# Inverted Indexes

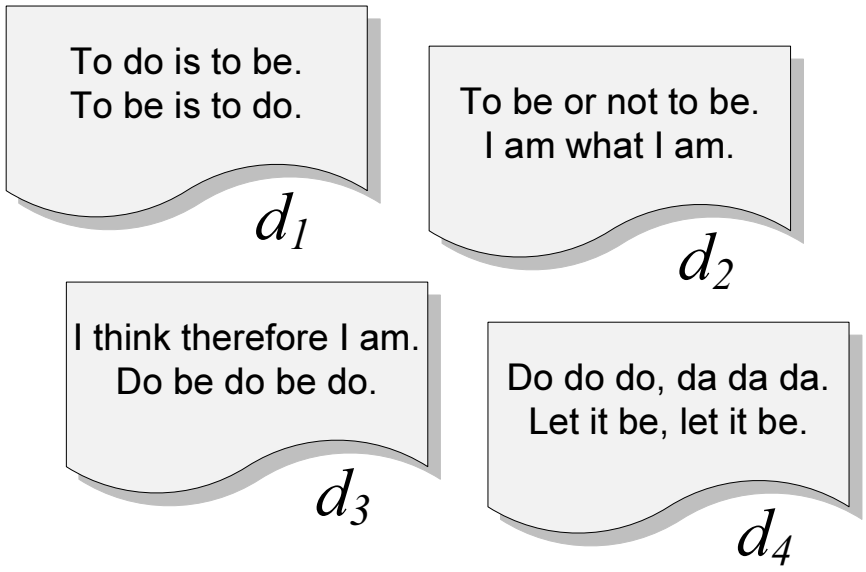
# Basic Concepts

- **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 |



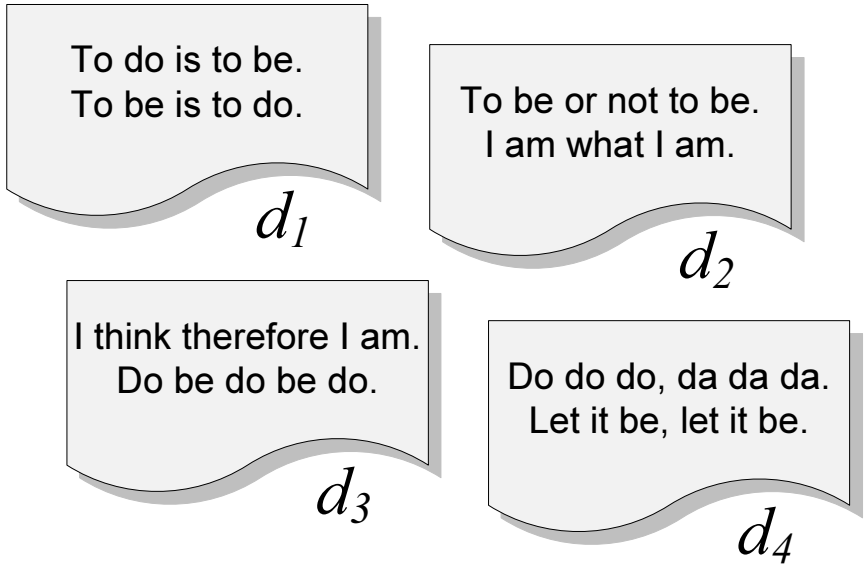
# Basic Concepts

- 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**

# 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] |

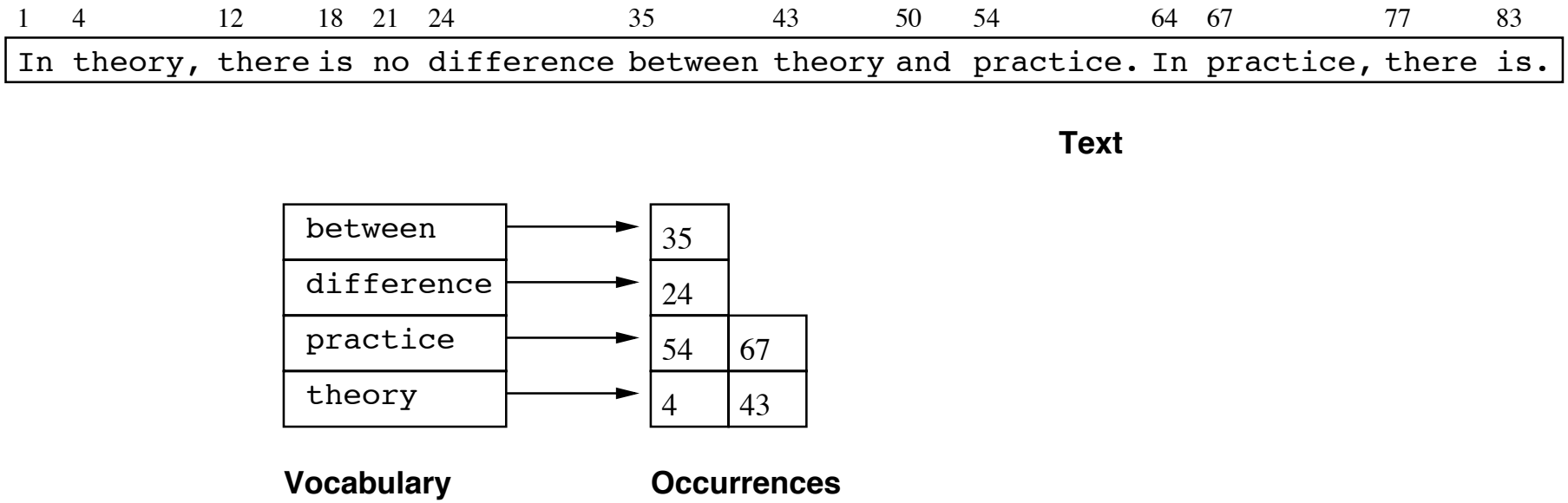


# Inverted Indexes

## Full Inverted Indexes

# Full Inverted Indexes

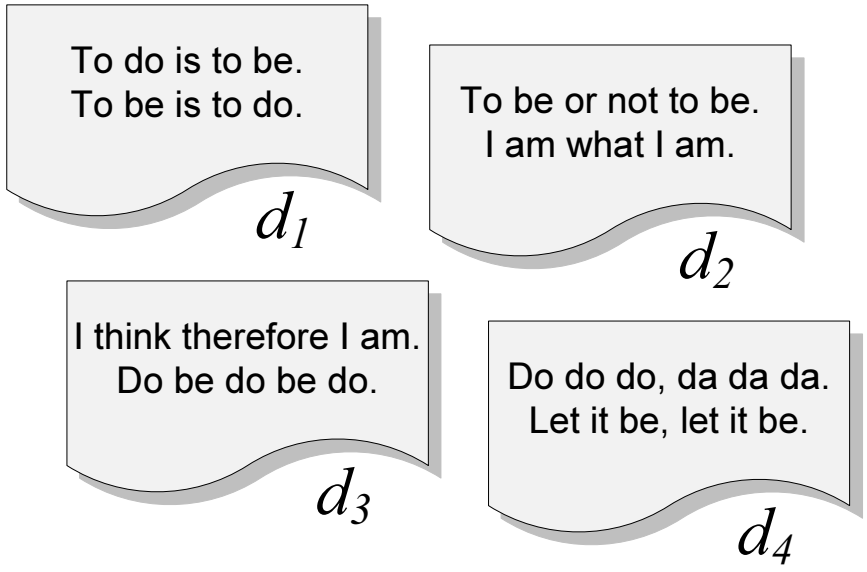
- The basic index is not suitable for answering phrase or proximity queries
- Hence, we need to add the positions of each word in each document to the index (full inverted index)



