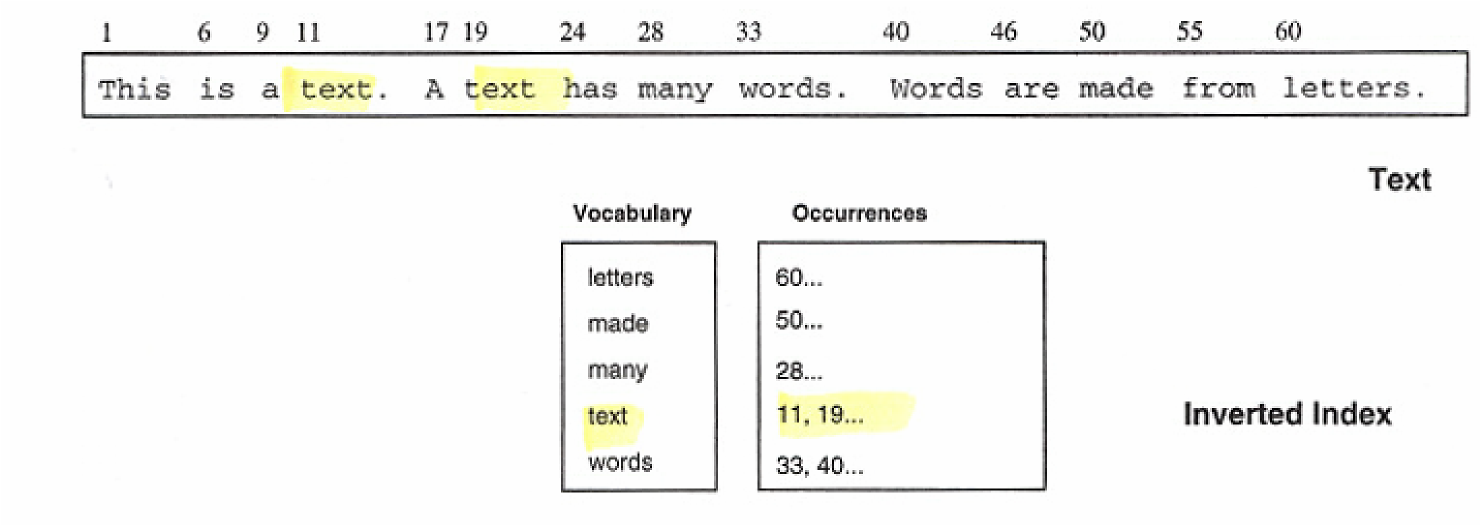
Indexing and Searching Textual Data

Motivation

- Just like in traditional RDBMSs searching for data may be costly
- In a RDB one can take (a lot of) advantage from the well defined structure of (and constraints on) the data
- Linear scan involves finding the occurrences of a pattern in a text when the text is not preprocessed
- Linear scan is appropriate when the text is small and it is the only choice if the text collection is very volatile (i.e. undergoes modifications very frequently) or the index space overhead cannot be afforded.
- Linear scan of the data is not feasible for non-trivial datasets (real life)
- Indices are not optional in IR (not meaning that they are in RDBMS)
- It is worthwhile building and maintaining an index when the collection is large and semi-static (updated at reasonably regular intervals)
- Main approaches:
  - Inverted files (or lists)
  - Suffix arrays
  - Signature files

Inverted Files

- There are two main elements:
  - vocabulary – set of unique terms
  - Occurrences – where those terms appear
- The occurrences can be recorded as terms or byte offsets
- Using term offset simplifies phrase and proximity queries, whereas byte offsets allow direct access to the matching text positions



1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.

Text

Vocabulary	Occurrences
letters	60...
made	50...
many	28...
text	11, 19...
words	33, 40...

Inverted Index

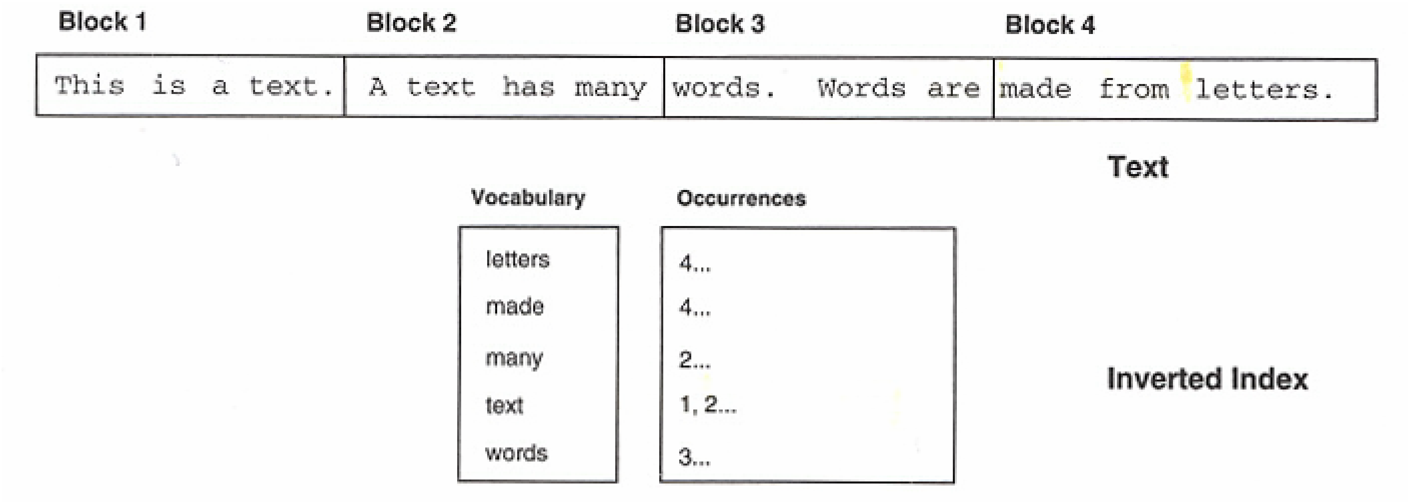
The words are converted to lower-case and some are not indexed. The occurrences point to character positions in the text

Inverted Files

- According to Heap’s law the vocabulary grows as $O(n^{\beta})$, where β is a constant between 0 and 1 dependent on the text, being between 0.4 and 0.6 in practice:
  - The number of indexed terms is often several orders of magnitude smaller when compared to the documents size (Mbs vs Gbs)
- The space consumed by the occurrence list is not trivial. Each time the term appears it must be added to a list in the inverted file:
  - O(n) extra space. Even omitting stopwords the space overhead of the occurrences is between 30% and 40% of the text size.
- That may lead to a quite considerable index overhead

Inverted Files

- Coarser addressing may be used
- All occurrences within a block (perhaps a whole document) are identified by the same block offset
- Much smaller overhead: only 5% overhead over the text size are obtained with this technique
- If the exact occurrence positions are required (e.g., proximity searches), then an online search over the qualifying blocks has to be performed (hardly feasible,specially on-line)



Block 1	Block 2	Block 3	Block 4
This is a text.	A text has many	words. Words are	made from letters.

