CS 5800: Searching Text

Ricardo Baeza-Yates

Northeastern University, 2018

Introduction

- Sequential search is fine when the text changes a lot or is not too big (e.g., editor)
- Indexed search is needed when the text is large
- The most used index is the **inverted index** (e.g., search engines)
- Can be built in memory in linear time
- Supports word queries as well as more complex queries and ranking

Sequential Search

Problem Definition

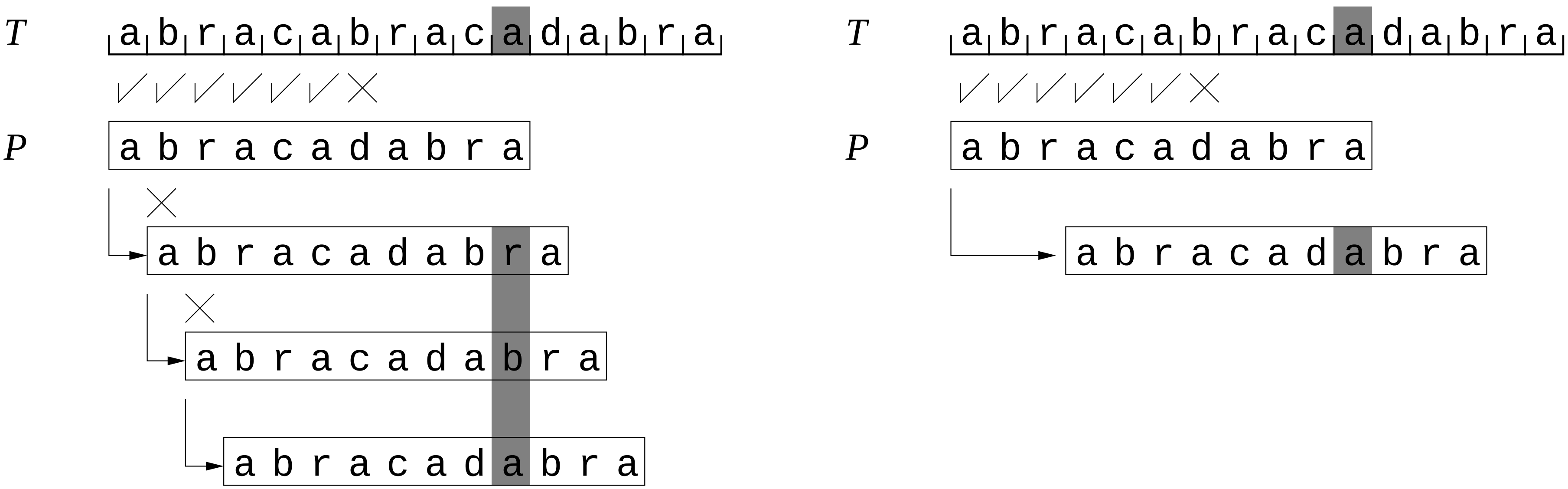
- In general the sequential search problem is:
  - Given a text $T = t_1 t_2 \ldots t_n$ and a pattern denoting a set of strings $\mathcal{P}$, find all the occurrences of the strings of $\mathcal{P}$ in T
- **Exact string matching:** the simplest case, where the pattern denotes just a single string $P = p_1 p_2 \ldots p_m$
- This problem subsumes many of the basic queries, such as word, prefix, suffix, and substring search
- We assume that the strings are sequences of characters drawn from an alphabet Σ of size σ

Simple Strings: Brute Force

- The **brute force** algorithm:
  - Try out all the possible pattern positions in the text and checks them one by one
- More precisely, the algorithm slides a **window** of length m across the text, $t_{i+1}t_{i+2}\ldots t_{i+m}$ for $0 \leq i \leq n-m$
- Each window denotes a potential pattern occurrence that must be verified
- Once verified, the algorithm slides the window to the next position

Simple Strings: Brute Force

- A sample text and pattern searched for using brute force



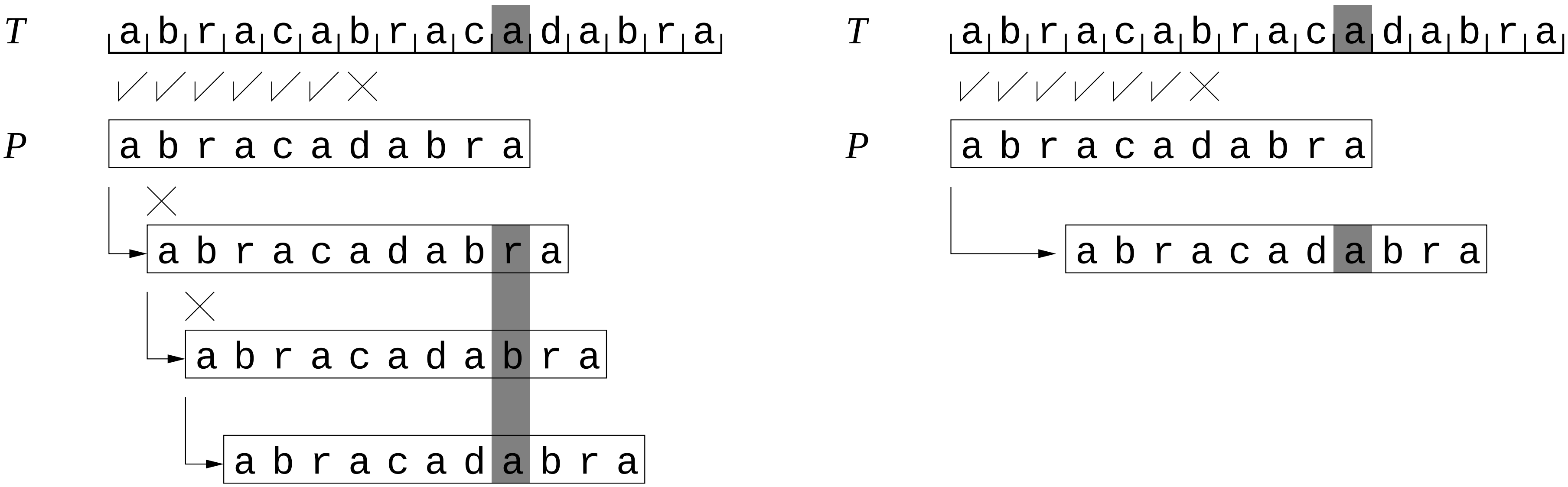
- The first text window is `abracabraca`
- After verifying that it does not match P, the window is shifted by one position

