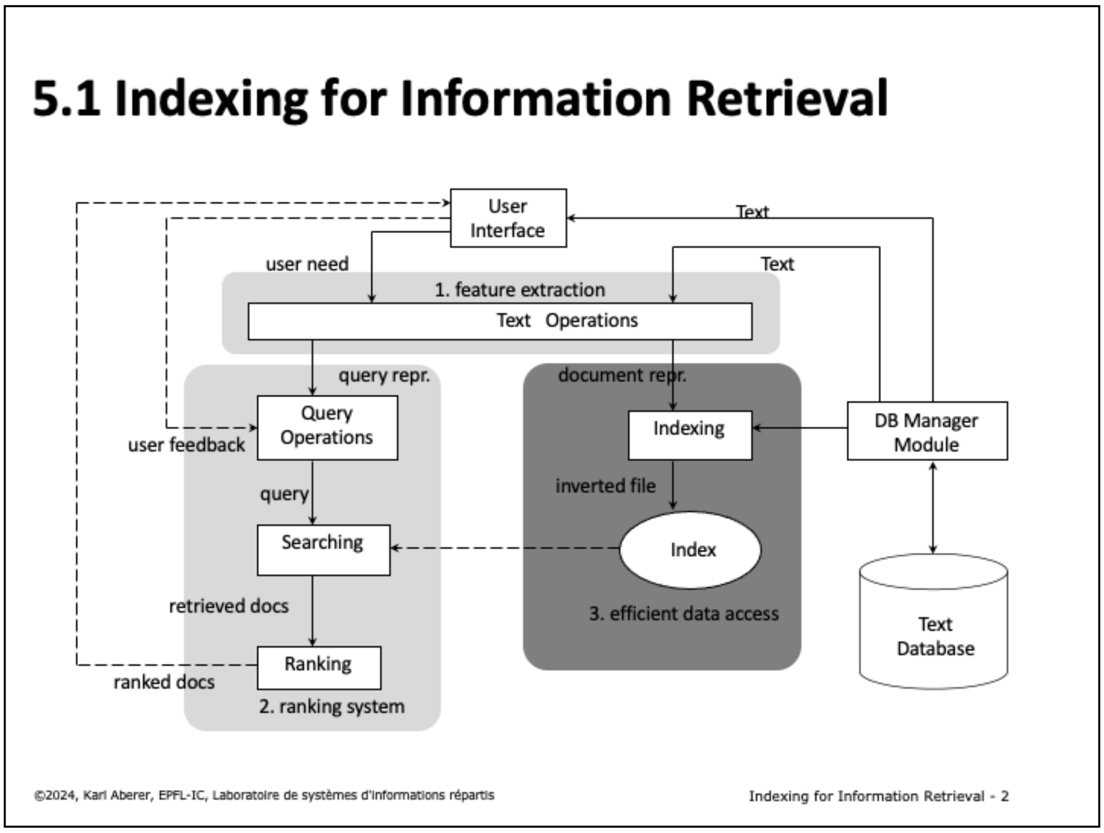
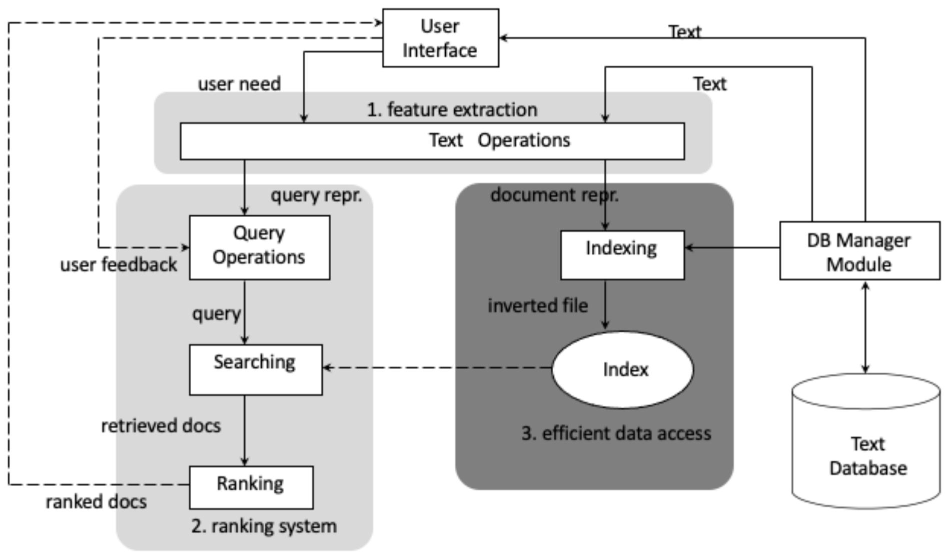
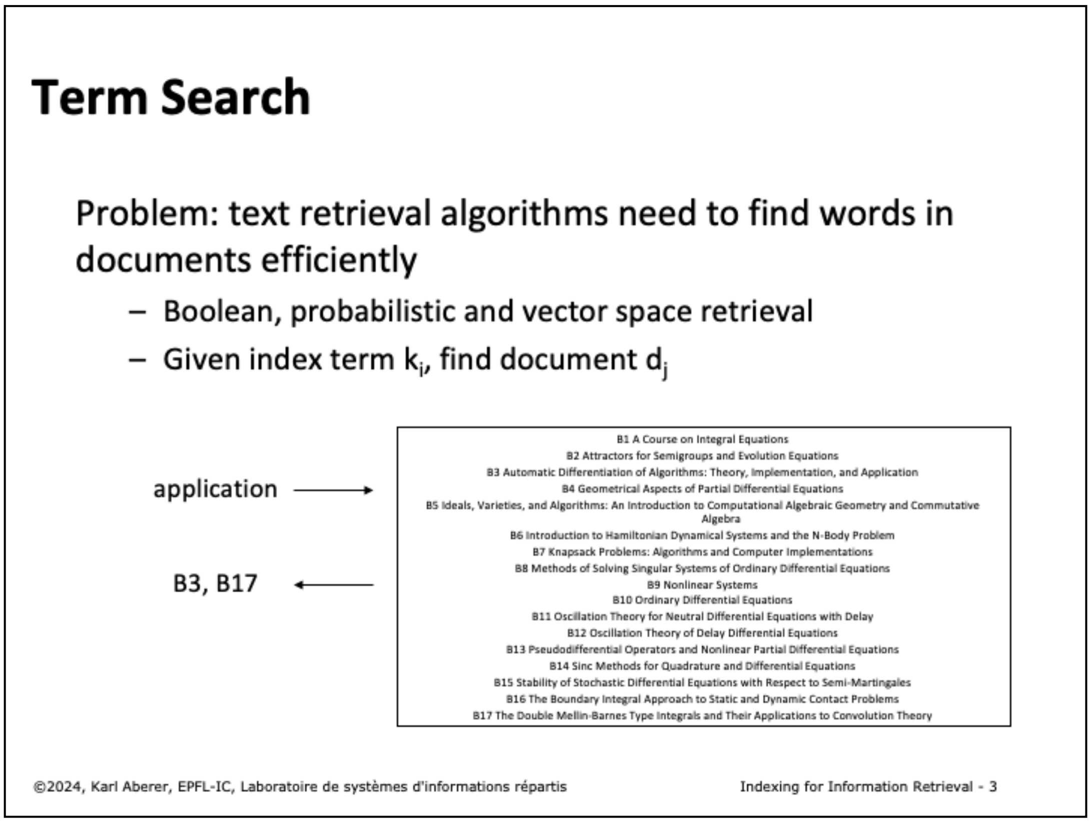
5. DATA INDEXING AND MINING



5.1 Indexing for Information Retrieval



For efficiently implementing a retrieval model in the ranking system, the statistical information about documents and the document access need to be supported by an indexing system. We will now introduce the basic methods that have been devised to that end.



Term Search

Problem: text retrieval algorithms need to find words in documents efficiently

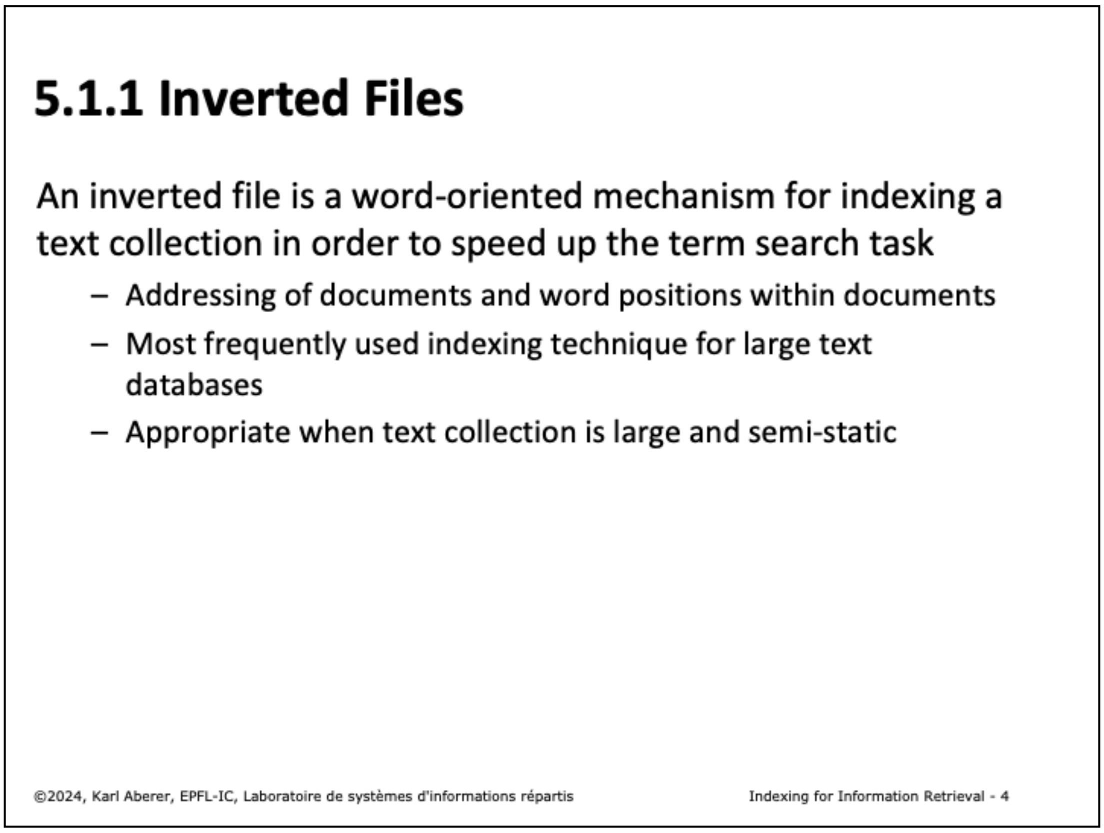
- Boolean, probabilistic and vector space retrieval
- Given index term k_i, find document d_j

application →

B3, B17 ←

B1 A Course on Integral Equations
B2 Attractors for Semigroups and Evolution Equations
B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4 Geometrical Aspects of Partial Differential Equations
B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6 Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7 Knapsack Problems: Algorithms and Computer Implementations
B8 Methods of Solving Singular Systems of Ordinary Differential Equations
B9 Nonlinear Systems
B10 Ordinary Differential Equations
B11 Oscillation Theory for Neutral Differential Equations with Delay
B12 Oscillation Theory of Delay Differential Equations
B13 Pseudodifferential Operators and Nonlinear Partial Differential Equations
B14 Sinc Methods for Quadrature and Differential Equations
B15 Stability of Stochastic Differential Equations with Respect to Semi-Martingales
B16 The Boundary Integral Approach to Static and Dynamic Contact Problems
B17 The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory

In order to implement text retrieval models efficiently, efficient search for term occurrences in documents must be supported. For that purpose, different indexing techniques exist, among which inverted files are the by far most widely used.

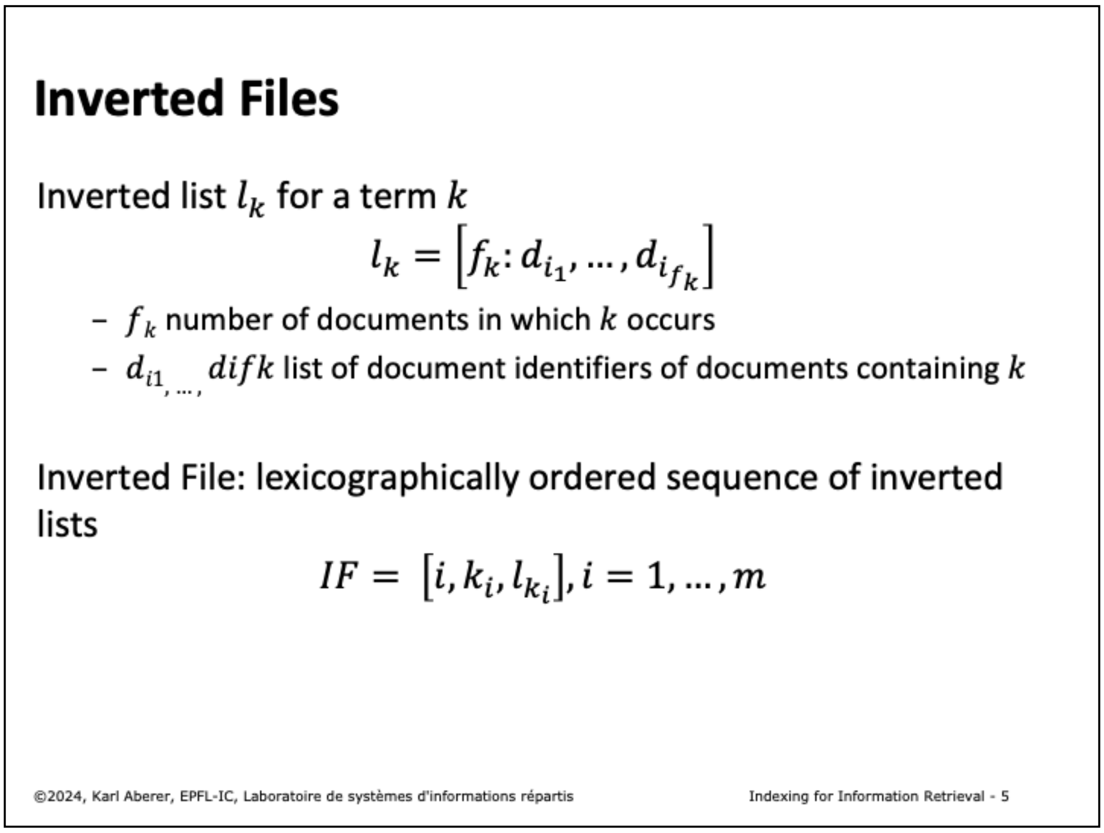


5.1.1 Inverted Files

An inverted file is a word-oriented mechanism for indexing a text collection in order to speed up the term search task

- Addressing of documents and word positions within documents
- Most frequently used indexing technique for large text databases
- Appropriate when text collection is large and semi-static

Inverted files support efficient addressing of words within documents. Inverted file are designed for supporting search on relatively static text collections. Therefore, their inverted files do not support continuous updates to the index when documents are updated in the corpus. Rather index construction is a one-shot process, analyzing a complete document collection. This distinguishes inverted files from typical database indexing techniques, such as B+-Trees.



Inverted Files

Inverted list l_k for a term k

$$l_k = \left[f_k: d_{i_1}, \dots, d_{i_{f_k}}\right]$$

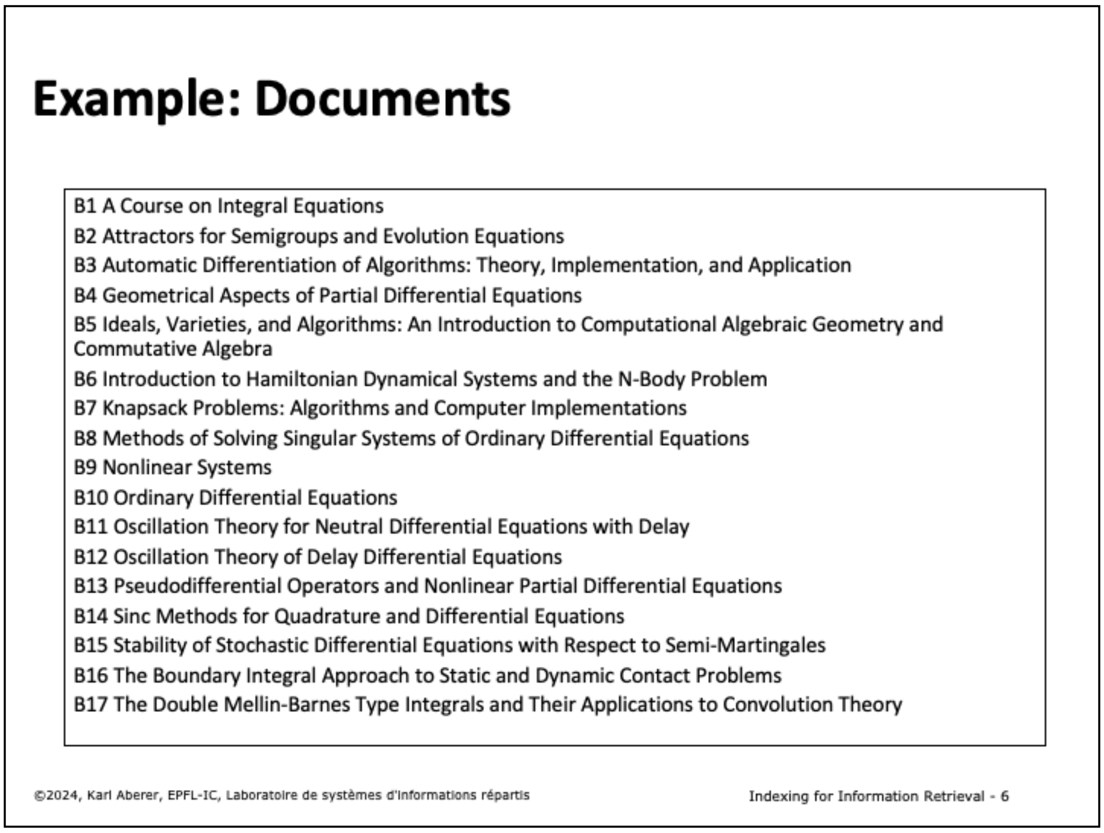
- f_k number of documents in which k occurs
- $d_{i1}, \dots, difk$ list of document identifiers of documents containing k

Inverted File: lexicographically ordered sequence of inverted lists

$$IF = \left[i, k_i, l_{k_i}\right], i = 1, \dots, m$$

Inverted files are constructed from inverted lists. Inverted lists enumerate all occurrences of the terms in documents, by storing the document identifiers and the global frequency of occurrences. Inverted files are constructed by concatenating the inverted lists for all terms occurring in the document collection.

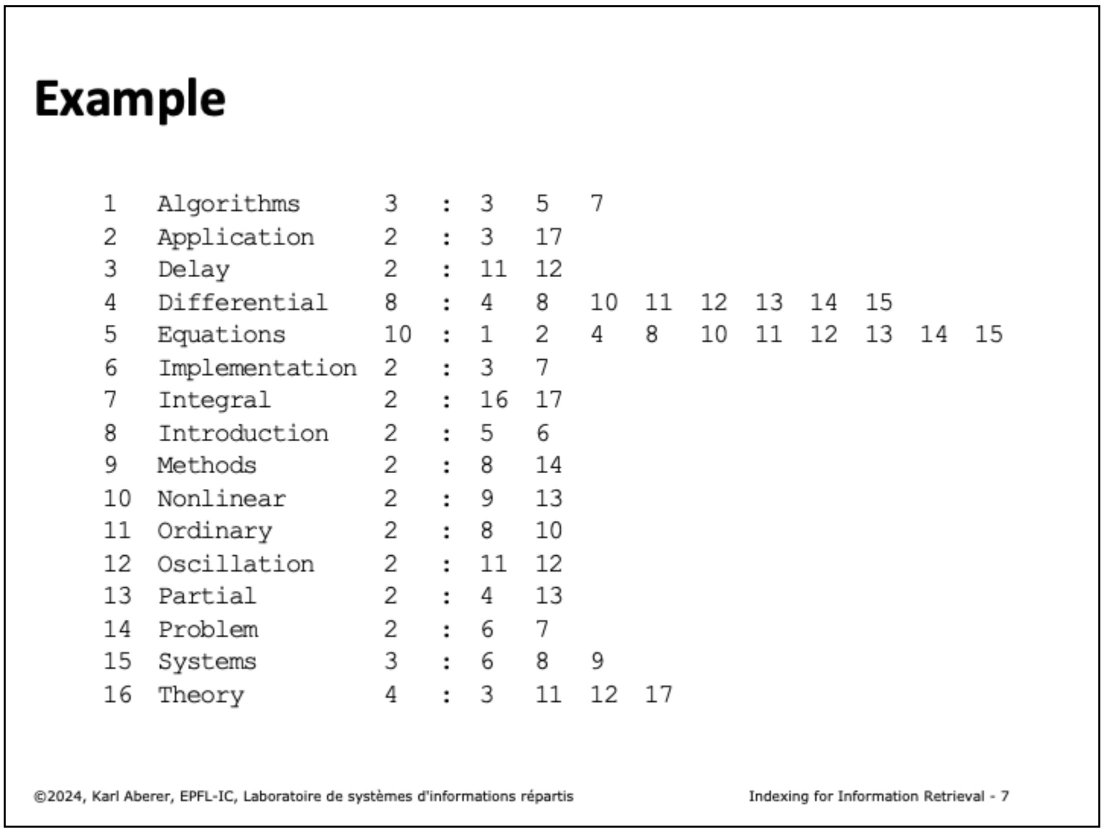
Storing the global frequency f_k is useful for determining inverse document frequency.