Text

Vocabulary	Occurrences
letters	4...
made	4...
many	2...
text	1, 2...
words	3...

Inverted Index

- The sample text split into four blocks, and an inverted index using block addressing built on it. The occurrences denote block numbers. Notice that both occurrences of ‘words’ collapsed into one.

Inverted Files

Index	Small collection (1Mb)		Medium collection (200Mb)		Large collection (2Gb)	
Addressing words	**45%**	**73%**	**36%**	**64%**	**35%**	**63%**
Addressing documents (10kb/document)	**19%**	**26%**	**18%**	**32%**	**26%**	**47%**
Addressing 64K blocks	**27%**	**41%**	**18%**	**32%**	**5%**	**9%**
Addressing 256 blocks	**18%**	**25%**	**1.7%**	**2.4%**	**0.5%**	**0.7%**

- Sizes of an inverted file as approximate percentages of the size the whole text collection. For each collection, the right column considers that stopwords are not indexed while the left column considers that all words are indexed.
- The blocks can be of fixed size (imposing a logical block structure over the text database) or they can be defined using the natural division of the text collection into files, documents, Web pages, or others
- The division into blocks of fixed size improves efficiency at retrieval time
- Alternatively, the division using natural cuts may eliminate the need for online traversal:
  - If one block per retrieval unit is used and the exact match positions are not required, there is no need to traverse the text for single-word queries, since it is enough to know which retrieval units to report. But if, on the other hand, many retrieval units are packed into a single block, the block has to be traversed to determine which units to retrieve

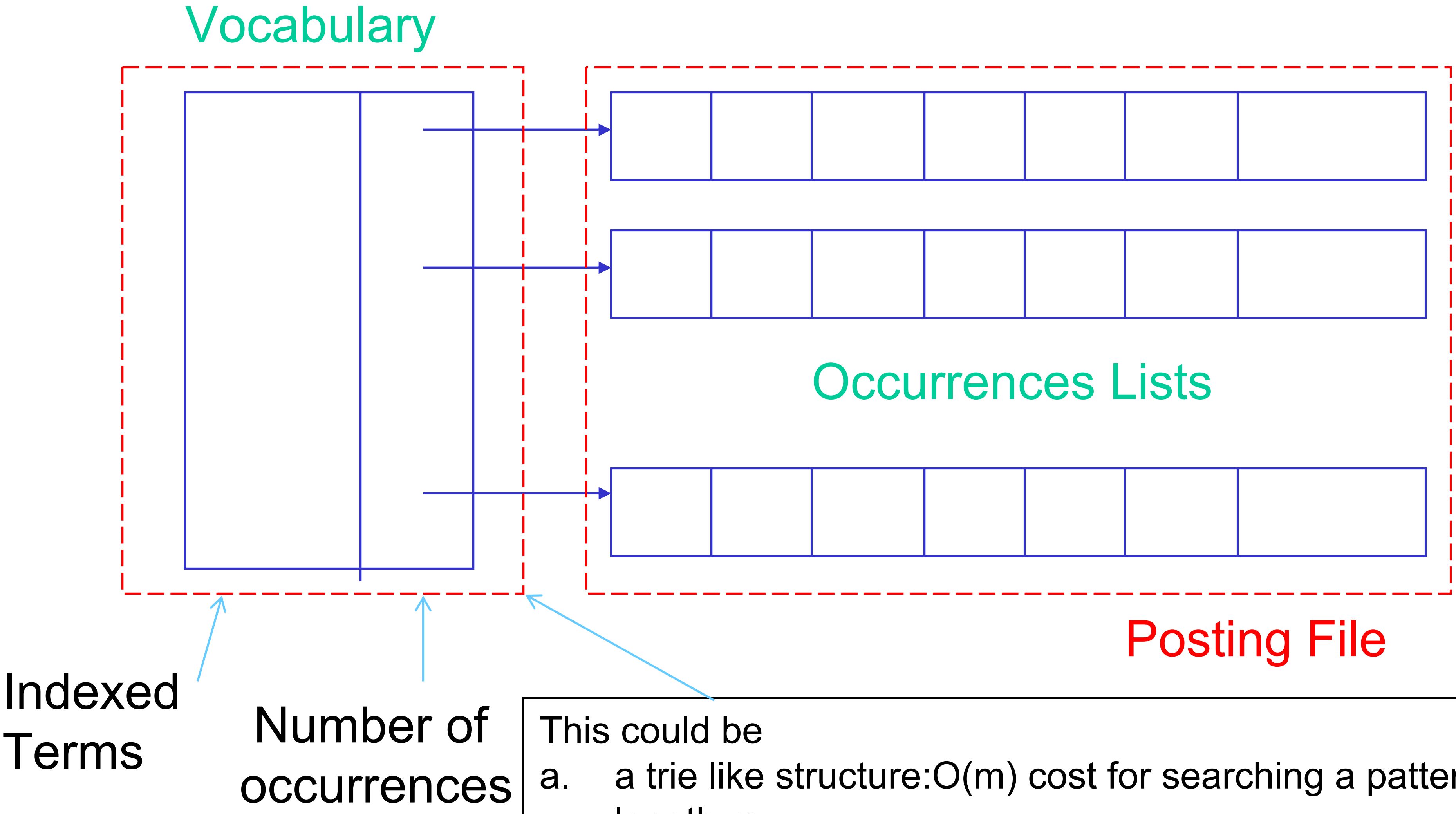
Inverted Files - searching

- Searching using an inverted file
  - Vocabulary search
    - The terms used in the query are searched separately. Notice that phrases and proximity queries are split into single words.
  - Retrieval of occurrences lists
    - The lists of the occurrences of all the words found are retrieved
  - Filtering answer
    - If the query was boolean then the retrieved lists have to be “booleany” processed as well
    - If the inverted file used blocking and the query used proximity (for instance) then the actual byte/term offset has to be obtained from the documents

Inverted Files - searching

- Processing the lists of occurrences (filtering the answer set) may be critical
- For instance, how to process a proximity query (involving two terms) ?
  - The lists are built in increasing order, so they may be traversed in a synchronous way, and each occurrence is checked for the proximity
  - If one list is much shorter than the others , it may be better to binary search its elements into longer lists instead of performing a linear merge. It is possible to prove using Zipf’s law that this is normally the case. This is important because the most time-demanding operation on inverted indices is the merging or intersection of the lists of occurrences.
- What if blocking is used ?
  - No positional information is kept, so a linear scan of the document is required. It is then better to intersect the lists to obtain the blocks which contain all the searched words and then sequentially search in those blocks only.
- The traversal and merging of the obtained lists are sensitive operations
- Using Heap’s and the Zipf’s laws, it has been demonstrated that the cost of solving queries is sublinear in the text size, even for complex queries involving list merging. The time complexity is O(n^a), where a depends on the query and is close to 0.4…0.8 for queries with reasonable selectivity.

