Modern information retrieval

Chapter 8 – Indexing and Searching

Indexing and Searching TOC

- Indexing techniques:
  - Inverted files
  - Suffix arrays
  - Signature files
- Technique used to search each type of index
- Other searching techniques

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 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)

Motivation

- Traditional indices, e.g., B-trees, are not well suited for IR

- Main approaches:
  - Inverted files (or lists)
  - Suffix arrays
  - Signature files

Inverted Files

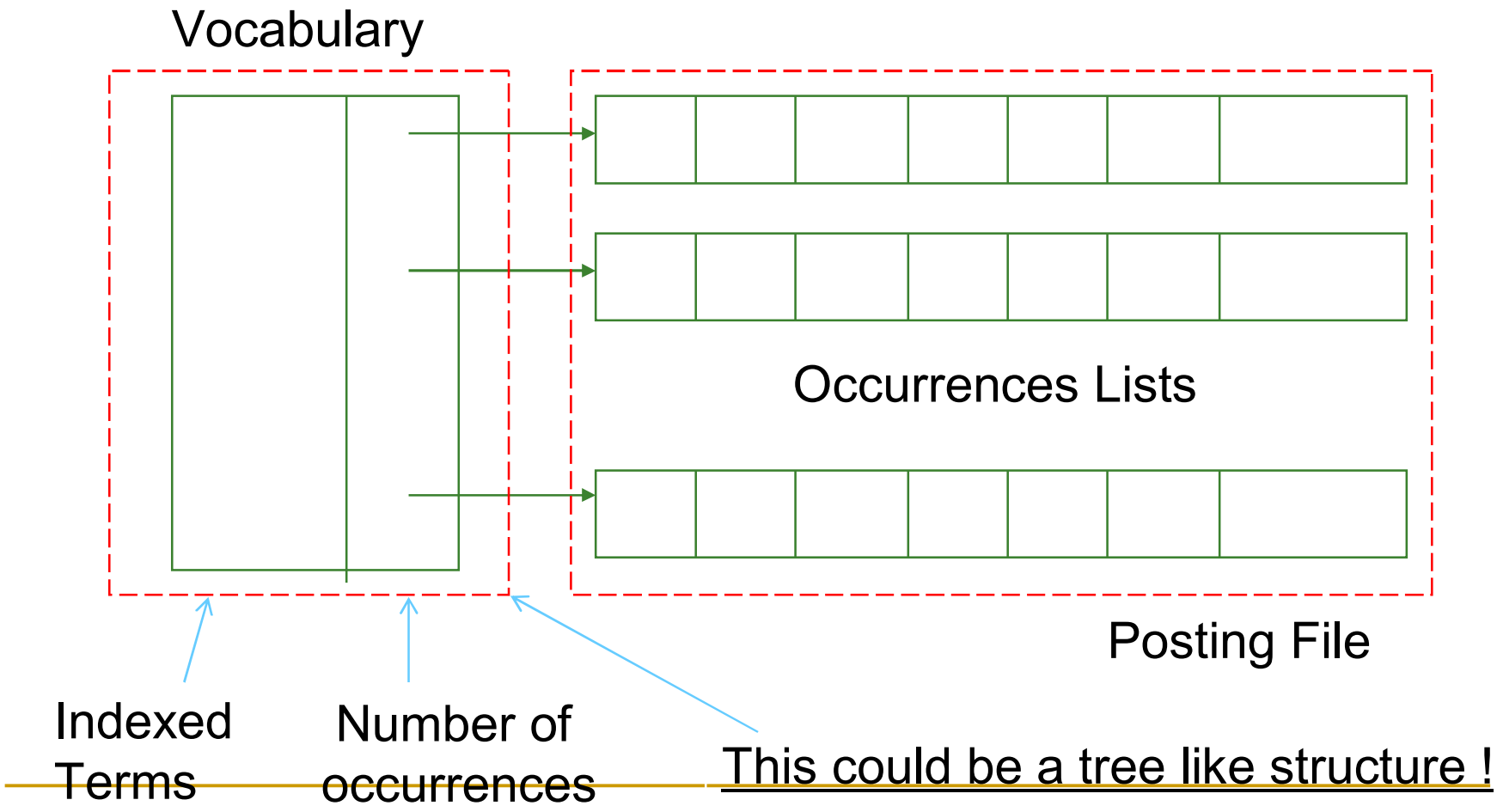
- There are two main elements:
  - vocabulary – set of unique terms
  - Occurrences – where those terms appear
- The occurrences can be recorded as terms or byte offsets
- Using term offset is good to retrieve concepts such as proximity, whereas byte offsets allow direct access

