Inverted Index

Overview

- Structure of an inverted index
- Building an inverted index
- Compression
  - Posting list compression
  - Term list compression
- Thresholding
  - Document
  - Query

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 queries will be supported).
        - » Position may be expressed as section, paragraph, sentence, location within sentence ,

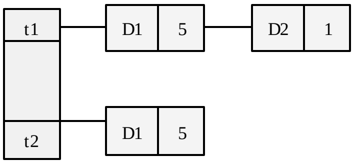
---

Inverted Index: Assumptions

- Assumptions
  - query will happen frequently
    - Find all documents that contain term *t*
  - delete will be rare
    - Delete document 52
  - update will be rare
    - Correct the spelling of term *t* in document 52
  - add will not happen too often
    - Add new documents

Inverted Index: Structure

- Term list
- Posting list



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.

Building an Inverted Index

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

Memory Management

- Usually we have enough memory to store the term list in a hash table in memory.
- If we are worried about the number of terms exhausing memory, a B-tree can be used instead (B-trees will take more space than a hash table).
- Without a perfect hash function (which requires knowledge of all distinct terms), the hash table will have collisions.

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.

Inverted Index Construction: Periodic write to disk

```
For each document d in the collection
    Begin
        numSubSet = 1
        While memory exists:
            For each term t in document d
                Find term t in the term dictionary
                If term t exists, add a node to its posting list
                Otherwise, add term t to the term dictionary
        Write SubSet of Inverted index to disk
        numSubSet = numSubSet + 1
        Free memory
    End
For I = 1 to numSubSet
    Merge SubSet I with Inverted Index
```

Output of Inverted Index

- Index
  - maps each term to a posting list which contains a document number and term frequency
- Document
  - maps each document number to a file or location, long name, weight, etc.
- Term
  - For each term, the total number of documents that contain the term. Might also contain the terms "type" – date, time, string, number, etc.

Compression of Inverted Index

- I/O to read a posting list is reduced if the inverted index takes less storage
- Stop words eliminate about half the size of an inverted index. "the" occurs in 7 percent of English text.
- Other compression
  - Posting List
  - Term Dictionary
- Half of terms occur only once *(hapax legomena)* so they only have one entry in their posting list
- Problem is some terms have very long posting lists -- in Excite's search engine 1997 occurs 7 million times.

Things to Compress

- Term name in the term list
- Term Frequency in each posting list entry
- Document Identifier in each posting list entry

Data Compression

- Applied to posting lists
  - term: (d_1,tf_1), (d_2,tf_2), ... (d_n,tf_n)
- Documents are ordered, so each d_i is replaced by the interval difference, namely, $d_i - d_{i-1}$
- Numbers are encoded using fewer bits for smaller, common numbers
- Index is reduced to 10-15% of database size

Compressing *tf:* Elias Encoding

X	γ
1	0
2	10 0
3	10 1
4	110 00
5	110 01
6	110 10
7	110 11
8	1110 000
63	111110 11111

To represent a value X:

- $\lfloor \log_2 X \rfloor$ ones representing the highest power of 2 not exceeding X
- a 0 marker
- $\lfloor \log_2 X \rfloor$ bits representing to represent the remainder X - 2^ $\lfloor \log_2 X \rfloor$ in binary.
- The smaller the integer, the fewer the bits used to represent the value. Most *tf's* are relatively small.

Elias Code

1 = 0
2 = 1 0 0
3 = 1 0 1
4 = 11 0 00
5 = 11 0 01
6 = 11 0 10
7 = 11 0 11
8 = 111 0 000
9 = 111 0 001

For 63, its 2^5 = 32 + 31 in binary (11111)
11111 0 11111

- 3 parts, not byte aligned
  1. n ones, one for each bit in part 3
  2. a 0 to mark the end of part 1.
  3. the next n numbers in binary

Instead of two bytes for the tf we now are using only a few bits.

Variable Length Compress Used for Document Identifier

- Document identifiers (the difference) may not all be small
- A generalization of Elias is to develop a vector V with the powers of some integer in its component.
- Examples
  - V <1,2,4,8,16,32>
  - V <2,4,8,16,32,64> ,etc.

Variable Length Encoding (cont.)

- Choose Vector V
- For an integer *x* to be compressed, find *k* such that sum of the vector components is greater than *x.*
- Encode *k-1* in unary.
- Now subtract the sum of the first *k-1* components of V from *x*. The difference is *d.*
- Encode a 0 stop bit
- Encode *d* in binary.

Variable Length Encoding (Example)

- For $x = 7$
- Using Vector <1,2,4,8,16>, it requires the sum of <1,2,4> to exceed x. Hence the index k is 3 and k-*1* is 2. Encode 2 in unary.
- The remainder is 7 - (1+2) = 3, encode this in binary after the stop bit.
- To encode x use *11011*

Changing V

- If V contains larger values, fewer bits will be needed to represent larger values.
- A constant b can be varied such that V is b, $2b, 4b, 8b, 16b, 32b, 64b$.
- b can be varied for *each posting list*
- Use the median of the document identifier differences for each posting list.
- Requires knowledge of how large a posting list, but you know this in the final stages of index development.