Inverted Files - layout



This could be

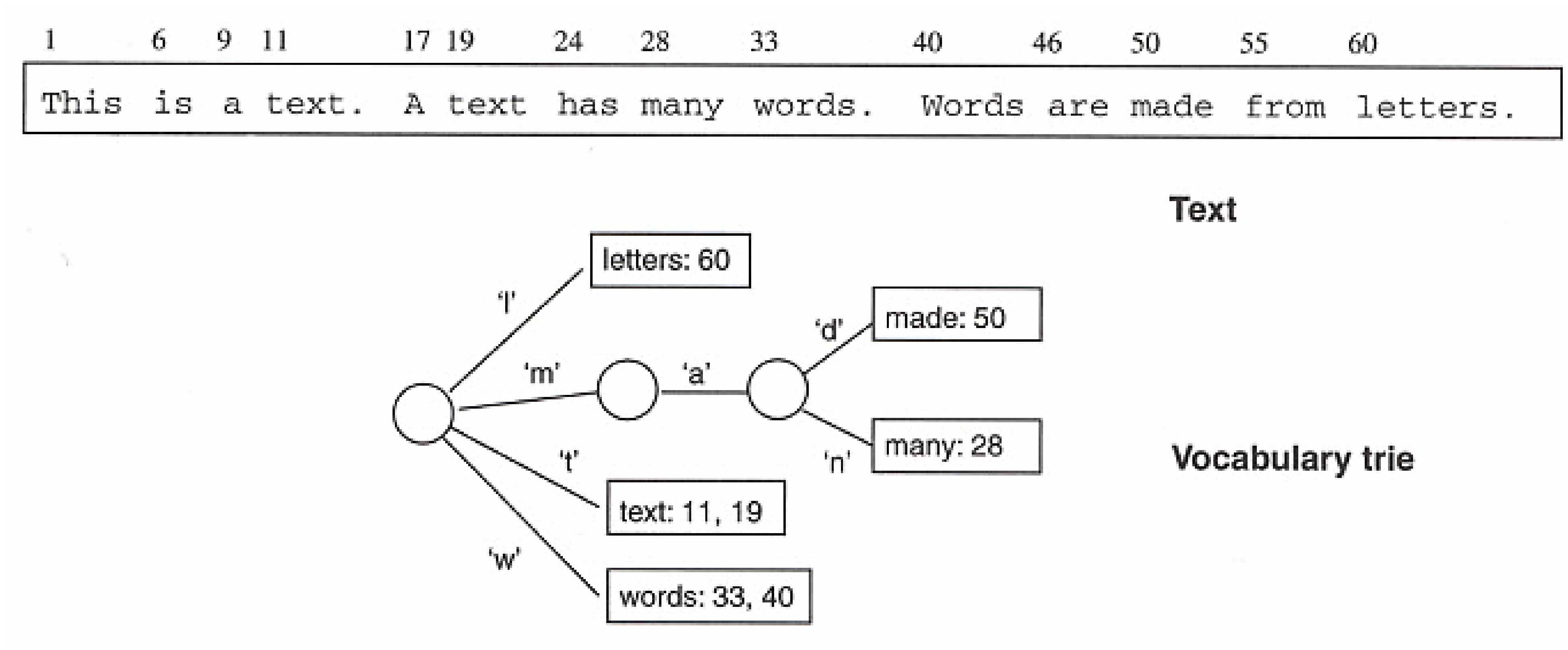
a. a trie like structure:O(m) cost for searching a pattern of length m

b. a hash table: O(1) cost for searching a pattern of length m

c. simple storing of words in lexicographical order

- O(log n) cost for binary searching a pattern in a text of n terms.
- cheaper in space

Inverted Files-layout



Implementing the cosine rule

- Concepts
  - Making the ranking process efficient in terms of time and space
  - Two main issues:
    - How to store the within-document frequencies
    - How to evaluate the cosine formula
- Within - document frequencies
  - Each inverted file entry must be augmented by including with each document pointer the number of times the term appears in that document
  - That is, $f_{d,t}$ must be stored in the inverted file entry along with the document number d_t
- Most $f_{d,t}$ values are small and are frequently either 1 or 2
- How should we code them?
- Unary code works well but Gamma code should be chosen if we use a simple code

Calculating the cosine value (cont.)

- Calculating the cosine value:

$$Cos(q, D_d) = \frac{\vec{D}_d \bullet \vec{q}}{W_d W_q}$$

$$= \frac{\sum_{t \in q \cap D_d} w_{td} * w_{tq}}{\sqrt{\sum_{t=1}^{n} w_{tq}^2} * \sqrt{\sum_{t=1}^{n} w_{id}^2}}$$

where $w_{td} = \frac{f_{t,d}}{\max_l f_{l,d}} \times \log \frac{N}{n_t}, \; w_{iq} = \left(0,5 + \frac{0,5 f_{t,q}}{\max_l f_{l,q}} \right) \times \log \frac{N}{n_t},$

n_t the number of documents which contain t, n the total number of terms in the vocabulary and N the total number of documents.

- Notes:
- t is in the vocabulary
- $f_{d,t}$ is now included in the corresponding occurrence list entry
- W_q is a constant for each query and will be disregarded
- W_d and $\max_l f_{l,d}$ must be computed and stored
- We need a set of accumulators to accumulate the document cosine values (since we will process query terms one by one and each query term will cause us to process the documents in the occurrence list of the term)

Calculating the cosine value (cont.)

