Building and Compression Techniques for Inverted Files

Roi Blanco

Dpt. Computing Science, University of A Coruna

December 13, 2005

- How to sort and manage large amounts of data effectively, so the retrieval can be done without undue inconveniences
- Algorithms for indexing text collections
- Compression techniques for inverted files, relative performance.

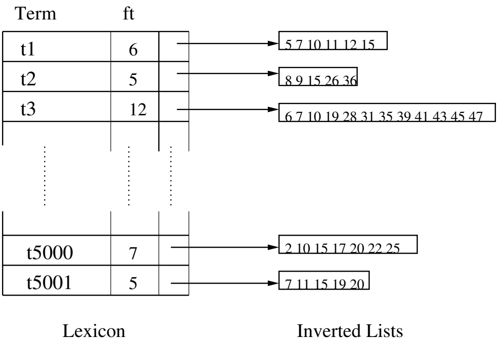
1. Inverted File Building
   - Introduction
   - Structure
   - Simple inversion algorithms
   - Indexing in terrier
   - Disk-based inversion
   - Merging algorithms
   - Single Pass Efficient IF Building
   - Dynamic Index Building

2. Compression for Inverted Files
   - Basics
   - Static coding
   - Cluster-based Compression
   - Query performance
   - Document Order

Introduction

- How to organize the information so that queries can be resolved and relevant portions of data located and extracted
- Domain-specific data structures, such as Inverted Files, bitmaps, signature files...
- Building an IF for a big collection can be challenging

Structure



- $< t; f_t; d_{t1}, d_{t2}, \ldots, d_{f_t} >$
- Also, positions, fields, etc.

Computational model

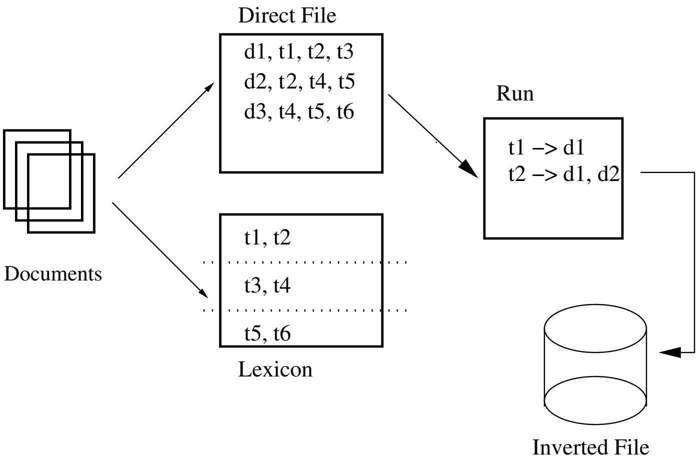
Table: Symbols for the analysis of the inversion approaches

Parameter	Symbol	Value
Main memory Size (MB)	M	256
Average disk seek time (sec)	t_s	9×10^{-3}
Disk transfer time (sec/byte)	t_t	5×10^{-8}
Time to parse one term (sec)	t_p	8×10^{-7}
Time to compress a byte (sec)	t_c	5×10^{-7}
Time to lookup a term in the lexicon (sec)	t_l	6×10^{-7}
Time to swap two 12-byte records (sec)	t_w	1×10^{-7}

Table: Statistics and symbols for three reference text collections

	Symbol	Col A	Col B	Col C
Size (Mb)	S	11	5,120	20,480
Distinct terms	n	41584	3×10^6	6.8×10^6
Term Occurrences ($\times 10^6$)	C	0.9	323.9	1,261.5
Documents	N	5440	928,113	3,560,951
Postings ($\times 10^6$)	P	0.5	140.6	543.2
Size of the Inverted File	I	0.7	180	697