Simple Strings: Horspool

- **Horspool's algorithm** is in the fortunate position of being very simple to understand and program
- It is the fastest algorithm in many situations, especially when searching natural language texts
- Horspool's algorithm uses one of the Boyer-Moore's ideas to shift the window in a smarter way
- A table d indexed by the characters of the alphabet is precomputed:
  - $d[c]$ tells how many positions can the window be shifted if the final character of the window is c
- In other words, $d[c]$ is the distance from the end of the pattern to the last occurrence of c in P, excluding the occurrence of p_m

Simple Strings: Horspool

- The diagram below repeats the previous example, now also applying Horspool's shift



Simple Strings: Horspool

- Pseudocode for Horspool's string matching algorithm

Horspool $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m)$

(1) **for** $c \in \Sigma$ **do** $d[c] \leftarrow m$

(2) **for** $j \leftarrow 1 \ldots m-1$ **do** $d[p_j] \leftarrow m - j$

(3) $i \leftarrow 0$

(4) **while** $i \leq n - m$ **do**

(5) $j \leftarrow 1$

(6)   **while** $j \leq m \ \wedge \ t_{i+j} = p_j$ **do** $j \leftarrow j + 1$

(7)   **if** $j > m$ **then**

(8)   report an occurrence at text position $i + 1$

(9) $i \leftarrow i + d[t_{i+m}]$

Small alphabets and long patterns

- When searching for long patterns over small alphabets Horspool's algorithm does not perform well
  - Imagine a computational biology application where strings of 300 nucleotides over the four-letter alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ are sought
- This problem can be alleviated by considering consecutive pairs of characters to shift the window
  - On other words, we can align the pattern with the last pair of window characters, $t_{i+m-1}t_{i+m}$
- In the previous example, we would shift by $4^2 = 16$ positions on average

Small alphabets and long patterns

- In general we can shift using q characters at the end of the window: which is the best value for q?
  - We cannot shift by more than m, and thus $\sigma^q \leq m$ seems to be a natural limit
  - If we set $q = \log_\sigma m$, the average search time will be $O(n \log_\sigma(m)/m)$
- Actually, this average complexity is optimal, and the choice for q we derived is close to correct
- It can be analytically shown that, by choosing $q = 2 \log_\sigma m$, the average search time achieves the optimal $O(n \log_\sigma(m)/m)$

Small alphabets and long patterns

- This technique is used in the **agrep** software
- A hash function is chosen to map q-grams (strings of length q) onto an integer range
- Then the distance from each q-gram of P to the end of P is recorded in the hash table
- For the q-grams that do not exist in P, distance $m - q + 1$ is used

Small alphabets and long patterns

- Pseudocode for the **agrep**'s algorithm to match long patterns over small alphabets (simplified)

Agrep $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m,\ q,\ h(\ ),\ N)$

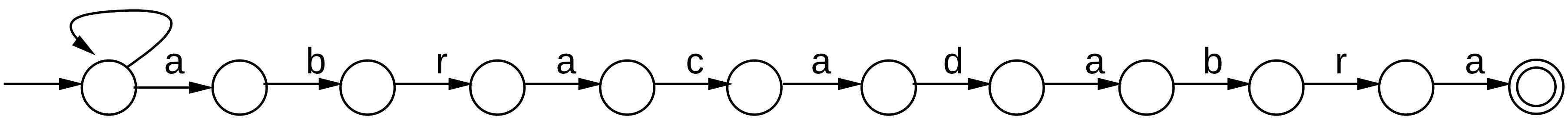
for $i \in [1, N]$ **do** $d[i] \leftarrow m - q + 1$
for $j \leftarrow 0 \ldots m - q$ **do** $d[h(p_{j+1} p_{j+2} \ldots p_{j+q})] \leftarrow m - q - j$
$i \leftarrow 0$
while $i \le n - m$ **do**
    $s \leftarrow d[h(t_{i+m-q+1} t_{i+m-q+2} \ldots t_{i+m})]$
    if $s > 0$ **then** $i \leftarrow i + s$
    else
        $j \leftarrow 1$
        while $j \le m \wedge t_{i+j} = p_j$ **do** $j \leftarrow j + 1$
        if $j > m$ **then** report an occurrence at text position $i + 1$
        $i \leftarrow i + 1$

Automata and Bit-Parallelism

- Horspool's algorithm, as well as most classical algorithms, does not adapt well to complex patterns
- We now show how **automata** and **bit-parallelism** allows to handle many complex patterns

Automata

- Figure below shows, on top, a NFA to search for the pattern $P = \texttt{abracadabra}$
  - The initial self-loop matches any character
  - Each table column corresponds to an edge of the automaton



B[a] =	0	1	1	0	1	0	1	0	1	1	0
B[b] =	1	0	1	1	1	1	1	1	0	1	1
B[r] =	1	1	0	1	1	1	1	1	1	0	1
B[c] =	1	1	1	1	0	1	1	1	1	1	1
B[d] =	1	1	1	1	1	1	0	1	1	1	1
B[*] =	1	1	1	1	1	1	1	1	1	1	1

Automata

- It can be seen that the NFA in the previous Figure accepts any string that finishes with $P =$ `'abracadabra'`
- The initial state is always active because of the self-loop that can be traversed by any character
- Note that several states can be simultaneously active
  - For example, after reading `'abra'`, NFA states 0, 1, and 4 will be active

Bit-parallelism

- **Bit-parallelism** takes advantage of the intrinsic parallelism of bit operations
- Bit masks are read right to left, so that the first bit of $b_m \ldots b_1$ is b_1
- Bit masks are handled with operations like:
  - $|$ to denote the bit-wise **or**
  - $\&$ to denote the bit-wise **and**, and
  - $\hat{}$ to denote the bit-wise **xor**
- Unary operation '$\sim$' complements all the bits
- $mask << i$ means shifting all the bits in $mask$ by i positions to the left, entering zero bits from the right ($mask >> i$ is analogous)
- Finally, it is possible to operate bit masks as numbers, for example adding or subtracting them

