

Efficiency: Indexing

(CS429)
Nazli Goharian
nazli@ir.iit.edu

Slides are *mostly* based on Information Retrieval Algorithms and Heuristics, Grossman, Frieder

© Goharian, Grossman, Frieder, 2002, 2009

Efficiency

- Difficult to analyze sequential IR algorithms: data and query dependency (query selectivity).
- $O(q(cf_{\max}))$ high estimate- half tokens are *hapax legomena* according to Zipf's law.
- No standard analytical model to estimate query performance, hence empirical efforts.

© Goharian, Grossman, Frieder, 2002, 2009

2

Efficiency Techniques

- Indexing
 - Compression
- Index Pruning (Top Doc)
- Efficient Query Processing
- Duplicate Document Detection

Indexing

- Scanning Text
 - Small document collection
- Inverted index [1960's]
 - Reducing I/O, thus, speeding query processing; storage overhead; time overhead to build index
- Signature files
 - Smaller and faster; less functionality
- Relational
 - Higher overhead; supports integration of structured data and text

Inverted Index

- Regardless of the retrieval strategy we need a data structure to efficiently store:
 - For each term in the document collection
 - The list of documents that contain the term
 - For each occurrence of a term in a document
 - The frequency the term appears in the document (tf)
 - The position in the document for which the term appears (only needed if proximity search is supported).
 - » Position may be expressed as section, paragraph, sentence, location within sentence.

© Goharian, Grossman, Frieder, 2002, 2009

5

Inverted Index

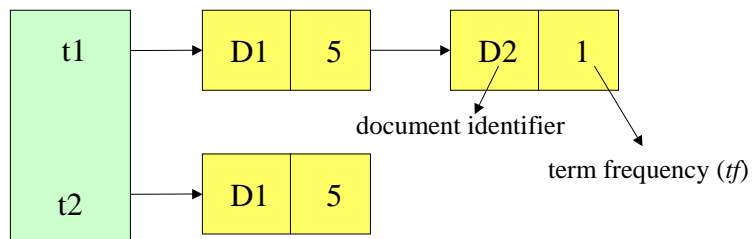
- Associates a *posting list* with each term
 - a: (D1,7) (D2,5) (D3,19) (D4,11)...
 - abacus: (D7,1)
 - abatement: (D15,1) (D23,2)
 - ...
 - zoology: (D8,1) (D32,2)
- Inverted because it lists for a term, all documents that contain the term.

© Goharian, Grossman, Frieder, 2002, 2009

6

Inverted Index: Structure

- Term list (Lexicon)- stores distinct terms and document frequency information (df , idf)
- Posting list- stores documents for a given term (estimate posting list size by zipfian distribution)



© Goharian, Grossman, Frieder, 2002, 2009

7

Zipf's Law [1949]

- Approximation of posting list size using term frequency distribution in a natural language:

*If terms are ordered by their collection frequency and assigned a rank, then:
The product of the frequency of a term in collection and its rank is constant*

© Goharian, Grossman, Frieder, 2002, 2009

8

Zipf's Law [1949]

Ranked Terms based on collection frequency	Rank (r)	freq (c/r)	Constant
Book	1	5000	c
Apple	2	5000/2	c
China	3	5000/3	c
Himalayas	4	5000/4	c

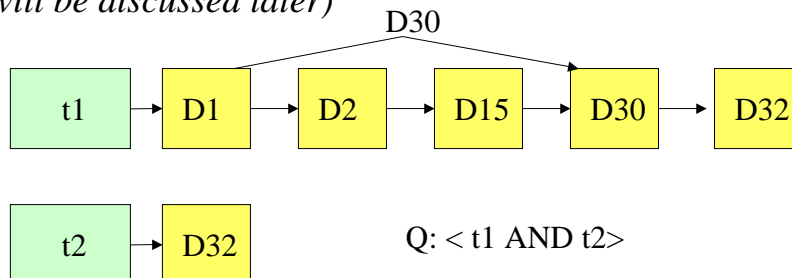
where, $C = f_{\max}$ (domain specific)

© Goharian, Grossman, Frieder, 2002, 2009

9

Skip Pointers

- To optimize the join operation of $O(m+n)$ for posting lists of size m and n
- To optimize search for a given document d in the PL (*will be discussed later*)



© Goharian, Grossman, Frieder, 2002, 2009

10

Positional (Proximity) Index

- Posting List nodes may maintain position of terms in each document for Proximity search.