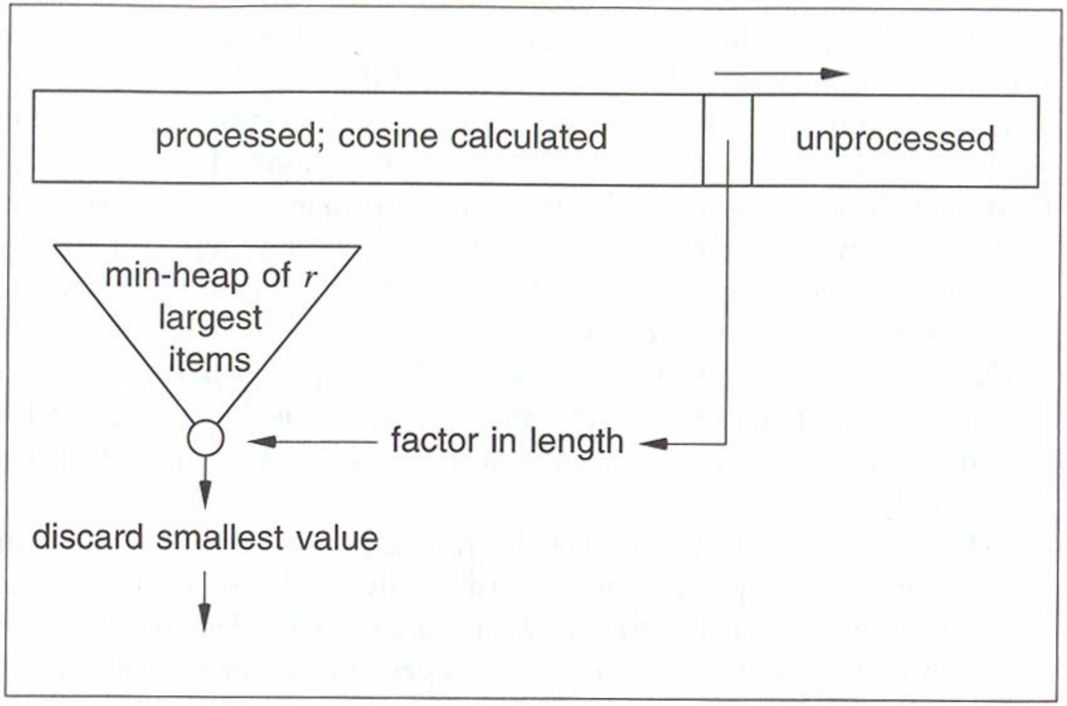
To retrieve r documents using the cosine measure,

```
1. Set $A \leftarrow \{ \}$. $A$ is the set of accumulators.
2. For each query term t in Q,
     (a) Stem t.
     (b) Search the vocabulary.
     (c) Record $n_t$ and the address of $I_t$, the inverted file entry for t.
     (d) Set $W_t \leftarrow \log(N / n_t)$.
(e) Read the inverted file entry $I_t$.
(f) For each $(d, f_{d,t})$ pair in $It$,
     i. If $A_d$ not in $A$ then
          Set $A_d \leftarrow 0$
          Set $A \leftarrow A + \{A_d\}$
     ii. Set $A_d \leftarrow A_d + f_{d,t} * (0.5 + [0.5 * f_{t,q} / \max_l f_{l,q}]) * W_t * W_t$.
3. For each $A_d$ in $A$,
   Set $A_d \leftarrow A_d/(W_d * \max_l f_{l,d})$.
   $A_d$ is now proportional to the value $cosine(q, D_d)$.
4. For $1 \leq i \leq r$,
     (a) Select $d$ such that $A_d = \max\{A\}$.
     (b) Look up the address of document $d$.
     (c) Retrieve document $d$ and present it to the user.
     (d) Set $A \leftarrow A - \{A_d\}$
```

Calculating the cosine value (cont.)

- We only present the top r << N documents, so we should not pay the price of a full sort of the cosine values:
  - Conventional sorting algorithms require at least N log N comparisons to sort N records, and for N ~1000000 this corresponds to 20 million operations, or several seconds on typical computers.
  - How to reduce the sorting time?
    - Sort only the accumulators that are nonzero. In many cases, the set A contain accumulators for only a small proportion of the documents.
    - Use of a min-heap structure

Calculating the cosine value (cont.)



Selection using a min-heap of r items:

O(N log r) steps are required

To select the top *r* cosine values:

```
1. Set $H \leftarrow \{\}$. $H$ is the min-heap.
2. For $1 \le d \le r$,
    (a) Record address of document $d$.
    (b) Set $H \leftarrow H + \{A_d\}$
3. Build $H$ into a heap.
4. For $r + 1 \le d \le N$,
    (a) If $Ad > min\{H\}$ then
        i.   Set $H \leftarrow H - min\{H\} + \{A_d\}$
        ii.  Sift $H$
        iii. Record address of document $d$.
    $H$ now contains the top $r$ exact cosine values.
5. For $1 \le i \le r$,
    (a) Select $d$ such that $A_d = \max\{H\}$.
    (b) Retrieve document $d$ and present it to the user.
    (c) Set $H \leftarrow H - \{A_d\}$
```