Shift-And Algorithm

- The simplest bit-parallel algorithm allows matching single strings, and it is called **Shift-And** or **Baeza-Yates & Gonnet**
- The algorithm builds a table B which, for each character, stores a bit mask $b_m \dots b_1$
  - The mask in $B[c]$ has the i-th bit set if and only if $p_i = c$
- The state of the search is kept in a machine word $D = d_m \dots d_1$, where d_i is set if the state i is active
  - Therefore, a match is reported whenever $d_m = 1$
- Note that state number zero is not represented in D because it is always active and then can be left implicit

Shift-And Algorithm

- Pseudocode for the Shift-And algorithm

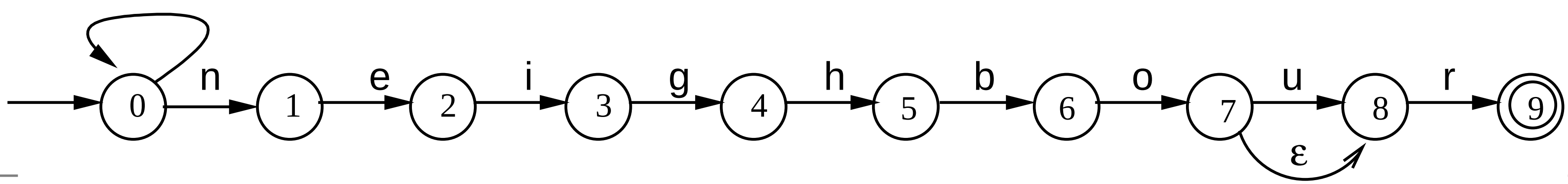
  Shift-And $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m)$

  (1) **for** $c \in \Sigma$ **do** $B[c] \leftarrow 0$
  (2) **for** $j \leftarrow 1 \ldots m$ **do** $B[p_j] \leftarrow B[p_j] \mid (1 << (j-1))$
  (3) $D \leftarrow 0$
  (4) **for** $i \leftarrow 1 \ldots n$ **do**
  (5) $\quad D \leftarrow ((D << 1) \mid 1)\ \&\ B[t_i]$
  (6) $\quad$ **if** $D\ \&\ (1 << (m-1)) \neq 0$
  (7) $\quad$ **then** report an occurrence at text position $i - m + 1$

- There must be sufficient bits in the computer word to store one bit per pattern position
  - For longer patterns, in practice we can search for $p_1 p_2 \ldots p_w$, and directly check the occurrences of this prefix for the complete P

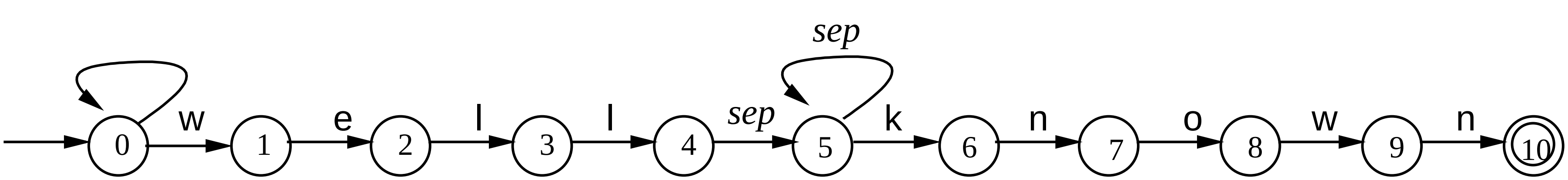
Extending Shift-And

- Shift-And can deal with much more complex patterns than Horspool
- The simplest case is that of **classes of characters**:
  - This is the case, for example, when one wishes to search in case-insensitive fashion, or one wishes to look for a whole word
- Let us now consider a more complicated pattern
  - Imagine that we search for `neighbour`, but we wish the `u` to be optional (accepting both English and American style)
- The Figure below shows an NFA that does the task using an ε-transition



Extending Shift-And

- Another feature in complex patterns is the use of **wild cards**, or more generally repeatable characters
  - Those are pattern positions that can appear once or more times, consecutively, in the text
- For example, we might want to catch all the transfer records in a banking log
- As another example, we might look for `well known`, yet there might be a hyphen or one or more spaces
  - For instance `'well known'`, `'well  known'`, `'well-known'`, `'well - known'`, `'well` $\backslash n$ `known'`, and so on



Extending Shift-And

- Figure below shows pseudocode for a Shift-And extension that handles all these cases

Shift-And-Extended $(T = t_1 t_2 \ldots t_n,\ m,\ B[\,],\ A,\ S)$

$I \leftarrow (A >> 1)\ \&\ (A\,\hat{}\,(A >> 1))$
$F \leftarrow A\ \&\ (A\,\hat{}\,(A >> 1))$
$D \leftarrow 0$
for $i \leftarrow 1 \ldots n$ **do**
    $D \leftarrow (((D << 1) \mid 1) \mid (D\ \&\ S))\ \&\ B[t_i]$
    $Df \leftarrow D \mid F$
    $D \leftarrow D \mid (A\ \&\ ((\sim (Df - I))\,\hat{}\,Df))$
    if $D\ \&\ (1 << (m-1)) \neq 0$
    then report an occurrence at text position $i - m + 1$

Inverted Indexes

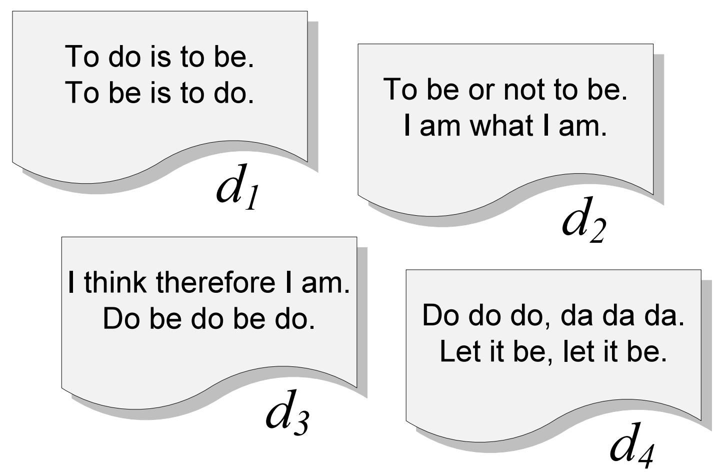
Basic Concepts

- **Inverted index**: a word-oriented mechanism for indexing a text collection to speed up the searching task
- The inverted index structure is composed of two elements: the **vocabulary** and the **occurrences**
- The vocabulary is the set of all different words in the text
- For each word in the vocabulary the index stores the documents which contain that word (inverted index)