Example: Documents

B1 A Course on Integral Equations
B2 Attractors for Semigroups and Evolution Equations
B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4 Geometrical Aspects of Partial Differential Equations
B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6 Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7 Knapsack Problems: Algorithms and Computer Implementations
B8 Methods of Solving Singular Systems of Ordinary Differential Equations
B9 Nonlinear Systems
B10 Ordinary Differential Equations
B11 Oscillation Theory for Neutral Differential Equations with Delay
B12 Oscillation Theory of Delay Differential Equations
B13 Pseudodifferential Operators and Nonlinear Partial Differential Equations
B14 Sinc Methods for Quadrature and Differential Equations
B15 Stability of Stochastic Differential Equations with Respect to Semi-Martingales
B16 The Boundary Integral Approach to Static and Dynamic Contact Problems
B17 The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory

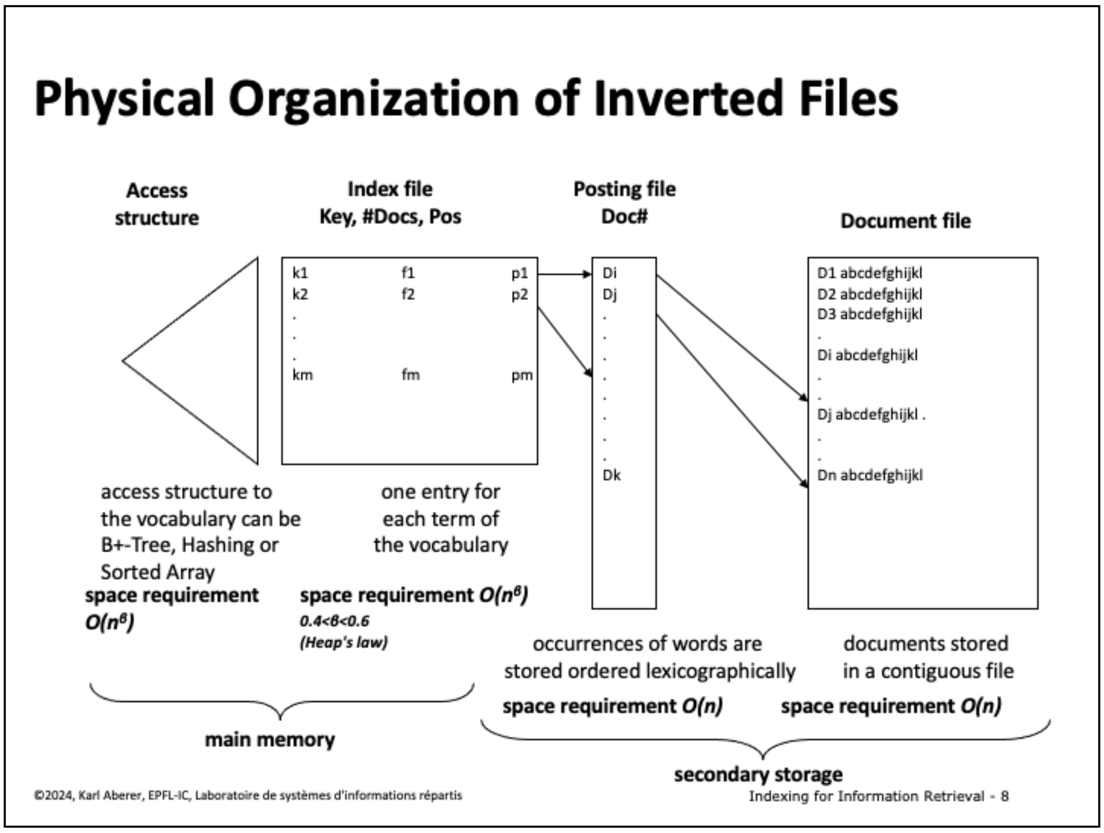
This is an example of a (simple) document collection that we will use in the following as running example.



Example

1	Algorithms	3	:	3 5 7
2	Application	2	:	3 17
3	Delay	2	:	11 12
4	Differential	8	:	4 8 10 11 12 13 14 15
5	Equations	10	:	1 2 4 8 10 11 12 13 14 15
6	Implementation	2	:	3 7
7	Integral	2	:	16 17
8	Introduction	2	:	5 6
9	Methods	2	:	8 14
10	Nonlinear	2	:	9 13
11	Ordinary	2	:	8 10
12	Oscillation	2	:	11 12
13	Partial	2	:	4 13
14	Problem	2	:	6 7
15	Systems	3	:	6 8 9
16	Theory	4	:	3 11 12 17

Here we show the inverted lists that are obtained for our example document collection. The concatenation of those lists constitutes the inverted file.



Physical Organization of Inverted Files

access structure to the vocabulary can be B+-Tree, Hashing or Sorted Array

space requirement *$O(n^\beta)$*

one entry for each term of the vocabulary

space requirement *$O(n^\beta)$*

$0.4<\beta<0.6$ (Heap's law)

occurrences of words are stored ordered lexicographically

space requirement *$O(n)$*

documents stored in a contiguous file

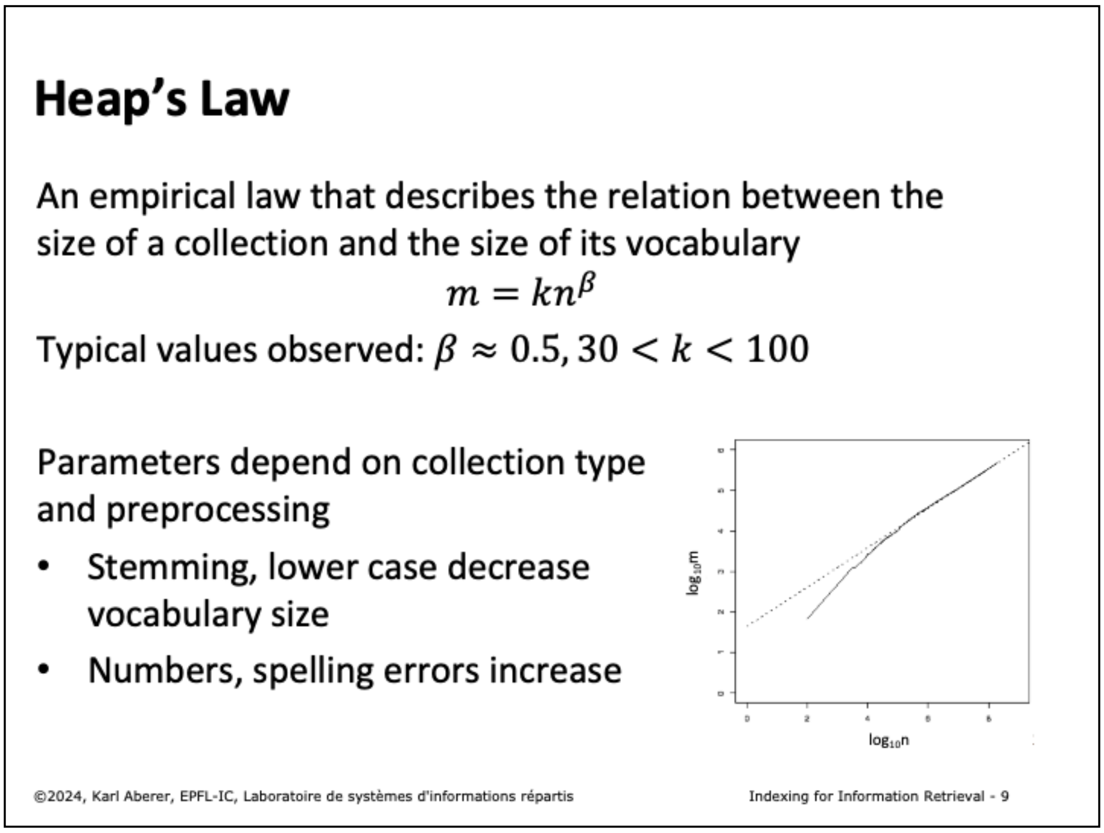
space requirement *$O(n)$*

main memory

secondary storage

Inverted files are a logical data structure, for which a physical storage organization needs to be designed.

The physical organization must consider the quantitative characteristics of the inverted file structure. A key observation is that the number of references to documents, corresponding to the occurrences of index terms in the documents is much larger than the number of index terms, and thus the number of inverted lists. As the number of index terms is much smaller, the index terms and the corresponding frequencies of occurrences can be kept in main memory, whereas the references to documents are kept in secondary storage. The access to the index file a data access structure is used. Examples of data access structures are binary search, hash tables or tree-based structures, such as B+-Trees or tries. The posting files consist of the sequence of all term occurrences of the inverted file. The index file is referring to the posting file by keeping for each index term a reference to the corresponding position in the posting file, at which the entries related to the index terms start. The occurrences stored in the posting file in turn refer to entries in the document file, which is also kept in secondary storage.



Heap's Law

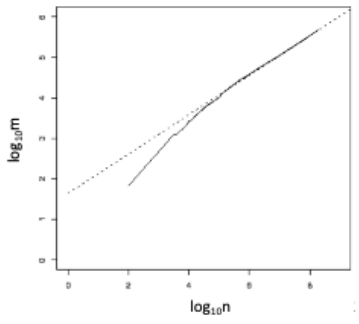
An empirical law that describes the relation between the size of a collection and the size of its vocabulary

$$m = kn^{\beta}$$

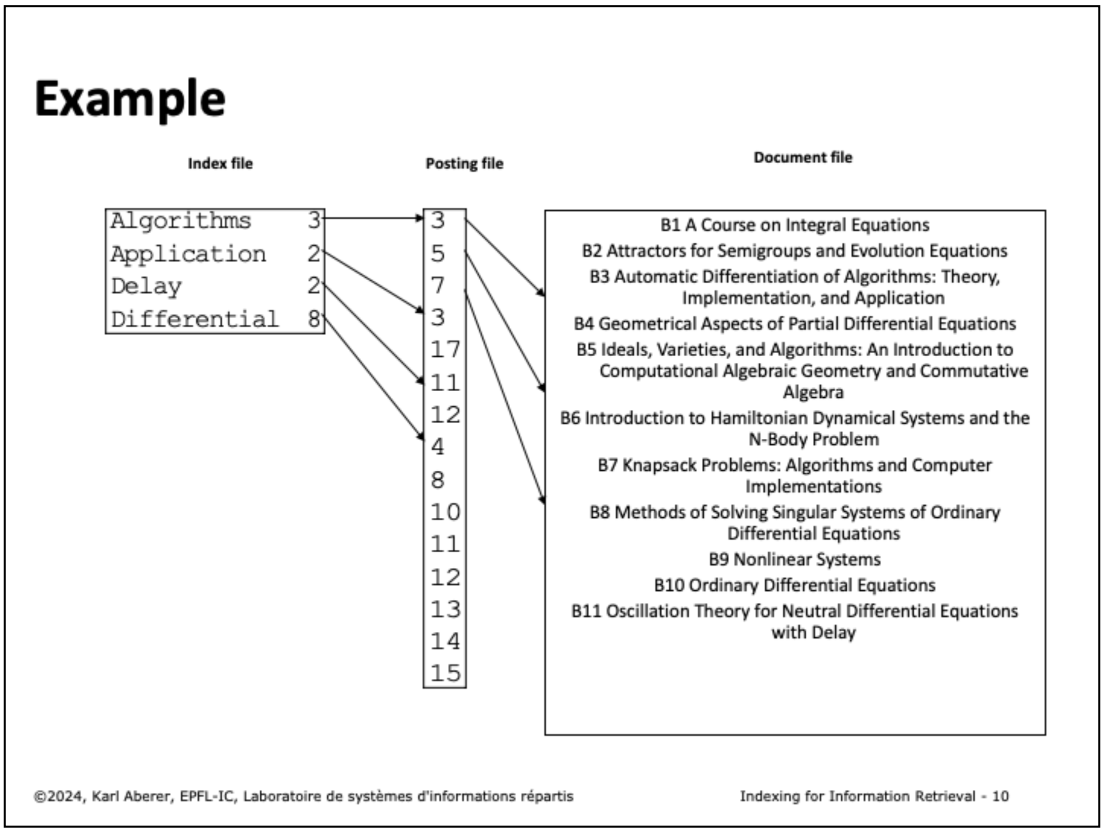
Typical values observed: $\beta \approx 0.5, 30 < k < 100$

Parameters depend on collection type and preprocessing

- Stemming, lower case decrease vocabulary size
- Numbers, spelling errors increase



Empirical studies show that for a document collection of size n the number of different index terms is typically O(n^β), where β is roughly 0.5 (Heap's law). More precisely, the relationship can be described as k n^β, where typical values of k are 30 < k < 100. For example, a document collection of size n= 10^6 could have approximately m=100 * 10^3=10^5 index terms.



Example

Index file

Algorithms	3
Application	2
Delay	2
Differential	8

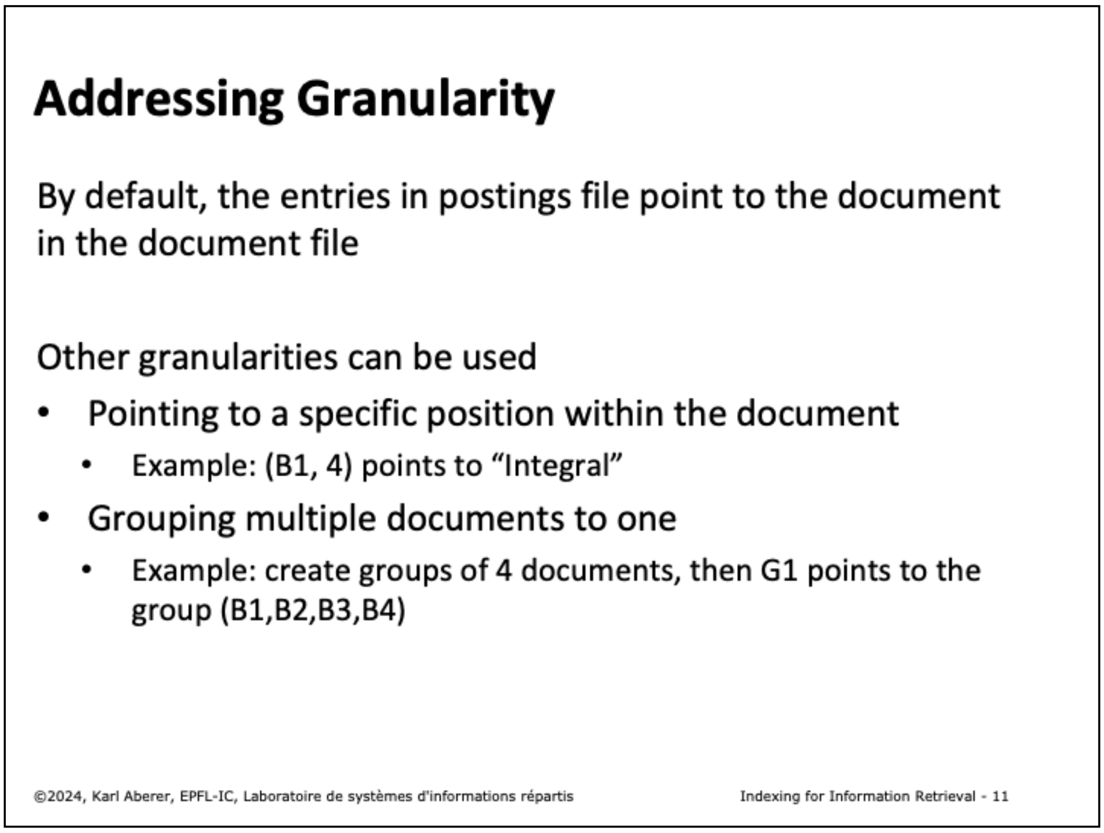
Posting file

```
3
5
7
3
17
11
12
4
8
10
11
12
13
14
15
```

Document file

B1 A Course on Integral Equations
B2 Attractors for Semigroups and Evolution Equations
B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4 Geometrical Aspects of Partial Differential Equations
B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6 Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7 Knapsack Problems: Algorithms and Computer Implementations
B8 Methods of Solving Singular Systems of Ordinary Differential Equations
B9 Nonlinear Systems
B10 Ordinary Differential Equations
B11 Oscillation Theory for Neutral Differential Equations with Delay

Here we illustrate the physical organization of the inverted file for the running example. Note that only part of the data in the document file is shown.



Addressing Granularity

By default, the entries in postings file point to the document in the document file

Other granularities can be used

- Pointing to a specific position within the document
  - Example: (B1, 4) points to “Integral”
- Grouping multiple documents to one
  - Example: create groups of 4 documents, then G1 points to the group (B1,B2,B3,B4)

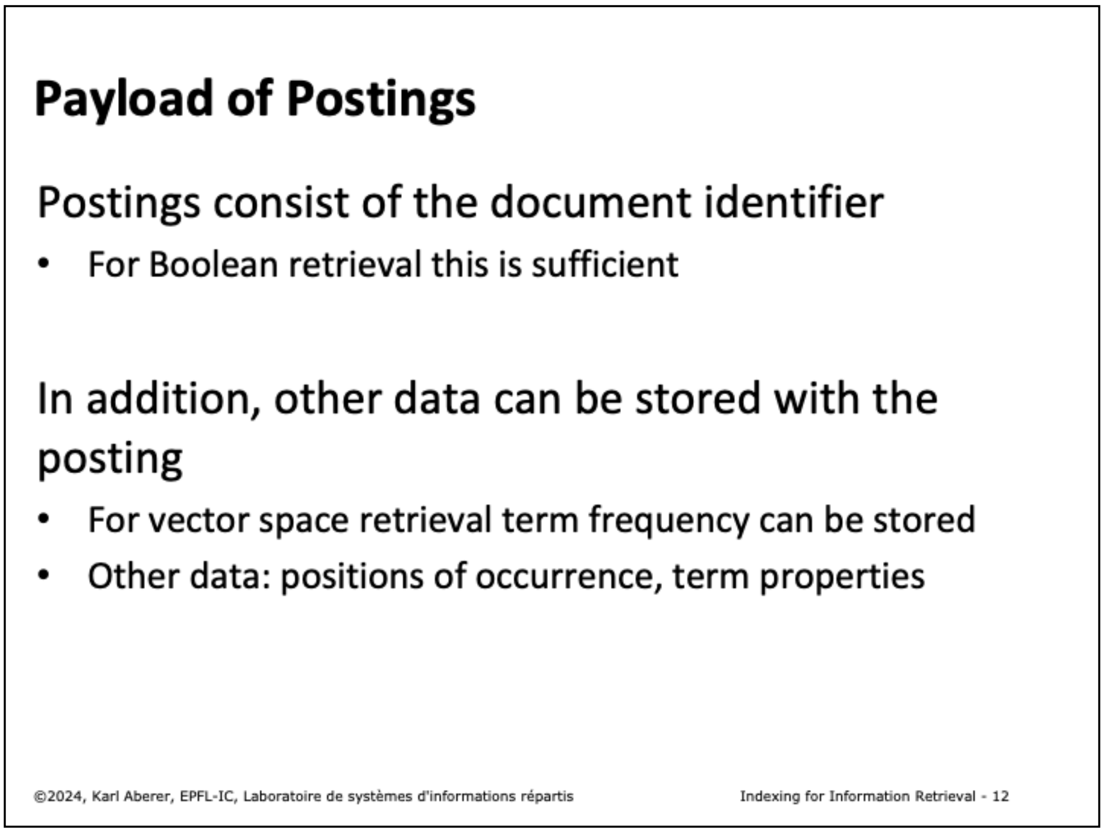
The addressing granularity determines of how positions of index terms are recorded in the posting file. There exist three main options:

• pointing to the start of the document in which the index term occurs (this is how we introduced inverted files)

• exact position of the index term within the document, or a sentence or paragraph containing the index term.

• grouping of multiple documents into blocks and pointing to the start of a block

The larger the granularity, the fewer entries occur in the posting file. In turn, with coarser granularity additional post-processing is required in order to determine exact positions of index terms within the documents.