Vocabulary	Occurrences (byte offset)
…	…

Inverted Files

- 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
- That may lead to a quite considerable index overhead

Inverted Files - layout



Example

- Text:

1 6 12 16 18 25 29 36 40 45 54 58 66 70

That house has a garden. The garden has many flowers. The flowers are beautiful

- Inverted file

Vocabulary	Occurrences
beautiful	70
flowers	45, 58
garden	18, 29
house	6

Inverted Files

- Coarser addressing may be used

Terms	Occurrences (**block** offset)
…	…

- All occurrences within a block (perhaps a whole document) are identified by the same block offset
- Much smaller overhead
- Some searches will be less efficient, e.g., proximity searches. Linear scan may be needed, though hardly feasible (specially on-line)

Space Requirements

- The space required for the vocabulary is rather small. According to *Heaps'* law the vocabulary grows as $O(n^{\beta})$, where β is a constant between 0.4 and 0.6 in practice
- On the other hand, the occurrences demand much more space. Since each word appearing in the text is referenced once in that structure, the extra space is $O(n)$
- To reduce space requirements, a technique called *block addressing* is used

Block Addressing

- The text is divided in blocks
- The occurrences point to the blocks where the word appears
- Advantages:
  - the number of pointers is smaller than positions
  - all the occurrences of a word inside a single block are collapsed to one reference
- Disadvantages:
  - online search over the qualifying blocks if exact positions are required

Example

- Text:

That house has a	garden. The garden has	many flowers. The flowers	are beautiful
Block 1	**Block 2**	**Block 3**	**Block 4**

- Inverted file:

Vocabulary	Occurrences
beautiful	4
flowers	3
garden	2
house	1

Inverted Files

Index	Small collection (1Mb)		Medium collection (200Mb)		Large collection (2Gb)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	19%	26%	18%	32%	26%	47%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%