Basic Concepts

- **Term-document matrix**: the simplest way to represent the documents that contain each word of the vocabulary

Vocabulary	n_i	d_1	d_2	d_3	d_4
to	2	4	2	-	-
do	3	2	-	3	3
is	1	2	-	-	-
be	4	2	2	2	2
or	1	-	1	-	-
not	1	-	1	-	-
I	2	-	2	2	-
am	2	-	2	1	-
what	1	-	1	-	-
think	1	-	-	1	-
therefore	1	-	-	1	-
da	1	-	-	-	3
let	1	-	-	-	2
it	1	-	-	-	2



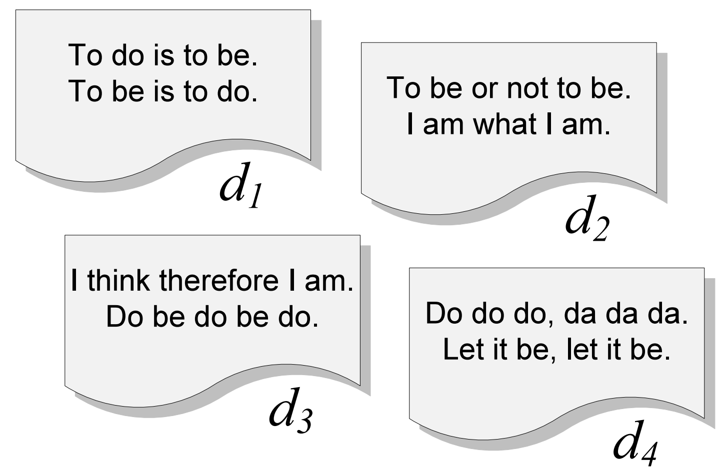
Basic Concepts

- The main problem of this simple solution is that it requires too much space
- As this is a sparse matrix, the solution is to associate a list of documents with each word
- The set of all those lists is called the **occurrences** and each one an **inverted list**

Basic Concepts

- Basic inverted index

Vocabulary	n_i	Occurrences as inverted lists
to	2	[1,4],[2,2]
do	3	[1,2],[3,3],[4,3]
is	1	[1,2]
be	4	[1,2],[2,2],[3,2],[4,2]
or	1	[2,1]
not	1	[2,1]
I	2	[2,2],[3,2]
am	2	[2,2],[3,1]
what	1	[2,1]
think	1	[3,1]
therefore	1	[3,1]
da	1	[4,3]
let	1	[4,2]
it	1	[4,2]



Resolving Simple Queries

Single Word Queries

- The simplest type of search is that for the occurrences of a single word
- The vocabulary search can be carried out using any suitable data structure
  - Ex: hashing, tries, or B-trees
- The first two provide $O(m)$ search cost, where m is the length of the query
- We note that the vocabulary is in most cases sufficiently small so as to stay in main memory
- The occurrence lists, on the other hand, are usually fetched from disk

Multiple Word Queries

- If the query has more than one word, we have to consider two cases:
  - conjunctive (AND operator) queries
  - disjunctive (OR operator) queries
- **Conjunctive queries** imply to search for all the words in the query, obtaining one inverted list for each word
- Following, we have to **intersect** all the inverted lists to obtain the documents that contain all these words
- For **disjunctive queries** the lists must be **merged**
- The first case is popular in the Web due to the size of the document collection

List Intersection

- The most time-demanding operation on inverted indexes is the merging of the lists of occurrences
  - Thus, it is important to optimize it
- Consider one pair of lists of sizes m and n respectively, stored in consecutive memory, that needs to be intersected
- If m is much smaller than n, it is better to do m binary searches in the larger list to do the intersection
- If m and n are comparable, Baeza-Yates devised a double binary search algorithm
  - It is $O(\log n)$ if the intersection is trivially empty
  - It requires less than $m + n$ comparisons on average

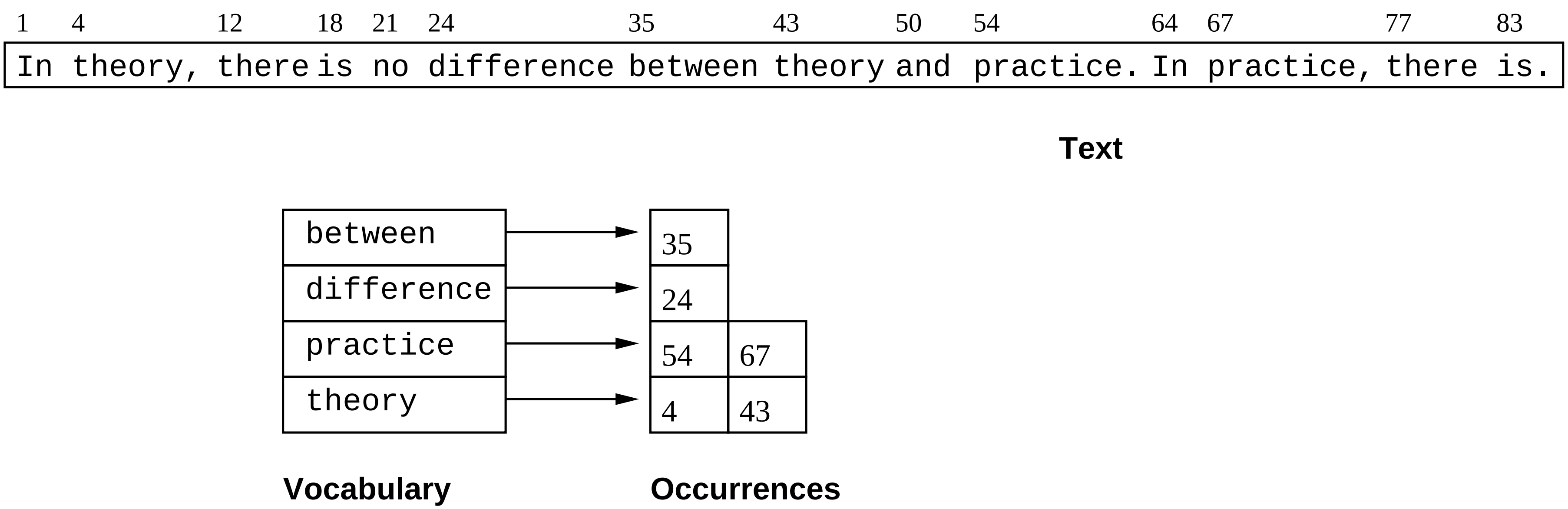
List Intersection

- When there are more than two lists, there are several possible heuristics depending on the list sizes
- If intersecting the two shortest lists gives a very small answer, might be better to intersect that to the next shortest list, and so on
- The algorithms are more complicated if lists are stored non-contiguously and/or compressed