Payload of Postings

Postings consist of the document identifier

- For Boolean retrieval this is sufficient

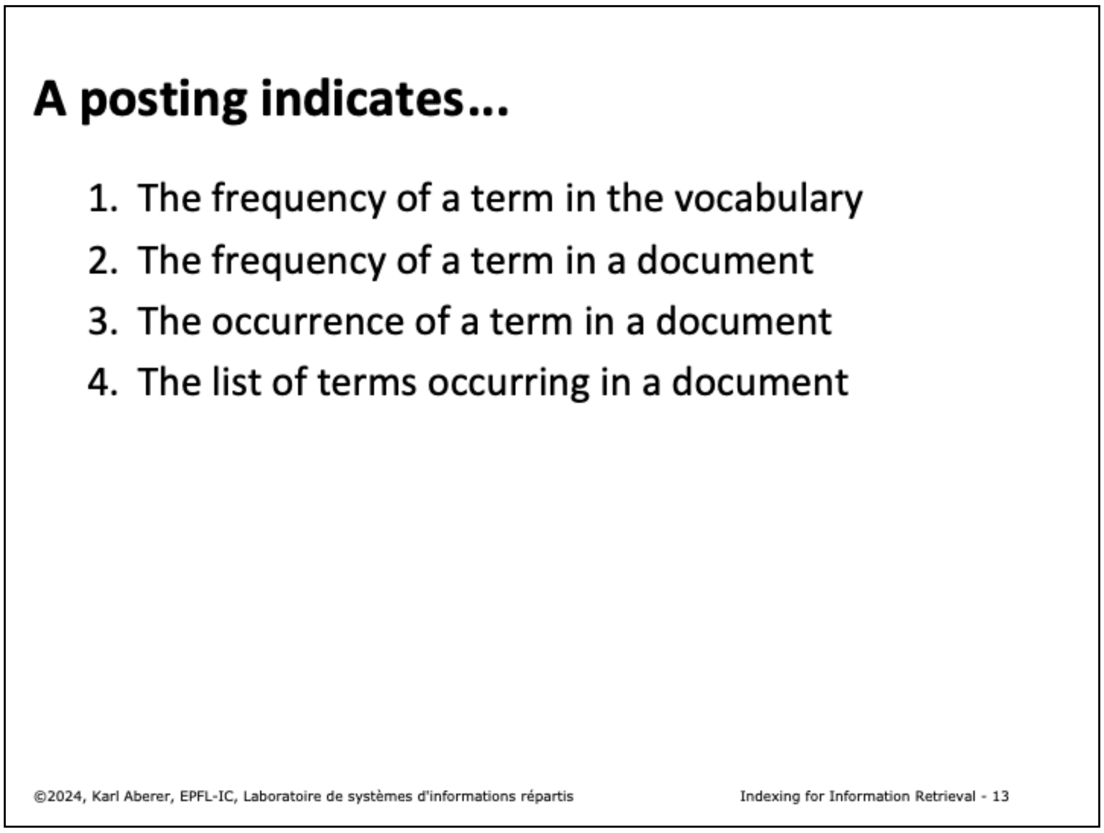
In addition, other data can be stored with the posting

- For vector space retrieval term frequency can be stored
- Other data: positions of occurrence, term properties

In the simplest case the postings file consists of the document identifiers. This is sufficient for implementing a Boolean retrieval system, since the only necessary infomration for query evalaution is on the presence or absence of a term in the document. Any additional information about the term needs to be computed on the fly from the document content.

To make retrieval more efficient, additional information about the term can be stored in the postings file. For vector space retrieval it is useful to also store the term frequency in order to avoid repeated scans of the document content, whenever the term frequency is requested.

Other information about terms related to the document that can be stored in a postings file is the positions of occurrence, the context in which the term occurs (e.g. in the title) or results from some further text processing (e.g. using natural language processing such as named entity recognition about which will learn later in this lecture).

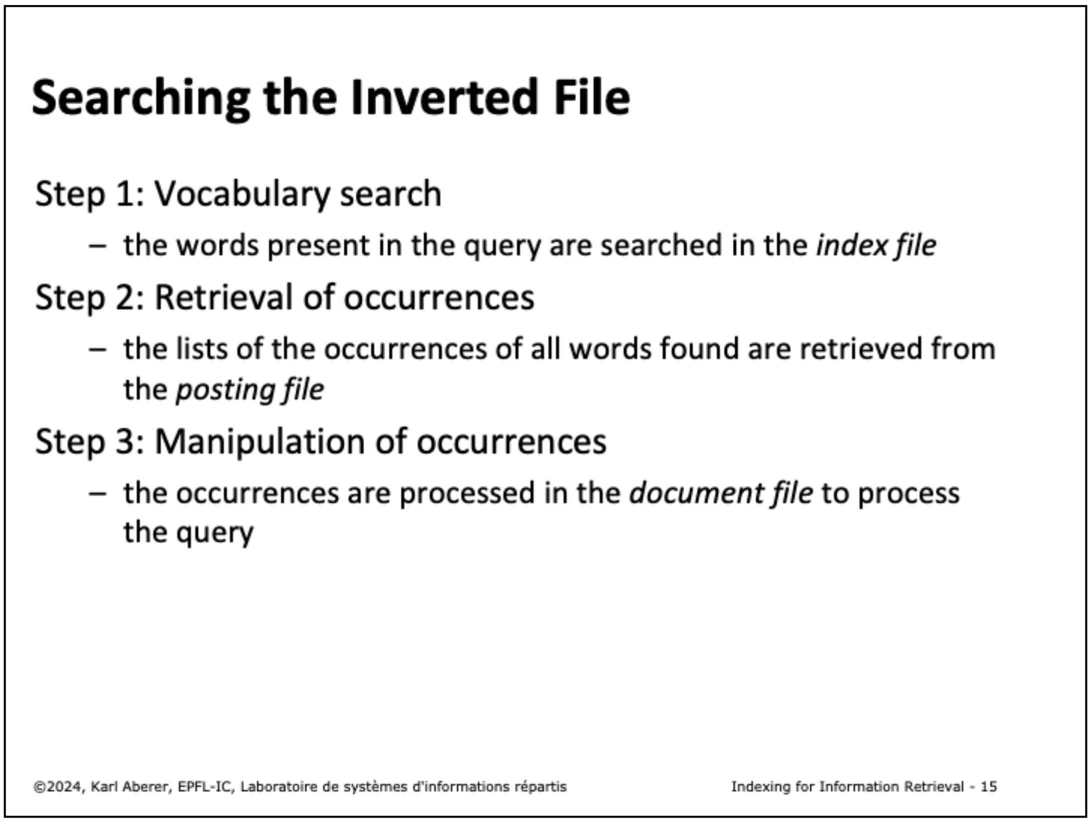


A posting indicates…

1. The frequency of a term in the vocabulary
2. The frequency of a term in a document
3. The occurrence of a term in a document
4. The list of terms occurring in a document

When indexing a document collection using an inverted file, the main space requirement is implied by ...

1. The access structure
2. The vocabulary
3. The index file
4. The postings file



Searching the Inverted File

Step 1: Vocabulary search

- the words present in the query are searched in the *index file*

Step 2: Retrieval of occurrences

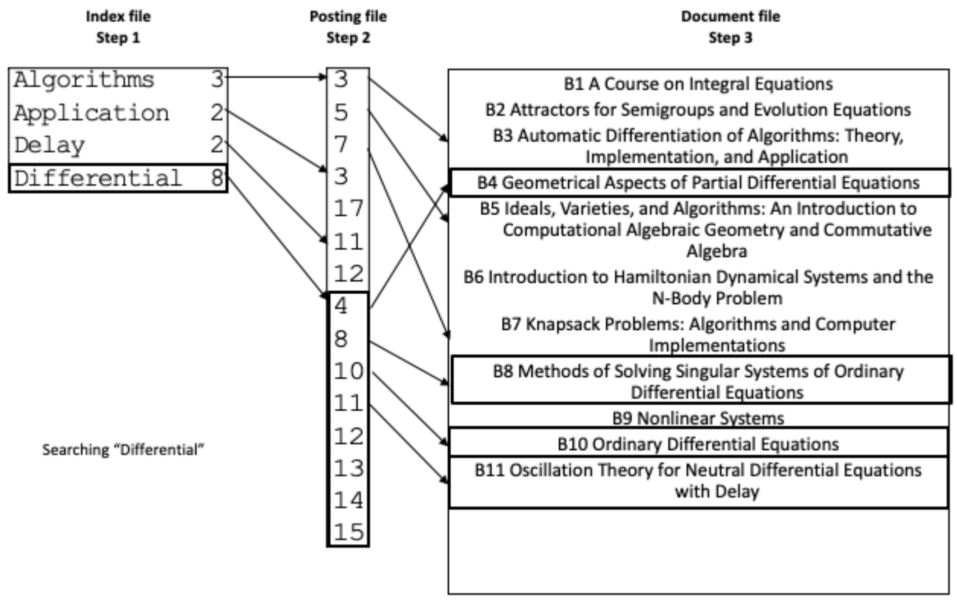
- the lists of the occurrences of all words found are retrieved from the *posting file*

Step 3: Manipulation of occurrences

- the occurrences are processed in the *document file* to process the query

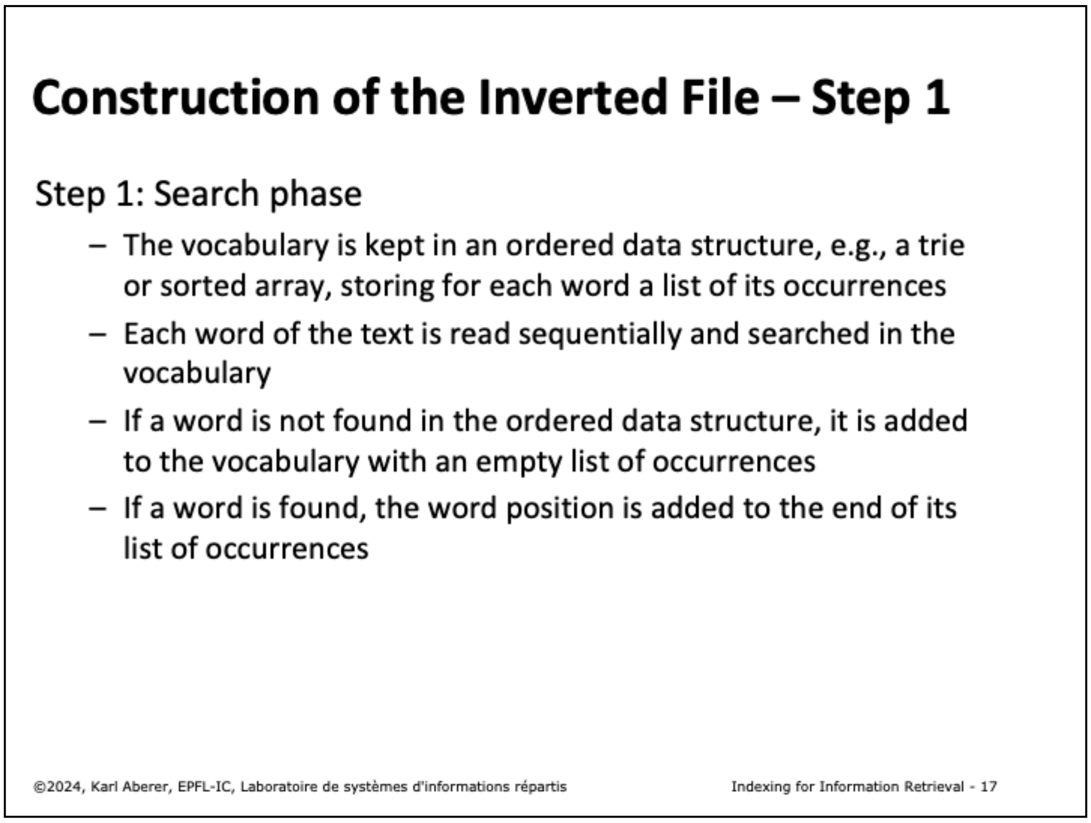
Search in an inverted file is a straightforward process. Using the data access structure, first the index terms occurring in the query are searched in the index file. Then the occurrences can be sequentially retrieved from the postings file. Afterwards the corresponding document portions are accessed and can be processed (e.g. for counting term frequencies).

Example



Index file
Step 1
Posting file
Step 2
Document file
Step 3
Algorithms 3
Application 2
Delay 2
Differential 8
3
5
7
3
17
11
12
4
8
10
11
12
13
14
15
B1 A Course on Integral Equations
B2 Attractors for Semigroups and Evolution Equations
B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4 Geometrical Aspects of Partial Differential Equations
B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6 Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7 Knapsack Problems: Algorithms and Computer Implementations
B8 Methods of Solving Singular Systems of Ordinary Differential Equations
B9 Nonlinear Systems
B10 Ordinary Differential Equations
B11 Oscillation Theory for Neutral Differential Equations with Delay
Searching "Differential"

Here we illustrate the the 3 steps of the search process.

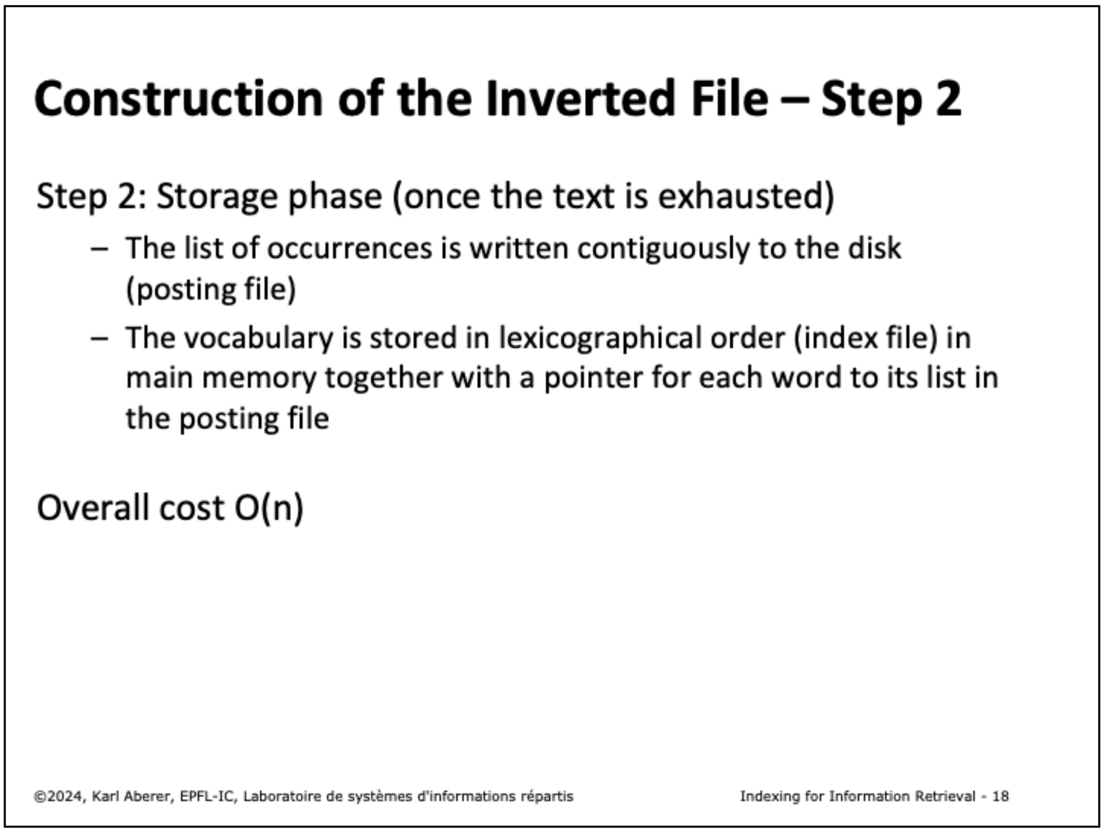


Construction of the Inverted File – Step 1

Step 1: Search phase

- The vocabulary is kept in an ordered data structure, e.g., a trie or sorted array, storing for each word a list of its occurrences
- Each word of the text is read sequentially and searched in the vocabulary
- If a word is not found in the ordered data structure, it is added to the vocabulary with an empty list of occurrences
- If a word is found, the word position is added to the end of its list of occurrences

The index construction is performed by first constructing dynamically an order data structure (we will use a trie structure), in order to generate a sorted vocabulary and to create a list of the occurrences of index terms in the text.



Construction of the Inverted File – Step 2

Step 2: Storage phase (once the text is exhausted)

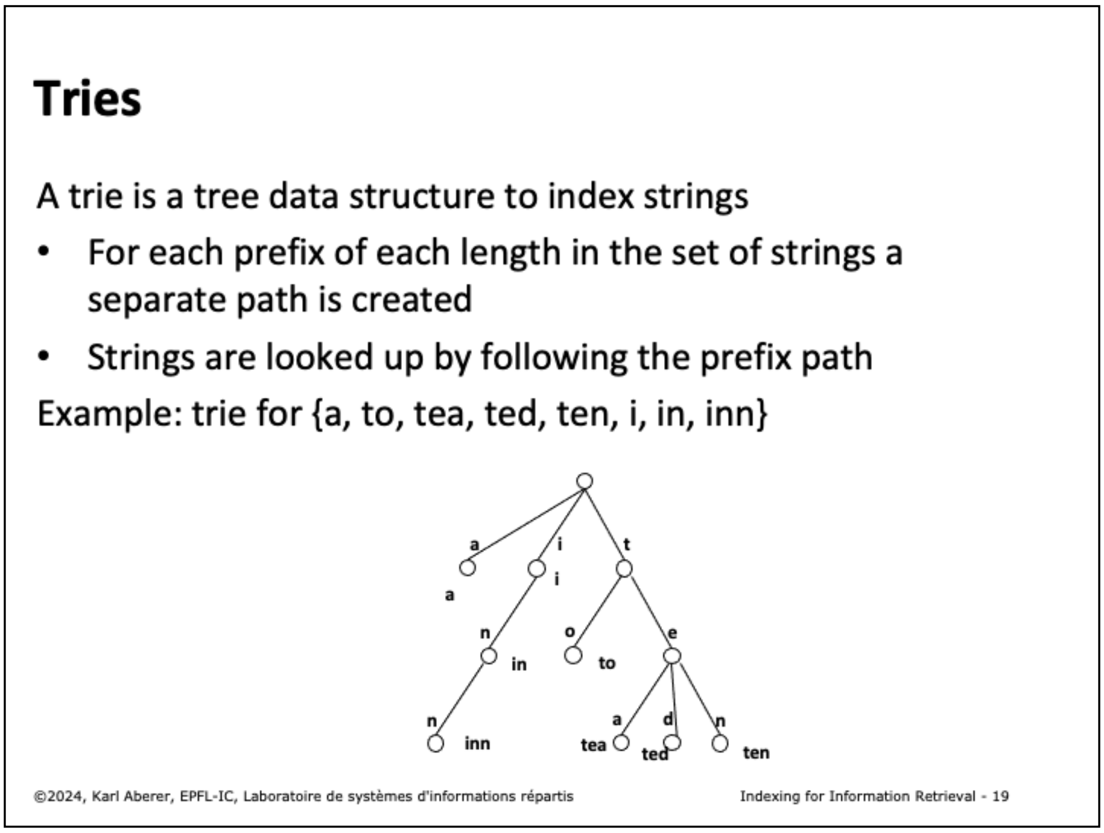
- The list of occurrences is written contiguously to the disk (posting file)
- The vocabulary is stored in lexicographical order (index file) in main memory together with a pointer for each word to its list in the posting file

Overall cost O(n)

©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

Indexing for Information Retrieval - 18

After the complete document collection has been traversed, the ordered data structure is sequentially traversed, and the posting file is written to secondary storage. The ordered data structure itself can be used as a data access structure for the index file that is kept in main memory.

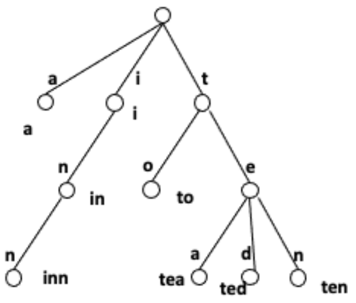


Tries

A trie is a tree data structure to index strings

- For each prefix of each length in the set of strings a separate path is created
- Strings are looked up by following the prefix path

Example: trie for {a, to, tea, ted, ten, i, in, inn}



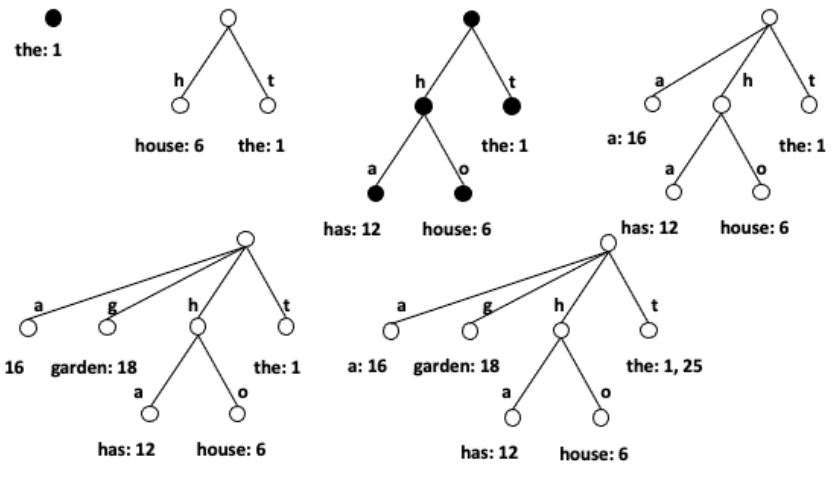
In the following we will us tries as ordered data structure. The trie data structure is a so - called lexicographical index structure specifically suited for text processing. Variations of this data structure, specifically PAT trees can be used for efficient text processing including evaluation of regular expressions.