# 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

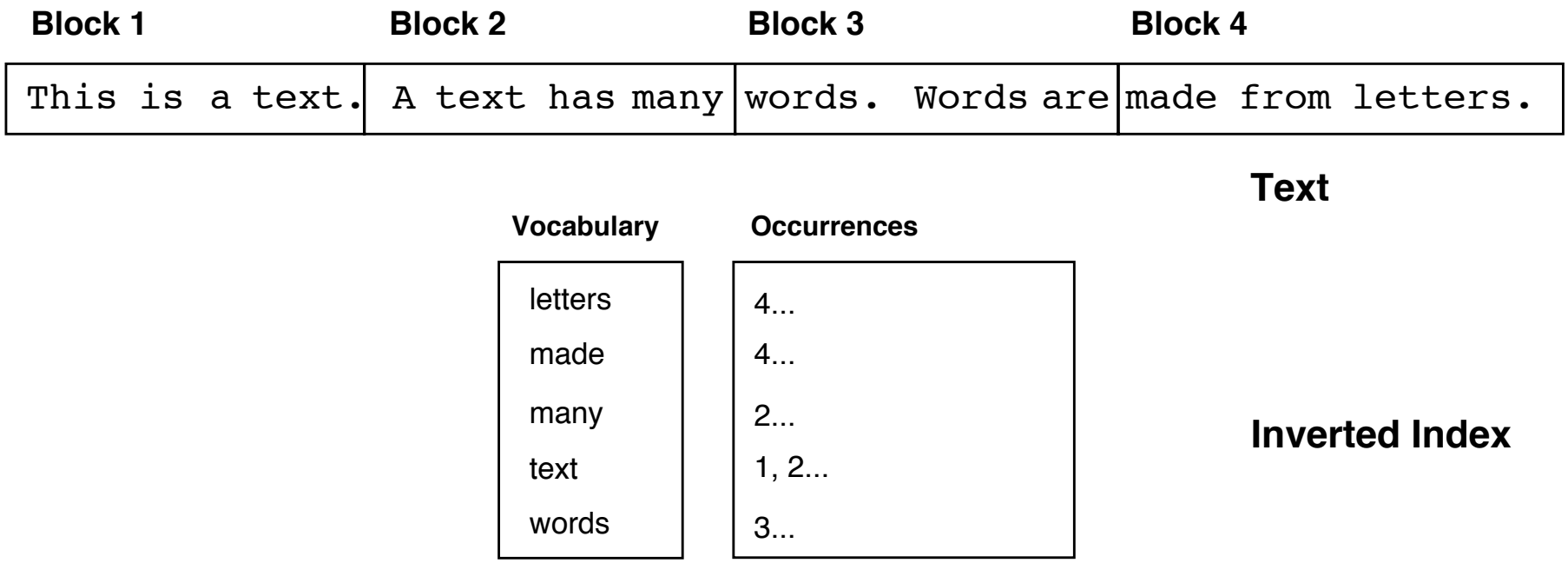
- The space required for the vocabulary is rather small
- Heaps' law: the vocabulary grows as $O(n^{\beta})$, where
  - $n$ is the collection size
  - $\beta$ is a collection-dependent constant between 0.4 and 0.6
- For instance, in the TREC-3 collection, the vocabulary of 1 gigabyte of text occupies only 5 megabytes
- This may be further reduced by stemming and other normalization techniques

# Full Inverted Indexes

- The occurrences demand much more space
- 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

# 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