Indexing in terrier



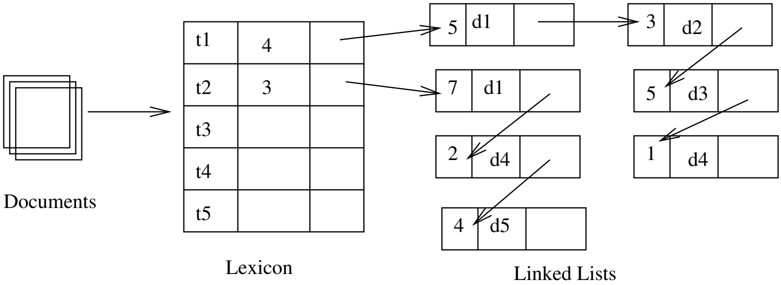
- Needs a Direct File
- Performs several passes to the Direct File loading terms in memory and writing the postings back to disk
- Requires the whole lexicon in memory
-

$$
\begin{aligned}
T = St_t + C(t_p + t_l) + & \quad \text{(read, parse, lookup)} \\
l'(t_c + t_t) + & \quad \text{(direct file)} \\
kl'(t_t + t_c) + kt_s + l(t_c + t_t) & \quad \text{(inverted file)}
\end{aligned}
$$

Simple in-memory inversion

- Inversion matrix filled by columns
- Two passes over the text.
- The space needed is $\approx 2(4) \times |n||N|$
- Saving space by storing the rows as 12-bytes linked lists (random order access)
- Time considering random disk accesses in virtual memory

Disk-based inversion

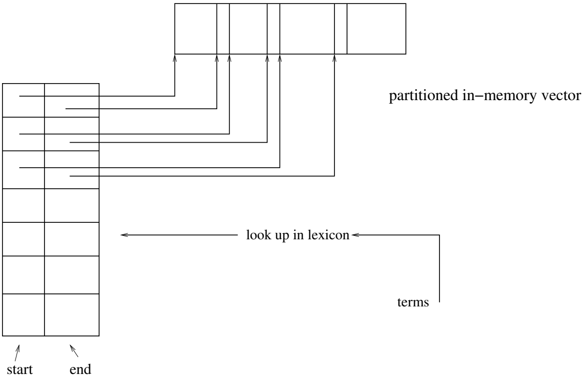


- Harman and Candela (1990)
- Accumulate postings on disk and not on main memory
- Allocate a new file on disk for the inverted lists in lexicographical order.
- Single pass over the collection
- Needs an in-memory lexicon
- Extra disk space, random accesses to the posting file makes the running times not practical
- Time depends as well in the size of the IF that can be mapped into memory.
- $$\begin{aligned} T = {} & St_t + C(t_p + t_l) + 10Pt_t + && \text{(read, parse, lookup, write)} \\ & Pt_s/v + 10Pt_t + && \text{(traverse lists)} \\ & I(t_c + t_t) && \text{(compress, write out IF).} \end{aligned}$$

Two-pass in-memory inversion

- Gathers statistics over the text for efficient allocation/compression of data structures
- Postings belonging to the same term are stored together.
- The collection can be subdivided in several loads.
- $$\begin{aligned} T = {} & St_t + C(t_p + t_l) + && \text{(first pass)} \\ & St_t + C(t_p + t_l) + I' t_c + && \text{(second pass)} \\ & I' t_c + I(t_c + t_t) && \text{(decompress, recompress, write out)} \end{aligned}$$