Full Inverted Indexes

Full Inverted Indexes

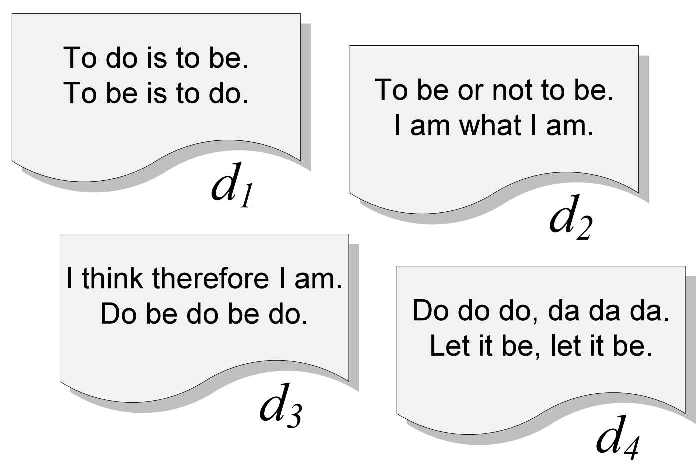
- The basic index is not suitable for answering phrase or proximity queries
- Hence, we need to add the positions of each word in each document to the index (full inverted index)



Full Inverted Indexes

- In the case of multiple documents, we need to store one occurrence list per term-document pair

Vocabulary	n_i	Occurrences as full inverted lists
to	2	[1,4,[1,4,6,9]],[2,2,[1,5]]
do	3	[1,2,[2,10]],[3,3,[6,8,10]],[4,3,[1,2,3]]
is	1	[1,2,[3,8]]
be	4	[1,2,[5,7]],[2,2,[2,6]],[3,2,[7,9]],[4,2,[9,12]]
or	1	[2,1,[3]]
not	1	[2,1,[4]]
I	2	[2,2,[7,10]],[3,2,[1,4]]
am	2	[2,2,[8,11]],[3,1,[5]]
what	1	[2,1,[9]]
think	1	[3,1,[2]]
therefore	1	[3,1,[3]]
da	1	[4,3,[4,5,6]]
let	1	[4,2,[7,10]]
it	1	[4,2,[8,11]]



Full Inverted Indexes

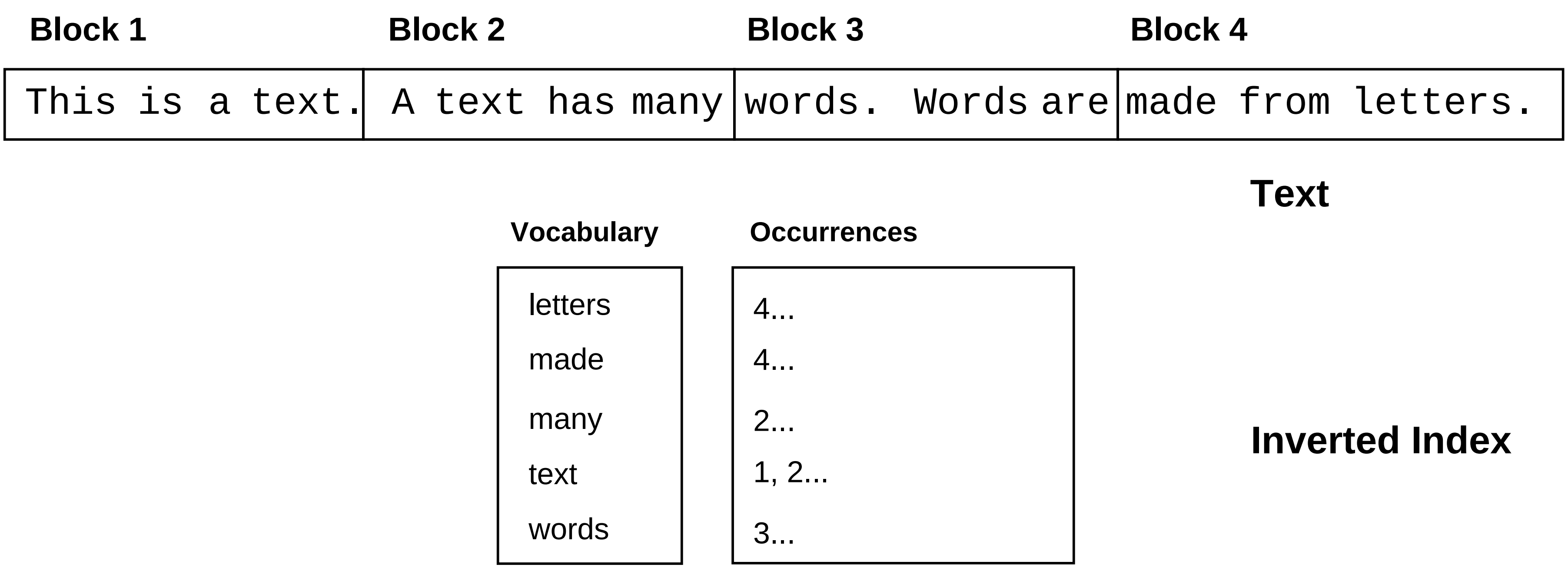
- The space required for the vocabulary is rather small
- Heaps' law: the vocabulary grows as $O(n^{\beta})$, where
  - n is the collection size
  - β is a collection-dependent constant between 0.4 and 0.6
- For instance, the vocabulary of 1 gigabyte of text occupies about 5 megabytes
- This may be further reduced by stemming and other normalization techniques

Full Inverted Indexes

- The occurrences demand much more space
- Functional words, also called **stopwords** can be omitted
- The extra space will be $O(n)$ and is around
  - 40% of the text size if stopwords are omitted
  - 80% when stopwords are indexed
- Document-addressing indexes are smaller, because only one occurrence per file must be recorded, for a given word
- Depending on the document (file) size, document-addressing indexes typically require 20% to 40% of the text size

Full Inverted Indexes

- To reduce space requirements, a technique called **block addressing** is used
- The documents are divided into blocks, and the occurrences point to the blocks where the word appears