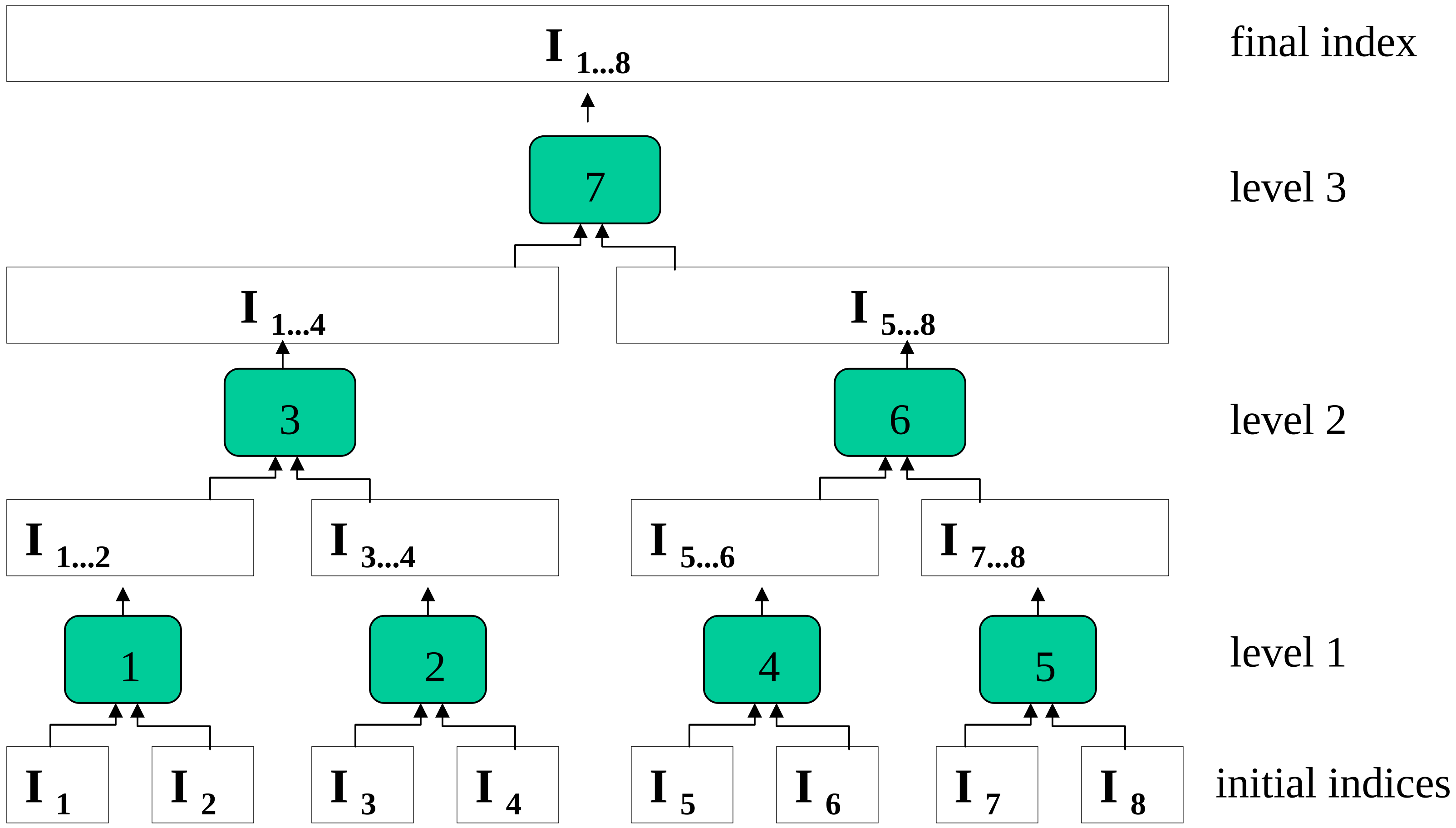
Inverted Files - construction

- Each word of the text is read and searched in the vocabulary
- If it is not found, it is added to the vocabulary with a empty list of occurrences and the new position is added to the end of its list of occurrences
- Once the text is exhausted, the vocabulary is written to disk with the list of occurrences. Two files are created:
  - in the first file, the list of occurrences are stored contiguously (posting file)
  - in the second file, the vocabulary is stored in lexicographical order and, for each word, a pointer to its list in the first file is also included. This allows the vocabulary to be kept in memory at search time
- The overall process is *O(n)* worst-case time
- For large texts, building the index in main memory is not feasible (wouldn’t fit, and swapping would be unbearable)
- Building it entirely in disk is not a good idea either (would take a long time)
- An option is to use the previous algorithm until the main memory is exhausted. When no more memory is available, the partial index $\mathbf{I}_i$ obtained up to now is written to disk and erased the main memory before continuing with the rest of the text
- Once the text is exhausted, a number of partial indices $\mathbf{I}_i$ exist on disk
- The partial indices are merged to obtain the final index

Inverted Files - construction

- The procedure works as follows:
  - Build and save partial indices I_1 , I_2 , …, I_n
  - Merge I_j and I_{j+1} into a single partial index $I_{j,j+1}$
    - Merging indices mean that their sorted vocabularies are merged, and if a term appears in both indices then the respective lists should be merged (keeping the document order)
  - Then indices $I_{j,j+1}$ and $I_{j+2,j+3}$ are merged into partial index $I_{j,j+3}$, and so on and so forth until a single index is obtained
  - Several partial indices can be merged together at once. In practice, it is a good idea to merge even 20 partial indices at once.

Example



$I_{1...8}$ final index

7 level 3

$I_{1...4}$ $I_{5...8}$

3 6 level 2

$I_{1...2}$ $I_{3...4}$ $I_{5...6}$ $I_{7...8}$

1 2 4 5 level 1

I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 initial indices

Inverted Files - construction

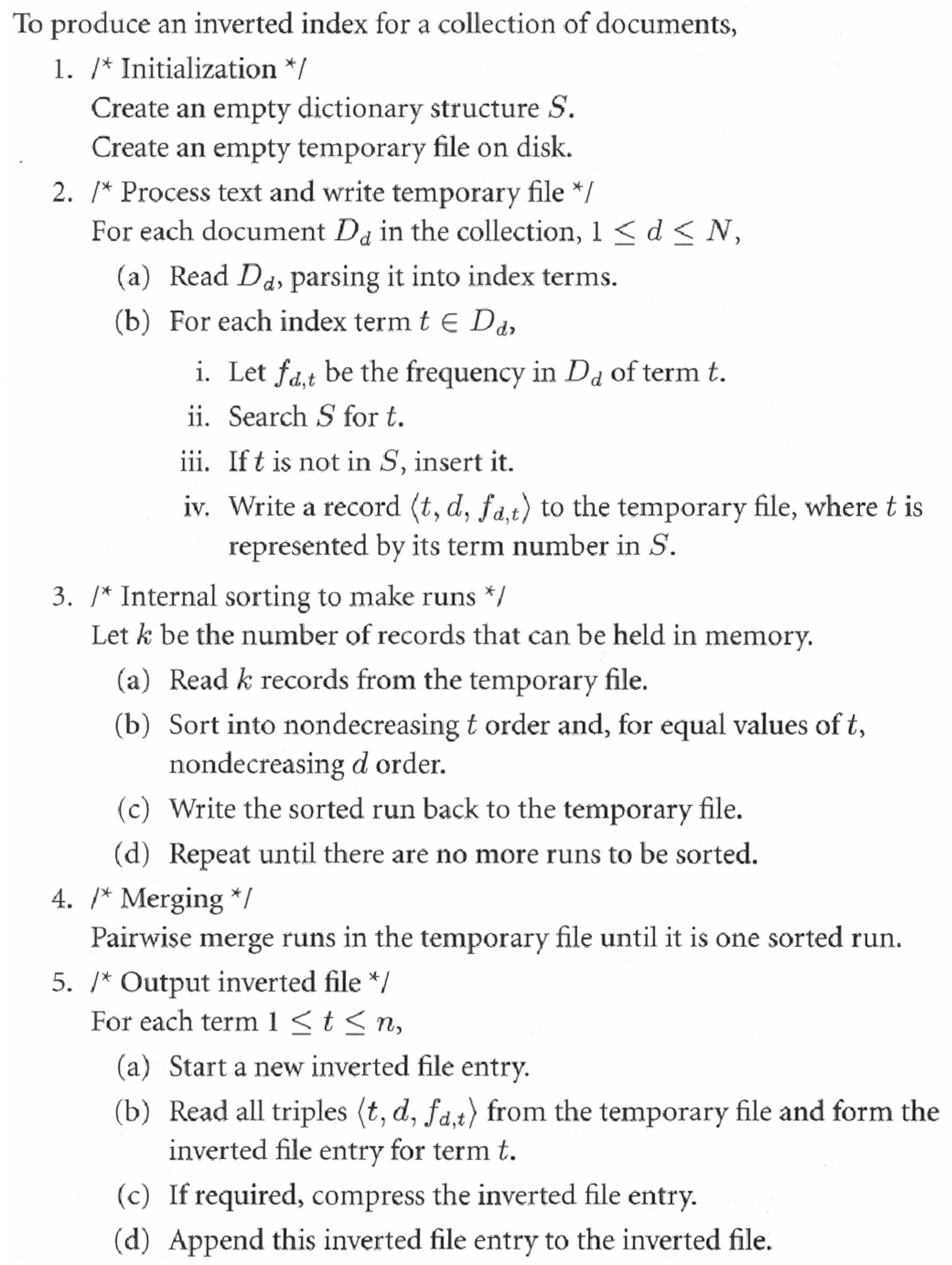
- This procedure takes O(n log (n/M)) time plus O(n) to build the partial indices – where n is the text size in characters and M is the amount of main memory available
- Adding a new document is a matter of merging its (partial) index (indices) to the index already built
- Deletion can be done in O(n) time – scanning over all lists of terms occurring in the deleted document

Sort-based construction

Concept

- Use of disk is inescapable for the size of collection
- Sequential access is the only efficient processing mode for
- large disk files (transfer rates high, random seeks low)
- Algorithm
  - Parse text into triples <t,d,fd,t> and write temporary file
  - Mergesort the temporary file into non-descending t,d order
  - Sorting the temporary file, gives the required inverted file order:
    - records are ordered by increasing t and, within equal values of t, by increasing d.