Example

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

the house has a garden. the garden has many flowers. the flowers are beautiful

(each word = one document, position = document identifier)

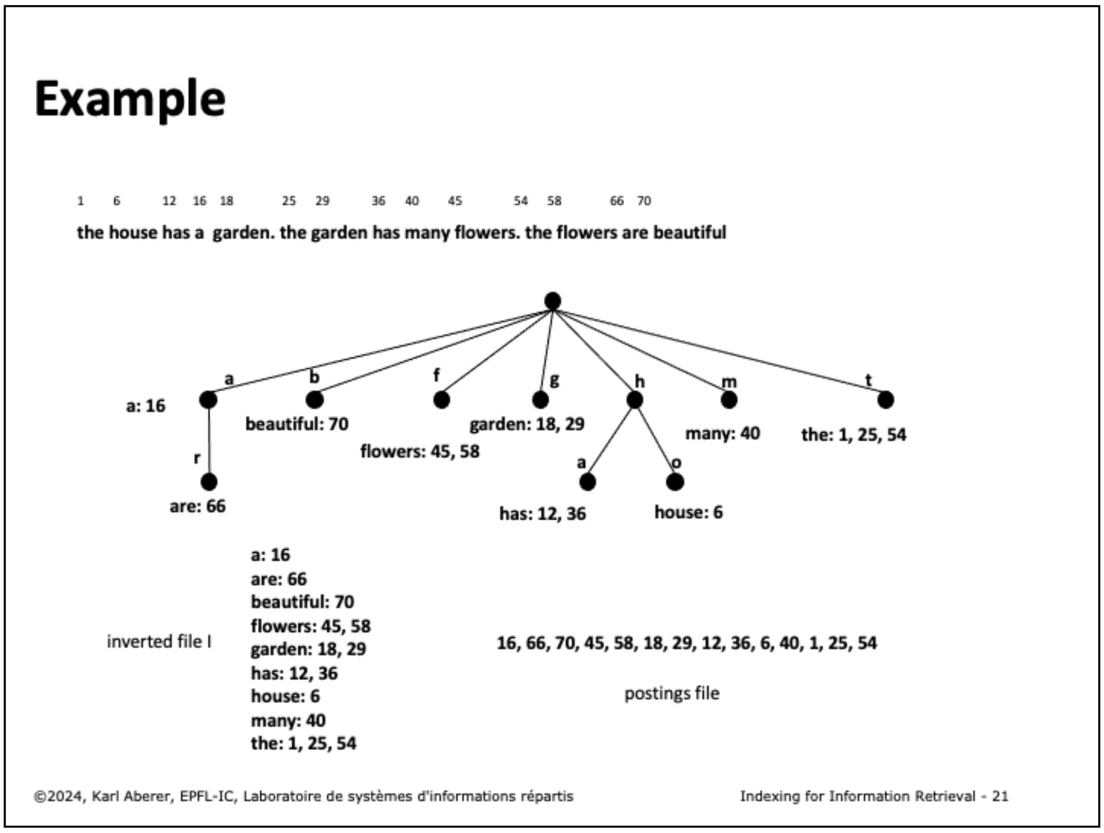


©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

Indexing for Information Retrieval - 20

In this example we use word positions as addressing granularity for the postings file, since we have only a single document.

We demonstrate the initial steps of constructing the trie structure and adding to it the occurrences of index terms. The changes to the trie structure are highlighted for each step. Note that in the last step the tree structure of the trie does not change, since the index term "the" is already present.



Example

```
1    6     12  16  18        25   29       36   40    45         54   58        66   70
the house has a  garden. the garden has many flowers. the flowers are beautiful
```

inverted file I

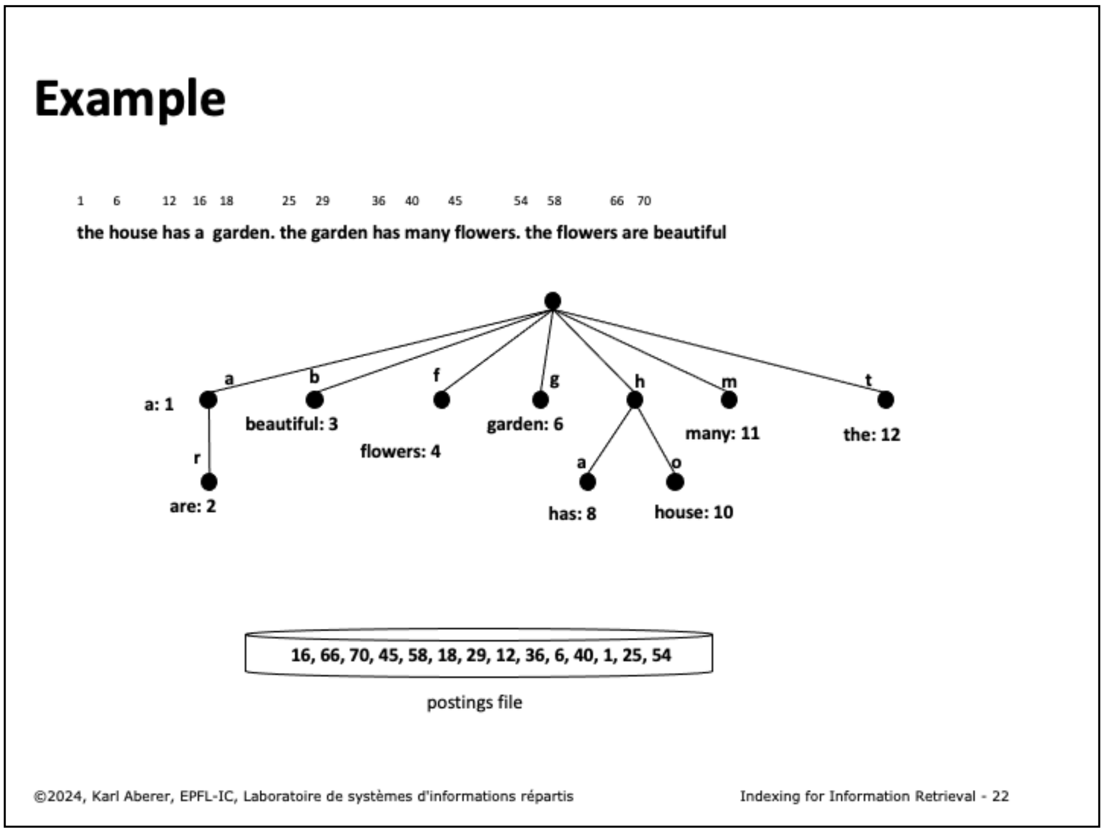
a: 16
are: 66
beautiful: 70
flowers: 45, 58
garden: 18, 29
has: 12, 36
house: 6
many: 40
the: 1, 25, 54

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

postings file

©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis Indexing for Information Retrieval - 21

Once the complete trie structure is constructed the inverted file can be derived from it. For doing this, the trie is traversed top-down and left-to-right. Whenever an index term is encountered it is added at the end of the inverted file. Note that if a term is prefix of another term (such as "a" is prefix of "are") index terms can occur on internal nodes of the trie. After to the construction of the inverted file the posting file can be extracted from it.

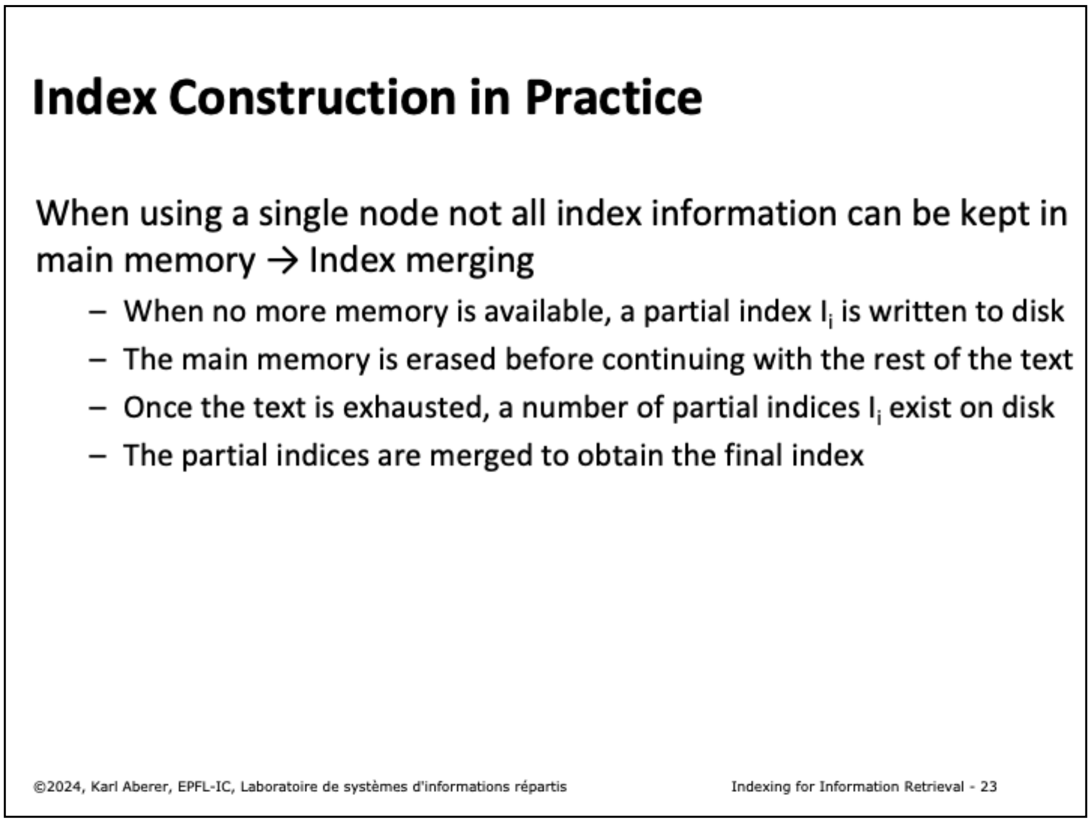


Example

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

the house has a garden. the garden has many flowers. the flowers are beautiful

The resulting physical organization of the inverted file is shown here. The trie structure can be used as an access structure to the index file in main memory. Thus, the entries of the index files occur as leaves (or internal nodes) of the trie. Each entry has a reference to the position of the postings file that is kept in secondary storage.



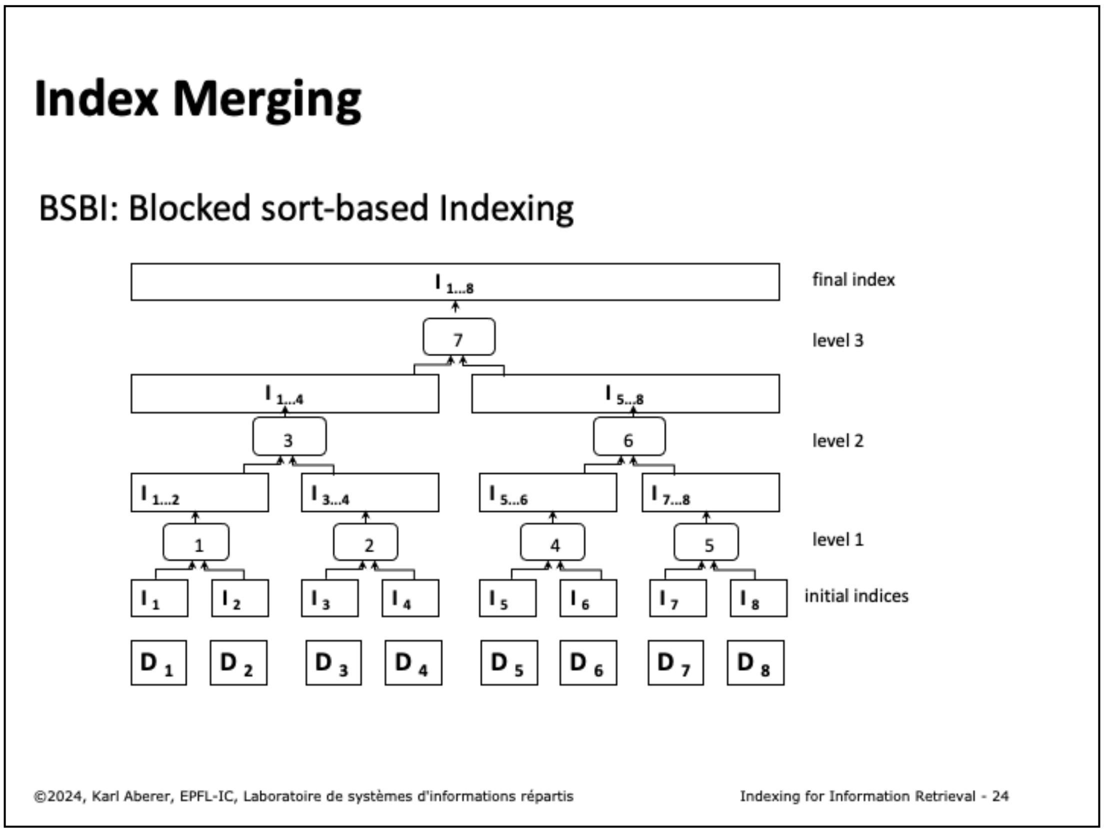
Index Construction in Practice

When using a single node not all index information can be kept in main memory → Index merging

- When no more memory is available, a partial index I_i is written to disk
- The main memory is erased before continuing with the rest of the text
- Once the text is exhausted, a number of partial indices I_i exist on disk
- The partial indices are merged to obtain the final index

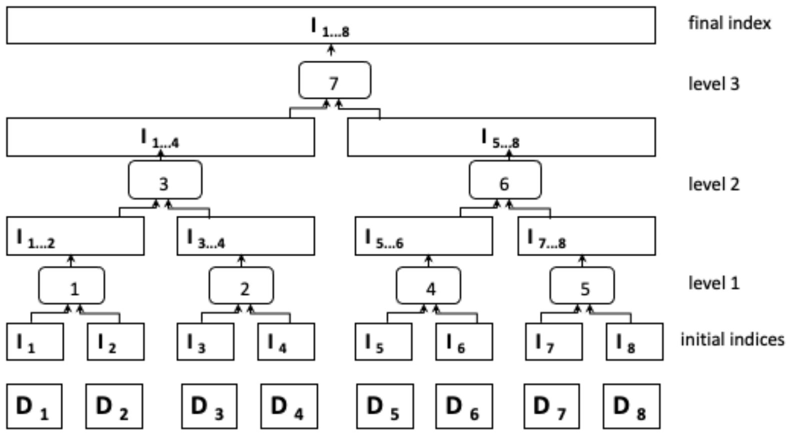
In practice, the available main memory needed to keep the inverted file during index construction is insufficient in size, since the storage space consumed by the postings is proportional to the size of the document collection.

In this case, the index construction process needs be partitioned into smaller steps: in a first phase, the document collection is sequentially traversed, and partial indices are written to the disk whenever the main memory becomes full. This results in a set of partial indices, indexing consecutive partitions of the text. In a second phase the partial indices need to be merged into one index.

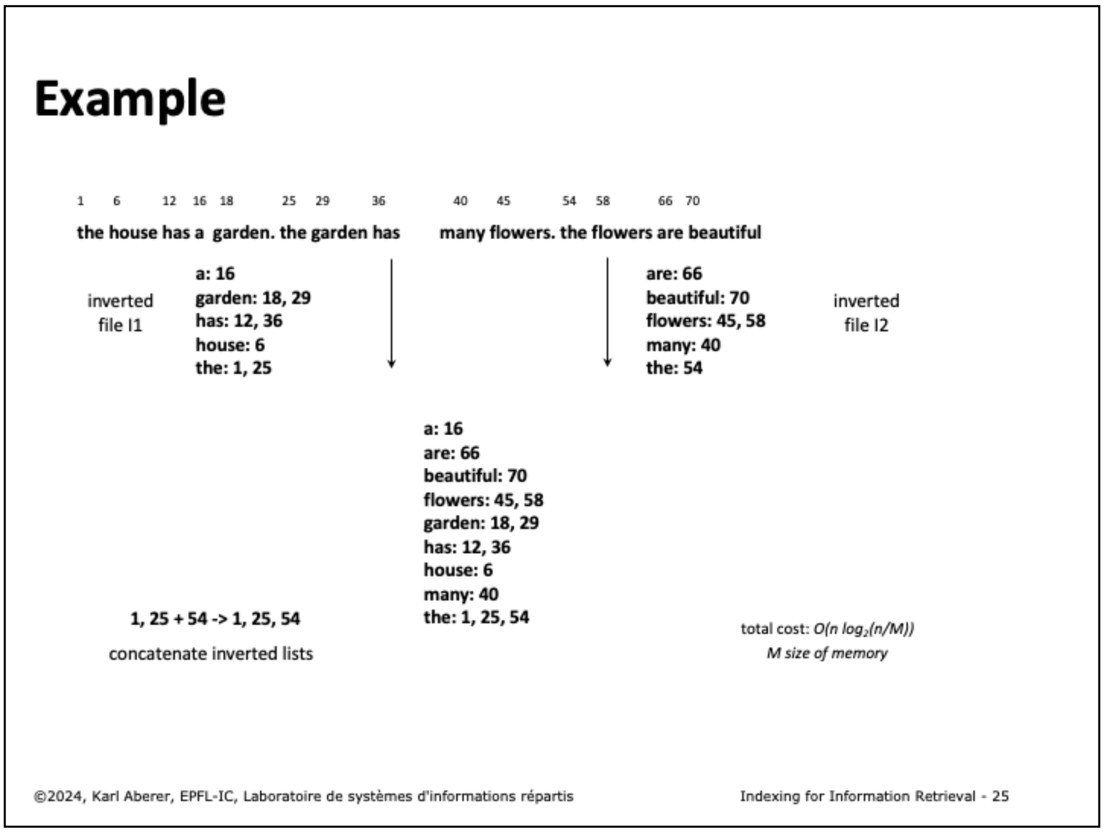


Index Merging

BSBI: Blocked sort-based Indexing



This figure illustrates the merging process: 8 partial indices have been constructed. Step by step the indices are merged, by merging two indices into one, until one final index remains. The merging can be performed, such that the two partial indices which are to be merged are in parallel scanned on the disk, and while scanning the resulting index is written sequentially to the disk.



Example

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

the house has a garden. the garden has many flowers. the flowers are beautiful

inverted file I1

a: 16
garden: 18, 29
has: 12, 36
house: 6
the: 1, 25

inverted file I2

are: 66
beautiful: 70
flowers: 45, 58
many: 40
the: 54

a: 16
are: 66
beautiful: 70
flowers: 45, 58
garden: 18, 29
has: 12, 36
house: 6
many: 40
the: 1, 25, 54

1, 25 + 54 -> 1, 25, 54

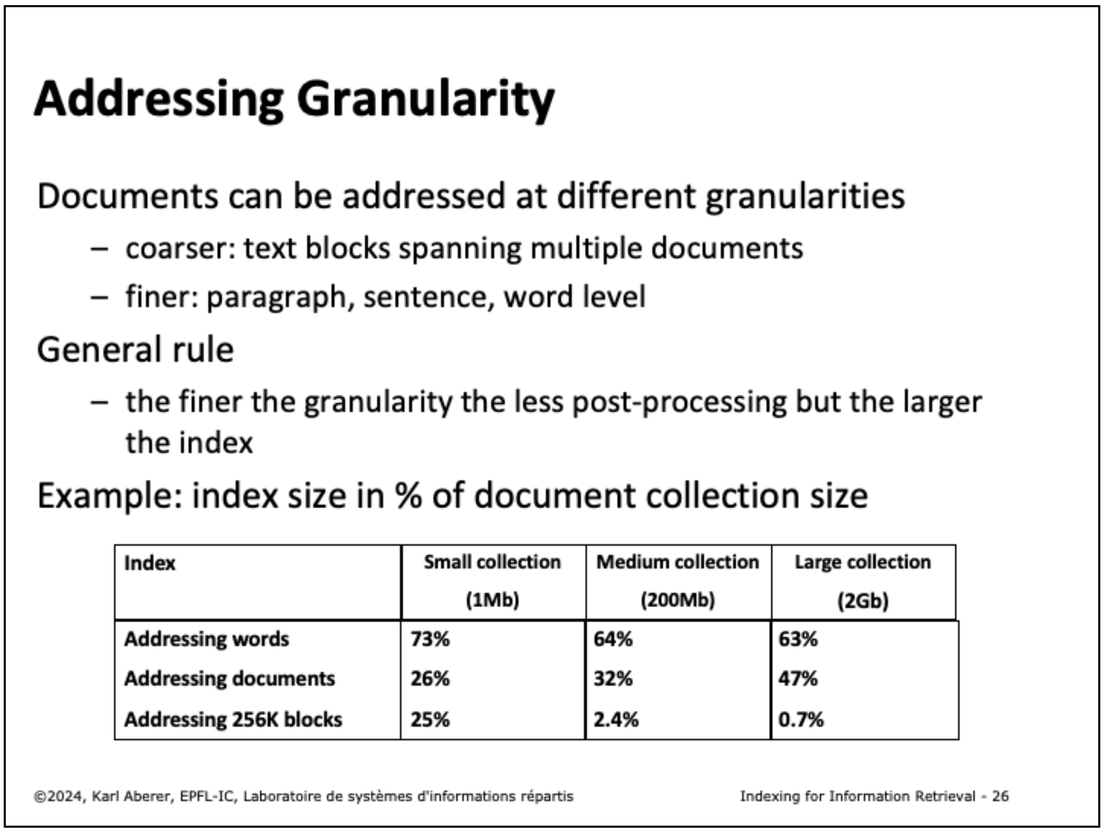
concatenate inverted lists

total cost: *O(n log_2(n/M))*
M size of memory

Merging the indices requires first merging the vocabularies. As we mentioned earlier, the vocabularies are comparably small and thus the merging of the vocabularies can take place in main memory. In case a vocabulary term occurs in both partial indices, their list of occurrences from the posting file need to be combined. Here we can take advantage of the fact that the partial indices have been constructed by sequentially traversing the document file. Therefore, these lists can be directly concatenated without sorting.