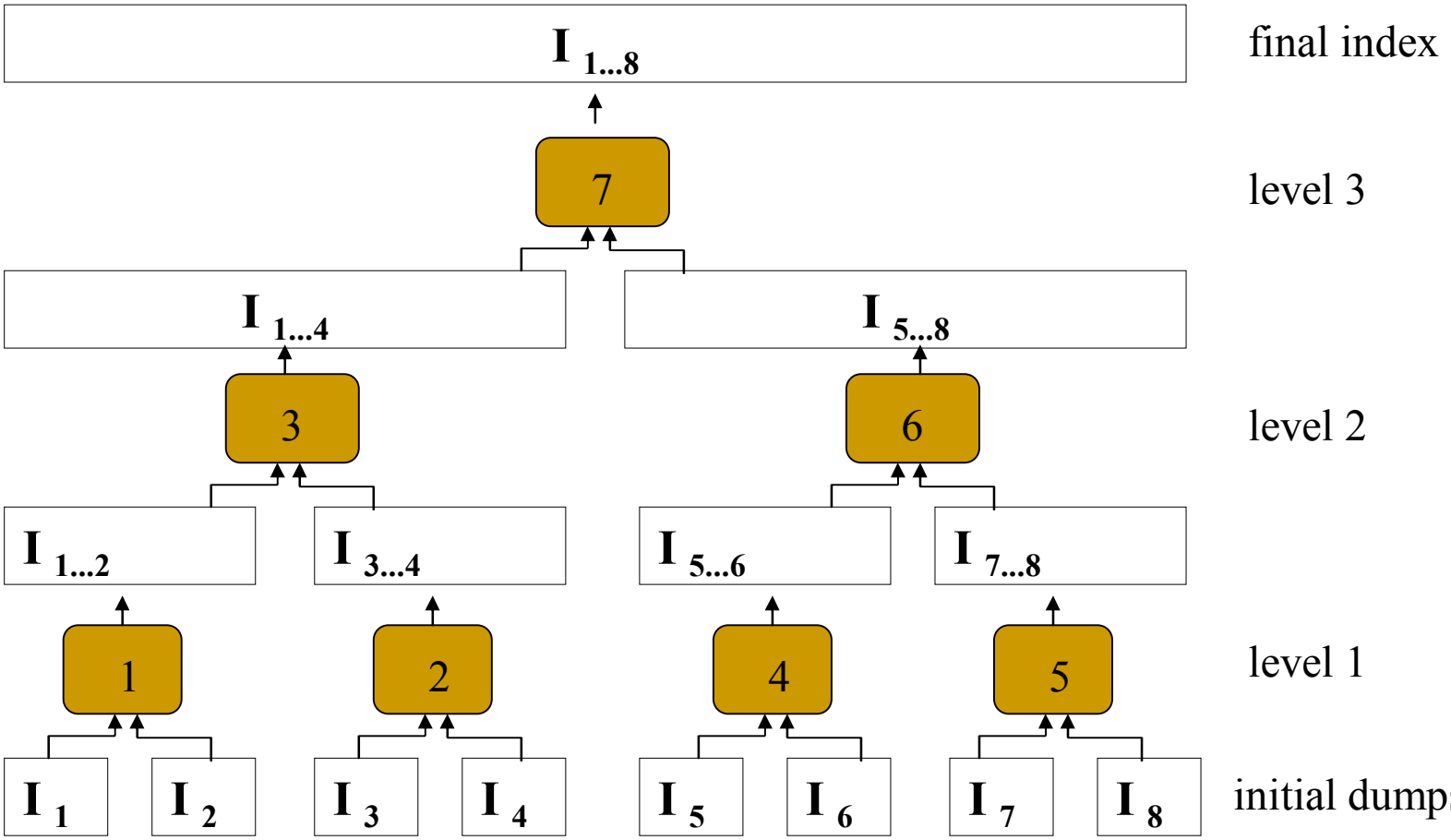
Inverted Files - construction

- 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)
- One idea is to build several partial indices in main memory, one at a time, saving them to disk and then merging all of them to obtain a single index

Inverted Files - construction

- The procedure works as follows:
  - Build and save partial indices I1, I2, …, In
  - Merge Ij and Ij+1 into a single partial index Ij,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 Ij,j+1 and Ij+2,j+3 are merged into partial index Ij,j+3, and so on and so forth until a single index is obtained
  - Several partial indices can be merged together 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 dumps

Inverted Files - construction

- This procedure takes O(n log (n/M)) time plus O(n) to build the partial indices – where n is the document size 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

Searching

- The search algorithm on an inverted index follows three steps:
  - Vocabulary search: the words present in the query are searched in the vocabulary
  - Retrieval occurrences: the lists of the occurrences of all words found are retrieved
  - Manipulation of occurrences: the occurrences are processed to solve the query

Searching

- Searching task on an inverted file always starts in the vocabulary (It is better to store the vocabulary in a separate file)
- The structures most used to store the vocabulary are *hashing, tries* or *B-trees*
- An alternative is simply storing the words in lexicographical order (cheaper in space and very competitive with *O(log v)* cost)

Inverted Files - searching

- Searching using an inverted file
  - Vocabulary search
    - The terms used in the query (decoupled in the case of phrase or proximity queries) are searched separately
  - Retrieval of occurrences lists
  - 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
- What if blocking is used ?
  - No positional information is kept, so a linear scan of the document is required
- The traversal and merging of the obtained lists are sensitive operations

Construction

- All the vocabulary is kept in a suitable data structure storing for each word a list of its occurrences
- 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

Compression of Inverted Index

- I/O to read a occurrence 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
  - Occurrence List
  - Term Dictionary
- Half of terms occur only once *(hapax legomena)* so they only have one entry in their Occurrence 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 occurrence list entry
- Document Identifier in each occurrence list entry

Data Compression

- Applied to occurrence 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.

Conclusion

- Inverted file is probably the most adequate indexing technique for database text
- The indices are appropriate when the text collection is large and semi-static
- Otherwise, if the text collection is volatile online searching is the only option
- Some techniques combine online and indexed searching

Suffix Trees/Arrays

- Suffix trees are a generalization of inverted files
- For traditional queries, i.e., those based on simple terms, inverted files are the structure of choice
- This type of index treats the text to be indexed as a finite, but long string.
- Thus each position in the text represent a suffix of that text, and each suffix is uniquely identified by its position
- Such position determine what is and what is not indexed and is called index point

Suffix Trees

- Note that the choice of index points is crucial to the retrieval capabilities
- Consider the document:

  This is a text. A text has many words. Words …
- And the index points/suffixes:

  11: text. A text has many words. Words …

  19: text has many words. Words …

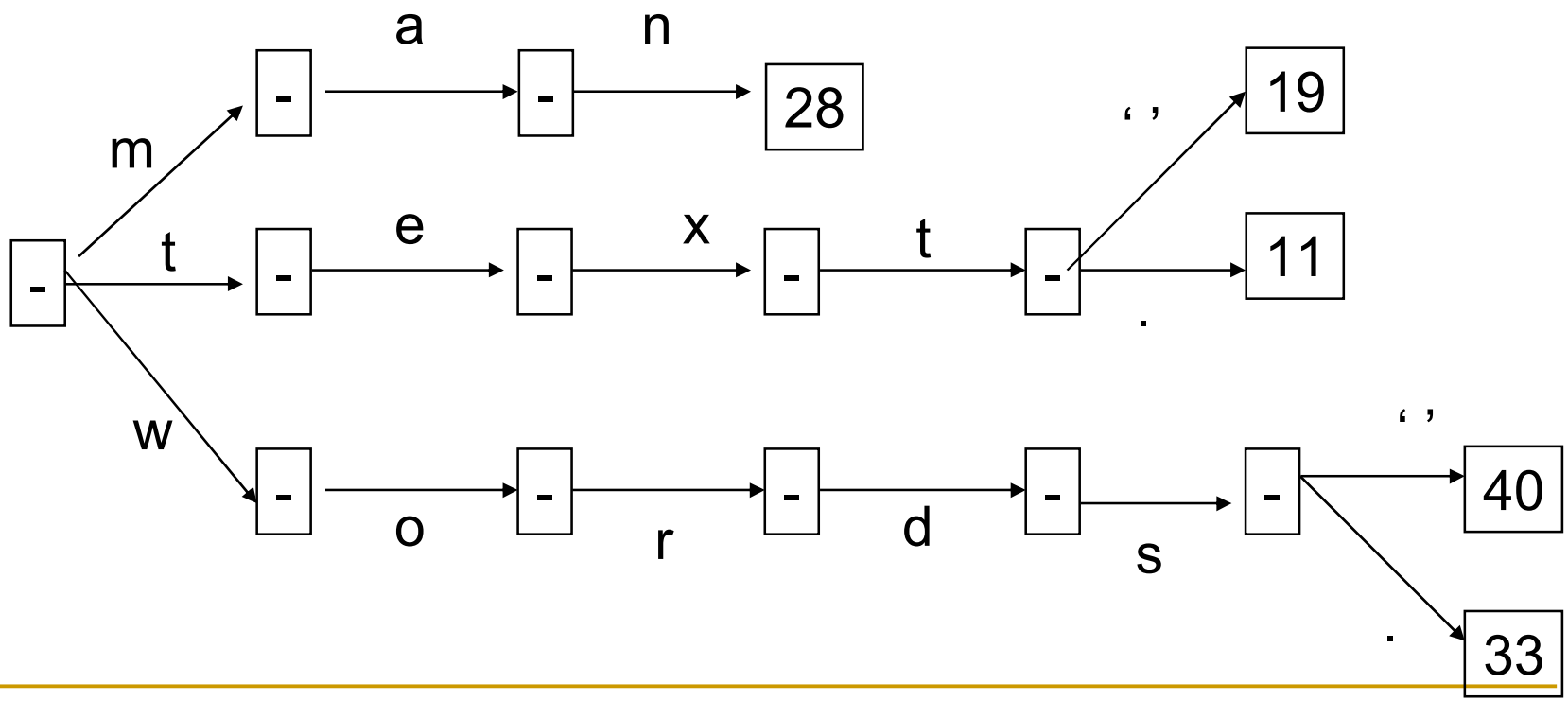
  28: many words. Words …

  33: words. Words …

  40: Words …

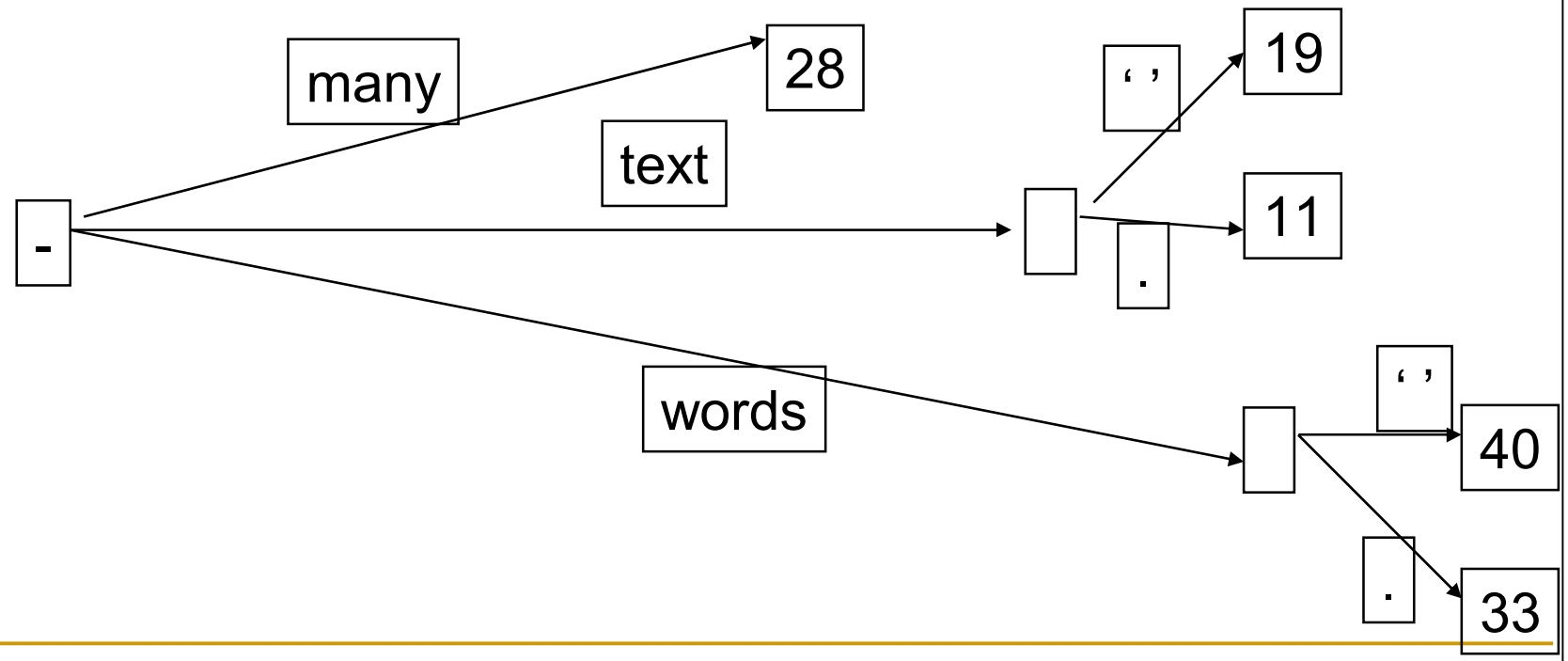
Suffix Trees

- Suffix **Trie**



Suffix Trees

- Suffix **Tree**



Suffix Trees

- Even though the example showed only terms, one can use suffix trees to index and search phrases
- For practical purposes a rather small phrase length (typically much smaller than the text) should be used
- Suffix arrays are a better implementation of suffix trees

Suffix Arrays

- Careful inspection of the leaves in the suffix tree shows that there is a lexicographical order among the index points
- An equivalent suffix array would be:

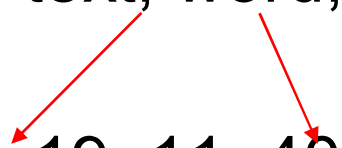
  [28, 19, 11, 40, 33] for

  (many, text_, text., words_, words.}
- This allows the search to be processed as a binary search, but note that one must use the array and the text at query time
- The number of I/Os may become a problem

Suffix Arrays

- One idea is to use a supra-index
- A supra-index will contain a sample of the indexed terms, e.g.,

  […, text, word, …]

  [28, 19, 11, 40, 33]

- This alleviates the binary search, but makes it quite similar to inverted files

Suffix Arrays

- This similarity is true if only single terms are indexed, but suffix trees/arrays aim more than that, they aim mostly at phrase queries
- However, the array (or tree) is unlikely to fit in main memory, whereas for the inverted files the vocabulary could fit in main memory
- This requires careful implementation to reduce I/Os
- Good implementations show reduction of over 50% compared to naive ones