Example

- Suppose a posting list had:

  term --> d_4 d_{10} d_{20} d_{30} d_{35}
- Differences are 6, 10, 10, 5 so median is 10
- V is now <10, 20, 30, 40>
- To encode the differences we have:

$\mathbf{4_{10}}$	$\mathbf{6_{10}}$	$\mathbf{10_{10}}$	$\mathbf{10_{10}}$	$\mathbf{5_{10}}$
00011	00101	01001	01001	00100

- Note: We never needed *any* leading bits. With a vector of <1,2,4,8,16> we would have had:

$\mathbf{4_{10}}$	$\mathbf{6_{10}}$	$\mathbf{10_{10}}$	$\mathbf{10_{10}}$	$\mathbf{5_{10}}$
11000	11010	1110010	1110010	11001

Variable length we used 25 bits. Regular Elias we used 29 bits.

Example 2

- To encode 15 with vector of <10, 20, …>
  - k+1 = 2, encode this in unary as 11
  - 10 < 15 <= 30
  - Encode the stop bit 0
  - Encode r = 15 - 10 - 1 = 4, encode this in binary as 0100. See p. 141.
  - So we have 1100100 (seven bits)
- In Elias code vector is <1,2,4,8, 16>
  - so k = 3
  - 1 + 2 + 4 < 15 <= 15
  - k+1 = 4, encode this in unary
  - residual is r = 15 - (1 + 2 + 4) -1 = 7
  - Encode 7 in binary, 111
- So we have 11110111 (eight bits)

Byte-Aligned codes

0-63	00xxxxxx
64-16K	01xxxxxx xxxxxxxx
16K-4M	10xxxxxx xxxxxxxx xxxxxxxx
4M-1G	11xxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
0	00000000
1	00000001
...	...
63	00111111
64	01000000 00000000
65	01000000 00000001

The hope here is that the document distance between posting list nodes will be small.

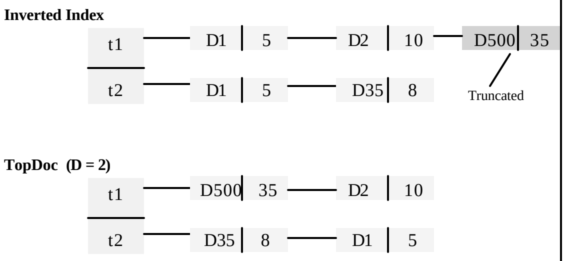
Compression Summary

- Pro
  - Can reduce I/O for query of inverted index.
  - Reduce storage requirements of inverted index.
- Con
  - Takes longer to build the inverted index.
  - Software becomes *much* more complicated.
  - Uncompress required at query time – note that this time is usually offset by dramatic reduction in I/O.

Top Docs

- Other structures may be built at index creation to optimize performance.
- Instead of retrieving the whole posting list, we might want to only retrieve the top *x* documents where the documents are ranked by weight.
- A separate structure with sorted, truncated posting lists may be produced.

Inverted Index and TopDoc



Inverted Index

t1 — D1 | 5 — D2 | 10 — D500 | 35

t2 — D1 | 5 — D35 | 8

Truncated

TopDoc (D = 2)

t1 — D500 | 35 — D2 | 10

t2 — D35 | 8 — D1 | 5

Top Doc Summary

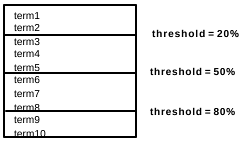
- Pro
  - Avoids need to retrieve the entire posting list
  - Dramatic savings on efficiency for large posting lists
- Con
  - Not feasible for Boolean queries
  - Can miss some relevant documents due to truncation

Query Threshold

Consider a query with terms t_1, t_2, t_3, ..., t_n.

Sort the terms by their frequency across the collection (least frequent terms appear first).

Define a threshold as the percentage of terms taken in the original query in a newly created reduced query.



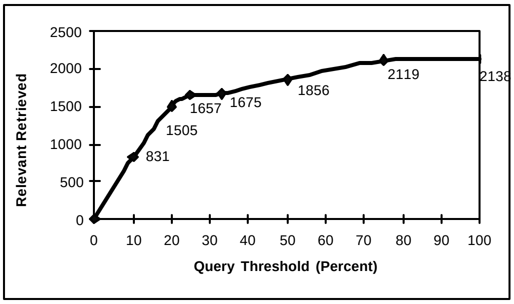
term1
term2
term3
term4
term5
term6
term7
term8
term9
term10

threshold = 20%

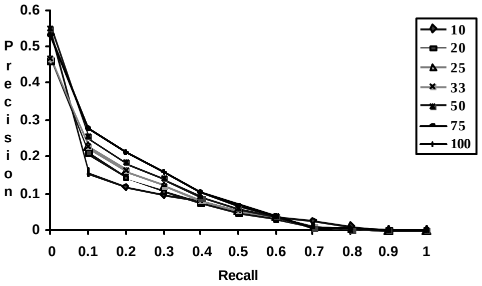
threshold = 50%

threshold = 80%

Relevant Retrieved for Varying Query Thresholds



Precision/Recall



Threshold Summary

- Pro
  - Avoids large posting lists
  - Dramatic savings on efficiency when large posting list is not retrieved
  - Effectiveness does not degrade (as long as we do not threshold too much) because we are omitting only those terms with long posting lists
- Con
  - Still can have some very long posting lists