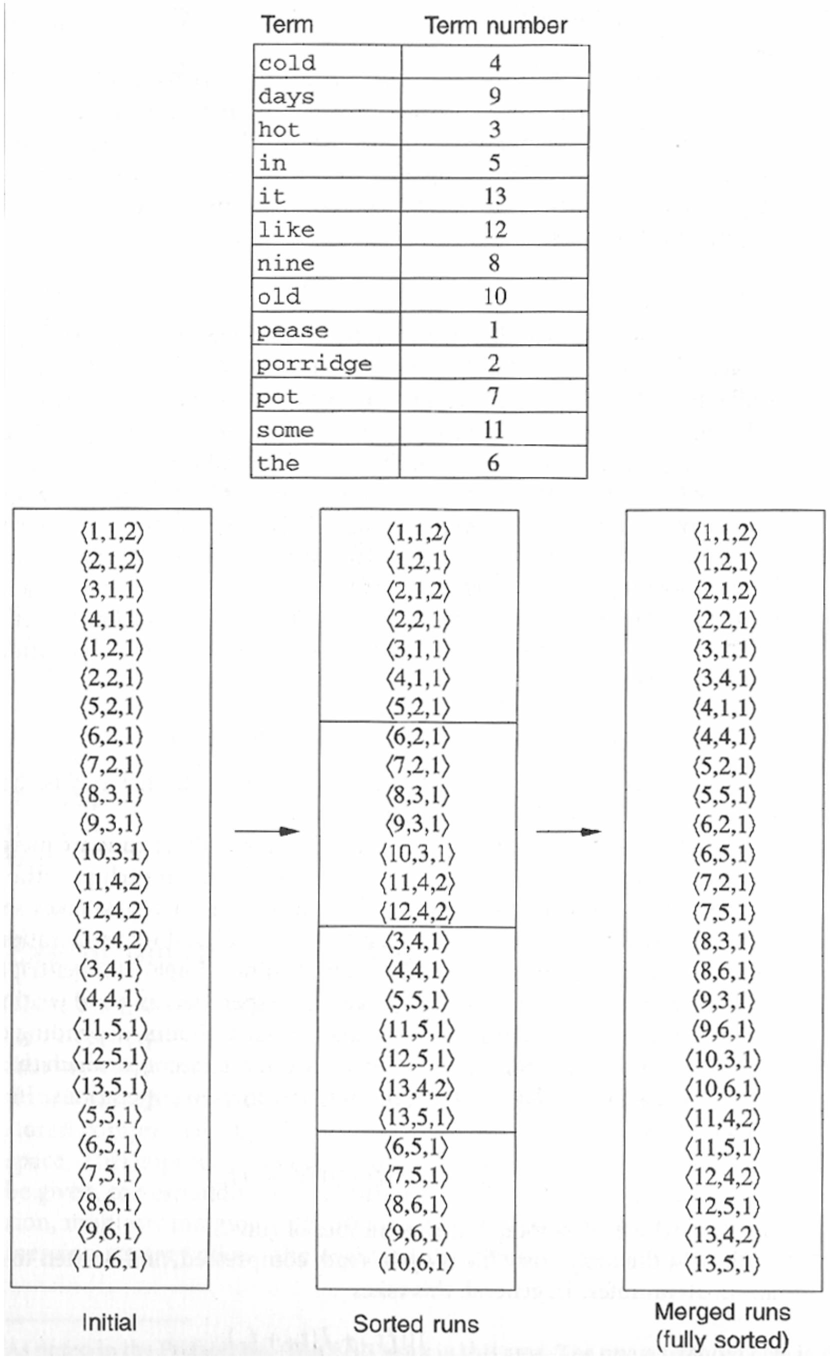
Sort-based construction



To produce an inverted index for a collection of documents,

```
1. /* Initialization */
   Create an empty dictionary structure $S$.
   Create an empty temporary file on disk.
2. /* Process text and write temporary file */
   For each document $D_d$ in the collection, $1 \leq d \leq N$,
   (a) Read $D_d$, parsing it into index terms.
   (b) For each index term $t \in D_d$,
       i.   Let $f_{d,t}$ be the frequency in $D_d$ of term $t$.
       ii.  Search $S$ for $t$.
       iii. If $t$ is not in $S$, insert it.
       iv.  Write a record $\langle t, d, f_{d,t} \rangle$ to the temporary file, where $t$ is
            represented by its term number in $S$.
3. /* Internal sorting to make runs */
   Let $k$ be the number of records that can be held in memory.
   (a) Read $k$ records from the temporary file.
   (b) Sort into nondecreasing $t$ order and, for equal values of $t$,
       nondecreasing $d$ order.
   (c) Write the sorted run back to the temporary file.
   (d) Repeat until there are no more runs to be sorted.
4. /* Merging */
   Pairwise merge runs in the temporary file until it is one sorted run.
5. /* Output inverted file */
   For each term $1 \leq t \leq n$,
   (a) Start a new inverted file entry.
   (b) Read all triples $\langle t, d, f_{d,t} \rangle$ from the temporary file and form the
       inverted file entry for term $t$.
   (c) If required, compress the inverted file entry.
   (d) Append this inverted file entry to the inverted file.
```

Sort-based construction



Term	Term number
cold	4
days	9
hot	3
in	5
it	13
like	12
nine	8
old	10
pease	1
porridge	2
pot	7
some	11
the	6

Initial	Sorted runs	Merged runs (fully sorted)
⟨1,1,2⟩	⟨1,1,2⟩	⟨1,1,2⟩
⟨2,1,2⟩	⟨1,2,1⟩	⟨1,2,1⟩
⟨3,1,1⟩	⟨2,1,2⟩	⟨2,1,2⟩
⟨4,1,1⟩	⟨2,2,1⟩	⟨2,2,1⟩
⟨1,2,1⟩	⟨3,1,1⟩	⟨3,1,1⟩
⟨2,2,1⟩	⟨4,1,1⟩	⟨3,4,1⟩
⟨5,2,1⟩	⟨5,2,1⟩	⟨4,1,1⟩
⟨6,2,1⟩	⟨6,2,1⟩	⟨4,4,1⟩
⟨7,2,1⟩	⟨7,2,1⟩	⟨5,2,1⟩
⟨8,3,1⟩	⟨8,3,1⟩	⟨5,5,1⟩
⟨9,3,1⟩	⟨9,3,1⟩	⟨6,2,1⟩
⟨10,3,1⟩	⟨10,3,1⟩	⟨6,5,1⟩
⟨11,4,2⟩	⟨11,4,2⟩	⟨7,2,1⟩
⟨12,4,2⟩	⟨12,4,2⟩	⟨7,5,1⟩
⟨13,4,2⟩	⟨3,4,1⟩	⟨8,3,1⟩
⟨3,4,1⟩	⟨4,4,1⟩	⟨8,6,1⟩
⟨4,4,1⟩	⟨5,5,1⟩	⟨9,3,1⟩
⟨11,5,1⟩	⟨11,5,1⟩	⟨9,6,1⟩
⟨12,5,1⟩	⟨12,5,1⟩	⟨10,3,1⟩
⟨13,5,1⟩	⟨13,4,2⟩	⟨10,6,1⟩
⟨5,5,1⟩	⟨13,5,1⟩	⟨11,4,2⟩
⟨6,5,1⟩	⟨6,5,1⟩	⟨11,5,1⟩
⟨7,5,1⟩	⟨7,5,1⟩	⟨12,4,2⟩
⟨8,6,1⟩	⟨8,6,1⟩	⟨12,5,1⟩
⟨9,6,1⟩	⟨9,6,1⟩	⟨13,4,2⟩
⟨10,6,1⟩	⟨10,6,1⟩	⟨13,5,1⟩

