Inverted Index Algorithm and Compression

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 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
```

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: $(d_1,tf_1), (d_2,tf_2), \ldots (d_n,tf_n)$
- 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 or equal to *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 (cont.)

- Formula to find k is…

$$\Sigma_{i=1}^{k-1}(V_i) < x \le \Sigma_{i=1}^{k}(V_i)$$

- remainder = $d = x - \Sigma_{i=1}^{k-1}(V_i) - 1$
- Now the result will be made of 3 parts
  - Encode *k-1* in unary.
  - Encode a **0** stop bit
  - Encode the remainder ***d*** in $[\log_2 V_k]$ bits.

Variable Length Encoding (Example 1)

- 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) - 1 = 3, encode this in binary after the stop bit.
- To encode *x* use *11011*

Example 2

- To encode 9 with vector of <1, 2, 4, 8, 16, 32>
  - If k=3 : 1+2+4 = 7
  - And if k = 4 : 1+2+4+8 = 15
    - We select k = 4 …. So encode (k -1) in unary (which is *111*)
  - Encode the stop bit 0
  - Encode r = 9 -7 -1 = 1, encode this in binary as *001* {we encode in 3 bits as $\lceil \log_2 8 \rceil = 3$ }
  - So we have *1110001* (seven bits)
- To encode 9 with new vector that starts with 2 of <2, 4, 8, 16, 32>
  - If k=3
    - we get the equation as : (2 + 4)=6 < 9 <= 14 = (2+4+8)
  - we select k = 3 … so encode encode (k-1) in unary (which is *11*)
  - Encode the stop bit 0
  - Encode r = 9 -6 - 1 = 2, encode this in binary as *010* {we encode in 3 bits as $\lceil \log_2 8 \rceil = 3$ }
  - So we have *110010* (six bits)

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 3

- 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 = *b*
- V is now <10, 20, 30, 40>
- To encode the differences we have:

4_{10}	6_{10}	10_{10}	10_{10}	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:

4_{10}	6_{10}	10_{10}	10_{10}	5_{10}
11000	11010	1110010	1110010	11001

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

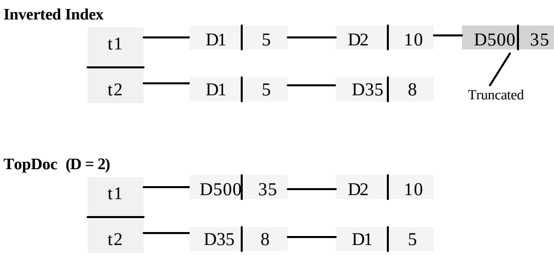
Compression Summary

- 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



Top Doc Summary

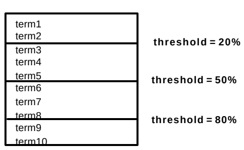
- Pro
  - Avoids need to retrieve the entire posting list
  - Dramatic savings on efficiency for large posting lists
- Con
  - Not feasible for Boolean queries
  - Can miss some relevant documents due to truncation

Query Threshold

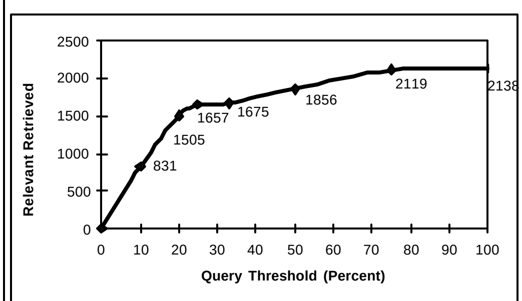
Consider a query with terms t_1, t_2, t_3, ..., t_n.

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

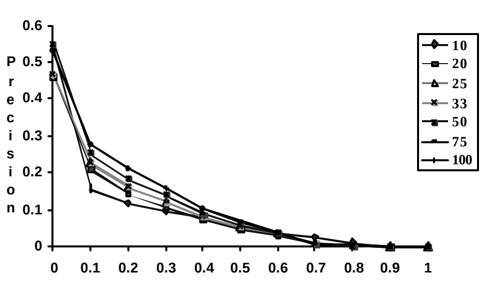
Define a threshold as the percentage of terms taken in the original query in a newly created reduced query.



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