Apple, 3 → (D1,2, {1,5}) (D2,1, {10}) (D3,3, {1,7, 15}) ...

- An alternative to phrasing
- Expands the PL storage requirements
- Using both phrase and proximity can be combined.

Index Construction Algorithms

All depends on the hardware availability

- Memory-based
 - Assumption: enough memory is available to construct and maintain the entire inverted index.
 - Good if enough memory and small collection
- Disk-based
 - No memory assumption; scaling to large collections
 - Various implementations exist

Term List (Lexicon)

- Usually we have enough memory to store the term list [dictionary] in a hash table in memory.
- Without a perfect hash function (which requires knowledge of all distinct terms), the hash table will have collisions.

Building a Memory-based Inverted Index

- For each document d in the collection
 - For each term t in document d
 - Find term t in the lexicon
 - If term t exists, add a node to its posting list
 - Otherwise,
 - Add term t to the lexicon
 - Add a node to the posting list
- After all documents have been processed, write the inverted index to disk.

Memory-based Inverted Index

- Phase I (parse and read)
 - For each document
 - Identify distinct terms in the document
 - Update, in memory the posting list for each term
- Phase II (write)
 - For each distinct term in the index
 - Write the inverted index to disk (feel free to compress the posting list while writing it)

© Goharian, Grossman, Frieder, 2002, 2009

15

Memory Requirements

- While in memory the posting list is not compressed.
- Typical entry

DocID (4 bytes)	tf (2 bytes)	nextPointer (4 bytes)
--------------------	-----------------	--------------------------

- For an 800,000,000 word collection, 400,000,000 posting list entries were needed (many terms did not result in a posting list entry because of stop words removal and duplicate occurrences of a term within a document).
- With 400,000,000 posting list entries, at 10 bytes per entry, we obtain a memory requirement of 4GB.

© Goharian, Grossman, Frieder, 2002, 2009

16

Memory-Only Analysis

- Time to read and parse
 - $R = B t_r + F t_p$
- Time to write
 - $W = I(t_d + t_r)$
- Took about six hours to index a 5GB collection with a given hardware req.
- Fine, except for memory requirements.

© Goharian, Grossman, Frieder, 2002, 2009

17

Variables

B = Text size (assume 5 GB)

N = Number of Documents (about 500,000)

n = Distinct Terms (lexicon size)

F = Number of Words (about 800,000,000)

f = Number of posting list entries (about 400,000,000)

I = size of compressed file (about 400MB)

L = size of lexicon = 3 MB

Disk seek time = t_s = .01 seconds

Disk Transfer time = t_r = .000005 seconds

Time to compare and swap 10 byte records = t_c .

Time to parse, stem, and look up one term = t_p

M = Main memory available = (figure 40 MB, a low number)

© Goharian, Grossman, Frieder, 2002, 2009

18

Memory Management

- We usually don't have more memory than the size of the document collection.
- Periodically must write inverted index to disk.
- Algorithm must be changed to periodically write to disk a subset of the inverted index I and then merge the subsets.

Memory-based Inverted Index Summary

- Pro
 - Very fast algorithm that is easy to implement.
- Con
 - Doesn't work at all if you run out of main memory. Once memory runs out you start swapping. For small document collections this is a great algorithm, for anything realistic it requires a lot of memory.

Simple Alternatives (that do not Work at All!)

- Simply storing the posting lists on disk would require a tremendous amount of I/O. One estimate shows SIX WEEKS to run this for 5 GB collection.
- Alternatively, we could let the OS take care of the memory and just use virtual memory to solve the problem. This results in significant swapping (13 days).

Disk Resident (General Concept)

- Read fixed chunk of data into memory
- Tokenize
- *If needed* create the *term to term id* mappings
- build [term, doc] pairs; or [term, doc, tf] triples; or [term and its postings] per implementation decisions
- Create intermediate sorted files and write on disk
- Perform m-way merging of intermediate files in memory and write onto the disk
- The outcome is one final inverted index on disk.