Full Inverted Indexes

- The Table below presents the projected space taken by inverted indexes for texts of different sizes

Index granularity	Single document (1 MB)		Small collection (200 MB)		Medium collection (2 GB)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	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%

Full Inverted Indexes

- The blocks can be of fixed size or they can be defined using the division of the text collection into documents
- The division into blocks of fixed size improves efficiency at retrieval time
  - This is because larger blocks match queries more frequently and are more expensive to traverse
- This technique also profits from *locality of reference*
  - That is, the same word will be used many times in the same context and all the references to that word will be collapsed in just one reference

Answering More Complex Queries

Phrase and Proximity Queries

- Context queries are more difficult to solve with inverted indexes
- The lists of all elements must be traversed to find places where
  - all the words appear in sequence (for a phrase), or
  - appear close enough (for proximity)
  - these algorithms are similar to a list intersection algorithm
- Another solution for phrase queries is based on indexing two-word phrases and using similar algorithms over pairs of words
  - however the index will be much larger as the number of word pairs is not linear

More Complex Queries

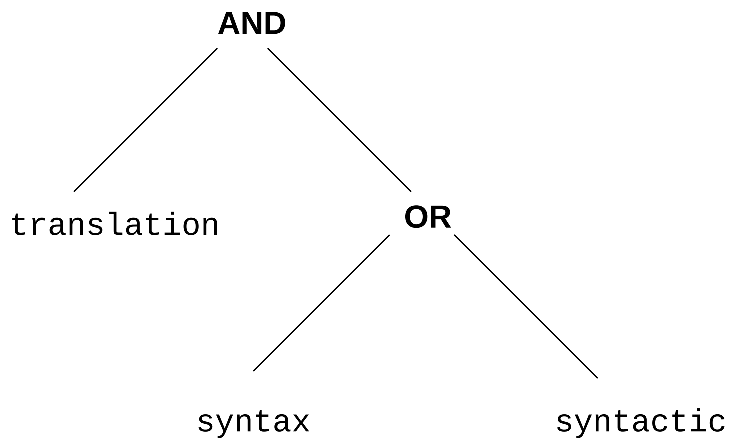
- Prefix and range queries are basically (larger) disjunctive queries
- In these queries there are usually several words that match the pattern
    - Thus, we end up again with several inverted lists and we can use the algorithms for list intersection

More Complex Queries

- To search for regular expressions the data structures built over the vocabulary are rarely useful
- The solution is then to **sequentially** traverse the vocabulary, to spot all the words that match the pattern
- Such a sequential traversal is not prohibitively costly because it is carried out only on the vocabulary

Boolean Queries

- In boolean queries, a **query syntax** tree is naturally defined



- Normally, for boolean queries, the search proceeds in three phases:
  - the first phase determines which documents to match
  - the second determines the likelihood of relevance of the documents matched
  - the final phase retrieves the exact positions of the matches to allow highlighting them during browsing, if required

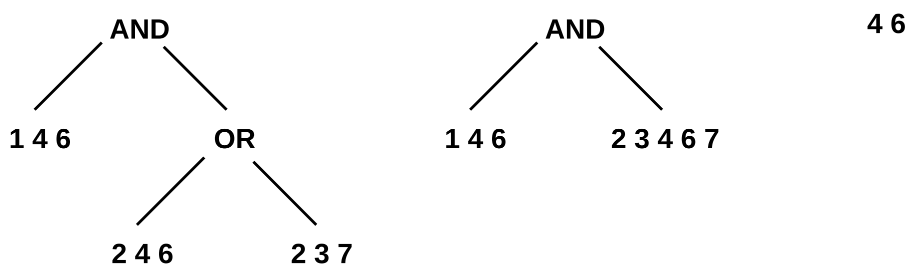
Boolean Queries

- Once the leaves of the query syntax tree find the classifying sets of documents, these sets are further operated by the internal nodes of the tree
- Under this scheme, it is possible to evaluate the syntax tree in **full** or **lazy** form
  - In the full evaluation form, both operands are first completely obtained and then the complete result is generated
  - In lazy evaluation, the partial results from operands are delivered only when required, and then the final result is recursively generated

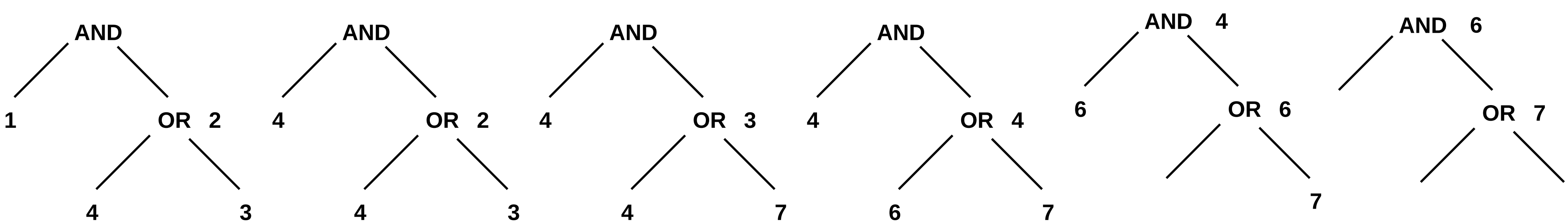
Boolean Queries

- Processing the internal nodes of the query syntax tree
  - In (a) full evaluation is used
  - In (b) we show lazy evaluation in more detail

a)



b)



Building an Inverted Index

Internal Algorithms

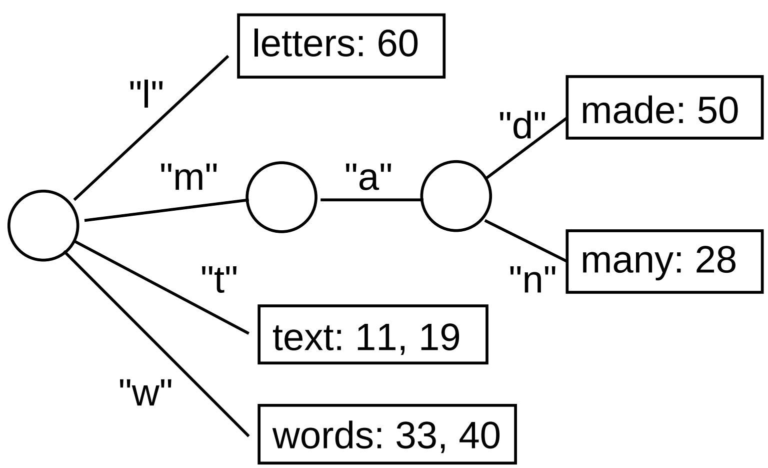
- Building an index in internal memory is a relatively simple and low-cost task
- A dynamic data structure to hold the vocabulary (B-tree, hash table, etc.) is created empty
- Then, the text is scanned and each consecutive word is searched for in the vocabulary
- If it is a new word, it is inserted in the vocabulary before proceeding