Signature Files

- Unlike the case of inverted files where, most of the time, there is a tree structure underneath, signature files use hash tables
- The main idea is to divide the document into blocks of fixed size and each block has assigned to it a signature (also fixed size), which is used to search the document for the queried pattern
- The block signature is obtained by OR'ing the hashed bitstrings of each term in the block

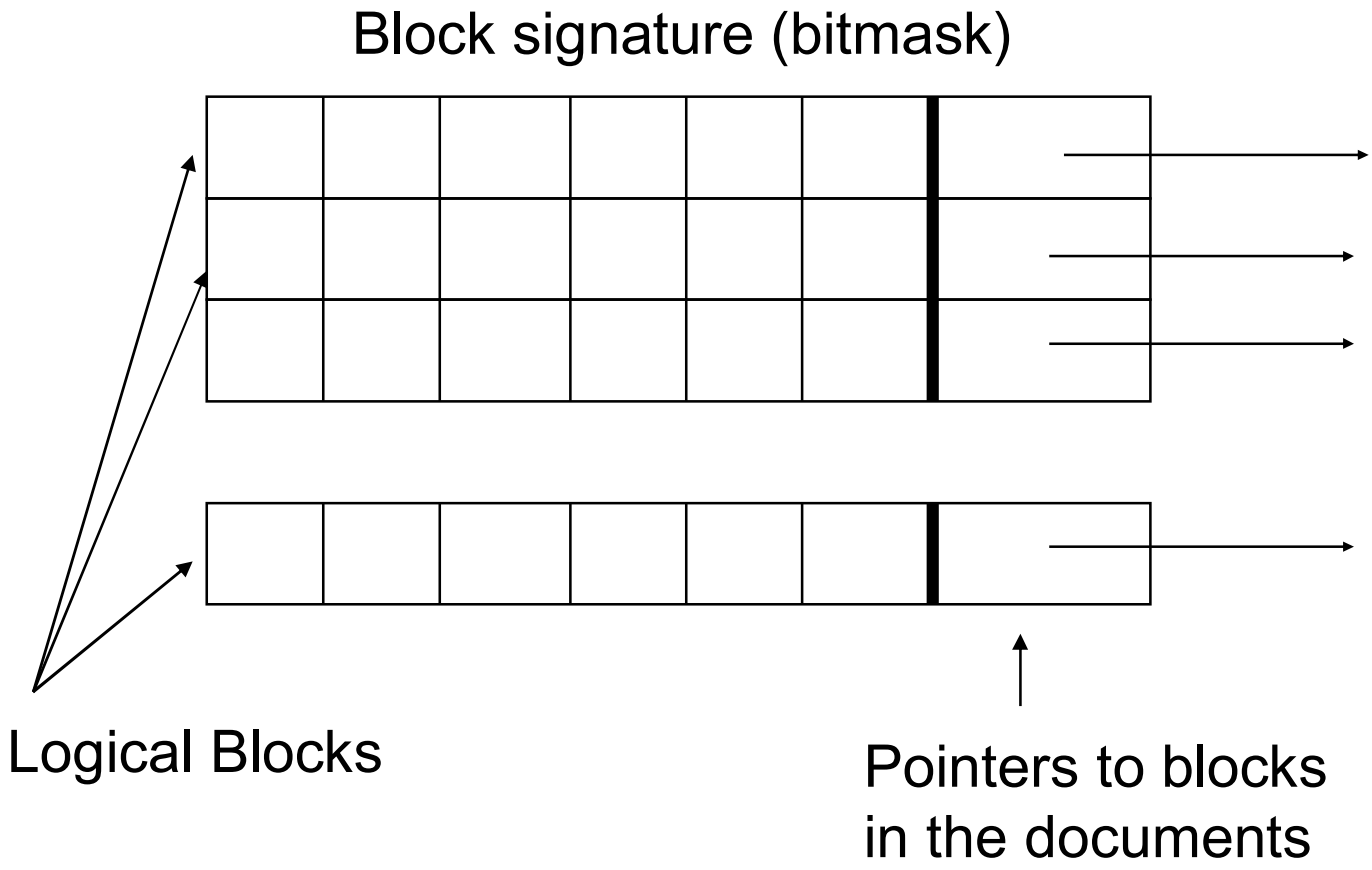
Signature Files

- Consider:
  - H(information) = 010001
  - H(text) = 010010
  - H(data) = 110000
  - H(retrieval) = 100010
  - The block signatures of a document D containing the text "textual retrieval and information retrieval" (after removing stopwords and stemming) for a block size of two terms – would be:
    - B1D = 110010 and
    - B2D = 110011