One Solution: Sort-Inversion

- Phase I
 - Create temp files of triples (termID, docID, tf)
- Phase II
 - Sort the triples using external mergesort
- Phase III
 - Merge the sorted triples files (2-way; m-way)
- Phase IV
 - Build Inverted index from sorted triples

© Goharian, Grossman, Frieder, 2002, 2009

23

Sort-Inversion (Cont'd)

- Phase I (parse and build temp file)
 - For each document
 - Parse text into terms, assign a term to a termID (use an internal *index* for this)
 - For each distinct term in the document
 - Write an entry to a temporary file with only triples <termID, docID, tf)
- Phase II (make sorted *runs*, to prepare for merge)
 - Do Until End of Temporary File
 - Sort the triples in memory by term id and doc id.
 - Write them out in a sorted run on disk.

© Goharian, Grossman, Frieder, 2002, 2009

24

Sort-Inversion (Cont'd)

Run1:	tid	did	tf	Sorted:	tid	did	tf
	1	d1	2		1	d1	2
	3	d1	1		1	d2	1
	5	d1	2		2	d1	4
	2	d1	4		2	d2	3
	4	d1	1		3	d1	1
	1	d2	1		4	d1	1
	2	d2	3		5	d1	2
	5	d2	3		5	d2	3

Run2:	tid	did	tf	Sorted:	tid	did	tf
	1	d3	2		1	d3	2
	2	d3	1		1	d4	2
	4	d3	3		2	d3	1
	2	d4	2		2	d4	2
	3	d4	1		3	d4	1
	5	d4	2		4	d3	3
	4	d4	1		4	d4	1
	1	d4	2		5	d4	2

© Goharian, Grossman, Frieder, 2002, 2009

25

Sort-Inversion (Cont'd)

- Phase III (merge the runs)
 - Repeat until there is only one *run*
 - Merge pair-wise (2-way) or m-way sorted runs into a single run.
- Phase IV
 - For each distinct term in final sorted run
 - Start a new inverted file entry.
 - Read all triples for a given term (these will be in sorted order)
 - Build the posting list (feel free to use compression)
 - Write (append) this entry to the inverted index into a binary file.

© Goharian, Grossman, Frieder, 2002, 2009

26

Sort-Inversion (Cont'd)

Sorted
Run1:

tid	did	tf
1	d1	2
1	d2	1
2	d1	4
2	d2	3
3	d1	1
4	d1	1
5	d1	2
5	d2	3

Merged:

tid	did	tf
1	d1	2
1	d2	1
1	d3	2
1	d4	2
2	d1	4
2	d2	3
2	d3	1
2	d4	2
3	d1	1
3	d4	1
4	d1	1
4	d3	3
4	d4	1
5	d1	2
5	d2	3
5	d4	2

Sorted
Run2:

tid	did	tf
1	d3	2
1	d4	2
2	d3	1
2	d4	2
3	d4	1
4	d3	3
4	d4	1
5	d4	2

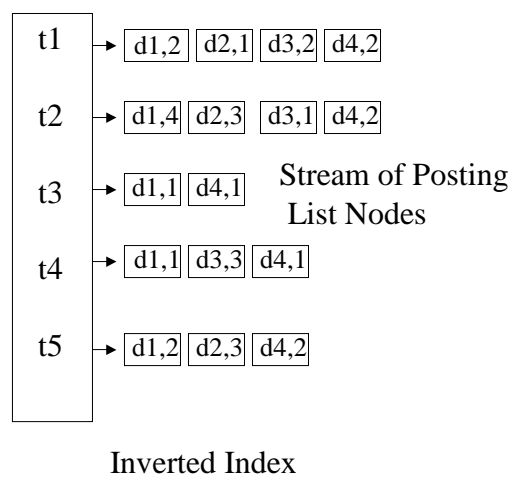
© Goharian, Grossman, Frieder, 2002, 2009

27

Sort-Inversion (Cont'd)

Final
Sorted
Run:

tid	did	tf
1	d1	2
1	d2	1
1	d3	2
1	d4	2
2	d1	4
2	d2	3
2	d3	1
2	d4	2
3	d1	1
3	d4	1
4	d1	1
4	d3	3
4	d4	1
5	d1	2
5	d2	3
5	d4	2