The computational complexity of the merging algorithm is *$O(n \log_2(n/M))$.* This implies that the additional cost of merging as compared to the purely main-memory based construction of inverted files is a factor of $O(\log_2(n/M)))$. This is small in practice, e.g., if the database size n is 64 times larger than the main memory size, then this factor would be 6.

This example illustrates how the merging process can be performed when the database is split into two partitions.



Addressing Granularity

Documents can be addressed at different granularities

- coarser: text blocks spanning multiple documents
- finer: paragraph, sentence, word level

General rule

- the finer the granularity the less post-processing but the larger the index

Example: index size in % of document collection size

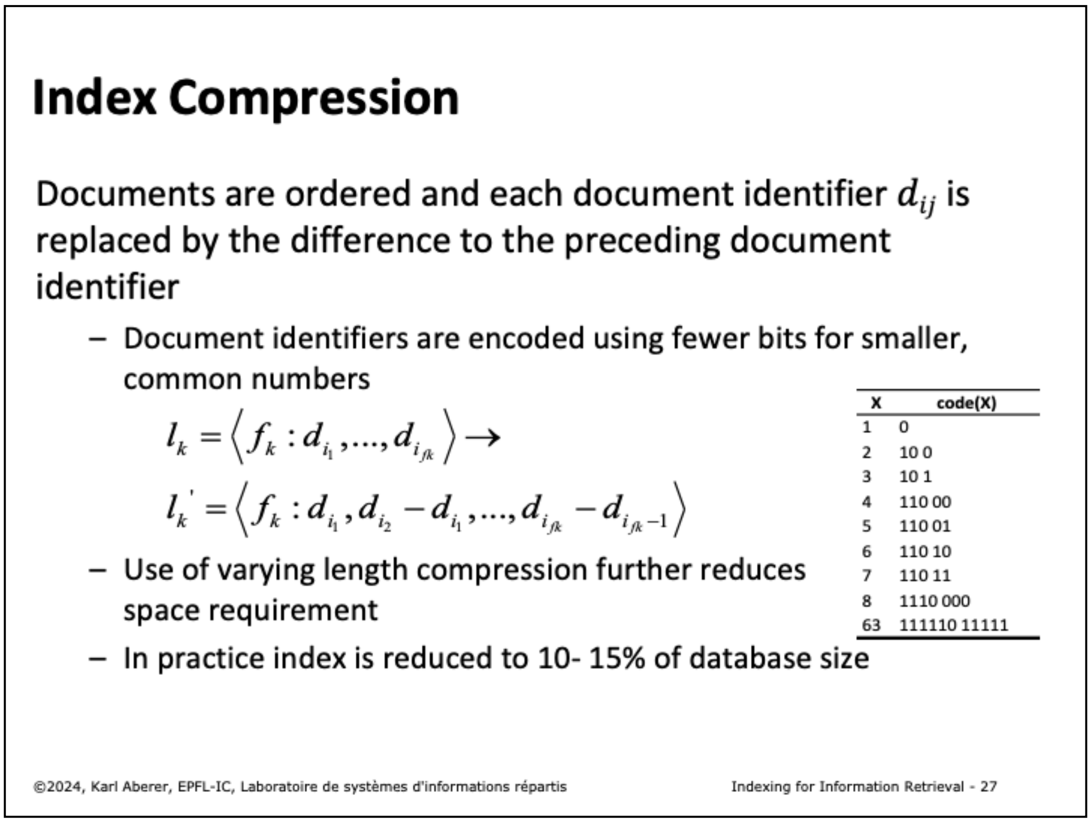
Index	Small collection (1Mb)	Medium collection (200Mb)	Large collection (2Gb)
Addressing words	73%	64%	63%
Addressing documents	26%	32%	47%
Addressing 256K blocks	25%	2.4%	0.7%

The posting file has the by far largest space requirements. An important factor determining the size of an inverted file is the addressing granularity used.

Experiments illustrate the substantial gains that can be obtained with coarser addressing granularities. Coarser granularities lead to a reduction of the index size for two reasons:

- a reduction in pointer size (e.g., from 4 Bytes for word addressing to 1 Byte with block addressing)
- and a lower number of occurrences.

Note that in the example for a 2GB document collection with 256K block addressing the index size is reduced by a factor of almost 100.



Index Compression

Documents are ordered and each document identifier d_{ij} is replaced by the difference to the preceding document identifier

- Document identifiers are encoded using fewer bits for smaller, common numbers

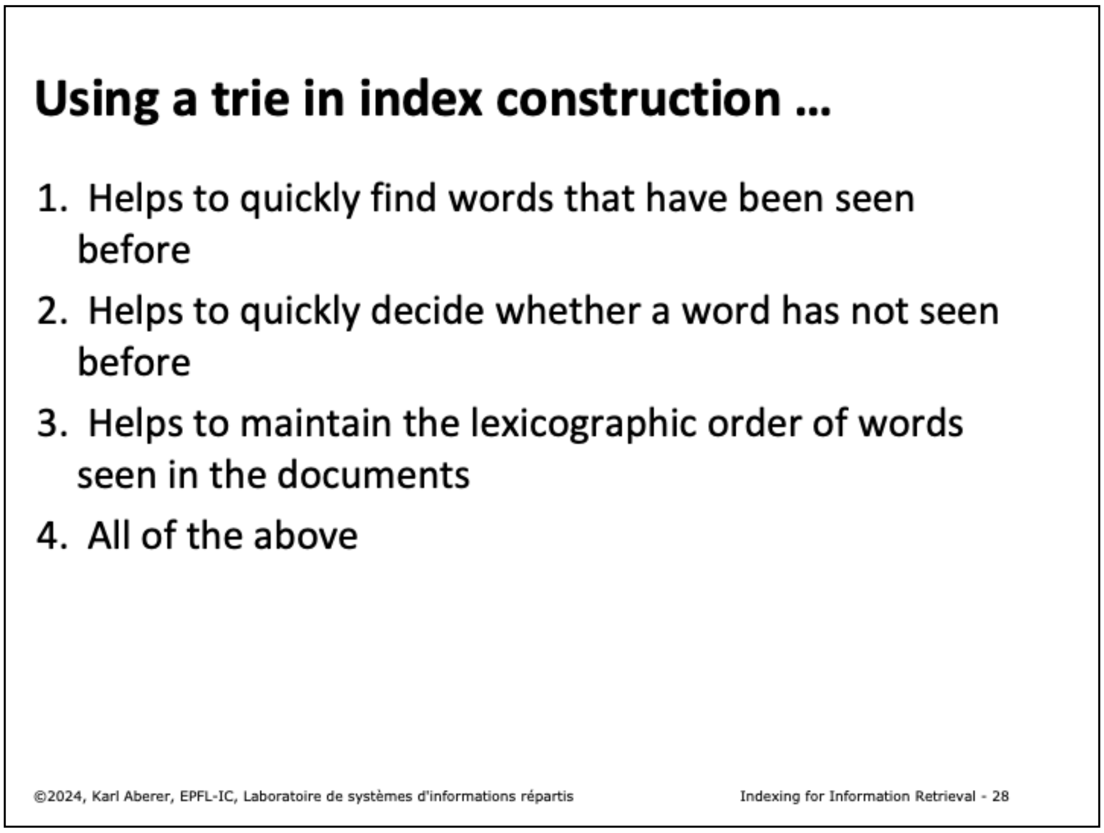
$$l_k = \left\langle f_k : d_{i_1}, \ldots, d_{i_{fk}} \right\rangle \rightarrow$$
$$l_k' = \left\langle f_k : d_{i_1}, d_{i_2} - d_{i_1}, \ldots, d_{i_{fk}} - d_{i_{fk}-1} \right\rangle$$

- Use of varying length compression further reduces space requirement
- In practice index is reduced to 10- 15% of database size

X	code(X)
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

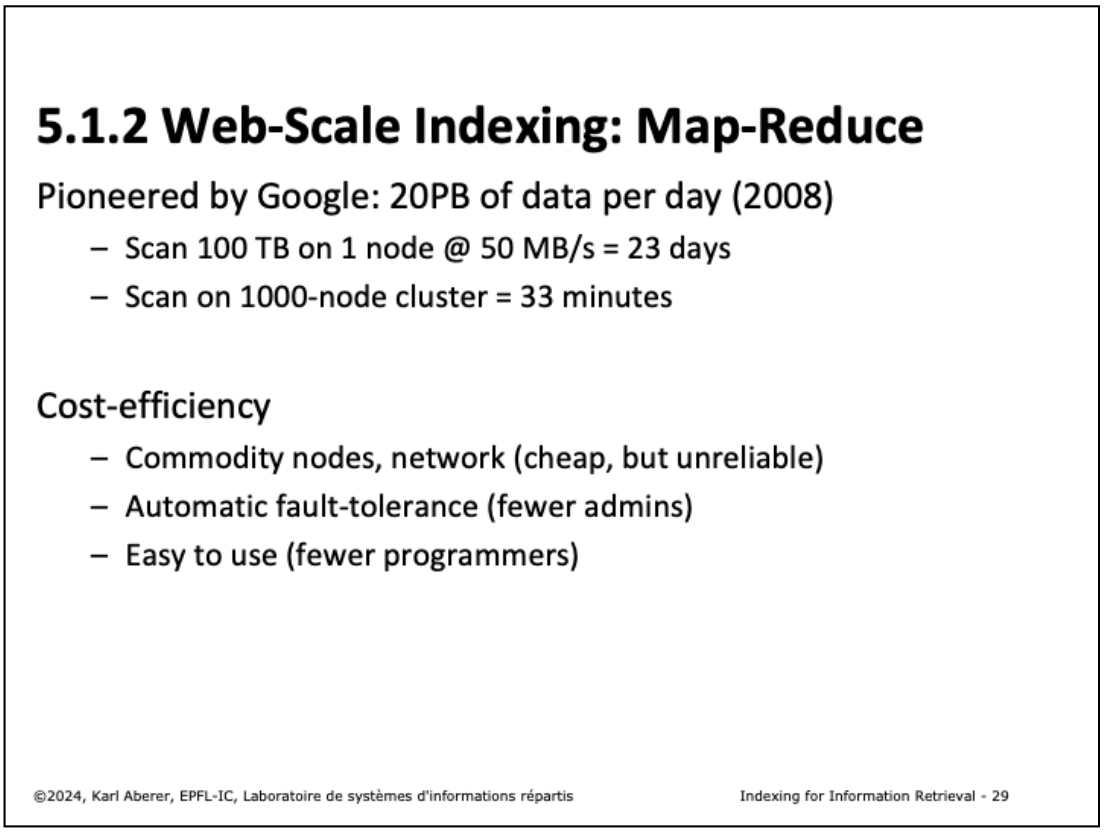
A further reduction of the index size can be achieved by applying compression techniques to the inverted lists. In practice, the inverted list of a single term can be rather long. A first improvement is achieved by storing only differences among subsequent document identifiers. Since they occur in sequential order, the differences are much smaller integers than the absolute position identifiers.

In addition, number encoding techniques can be applied to the resulting integer values. Since small values will be more frequent than large ones this leads to a further reduction in the size of the posting file.



Using a trie in index construction …

1. Helps to quickly find words that have been seen before
2. Helps to quickly decide whether a word has not seen before
3. Helps to maintain the lexicographic order of words seen in the documents
4. All of the above



5.1.2 Web-Scale Indexing: Map-Reduce

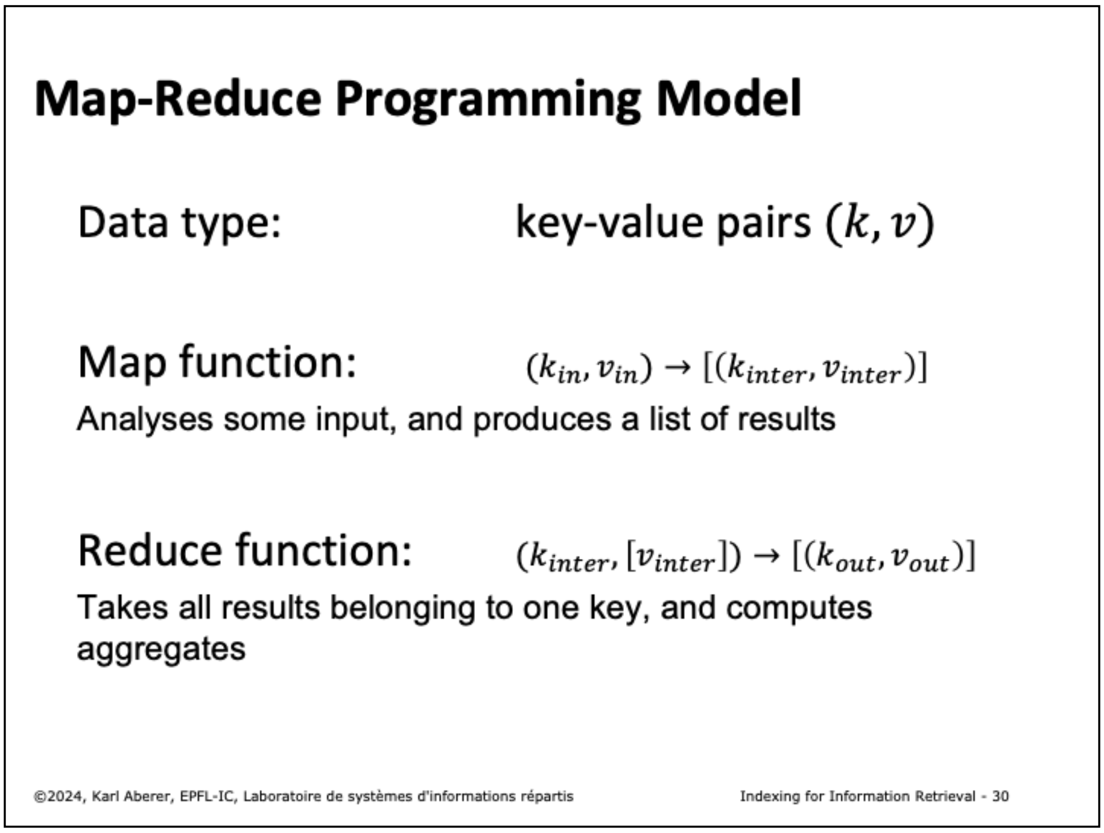
Pioneered by Google: 20PB of data per day (2008)

- Scan 100 TB on 1 node @ 50 MB/s = 23 days
- Scan on 1000-node cluster = 33 minutes

Cost-efficiency

- Commodity nodes, network (cheap, but unreliable)
- Automatic fault-tolerance (fewer admins)
- Easy to use (fewer programmers)

For Web-scale document collections traditional methods of index construction are not feasible. Therefore Google developed new approaches in terms of infrastructure and computing model to index very large document collections. A key element is the map-reduce programming model. It allows to parallelize index construction, within an infrastructure using potentially unreliable commodity hardware. The map-reduce programming model has been key in the ability of Google and later other Web platforms to scale up their applications. It actually led to a novel distributed programming paradigm and systems approach, that is tuned towards cost-efficiency and simplicity of programming.



Map-Reduce Programming Model

Data type: key-value pairs (k, v)

Map function: $(k_{in}, v_{in}) \rightarrow [(k_{inter}, v_{inter})]$

Analyses some input, and produces a list of results

Reduce function: $(k_{inter}, [v_{inter}]) \rightarrow [(k_{out}, v_{out})]$

Takes all results belonging to one key, and computes aggregates

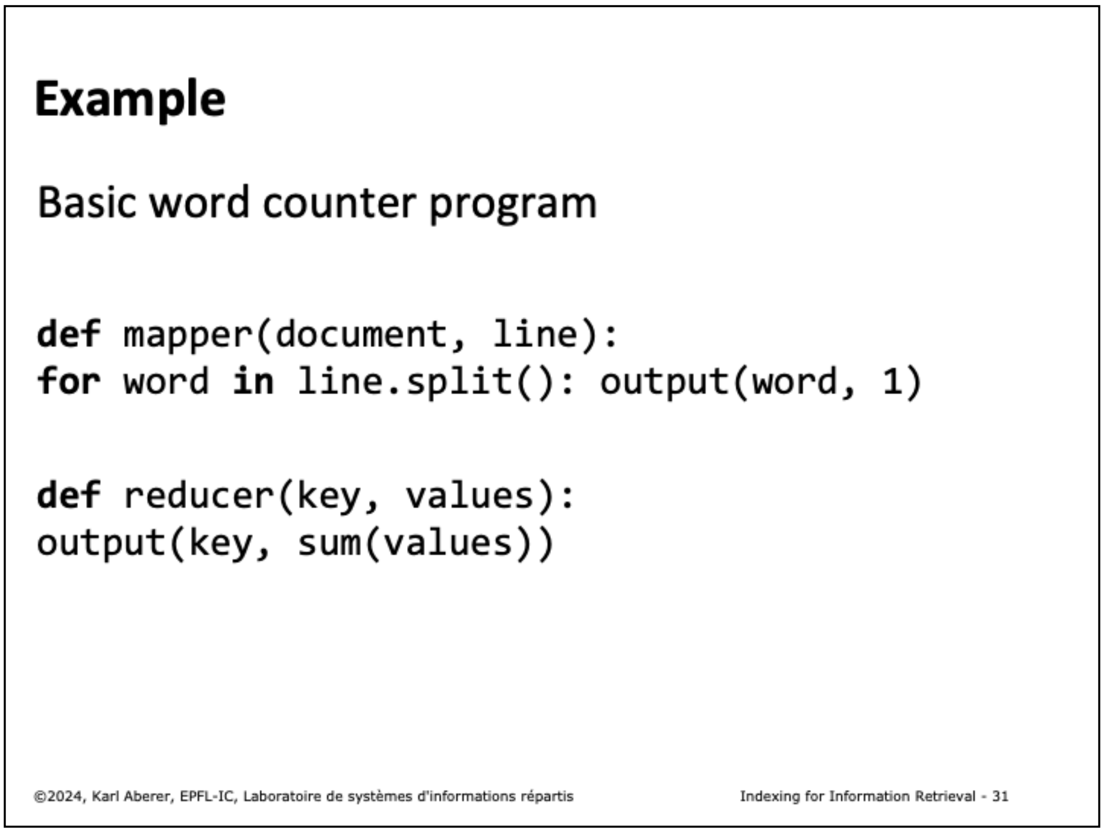
©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

Indexing for Information Retrieval - 30

The map-reduce programming model is based on manipulating key-value pairs (k, v) and lists of key value pairs $[(k, v)]$. It is based on two functions.

The map function receives some input data (typically a piece of text to analyze or index) and produces a list of key-value pairs. They constitute a partial results of the analysis (e.g., the counts of words in the text).

The reduce function receives as input all results for a given key value, that have been computed by different mapper functions. It computes then an aggregate output value (e.g., the total count of words in the document corpus).