Internal Algorithms

- A large array is allocated where the identifier of each consecutive text word is stored
- A full-text inverted index for a sample text with the incremental algorithm:

```
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.
```

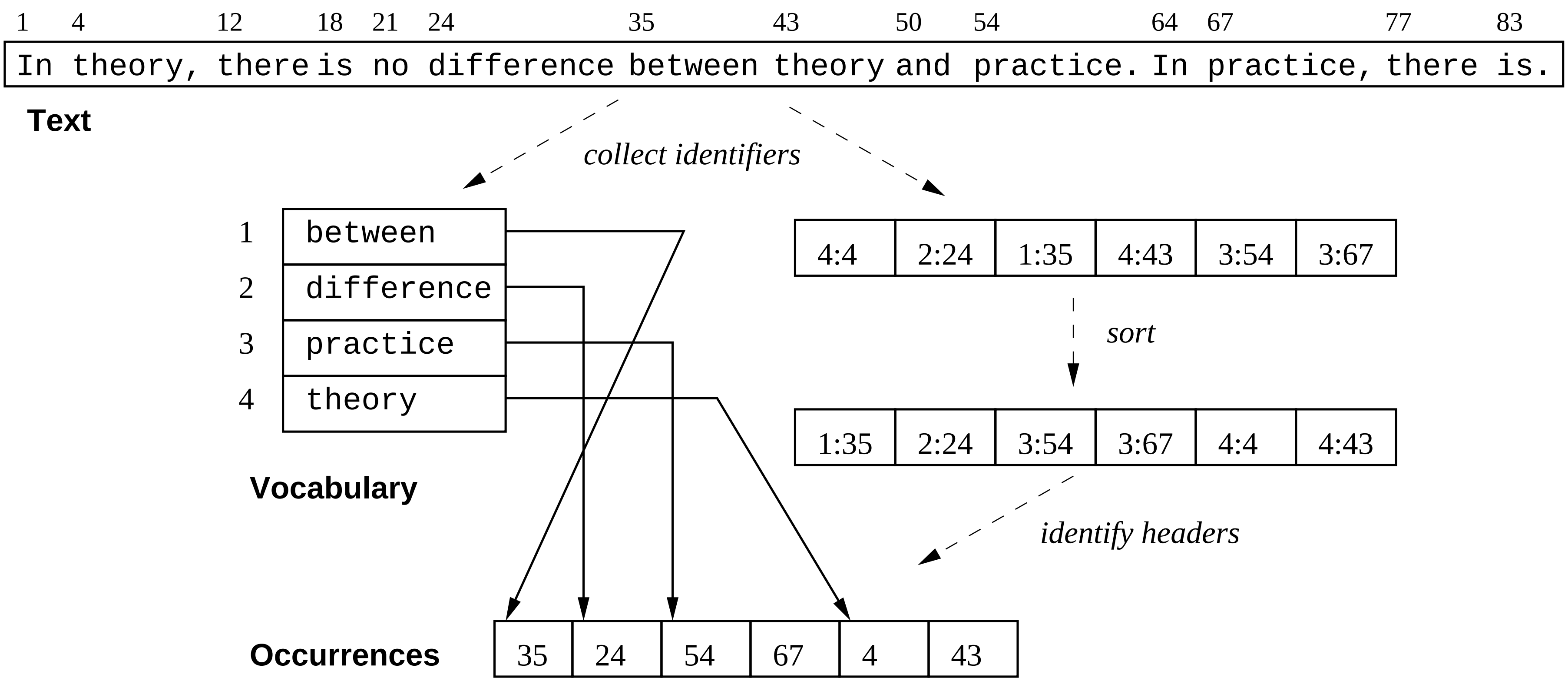
Text



Vocabulary trie

Internal Algorithms

- A full-text inverted index for a sample text with a sorting algorithm:



Internal Algorithms

- An alternative to avoid this sorting is to separate the lists from the beginning
  - In this case, each vocabulary word will hold a pointer to its own array (list) of occurrences, initially empty
- A non trivial issue is how the memory for the many lists of occurrences should be allocated
  - A classical list in which each element is allocated individually wastes too much space
  - Instead, a scheme where a list of blocks is allocated, each block holding several entries, is preferable