© Goharian, Grossman, Frieder, 2002, 2009

28

Alternatives

- Instead of triples:
 - tid, doc-id pairs: after sorting then create the posting with tf
 - For each term create the posting directly in memory (posting is the list of docs)
- Instead of term id:
 - No need for *term id* at all. Lexicon keeps the *terms*
 - No need for extra structure for the term to term id mapping

Time (estimates from testing)

- Read and Parse (5 hours)
 - Write temp file (30 minutes)
 - Sort (4 hours, 3 hours sort + 1 hour for r/w of temp file)
 - With 40MB and 400,000,000 triples in the temporary file we can hold 4,000,000, 10 byte triples in memory. So we have 100 runs.
 - Merge (7 hours for I/O, 2 hours computation)
 - $\log_{10} 100 = 7$ passes, each pass is a full r/w of temp file
 - compute time to do the merge (comparisons of runs)
 - Read sorted temp file and build inverted index (1.5 hours)
 - Total time (about 20 hours)
- *note: the timing are based on a given processing speed.

Analysis

- Time to read and parse, write file
 - $R = B t_r + F t_p + 10 f t_r$
- Time to sort
 - $20 f t_r$ (read and write) + $R(1.2k \log k) t_c$ (time to sort a run)
 - k number of triples that fit into memory
 - R number of runs to merge
- Time to merge
 - $(\log R) (20 f t_r + f t_c)$ (read, write, compare a run)
- Write the final inverted file
 - $10 f t_r + I(t_d + t_r)$

© Goharian, Grossman, Frieder, 2002, 2009

31

Disk-based Inverted Index Summary

- Pro
 - Not as fast as memory based, but at least is usable. Sort and merge time dominates cost.
- Con
 - Requires significant additional space.

© Goharian, Grossman, Frieder, 2002, 2009

32

Distributed Index

- Single index – traditional approach
 - Use single fast machine
 - Good for some applications (enterprise search)
- Distributed index
 - Use several fast machines (servers)
 - Good for indexing tens of billions of pages (large scale)

Distributed Index (Cont'd)

- Web search tools access data distributed on servers worldwide but indexed centrally.
- Most of these systems have a *partitioned index* with a *centralized control*.
- Partitioning of index across multiple machines, based on terms or documents
- Using content-index, sending requests to those server that have the data

Distributed Index Construction

- Not possible on a single machine
- Various architecture for distributed indexing
- MapReduce architecture (a term-partitioned index)
 - Master node assigns tasks to worker nodes to split up the computing jobs:
 - Parsing (map phase) & building localized <term, doc> pairs
 - Combining posting pairs for each term (reduce phase)

© Goharian, Grossman, Frieder, 2002, 2009

35

Signature Files

- A signature is an encoding of a document, using few bits.
- Each signature may represent multiple docs.
- Thus, Two-Phase query processing:
 - Phase 1: scan signatures and identify candidate signatures
 - Phase 2: scan original text of the candidate signatures

© Goharian, Grossman, Frieder, 2002, 2009

36

Construction of Signatures

- Often using one or more hashing functions for each term to set a bit in a signature:
 - h(information): 0101;
 - h(retrieval): 1010;
 - h(security): 0011
- OR the term signatures of a document to build document signature
 - D1: Information retrieval: 1111
 - D2: security information: 0111

© Goharian, Grossman, Frieder, 2002, 2009

37

Processing of Signatures

- Boolean AND between query and document
 - Q> information: 0101
 - D1: Information retrieval: 1111
 - D2: security information: 0111
 - ⇒ match: D1 and D2
 - Q> security: 0011
 - D1: Information retrieval: 1111
 - D2: security information: 0111
 - ⇒ match: D1 and D2 - *false positive (false drop)*

© Goharian, Grossman, Frieder, 2002, 2009

38

Processing of Signatures

- Boolean AND queries: all query terms must return true
- Boolean OR queries: some query terms must return true

Signature Files Summary

- Pros:
 - Useful if can fit into memory
 - Easy to add or remove documents (signatures) as compared to inverted index.
 - The order of signature in the signature file does not matter.
- Cons:
 - Two phased processing for false matches
 - Does not rank the retrieved documents

Relational Approach will be
discussed in a separate set of
slides!