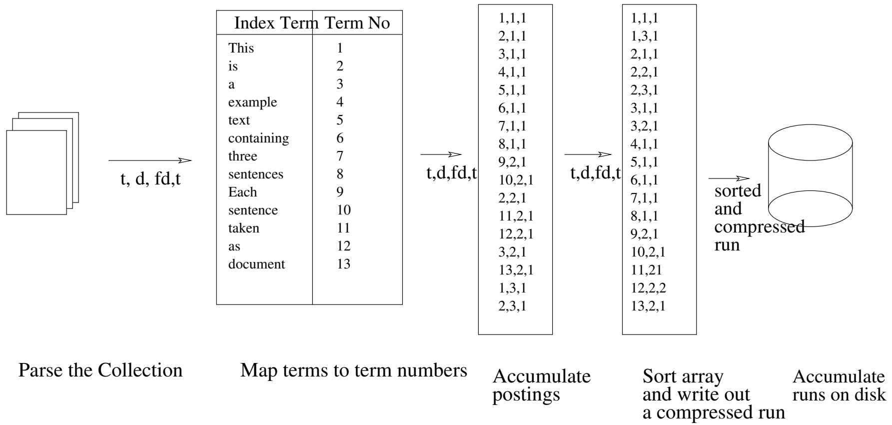
Merging example



Sort-based in-situ inversion

- Applicable to collections of any size
- Operates with limited amounts of memory and does not require large temporary disk space
- Memory threshold, K runs written to disk

Merging example



K-way merge

- Assign a memory buffer to each of the runs
- Lists not sorted in lexicographical order
- In-situ multiway merge with block permutations
- These two approaches require the whole vocabulary in main memory
-
$$
\begin{aligned}
T = {} & S t_t + C(t_p + t_l) + && \text{(read, parse)}\\
& K(1.2k \log_2 k) t_w + I(t_t + t_c) + && \text{(sort, compress, write)}\\
& P\lceil \log_2 K \rceil t_w + (I + I')(t_s/u + t_t + t_c) + && \text{(merge, recompress)}\\
& 2I(t_s/b + t_t) && \text{(permute).}
\end{aligned}
$$

Single Pass Efficient IF Building

- The basic idea is to assign to each index term in the lexicon a dynamic bitvector that accumulates the postings in a compressed format
- No statistics collected for compression
- Process the collection filling and compressing postings on the fly using a partial lexicon

- Merging is as in the sort based approach.
- Avoid keeping the lexicon in memory (less runs)
- Preserves the lexicographical order
- The drawback is keeping the lexicon terms indexed in the runs
- Strings can be front coded to save space
- $$\begin{aligned} T \approx\ & St_t + C(t_p + t_l) + && \text{(read, parse)} \\ & I'(t_c + t_t) + && \text{(compress, write)} \\ & P\lceil \log_2 K \rceil t_w + (I + I')(t_s/u + t_t + t_c) + && \text{(merge, recompress)} \\ & 2I(t_s/b + t_t) && \text{(permute)} \end{aligned}$$
- It is asymptotically more efficient for a fixed memory size

Dynamic Index Building

- Off-line vs On-line index construction
- Adds complexity and tensions (query throughput, document insertion rate, disk space) to the process of index building
- Postings are stored in contiguous disk partitions, so adding a new document is very slow
- The IF is stored in two parts, an in-memory part of recently included documents and an on-disk component.

Merge-based Index Update

- Merging event for index maintenance
- Different approaches:
  - Rebuilding
  - Remerge update
  - In-place update
  - Multiple partitions (geometric partitioning)

Compression in IFs

- More storage to index more documents
- Useful for IF building techniques
- Less bandwidth usage in distributed/mobile applications
- Faster access to disk

Static coding

- Modeling (Huffman, arithmetic) vs fixed codings
- Seeing the IF as an ascending list of integers ($d_k < d_{k+1}$)
- The list is stored as an initial position followed by a list of *d-gaps*
- $< 7; 2, 5, 10, 22, 27, 34, 45 > \longrightarrow < 7; 2, 3, 5, 12, 5, 7, 11 >$
- Best encoding depends on the number distribution

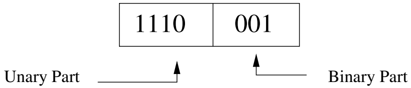
Types of static codings

- Non Parametric Codes
  - Binary
  - Unary
  - γ, δ
  - Variable-byte
- Parametric
  - Golomb
  - Skewed Golomb