Signature Files - searching

- To search for a given term we compare whether the term's bitstring could be "inside" the block signatures
- Consider we are searching for "text" in document D
  - H(text) = 010010 and B1D = 110010
  - H(text) bit-wise-AND B1D = 010010 = H(text)
  - Therefore "text" **could** be in B1D (it is in this particular case)
- Consider we are now searching for "data"
  - H(data) bit-wise-AND B1D = 110000 = H(data)
  - H(data) bit-wise-AND B2D = 110000 = H(data)
  - Though "data" is not in either block !
- Signature files may yield false hits …

Signature File - layout



Signature Files – design issues

- How to keep the probability of a false alarms low ? How to predict how good a signature is ?
- Consider:
  - B: the size (number of bits) of each term's bitstring
  - b: number of terms per document block
  - I: the minimum number of bits set (turned on) in B, this depends on the hashing function
- The value of I which minimizes the false hits (yielding probability of false hits equal to 2^{-I}) is
  - $I = B \ln(2)/b$
- B/b indicates the space overhead to pay

Signature Files - alternatives

- There are many strategies which can be used with signature files
- Signature compression
  - Given that the bitstring is likely to have many 0s a simple run-length encoding technique can be used
  - 0001 0010 0000 0001 is encoded as “328” (each digit in the code represents the number of 0s preceding a 1)
  - The main drawback is that comparing bitstrings require decoding at search time

Signature Files - alternatives

- Using the original signature file layout and a bitstring (queried term signature) all logical blocks are checked as to whether that block could contain that bitstring
- This may take a long time if the signature file is long (this depends on the block size)

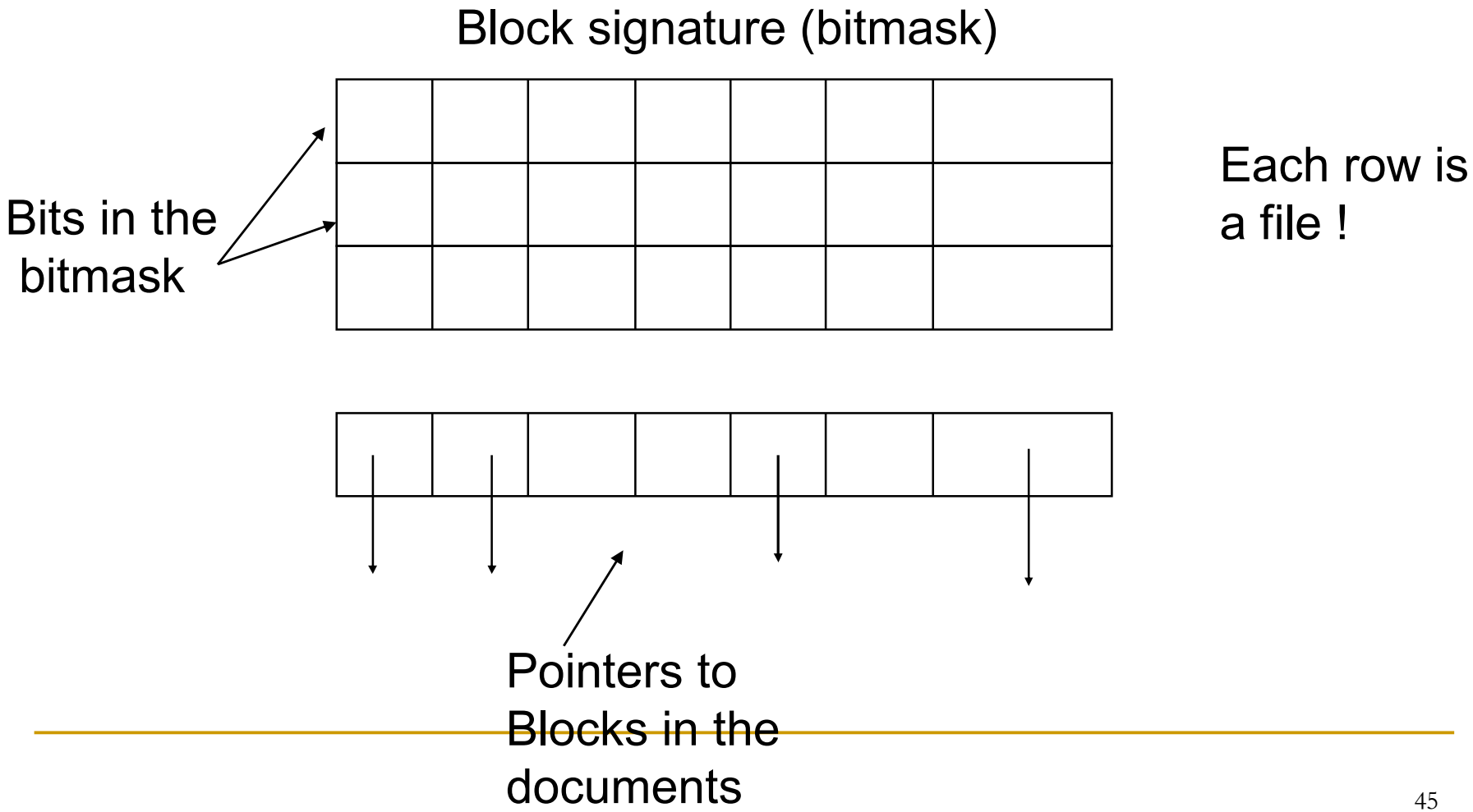
- Could we check only at the bits we are interested ? Yes, we can …