Inverted File Compression

- An inverted file is typically composed of
  1. vocabulary: a vector containing all the distinct terms in the text collection, stored in main memory if possible
  2. inverted list (IL) stored in disc: for each term in the vocabulary, a list of pointers to all occurrences of that term stored in ascending (or descending) sequence

- Important parameters:

	(in TREC, 99)
• N - number of documents in db	741.856
• n - number of (distinct) terms	535.346
• F - number of term occurrences	333.338.738
• f - number of inverted list entries	134.994.414
	Total size: 2G

- Uncompressed inverted files can require 50-100% of the space of the text they index
- Log N bits needed for each document id: total space is f log N bits for document-level inverted lists
- Inverted list compression helps to reduce size of index, cost of I/O

Inverted File Compression

- The IL for a term t contains f_t entries:

  $(d_1, d_2, \ldots, d_{f_t})$
- where $d_1, d_2, \ldots, d_{f_t}$ are the number of documents in which term t appears and $d_k < d_{k+1}$.
- The list of document numbers is in ascending order and all processing sequential can start from the beginning of the list:
  - The list can be stored as an initial position followed by a list of d-gaps, the differences $d_{k+1} - d_k$. For instance the list (3,5,20,21,23,76,77,78) can be stored as (3,2,15,1,2,53,1,1)
- If g1=d1, g2=d2-d1, …, we know that

  $$\sum_{i=1}^{f_t} g_t = d_{f_t} \leq N$$
- For long lists, most d-gaps are small.
- These facts can be used for compression
- Models:
  - Describe the probability density of gap sizes
  - Each gap has an associated probability, defined by its code length: $l_i \sim \log(1/p_i)$ (Shannon theorem)
  - Goal: higher probability gaps get coded in fewer bits, etc.
  - Global - every inverted file entry is compressed with the same model
  - Local – the inverted list of each term uses its own model (usually based upon a parameter such as the frequency of the term)
  - Local models outperform global models but are more complex to implement

Global, non-parametric methods

- Flat binary
  - $\lceil \log N \rceil$ bits per pointer (fixed-length representation)
  - Implicit probability model: each gap size is equally likely(uniformly random in 1 to N, p(*g*)=1/N)
- Unary coding:

represent each g>0 by g-1 digits 1, then 0

1 -> 0, 2 -> 10, 3 -> 110, 4-> 1110, …

  - Code length for g: g

➔ $$\sum_{i=1}^{f_t} g_t = d_{f_t} \leq N$$

  - Worst case for sum: N (hence for all IL's: nN, extremely large quantity)
  - P(g): *Pr[g] = 2^{-g} for gaps of length g*
  - Binary Exponential decay; favors small gaps, large gaps are coded in too many bits
  - if does not hold in practice: compression penalty

Global, non-parametric methods

- There are many codes whose implied probability distributions lie somewhere between the uniform distribution assumed by the binary code and the binary exponential decay implied by the unary code

Gamma (γ) code :

  - a number g is represented by
  - Prefix code: unary code for $1+\lfloor \log_2 g \rfloor$. *Specifies how many bits are required to code g*
  - Suffix code :binary code, with $\lfloor \log_2 g \rfloor$ digits, for $g-2^{\lfloor \log_2 g \rfloor}$
- *Example:*
  - *Encoding g = 10*
  - *Unary code:* $1+\lfloor \log_2 10 \rfloor$ *= 1+3 = 4 = 1110*
  - *Binary code:* $10-2^{\lfloor \log_2 10 \rfloor}$ *=10-8 = 2 = 010*
  - *Gamma code for x=10 is 1110010*
- *Decoding:* $g = 2^{c_u -1} + c_b$
  - *Extract unary code (c_u); Extract binary code (c_b)*
  - *g is represented in ≈ 1+2log$_2$ g bits (one log$_2$ g for power of 2 and one log$_2$ g for the remainder)*
  - *Implicit probability model: Pr(g) ≈ $2^{-(1+2\log_2 g)}$ = 1/2g^2 (remember g^2 = $2^{2\log_2 g}$) (inverse square)*

- Delta (δ) code :

prefix: represent $1+\lfloor \log g \rfloor$ in gamma code

suffix: represent $g-2^{\lfloor \log g \rfloor}$ in binary (as in gamma)

Examples:

7: $1+\lfloor \log 7 \rfloor = 3, \quad 1+\lfloor \log 3 \rfloor = 2,$ prefix is 101

suffix is 11, code is 10111

Code lenght for g: $1+2\lfloor \log(1+\lfloor \log g \rfloor) \rfloor + \lfloor \log g \rfloor$

$$\simeq 1+2\log\log g+\log g$$

Implicit Probability model: $p(g)=2^{-(1+2\lfloor 1+\lfloor \log g \rfloor \rfloor+\lfloor \log g \rfloor)} \approx \dfrac{1}{2g(\log g)^2}$

Golomb code

- Global, parametric code
- Semi-static, uses text collection statistics
- Very effective when the gap probability distribution is geometric:
  - The likelihood of a gap being of size g is the probability of having g-1 non-occurrences (within consecutive documents) of that particular term followed by one occurrence.
  - If a term occurs within a document with a probability p, the probability of a gap of size g is: $P(g) = (1-p)^{g-1}p$

    which is the geometric distribution
  - The probability p that any randomly selected document contains any randomly chosen term can be estimated as p=f / Nn.
- Golomb code:

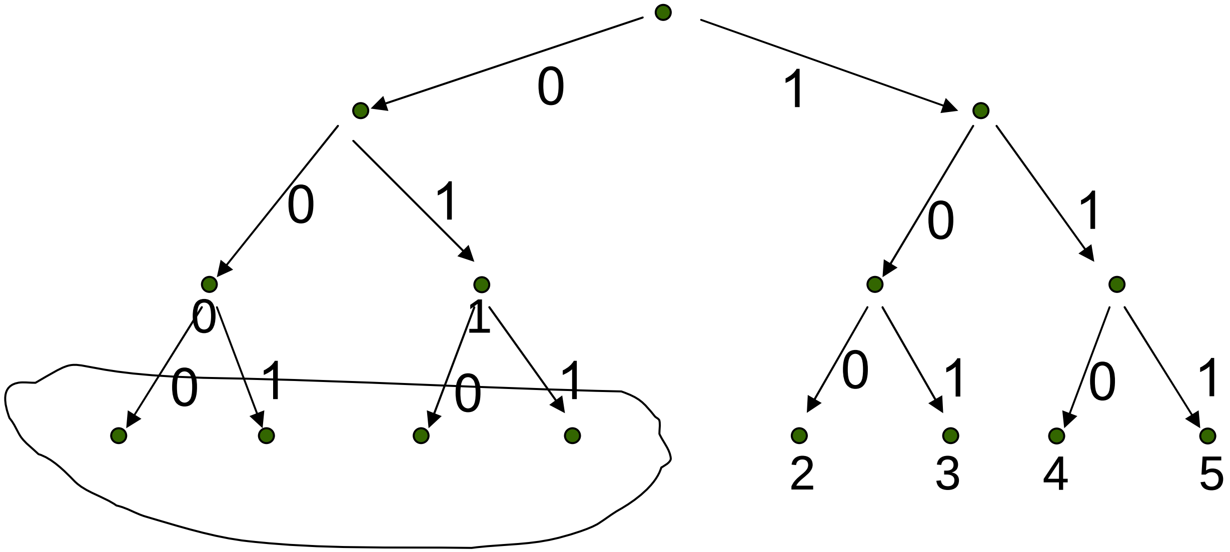
1. Select a basis b (based on text collection statistics)
2. *g*>0 → we represent *g*-1