Example

Basic word counter program

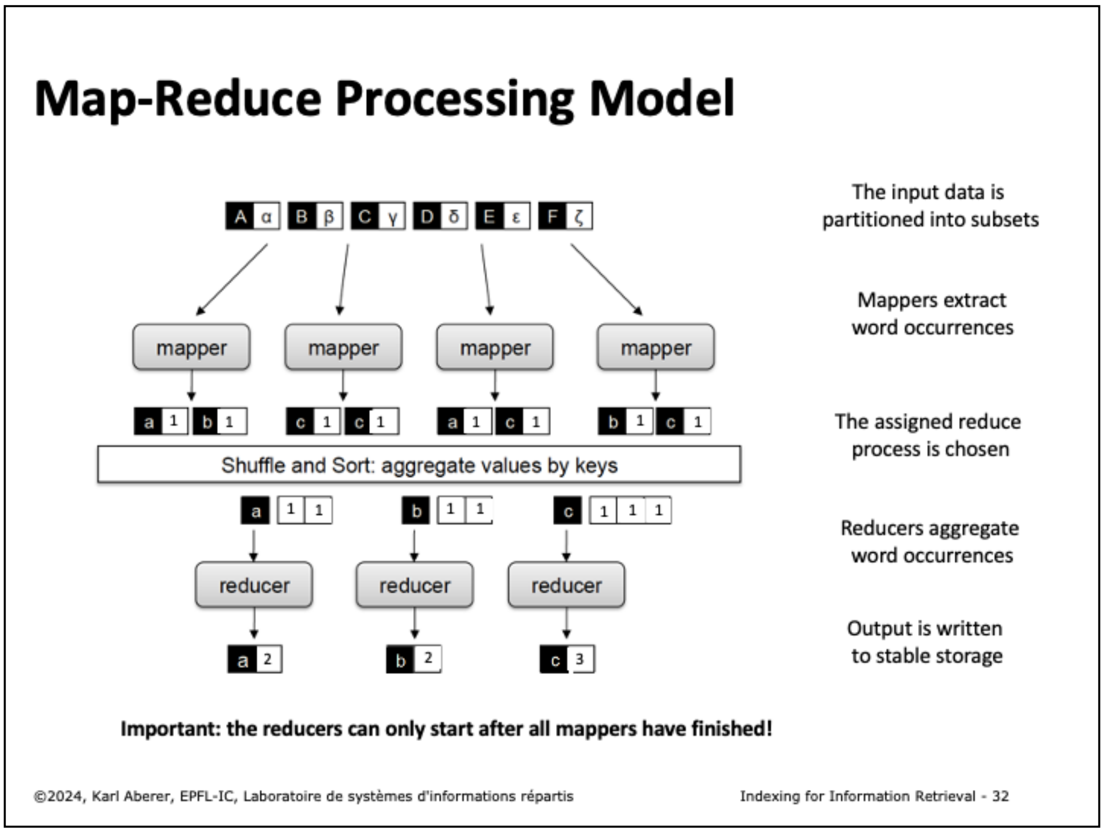
```
def mapper(document, line):
for word in line.split(): output(word, 1)

def reducer(key, values):
output(key, sum(values))
```

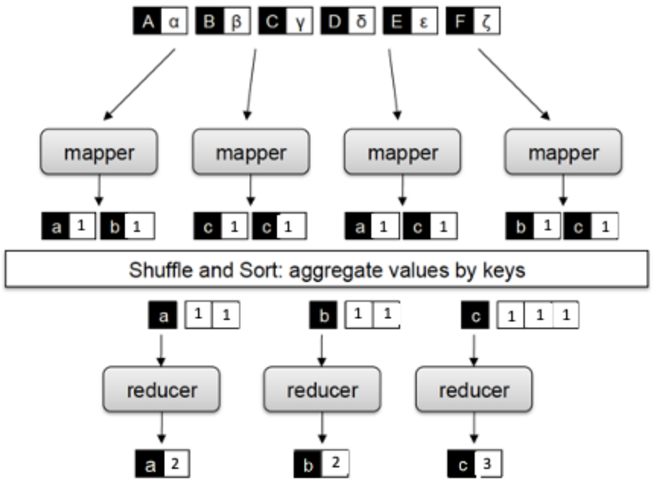
This is a basic illustration of the use of the map-reduce programming paradigm for a word counter program. A mapper processes the text by splitting it into words and producing for each word a key value pair (word, 1).

The reducer receives a list of all values associated with a key (a word), and outputs the aggregate value, the word count.

This is the code that a programmer has to write. The map-reduce system is then in charge of distributing the data, executing these functions in a distributed infrastructure and maling sure that the key-value pairs produced by the map function are properly aggregated and passed on to the reduce function.



Map-Reduce Processing Model



The input data is partitioned into subsets

Mappers extract word occurrences

The assigned reduce process is chosen

Reducers aggregate word occurrences

Output is written to stable storage

Important: the reducers can only start after all mappers have finished!

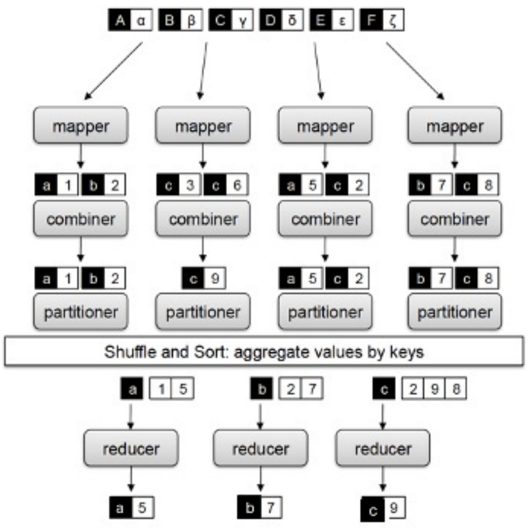
This figure illustrates the basic steps of a map-reduce computation for the basic example of word counting that are realized by the map-reduce infrastructure.

The system decides of how the document collection is partitioned and assigned to different mapper nodes. The mapper nodes extract word statistics for their partition of the document collection.

For each word, a reducer node is responsible. Based on the key, i.e., the word, the mapper nodes send their local results for the word to the responsible reducer node. This can be controlled, e.g., by hashing the key values and assigning hash values to reducer nodes. The reducer nodes aggregate the statistics that they receive from all the mapper nodes. When the reducer nodes have completed generating the partial indices for their key space, the results are written to the file system. The allocation of resources for the processes for mappers and reducers is performed automatically by the system and completely transparent to the developer of the code.

An important observation is that under this model, reducer nodes can start the aggregation only after all mapper nodes have finished, in order to assure that they receive all data. This can constitute an important bottleneck in such a system, since reducers can only start after the slowest mapper has terminated.

Refined Map-Reduce Programming Model



Combiners work like reducers,
but only on the local data of a mapper

```
def combiner(key, values):
  output(key, sum(values))
```

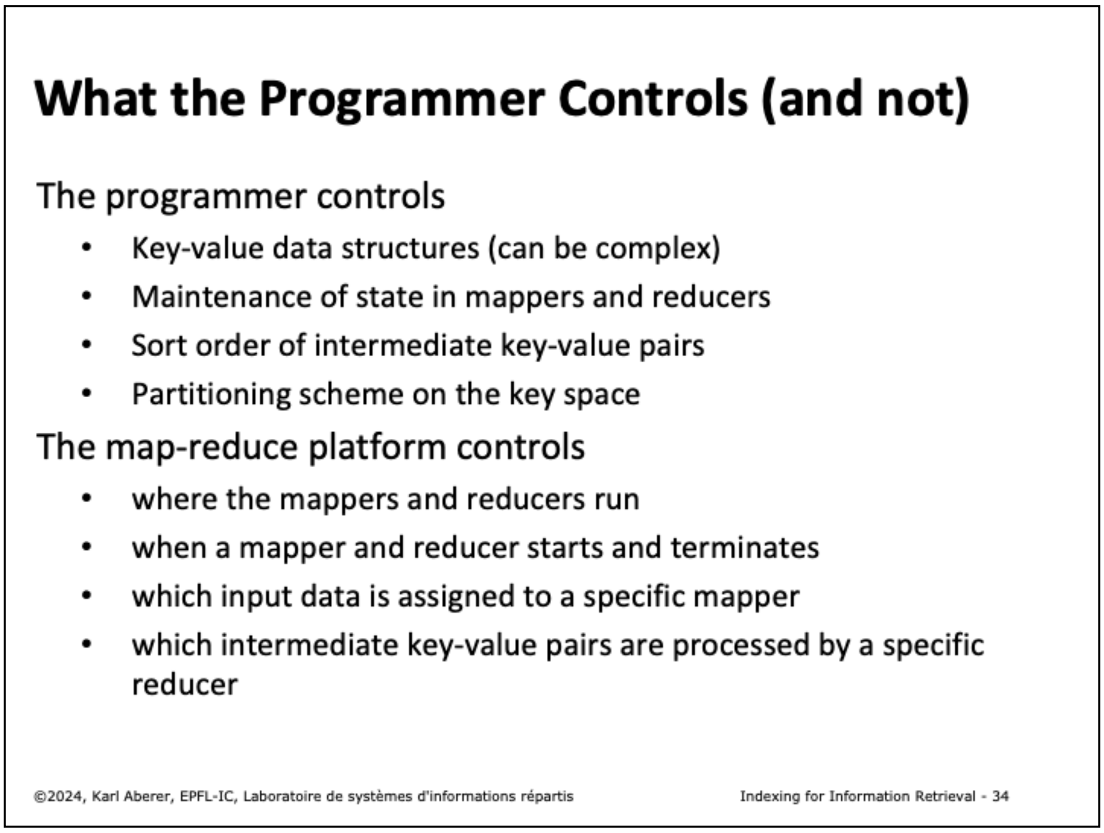
Partitioners allow to control the strategy
for distributing keys to reducers

Here we show some additional functions that are part of the map-reduce model. Note that the computation is different from the previous example. Mapper nodes already aggregate the statistics for a single document and the final output of the reducer is the maximum count instead of the sum.

Note that in this illustration the mapper emits only one key-value pair for each word and document, i.e., counts the total number of the occurrences of the word in the document (this is slightly smarter implementation than in the previous slide).

A combiner function can locally aggregate results on a node executing the mapper function (e.g., aggregating all counts of the same word), thus reducing the number of intermediate results.

Partitioners allow to control the strategy of distributing keys to reducer (to override the default strategy). This can be relevant if the programmer knows about the potential load distribution for keys and can thus optimize the allocation of reducer resources.



What the Programmer Controls (and not)

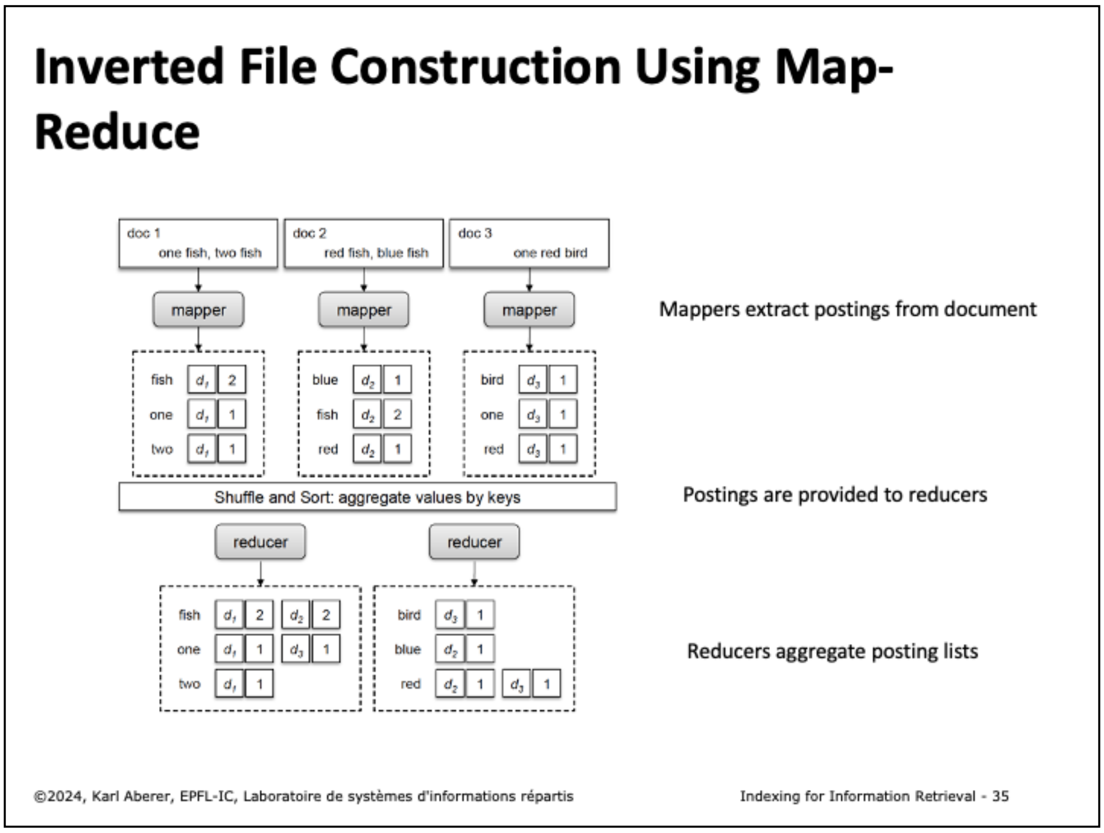
The programmer controls

- Key-value data structures (can be complex)
- Maintenance of state in mappers and reducers
- Sort order of intermediate key-value pairs
- Partitioning scheme on the key space

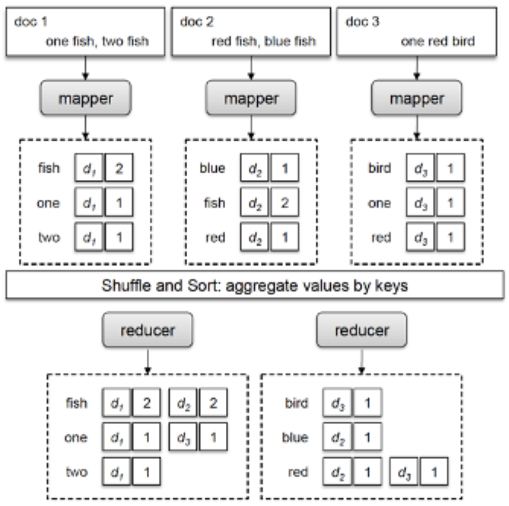
The map-reduce platform controls

- where the mappers and reducers run
- when a mapper and reducer starts and terminates
- which input data is assigned to a specific mapper
- which intermediate key-value pairs are processed by a specific reducer

The map-reduce model has been a successful approach to alleviate from the developers the tasks for dealing with the management of the distributed computing resources and concentrating on the logics of the data processing task.



Inverted File Construction Using Map-Reduce



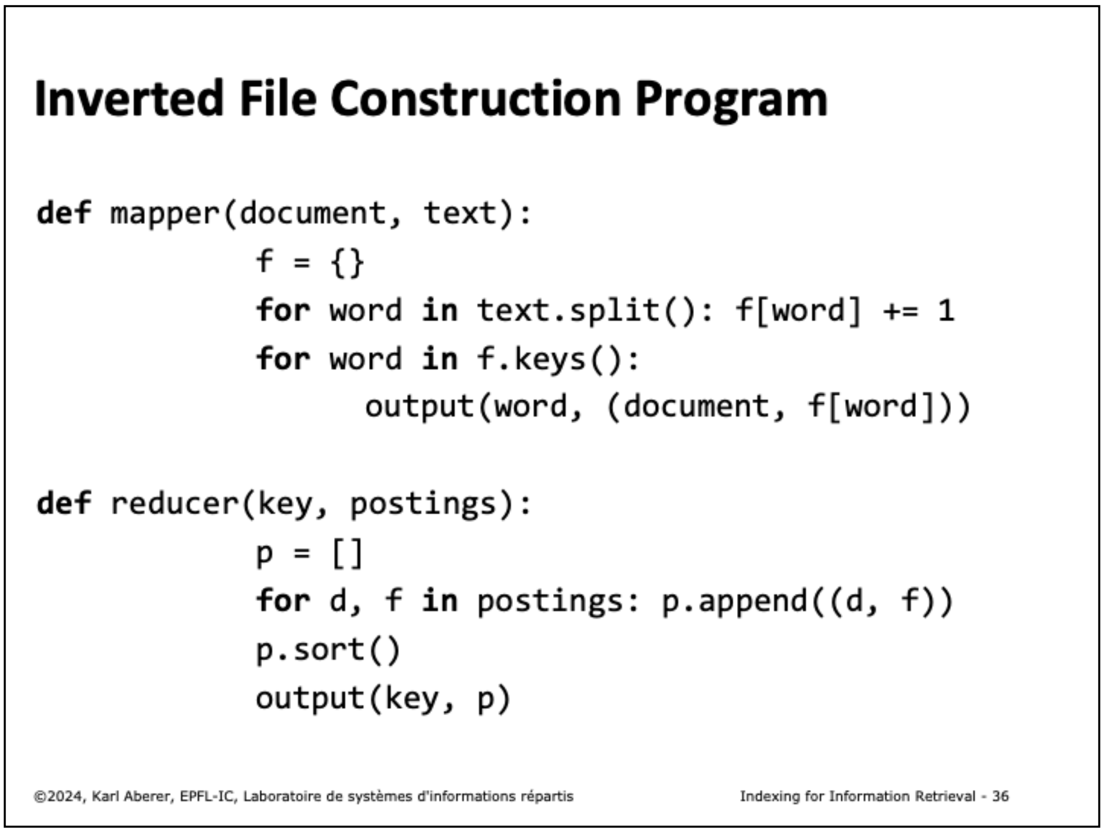
Mappers extract postings from document

Postings are provided to reducers

Reducers aggregate posting lists

We have illustrated the use of map-reduce so far performing for simple statistics tasks on text. The same programming paradigm can be used to perfrom the construction of an inverted file index. In this case, instead of producing simple counts, mappers produce inverted lists for the document inputs they receive. These inverted lists are then forwarded to the reducer in charge of the specific term. The reducers then aggregate the inverted files for a given term into a common inverted list. Finally, the full lists of postings can be stored.

Note that in this illustration the addressing granularity is at document level, i.e. words are associated with documents. The mappers compute the total frequency of the term in the given document.

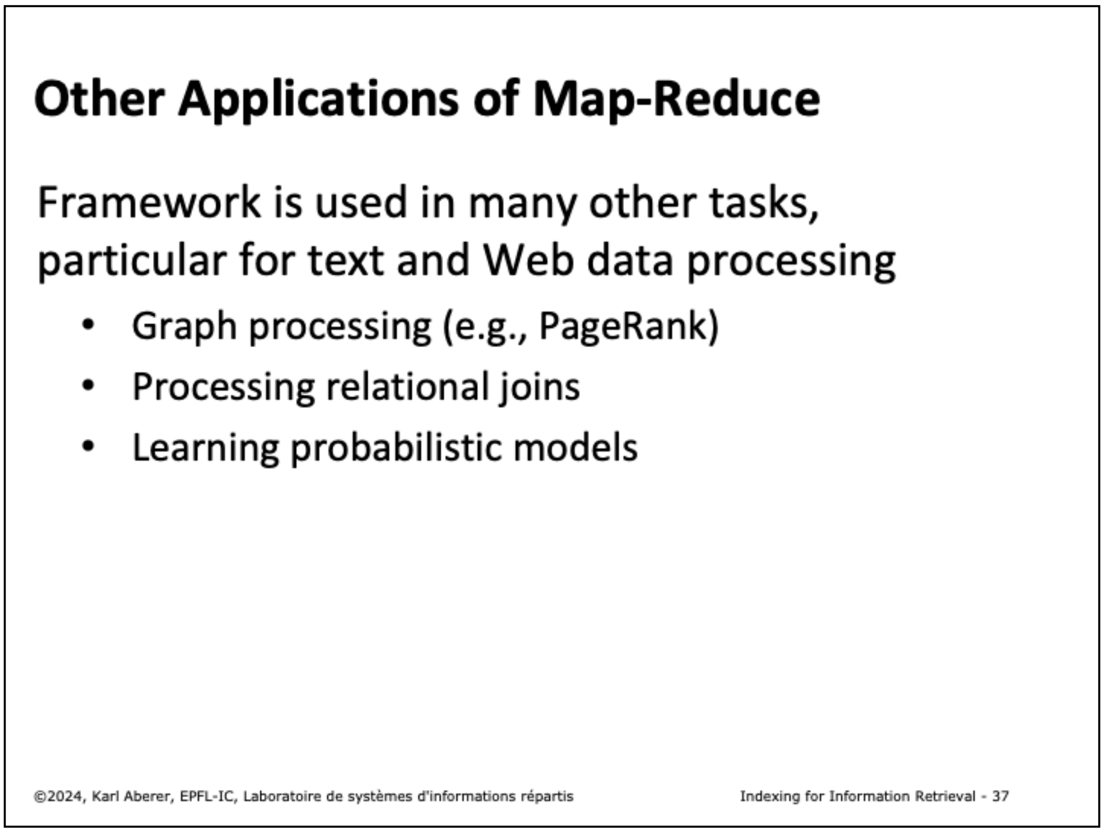


Inverted File Construction Program

```
def mapper(document, text):
            f = {}
            for word in text.split(): f[word] += 1
            for word in f.keys():
                  output(word, (document, f[word]))

def reducer(key, postings):
            p = []
            for d, f in postings: p.append((d, f))
            p.sort()
            output(key, p)
```

This is of how the inverted file construction illustrated on the previous slide could be implemented in the map-reduce model. Note that in the reducer the set of postings needs to be sorted, as they can be received from the mappers in arbitrary order.