Unary Coding

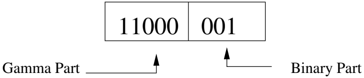
- Encode x with $(x-1)$ 0's followed with a 1
- $$\begin{aligned} 2 &= \quad 01 \\ 5 &= \quad 00001 \\ 9 &= \quad 000000001 \end{aligned}$$
- Too biased for short gaps
- $l(x) = x \Rightarrow Pr(x) = 2^{-x}$

γ Coding



- Split the integer in two parts:
  - $1 + \lfloor \log x \rfloor$ in unary
  - $x - 2^{\log x}$ in binary with $\lfloor \log x \rfloor$ bits
- Decoding is easy
  - Read unary code c_u
  - Read a binary code with $c_u - 1$ bits
- $l(x) = 1 + 2 \log x \Rightarrow Pr(x) = \frac{1}{2x^2}$

δ Coding



- Split the integer in two parts:
  - $1 + \lfloor \log x \rfloor$ in gamma
  - $x - 2^{\log x}$ in binary with $\lfloor \log x \rfloor$ bits
- Decoding
  - Read γ code as before with c_u
  - Read a binary code with $c_u - 1$ bits
- $l(x) = 1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor \Rightarrow Pr(x) = \frac{1}{2x(\log x)^2}$

Variable Byte Coding

- Use the minimum number of bytes to encode an integer with binary coding
- 7 bits to store the number, 1 bit to flag if there is another byte
- $$\begin{array}{ll} \text{if } 0 < x < 128 & 1 \text{ byte} \\ \text{if } 128 \leq x < 16384 & 2 \text{ bytes} \\ \text{if } 16384 \leq x < 2097152 & 3 \text{ bytes} \end{array}$$
- Fast decoding speed (byte oriented)
- A little wasteful for small integers

Bernoulli Model

- Use the density of the pointers in the Inverted File. If the number of pointers to be stored is f there is a probability of $f/(N \times n)$ that any random document contains any random term.
- The probability of a gap x can be modeled by a Bernoulli process (1 success).
- $Pr(x) = (1-p)^{x-1}p$ (Geometric Distribution)

Golomb Coding

- The probabilities of the geometric distribution can be generated by a coding method:
- x is coded in two parts
  - $q+1$ in unary, where $q = \lfloor (x-1)/b \rfloor$
  - The remainder $x - qb - 1$ in binary with $\log b$ bits
- b can be chosen so that it minimizes the average number of $\frac{N \times n}{f}$
- Rice codes $(b = 2^x)$

Local Golomb

- It is possible to drive a different code for each posting list
- Choose a different b according to f_t
- Small overhead for b

Cluster-based Compression

- Context-sensitive compression
- Term appearances are not random events, but clusters
- Methods for skewing the probability distribution (LLRUN) to capture the true distribution of d-gaps.
- Markov-3 algorithm, Markov chain with different states that represent if a term is in a cluster or not.

Interpolative Coding

- Exploits clustering implicitly
- Codes sequences of d-gaps in a block
- Encodes the integer in the posting list recursively
- Integers can be encoded with 0 bits
- Unique-Order Interpolative Coding

Query performance

Cheng et al, IP&M 2005

	LATimes		FBIS	
	time(μs)	bpg	time(μs)	bpg
γ	2047	5.38	2148	5.63
Golomb	2482	5.27	2635	5.31
LLRUN	2789	4.63	2905	4.78
Interpolative	4118	4.58	4261	4.65
Variable Byte	3470	8.88	3877	8.89
Carryover-12	2263	6.23	2316	6.13
Unique-Order IC	2107	4.66	2117	4.74

Document Order

- Methods that reorders the document identifiers in the collection, so that the clustering property is enhanced.
- Improvements up to 20% in compression ratios
- New compression techniques that use the fact of a clustered order for faster decompression times
- New fast algorithms for assignment
- Improvements in query performance (up to 20% again)