- Prefix: let $q = \lfloor (g-1)/b \rfloor$ (integer division), represent in unary, *q*+1
- Suffix: the remainder is (*g*-1)-*qb* (in [0..b-1]) represented by a binary tree code
  - some leaves at distance $\lfloor \log b \rfloor$
  - the others at distance $\lceil \log b \rceil$

- The binary tree code:

$$\text{let} \quad k = \lceil \log\ b \rceil, \;\; j = 2^k - b$$

- cut 2j leaves from the full binary tree of depth k
- assign leaves, in order, to the values in [0..b-1]
- Example: b=6



Summary of Golomb code

Prefix: unary code of $1+\lfloor (g-1)/b \rfloor$

Suffix: code of length between $\lfloor \log b \rfloor$ and $\lceil \log b \rceil$

length $\approx 1+\lfloor (g-1)/b \rfloor + \lceil \log b \rceil$

Implicit Probability mode; p(g) $\approx \frac{1}{2b} \cdot \frac{1}{2^{(g-1)/b}} \approx \frac{1}{2b} \cdot \frac{1}{(2^{1/b})^{g-1}}$

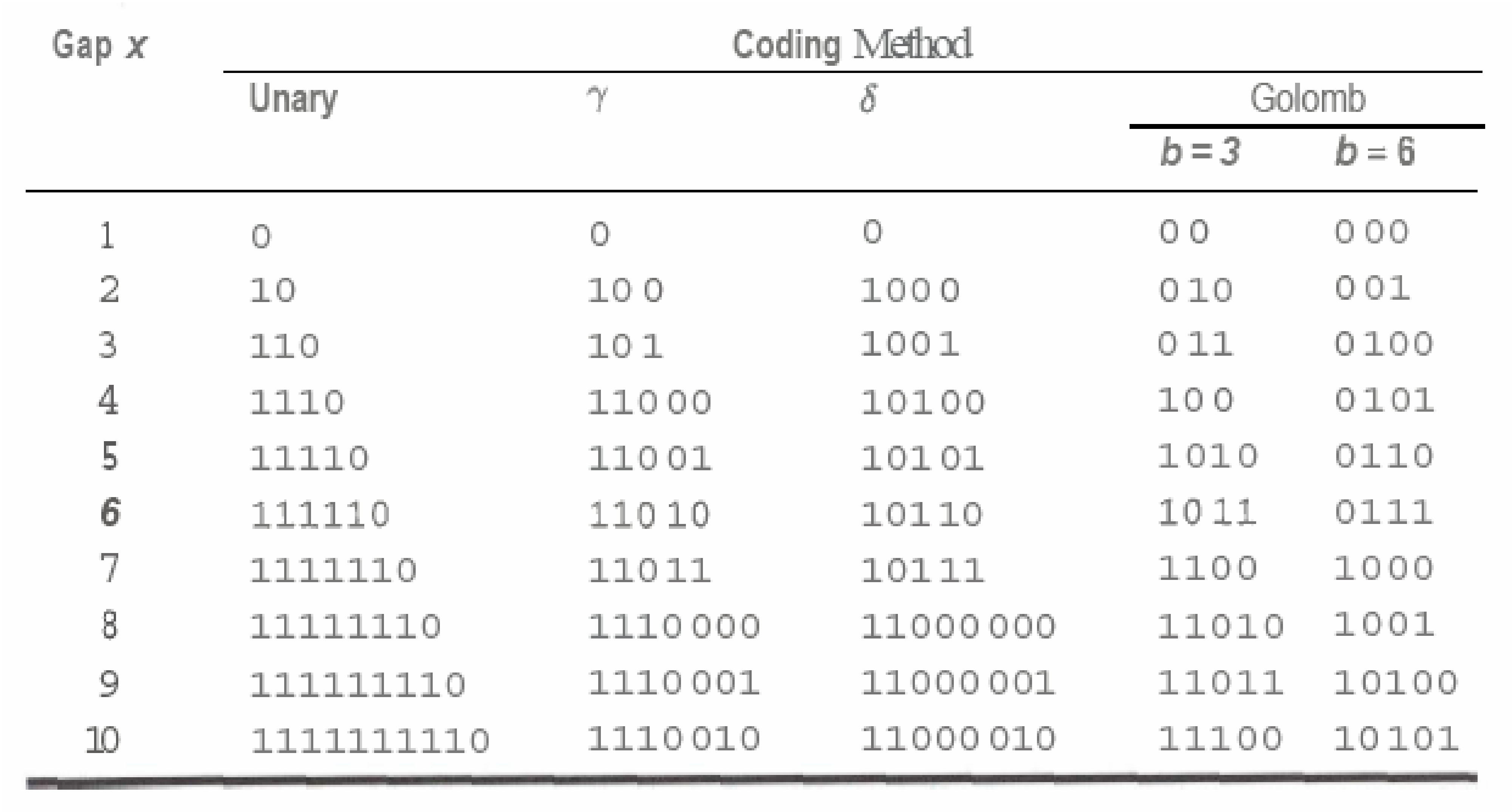
- Exponential decay like unary, slower rate, affected by *b*
- Q: how is *b* chosen?
- The Golomb code matches the entropy of geometric distribution for:

$$b = \left\lceil \frac{\log_2(2-p)}{-\log_2(1-p)} \right\rceil$$

- Assuming that p=f/(Nn)<<1, a useful simplification is

$$b \approx \frac{\log_e 2}{p} \approx 0.69 \cdot \frac{N \cdot n}{f}$$

Example codes for integer



Gap *x*	Coding Method				
	Unary	γ	δ	Golomb	
				b = 3	*b* = 6
1	0	0	0	00	000
2	10	100	1000	010	001
3	110	101	1001	011	0100
4	1110	11000	10100	100	0101
5	11110	11001	10101	1010	0110
6	111110	11010	10110	1011	0111
7	1111110	11011	10111	1100	1000
8	11111110	1110000	11000000	11010	1001
9	111111110	1110001	11000001	11011	10100
10	1111111110	1110010	11000010	11100	10101