Signature Files - alternatives

- Original signature file layout:
  - Each row has a bitmask corresponding to a block
  - Each column tells whether that bit is set or not
- Bit-sliced signature files layout:
  - Each row correspond to a particular bit in the bitmasks
  - Each column is the bitmask for the blocks

- The signature file is “transposed”

Signature Files - alternatives



Signature Files - alternatives

- Advantages of bit-slicing:
- Given a term bitstring one can read only the lines (files) corresponding to those set bits and if they are set in a given column, then the whole column may be retrieved.
- Empty answers are found very fast
- Insertion is expensive, as all rows must be updated, but in an append-only fashion, this is good for WORM media types

Signature Files - alternatives

- It has been proposed that m = 50% (half of the bits to be 1)
- A typical value of m is 10, that implies 10 accesses for a single term query
- Some authors have suggested to use a smaller m (thus smaller number of accesses)
- The trade-off is that to maintain low probability of false-hits, the signature has to be larger, thus the space required for the signature grows larger as well

Searching - Motivation

- Thus far, we've seen how to pre-process texts (stemming, "stopping" words, compressing, etc)
- We've also seen how to build, process, improve and assess the quality of queries and the returned answer sets
- In the last classes, we saw how to effectively index the texts
- Now, how do we search text ? It goes beyond searching the indices, specially in the cases where the indices are not enough to answer queries (e.g., proximity queries)

Sequential Searching

- If no index is used this is the only option, however sometimes (e.g., when blocking is used) it is necessary even if an index exists
- The problem is “simple”
  - Given a pattern P of length n and a text T of length m (m >> n) find all positions in T where P occurs
- There is much more work on this than we can cover, including many theoretical results. Thus we will cover some well known approaches

Brute Force

- The name says it all …
- Starting from all possible initial positions (i.e., all positions) whether the pattern could start at that position
- It takes O(mn) time in the worst case, but O(n) in the average case – not that bad

- The most important thing is that it suggests the use of a sliding window over the text. The idea is to see whether we can see the pattern through the window

Knuth-Morris-Pratt

- Consider the following text:

  TEXTURE OF A TEXT IS NOT TEXTUAL
- And that the queried term is ‘TEXT ’ (note the blank at the end !)
- The window is slid

  TEXTURE OF A TEXT IS NOT TEXTUAL
- At this point there cannot be a match so the window can be slid to

  TEXTURE OF A TEXT IS NOT TEXTUAL
- Is that correct ?

Knuth-Morris-Pratt

- Only if we are not interested in the occurrence of pattern within terms, otherwise not
- For instance, search for ABRACADABRA within ABRAC**ABRACADABRA**
- The previous idea would do:

  ABRACABRACADABRA and then skip to

  ABRACABRACADABRA missing the pattern
- The correct move would be to

  ABRACABRACADABRA
- How come ?

Knuth-Morris-Pratt

- The processing time of this algorithm is O(n), which is linear, thus better than the Brute Force Algorithm
- However, in the average case the Brute Force algorithm is O(n) as well

- Many, many others algorithms exist …