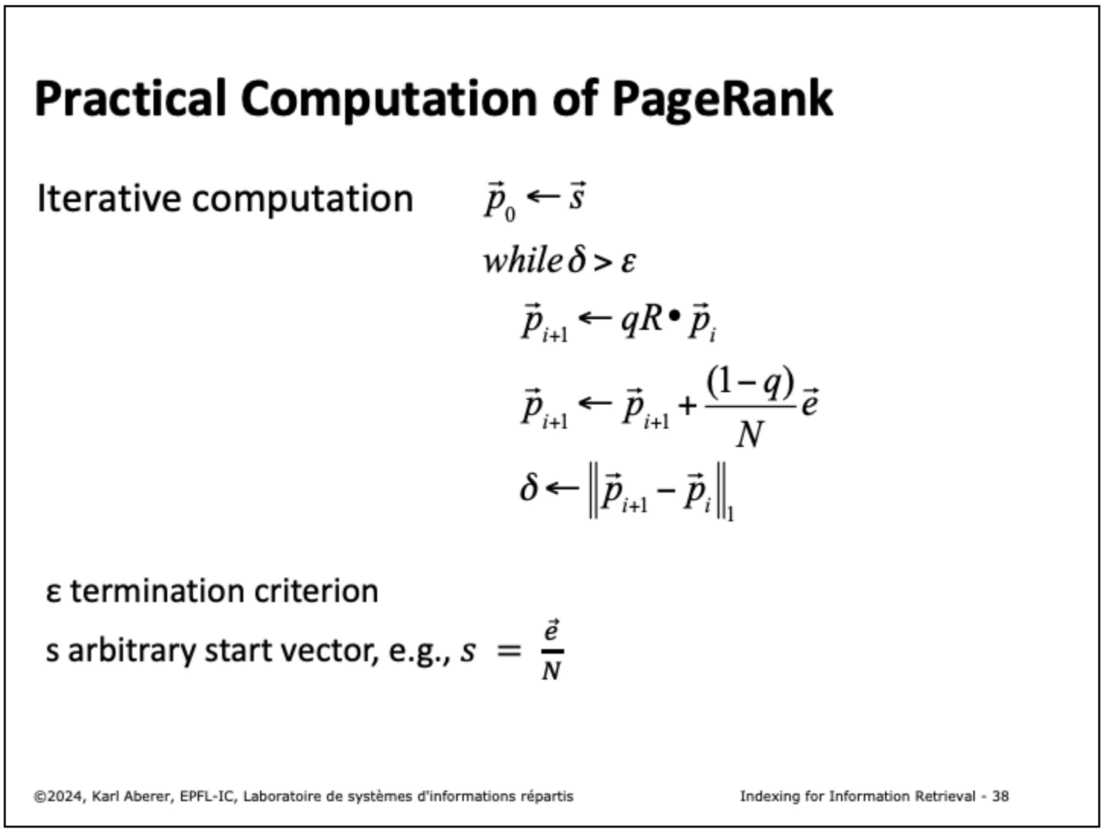


Other Applications of Map-Reduce

Framework is used in many other tasks, particular for text and Web data processing

- Graph processing (e.g., PageRank)
- Processing relational joins
- Learning probabilistic models

Though the initial motivation for developing map-reduce was the construction of Web-scale index structures, it turned out that the same model can be used for many other data processing tasks that allow for a high degree of parallel processing. These are called embarrassingly parallel workloads, where the input can be split with little effort into many partitions that can be processed in parallel.



Practical Computation of PageRank

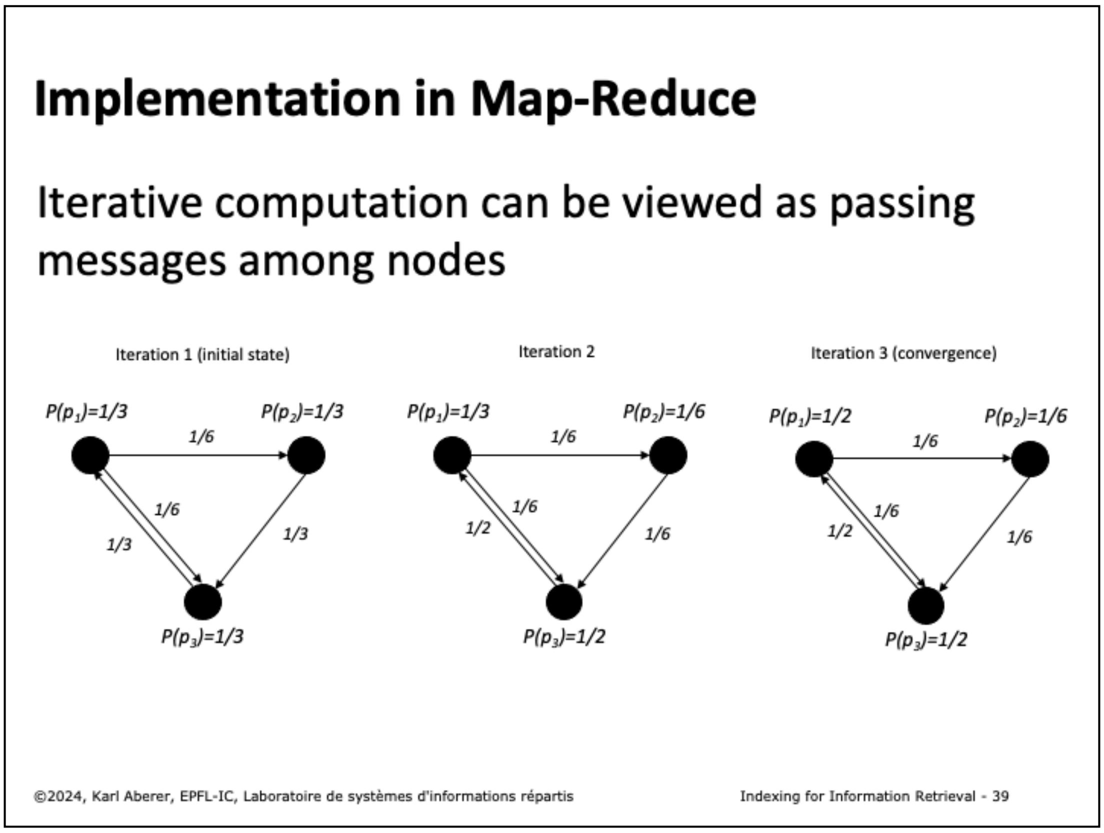
Iterative computation

```
$\vec{p}_0 \leftarrow \vec{s}$
$while\, \delta > \varepsilon$
    $\vec{p}_{i+1} \leftarrow qR \bullet \vec{p}_i$
    $\vec{p}_{i+1} \leftarrow \vec{p}_{i+1} + \frac{(1-q)}{N}\vec{e}$
    $\delta \leftarrow \left\| \vec{p}_{i+1} - \vec{p}_i \right\|_1$
```

ε termination criterion

s arbitrary start vector, e.g., $s = \frac{\vec{e}}{N}$

For the practical computation of the PageRank ranking an iterative approach can be used. The vector e is used to add a source of rank. It can uniformly distribute weights to all pages, but it could also incorporate pre-existing knowledge on the importance of pages and bias the ranking towards them. The vector can also be used as initial probability distribution.

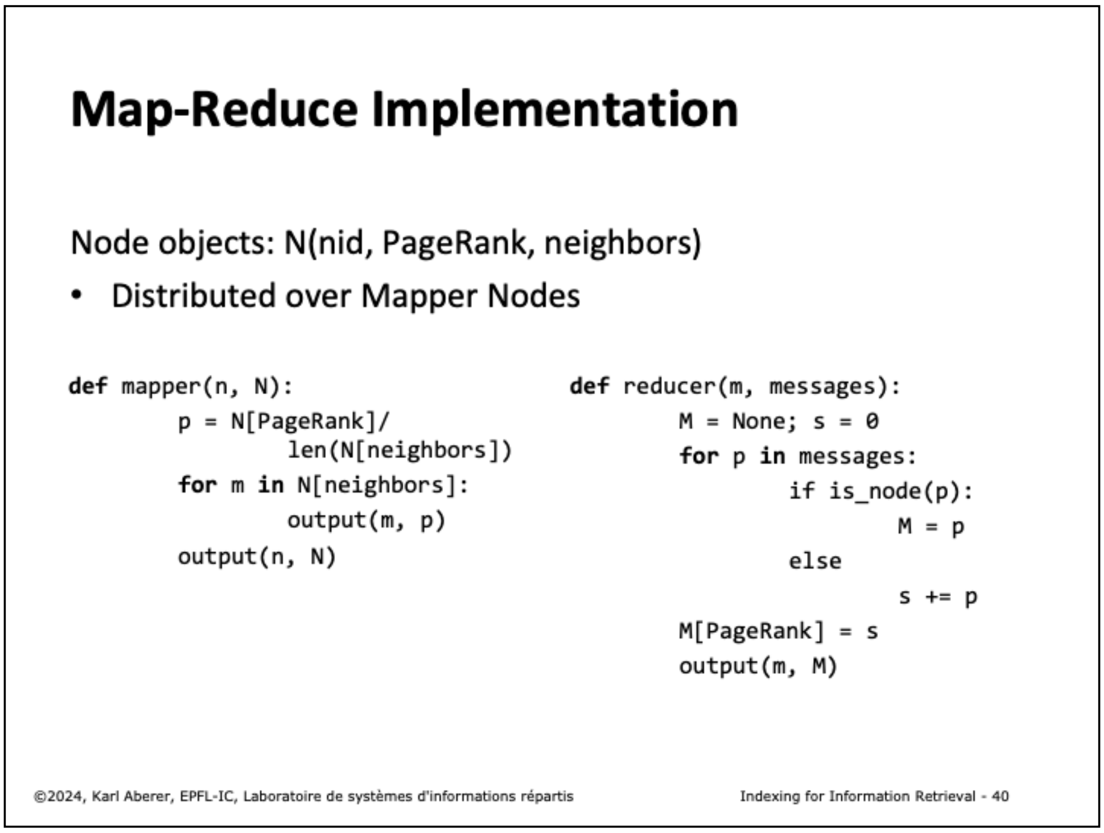


Implementation in Map-Reduce

Iterative computation can be viewed as passing messages among nodes

For practical computation of a ranking for a Web graph, the link matrices are huge and can potentially not be processed on a single node. This problem is similar to the one we have addressed for the construction of inverted files and for which one solution was the use of the map-reduce computing paradigm. We can use this paradigm as well for computing PageRank in a distributed way.

To that end, we consider the iterative computation of the PageRank values as a process of transmitting messages among the nodes. In the computation of the PageRank values a node receives in each step from the incoming links weights. It then aggregates those weights, before the next round of computation is performed. Representing the computation in this form leads directly to an approach for implementing it in the map-reduce programming model.



Map-Reduce Implementation

Node objects: N(nid, PageRank, neighbors)

- Distributed over Mapper Nodes

```
def mapper(n, N):
        p = N[PageRank]/
                len(N[neighbors])
        for m in N[neighbors]:
                output(m, p)
        output(n, N)
```

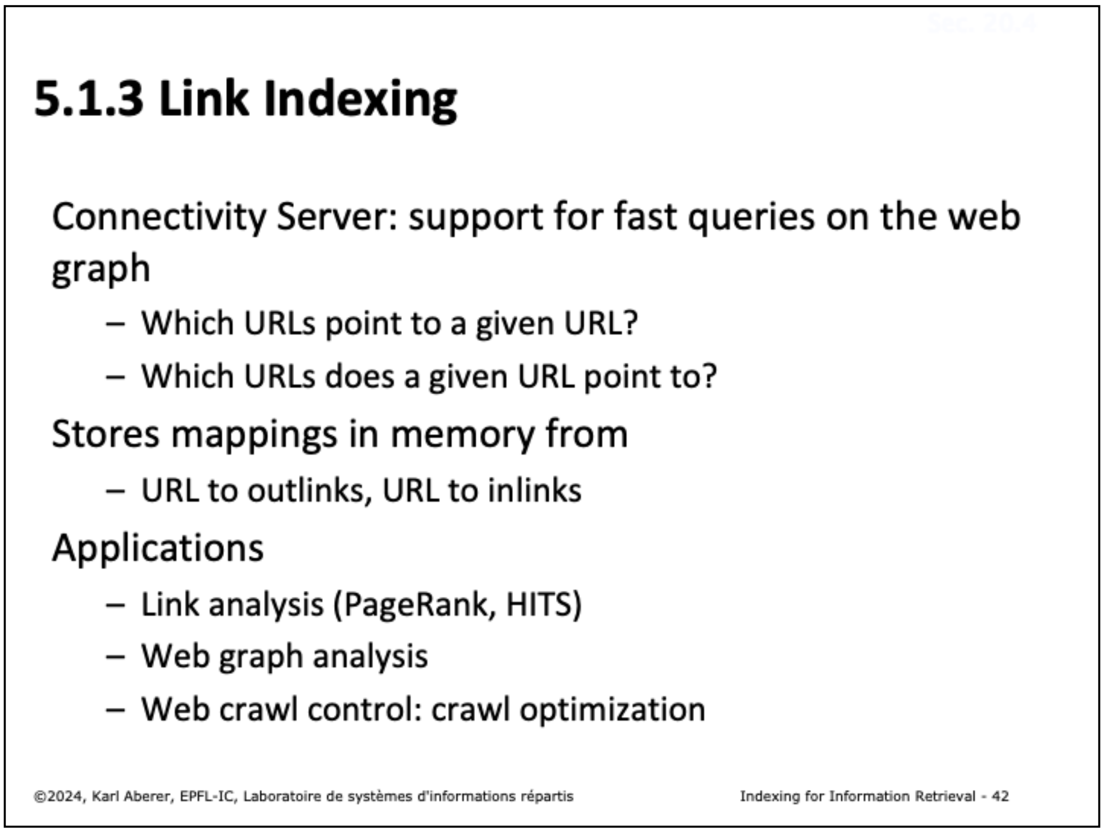
```
def reducer(m, messages):
        M = None; s = 0
        for p in messages:
                if is_node(p):
                        M = p
                else
                        s += p
        M[PageRank] = s
        output(m, M)
```

In the map-reduce implementation two types of outputs are processed. Node objects that contain a node identifier nid, the current PageRank values, and a list of the neighbour ids. The mapper receives a node object N. It computes the contribution of its rank to its neighbours and outputs the neighbour node ids together with their weights. It also outputs the nodeid together with the node object.

The reducers collect all messages concerning a given node m. It receives both the node object and all weights. The weights are added up and used to update the node object. The reducer outputs the updated node object that will be serving as inputs to mappers in a subsequent map-reduce phase. Therefore, each execution of a mapreduce job corresponds to one iteration of the PageRank algorithm.

Maintaining the order of document identifiers for vocabulary construction when partitioning the document collection is important ...

1. in the index merging approach for single node machines
2. in the map-reduce approach for parallel clusters
3. in both
4. in neither of the two



5.1.3 Link Indexing

Connectivity Server: support for fast queries on the web graph

- Which URLs point to a given URL?
- Which URLs does a given URL point to?

Stores mappings in memory from

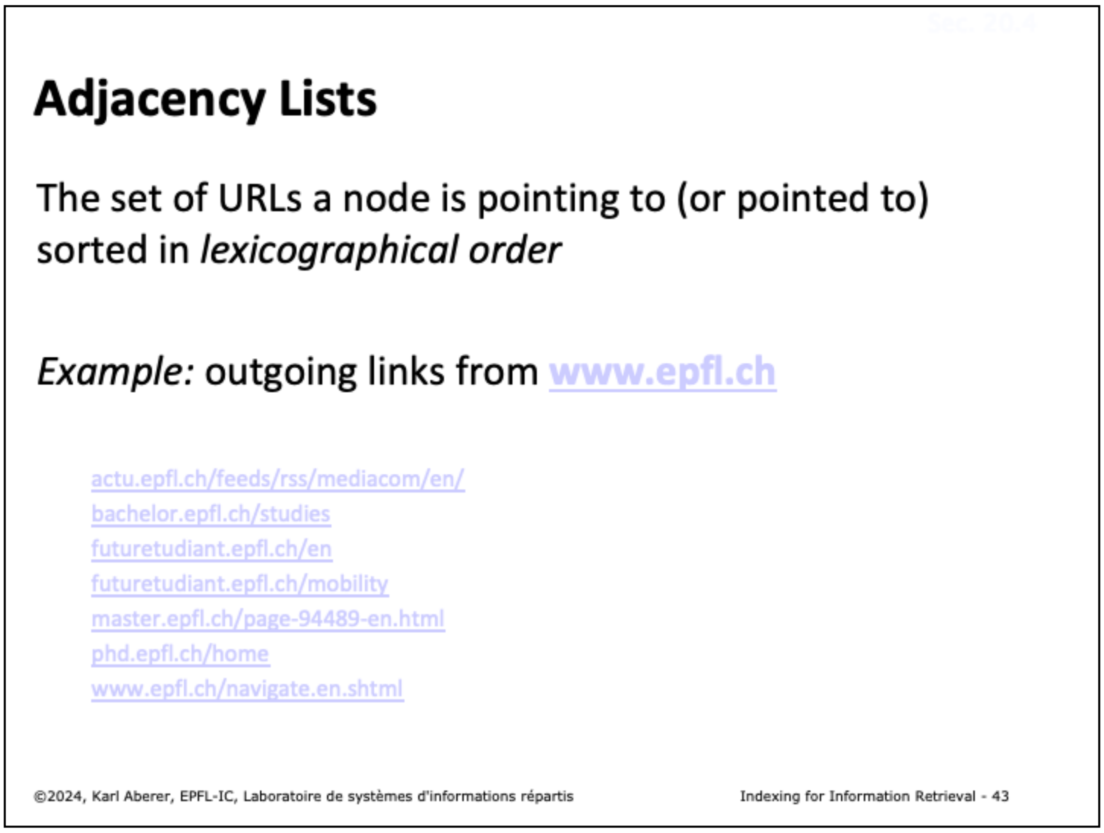
- URL to outlinks, URL to inlinks

Applications

- Link analysis (PageRank, HITS)
- Web graph analysis
- Web crawl control: crawl optimization

For text retrieval we introduced inverted files as an indexing structure for fast lookup of documents based on text queries. Using such an index it becomes possible to efficiently compute that statistics needed for the ranking model.

Similarly as for link-based ranking, for computing statistics on the Web graph we need an efficient indexing structure for the link graph. This is called a connectivity server. It allows to answer efficiently the queries relevant for Web graph analysis, namely which URLs a page points to and is pointed to. Beyond performing Web graph analysis such a connectivity server has also other applications, especially for controlling Web crawling.


Sec. 20.4

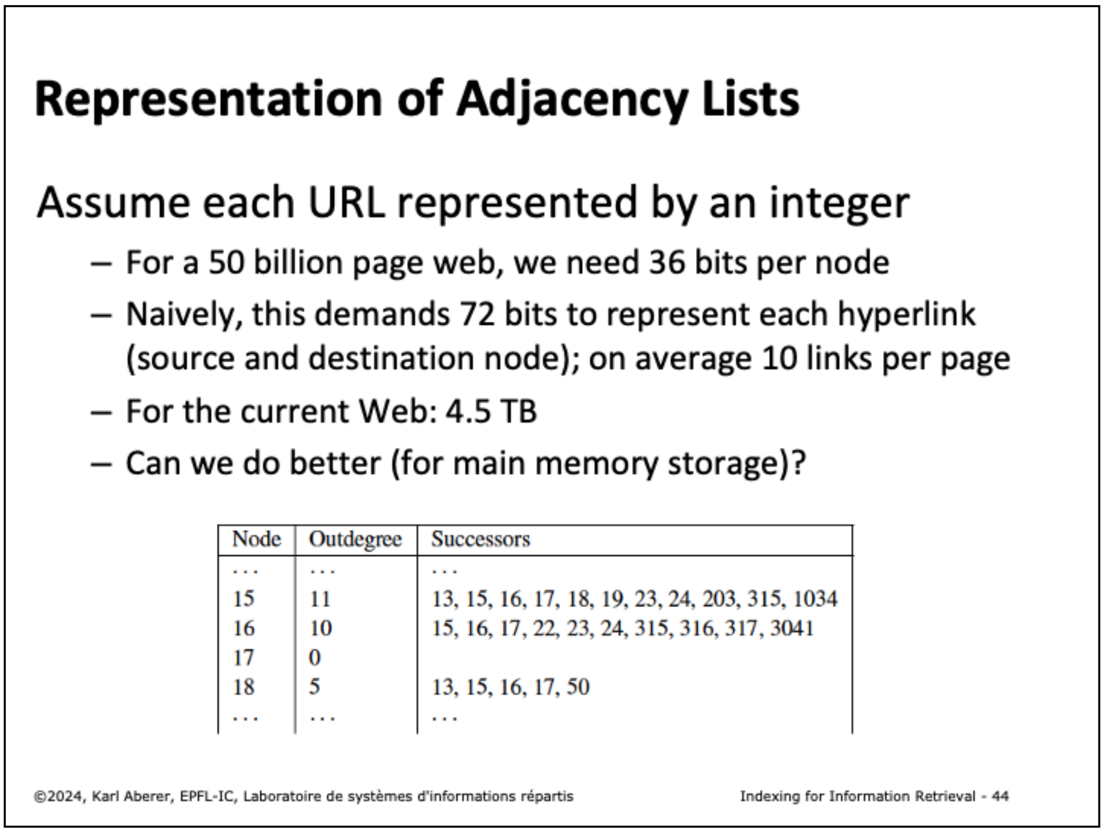
Adjacency Lists

The set of URLs a node is pointing to (or pointed to) sorted in *lexicographical order*

Example: outgoing links from www.epfl.ch

actu.epfl.ch/feeds/rss/mediacom/en/
bachelor.epfl.ch/studies
futuretudiant.epfl.ch/en
futuretudiant.epfl.ch/mobility
master.epfl.ch/page-94489-en.html
phd.epfl.ch/home
www.epfl.ch/navigate.en.shtml

A connectivity server has to store all outgoing (and incoming) links to a web page. For example, the home page of EPFL contains a large set of outgoing links, some of which are shown here. As a first step, the lists of links are sorted in lexicographical order. As a result, we obtain the adjacency list for a Web page, which we can consider as the equivalent to the posting list of a document in text indexing.