Internal Algorithms

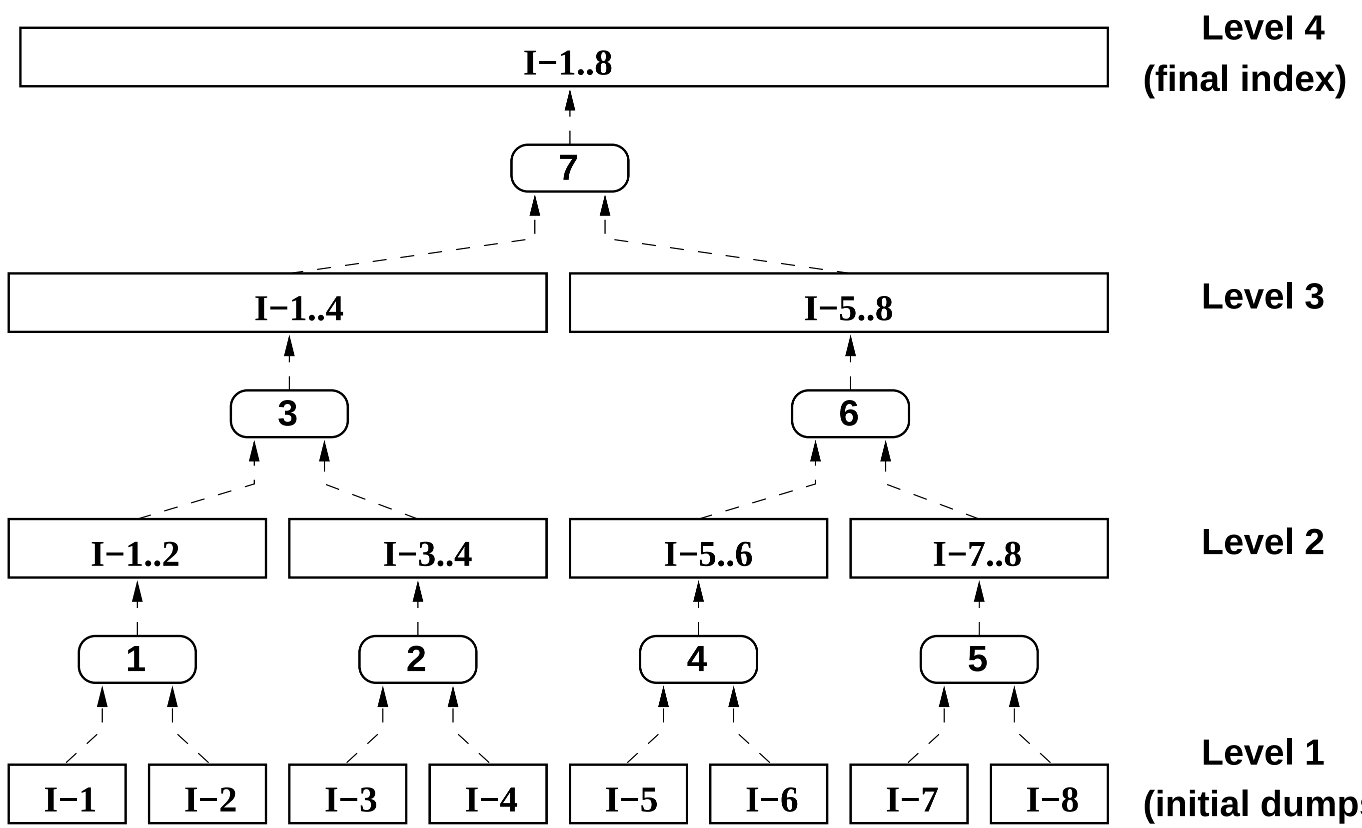
- Once the process is completed, the vocabulary and the lists of occurrences are written on two distinct disk files
- The vocabulary contains, for each word, a pointer to the position of the inverted list of the word
- This allows the vocabulary to be kept in main memory at search time in most cases

External Algorithms

- All the previous algorithms can be extended by using them until the main memory is exhausted
- At this point, the **partial** index I_i obtained up to now is written to disk and erased from main memory
- These indexes are then **merged** in a hierarchical fashion

External Algorithms

- Merging the partial indexes in a binary fashion
  - Rectangles represent partial indexes, while rounded rectangles represent merging operations



External Algorithms

- In general, maintaining an inverted index can be done in three different ways:
  - **Rebuild**
    - If the text is not that large, rebuilding the index is the simplest solution
  - **Incremental updates**
    - We can amortize the cost of updates while we search
    - That is, we only modify an inverted list when needed
  - **Intermittent merge**
    - New documents are indexed and the resultant partial index is merged with the large index
    - This in general is the best solution

Compressed Inverted Indexes

- It is possible to combine index compression and text compression without any complication
  - In fact, in all the construction algorithms mentioned, compression can be added as a final step
- In a full-text inverted index, the lists of text positions or file identifiers are in ascending order
- Therefore, they can be represented as sequences of **gaps** between consecutive numbers
  - Notice that these gaps are small for frequent words and large for infrequent words
  - Thus, compression can be obtained by encoding small values with shorter codes

Inverted Indexes

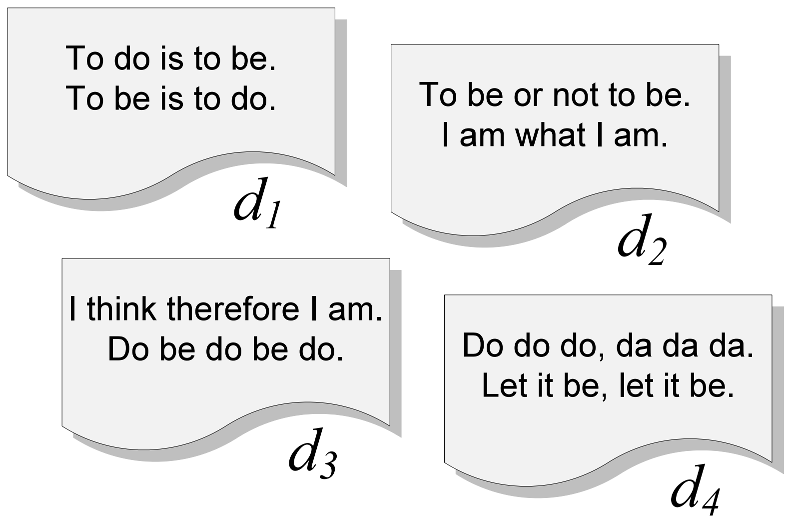
Ranking

Ranking

- How to find the top-k documents and return them to the user when we have weight-sorted inverted lists?
- If we have a single word query, the answer is trivial as the list can be already sorted by the desired ranking
- For other queries, we need to merge the lists

Ranking

- Suppose that we are searching the disjunctive query "`to do`" on the collection below



- As our collection is very small, let us assume that we are interested in the top-2 ranked documents
- We can use the following heuristic:
  - we process terms in IDF order (shorter lists first), and
  - each term is processed in TF order (simple ranking order)

Ranking

Ranking-in-the-vector-model(query terms t)

Create P as C-candidate similarities initialized to $(P_d, P_w) = (0, 0)$
Sort the query terms t by decreasing weight
$c \leftarrow 1$
for each sorted term t in the query **do**
  Compute the value of the threshold t_{add}
  Retrieve the inverted list for t, L_t
  for each document d in L_t **do**
    if $w_{d,t} < t_{add}$ **then break**
    $psim \leftarrow w_{d,t} \times w_{q,t} / W_d$
    if $d \in P_d(i)$ **then** $P_w(i) \leftarrow P_w(i) + psim$
    elif $psim > min_j(P_w(j))$ **then** $n \leftarrow min_j(P_w(j))$
    elif $c \leq C$ **then** $n \leftarrow c$; $c \leftarrow c + 1$
    if $n \leq C$ **then** $P(n) \leftarrow (d, psim)$
return the top-k documents according to P_w

- This is a variant of Persin's algorithm
- We use a priority queue P of C document candidates where we will compute partial similarities