f_n)$ 
7   $\text{MERGEBLOCKS}(f_1, \dots, f_n; f_{\text{merged}})$ 
```

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

# Outline

- 1 Recap
- 2 Introduction
- 3 BSBI algorithm
- 4 SPIMI algorithm**
- 5 Distributed indexing
- 6 Dynamic indexing

# 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
- 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)
1  output_file ← NEWFILE()
2  dictionary ← NEWHASH()
3  while (free memory available)
4  do token ← next(token_stream)
5     if term(token) ∉ dictionary
6         then postings_list ← ADDTODICTIONARY(dictionary,term(token))
7         else postings_list ← GETPOSTINGSLIST(dictionary,term(token))
8     if full(postings_list)
9         then postings_list ← DOUBLEPOSTINGSLIST(dictionary,term(token))
10    ADDTOSTRINGLIST(postings_list,docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms,dictionary,output_file)
13  return output_file
```

Merging of blocks is analogous to BSBI.

# SPIMI: Compression

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



# Exercise: Time 1 machine needs for Google size collection

BSBINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$ 
2  while (all documents have not been processed)
3  do  $n \leftarrow n + 1$ 
4     $block \leftarrow \text{PARSENEXTBLOCK}()$ 
5     $\text{BSBI-INVERT}(block)$ 
6     $\text{WRITEBLOCKTODISK}(block, f_n)$ 
7   $\text{MERGEBLOCKS}(f_1, \dots, f_n; f_{\text{merged}})$ 
```

symbol	statistic	value
$s$	average seek time	5 ms = $5 \times 10^{-3}$ s
$b$	transfer time per byte	$0.02 \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	$10^9 \text{ s}^{-1}$
$p$	lowlevel operation	$0.01 \mu\text{s} = 10^{-8}$ s
	number of machines	1
	size of main memory	8 GB
	size of disk space	unlimited
$N$	documents	$10^{11}$ (on disk)
$L$	avg. # word tokens per document	$10^3$
$M$	terms (= word types)	$10^8$
	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

Hint: You have to make several simplifying assumptions – that's

ok, just state them clearly.

# Outline

- 1 Recap
- 2 Introduction
- 3 BSBI algorithm
- 4 SPIMI algorithm
- 5 Distributed indexing**
- 6 Dynamic indexing

# 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!
- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system (assuming it does not tolerate failures)?
- Answer: 37%
- 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

# 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

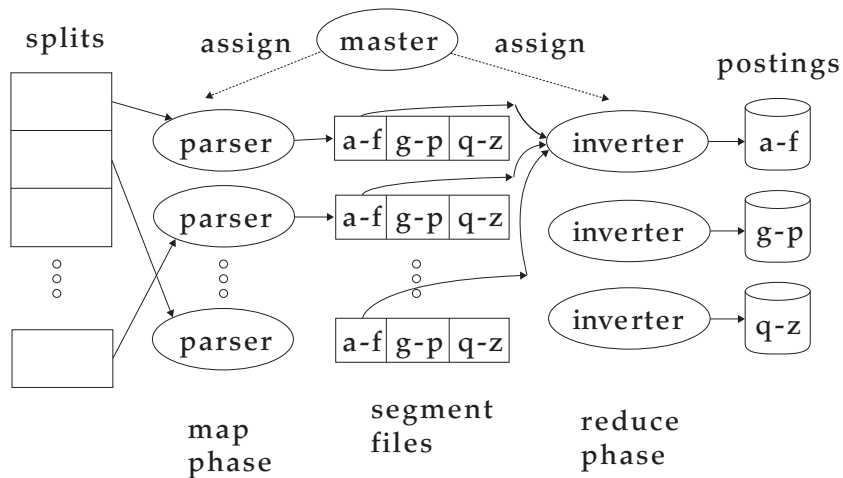
- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and **emits** (term,docID)-pairs.
- Parser writes pairs into  $j$  term-partitions.
- Each for a range of terms' first letters
  - E.g., a-f, g-p, q-z (here:  $j = 3$ )

# 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



# 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  $\rightarrow \text{list}(k, v)$   
reduce:  $(k, \text{list}(v)) \rightarrow \text{output}$

## Instantiation of the schema for index construction

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

## Example for index construction

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

# 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?

# Outline

- 1 Recap
- 2 Introduction
- 3 BSBI algorithm
- 4 SPIMI algorithm
- 5 Distributed indexing
- 6 Dynamic indexing**

# 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 **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
- Actually:
  - Merging of the auxiliary index into the main index is not that costly 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 into several files, collect small postings lists in one file etc.)



# Logarithmic merge

- Logarithmic merging amortizes the cost of merging indexes over time.
  - → 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 ( $l_0, l_1, \dots$ ) on disk
- If  $Z_0$  gets too big ( $> n$ ), write to disk as  $l_0$
- ... or merge with  $l_0$  (if  $l_0$  already exists) and write merger to  $l_1$  etc.

LMERGEADDTOKEN(*indexes*,  $Z_0$ , *token*)

```
1   $Z_0 \leftarrow \text{MERGE}(Z_0, \{\textit{token}\})$ 
2  if  $|Z_0| = n$ 
3    then for  $i \leftarrow 0$  to  $\infty$ 
4      do if  $l_i \in \textit{indexes}$ 
5        then  $Z_{i+1} \leftarrow \text{MERGE}(l_i, Z_i)$ 
6          ( $Z_{i+1}$  is a temporary index on disk.)
7           $\textit{indexes} \leftarrow \textit{indexes} - \{l_i\}$ 
8        else  $l_i \leftarrow Z_i$  ( $Z_i$  becomes the permanent index  $l_i$ .)
9           $\textit{indexes} \leftarrow \textit{indexes} \cup \{l_i\}$ 
10         BREAK
11      $Z_0 \leftarrow \emptyset$ 
```

LOGARITHMICMERGE()

```
1   $Z_0 \leftarrow \emptyset$  ( $Z_0$  is the in-memory index.)
2   $\textit{indexes} \leftarrow \emptyset$ 
3  while true
4  do LMERGEADDTOKEN(indexes,  $Z_0$ , GETNEXTTOKEN())
```

Binary numbers:  $l_3l_2l_1l_0 = 2^32^22^12^0$

- 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 of postings read so far)
- So query processing requires the merging of  $O(\log T)$  indexes
- Time complexity of index construction is  $O(T \log T)$ .
  - ... because each of  $T$  postings is merged  $O(\log T)$  times.
- Auxiliary index: index construction time is  $O(T^2)$  as each posting is touched in each merge.
  - Suppose auxiliary index has size  $a$
  - $a + 2a + 3a + 4a + \dots + na = a \frac{n(n+1)}{2} = O(n^2)$
- 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)
- **Distributed** index construction: MapReduce
- **Dynamic** index construction: how to keep the index up-to-date as the collection changes

# 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)
  - YouTube video: Google data centers