Representation of Adjacency Lists

Assume each URL represented by an integer

- For a 50 billion page web, we need 36 bits per node
- Naively, this demands 72 bits to represent each hyperlink (source and destination node); on average 10 links per page
- For the current Web: 4.5 TB
- Can we do better (for main memory storage)?

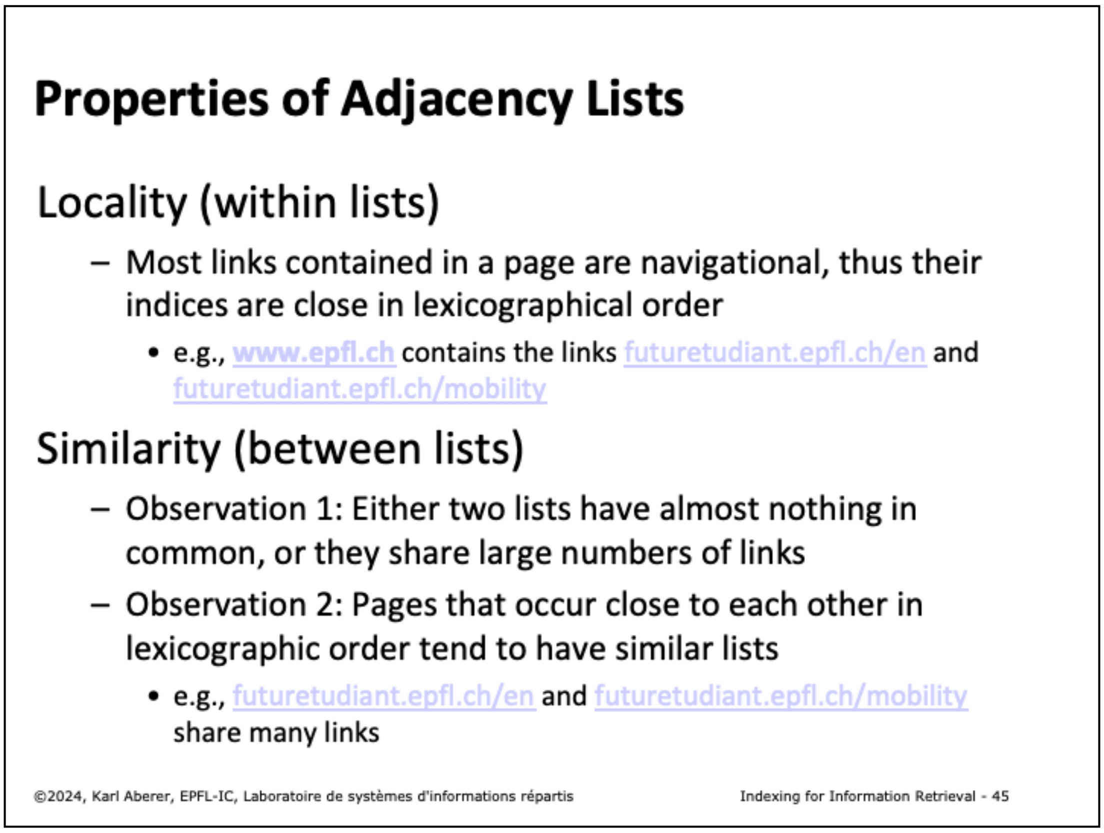
Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

As a first optimization of the representation of adjacency lists, we represent each URL by an integer, instead of storing it in its textual form. Using such an approach we can estimate the total size of adjacency lists for the current Web:

- The current (crawled) Web has around 50 billion pages (http://www.worldwidewebsize.com/, 2022)
- It is estimated that a page contains on average 10 links
- We need 32 bits for each URL, which demands 64 bits for the storage of a single link.

Therefore, the required storage is 4 TB.

Even with large memory sizes available today, this still is a significant index size. In the following, we will show how to reduce required storage to approximately ~3 bits/link which makes the size of the index much more manageable, i.e., about 20 times less storage. This will be achieved by systematically compressing the adjacency lists.



Properties of Adjacency Lists

Locality (within lists)

- Most links contained in a page are navigational, thus their indices are close in lexicographical order
  - e.g., www.epfl.ch contains the links futuretudiant.epfl.ch/en and futuretudiant.epfl.ch/mobility

Similarity (between lists)

- Observation 1: Either two lists have almost nothing in common, or they share large numbers of links
- Observation 2: Pages that occur close to each other in lexicographic order tend to have similar lists
  - e.g., futuretudiant.epfl.ch/en and futuretudiant.epfl.ch/mobility share many links

©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

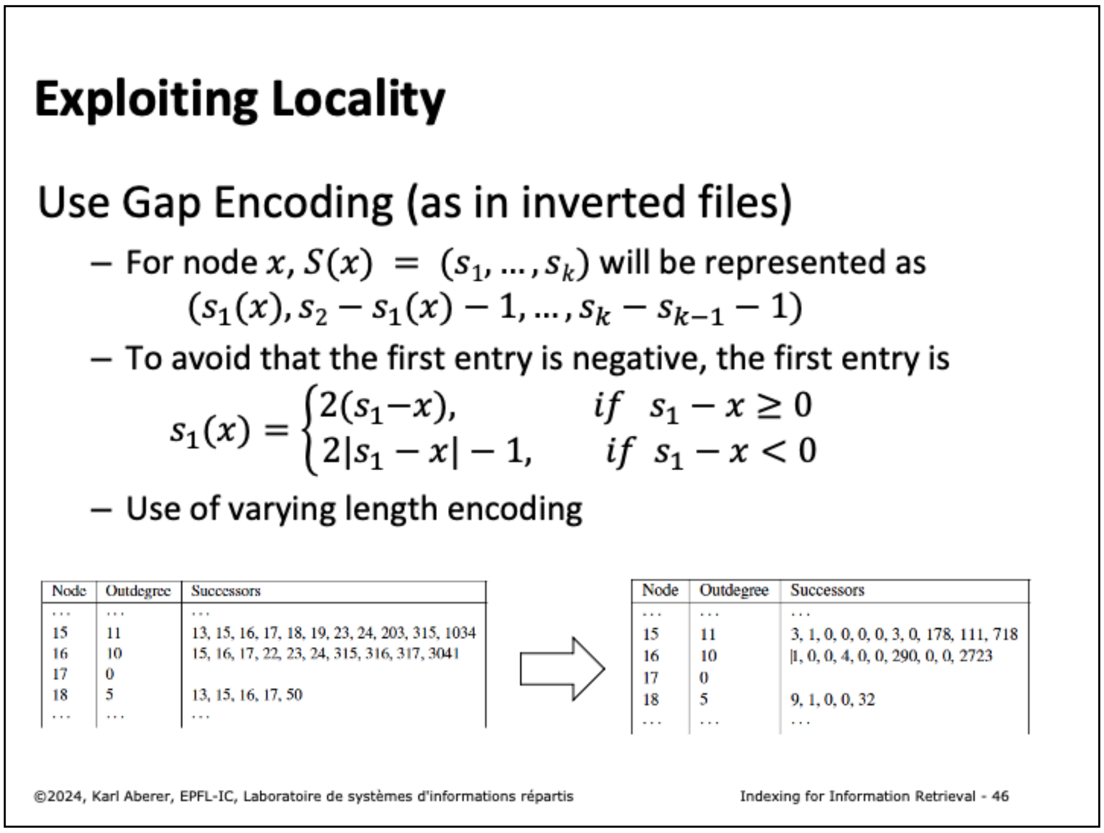
Indexing for Information Retrieval - 45

For compressing adjacency lists we can exploit several observations on typical properties of Web pages.

Locality: Most links contained in a page are for navigating withing the same Web site. If we compare the source and target URLs of these links, we observe that as a result they often share a long common prefix. Thus, if URLs are sorted lexicographically, the index of the URL of a Web page and the URLs of the targets of the links of the Web page are close to each other. Locality is a property of one adjacency list, thus is an intra-list similarity property.

Similarity: In general, we can assume that either pages have many common links, because the belong to the same web site, or they have almost nothing in common, because they are from different Web sites. Furthermore, pages that are from the same Website will have URLs that are similar in lexicographical order, and therefore it is more likely to find pages with many common outgoing links close to each other in lexicographic order. Similarity is a property of different adjacency lists, this an inter-list similarity property.

We will now show of how to exploit these two properties.



Exploiting Locality

Use Gap Encoding (as in inverted files)

- For node x, $S(x) = (s_1, \dots, s_k)$ will be represented as
  $(s_1(x), s_2 - s_1(x) - 1, \dots, s_k - s_{k-1} - 1)$
- To avoid that the first entry is negative, the first entry is

$$s_1(x) = \begin{cases} 2(s_1 - x), & if \;\; s_1 - x \geq 0 \\ 2|s_1 - x| - 1, & if \;\; s_1 - x < 0 \end{cases}$$

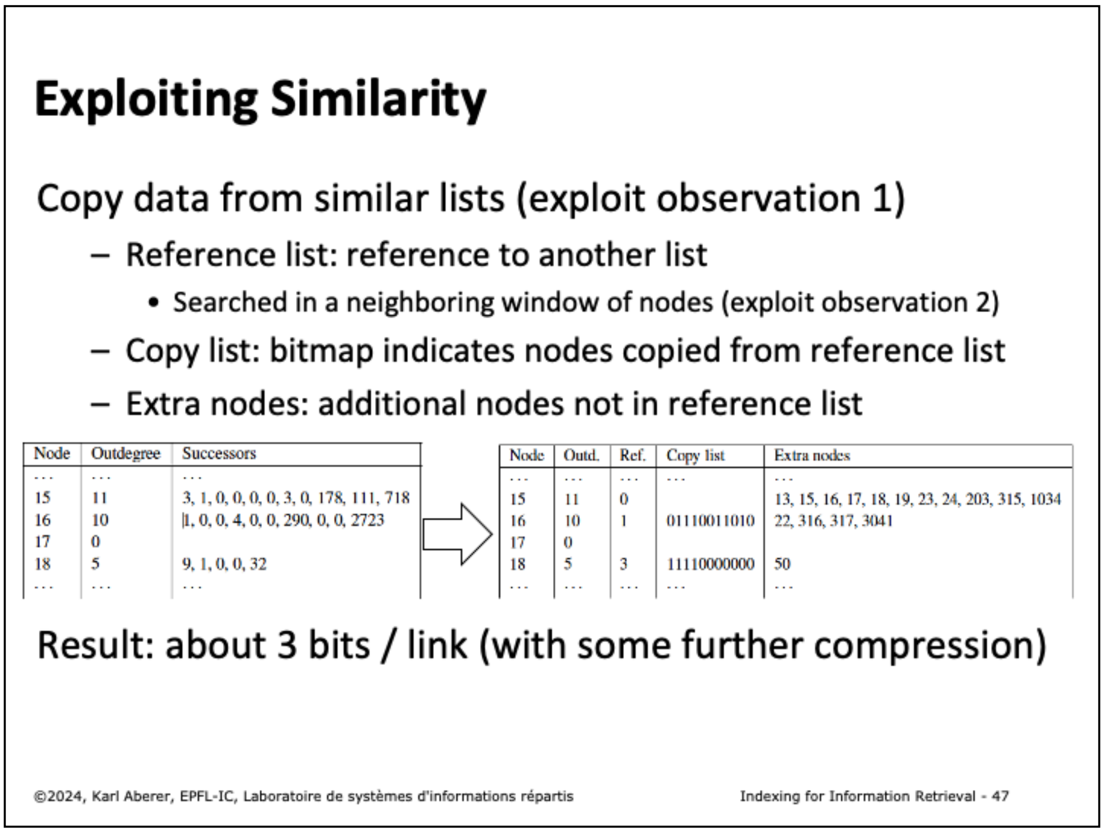
- Use of varying length encoding

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

$\Rightarrow$

Node	Outdegree	Successors
...	...	...
15	11	3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718
16	10	|1, 0, 0, 4, 0, 0, 290, 0, 0, 2723
17	0	
18	5	9, 1, 0, 0, 32
...	...	...

Locality can be exploited in a way analogous of how compression of posting lists for text indexing has been performed. Instead of storing the absolute integer indices of the URL identifiers, their differences are stored. In other words, we perform gap encoding. The resulting differences are then encoded using a varying length compression scheme, such as gamma coding, as it has been applied with inverted files.



Exploiting Similarity

Copy data from similar lists (exploit observation 1)

- Reference list: reference to another list
  - Searched in a neighboring window of nodes (exploit observation 2)
- Copy list: bitmap indicates nodes copied from reference list
- Extra nodes: additional nodes not in reference list

Node	Outdegree	Successors
...	...	...
15	11	3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718
16	10	|1, 0, 0, 4, 0, 0, 290, 0, 0, 2723
17	0	
18	5	9, 1, 0, 0, 32
...	...	...

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Result: about 3 bits / link (with some further compression)

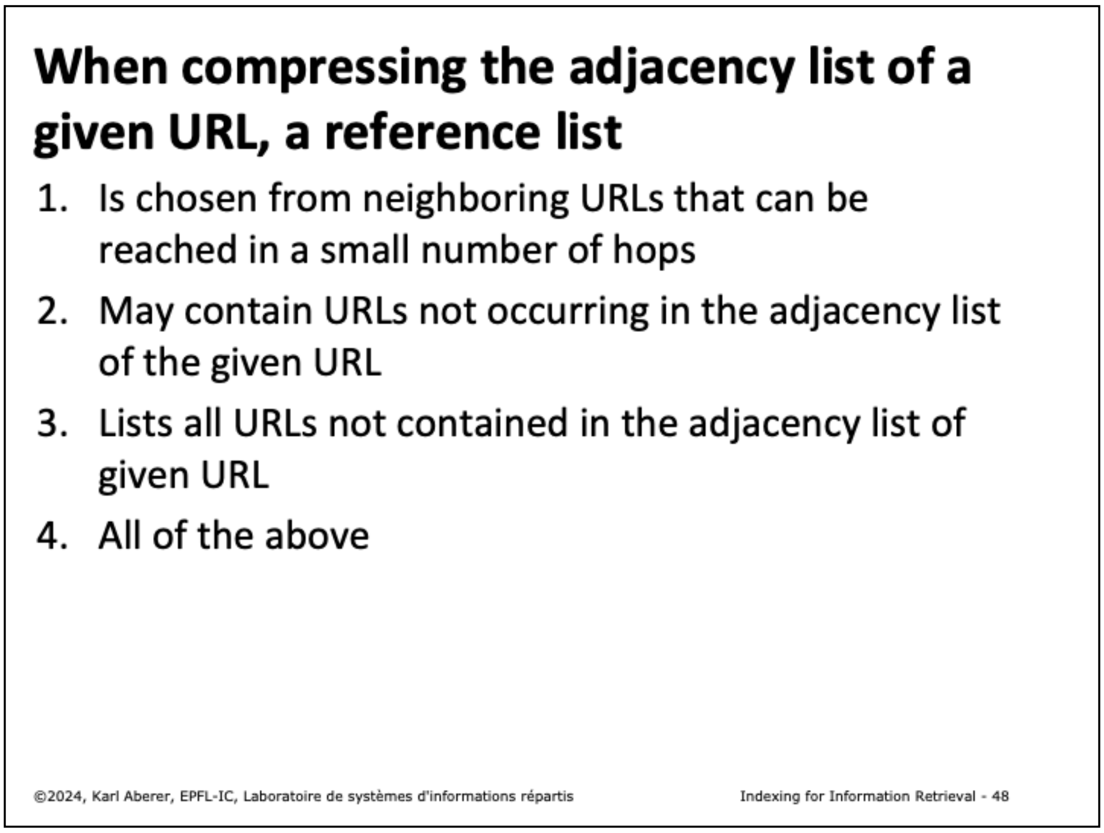
For exploiting similarity, we exploit the redundancy among similar lists. Based on the observation that similar lists are more likely to occur in a lexicographical neighborhood, in a first step a sliding window of neighboring lists that have already been processed is searched for a most similar list. If such a list is found, it is called reference list. Then for the given adjacency list to be compressed, only the data necessary to reconstruct the list from the reference list will be stored. For this it is necessary to store two types of data:

1. Copy data: this is a bitlist that indicates for every entry in the reference list whether the URL is also part of the given adjacency list. This covers all URLs that the given list can “inherit” from the reference list
2. Extra nodes: the given list can also contain URLs that are not part of the reference list. For those the indices of the URLs need to be stored explicitly.

In the example we use Node 15 for the reference list and compress Node 16 and 18. By comparing the lists we see that for the case of Node 18 all, but one URL appear in the reference list of Node 15. There are indicated in the bitlist. The adjacency list of Node 18 contains also one URLs that does not appear in the reference list, namely 50. This is listed in the list of extra nodes.

Candidates for potential reference lists are searched among neighboring lists using a window of predefined size. The choice of the window size is important, as larger windows increase chances of finding good candidates, but also increase the cost of compression.

Together with some further compression applied to the copy lists and the extra nodes, this index compression scheme achieves about 3 bits/link cost in the representation of the Web graph.

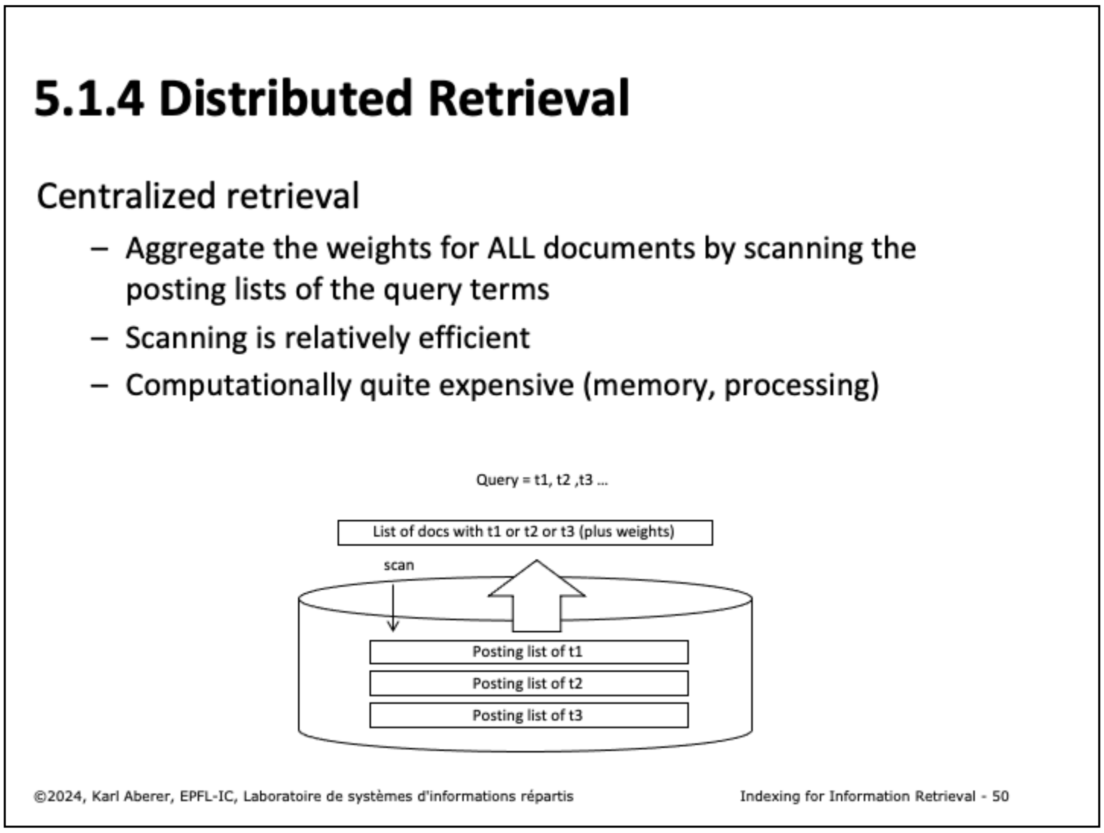


When compressing the adjacency list of a given URL, a reference list

1. Is chosen from neighboring URLs that can be reached in a small number of hops
2. May contain URLs not occurring in the adjacency list of the given URL
3. Lists all URLs not contained in the adjacency list of given URL
4. All of the above

Which is true?

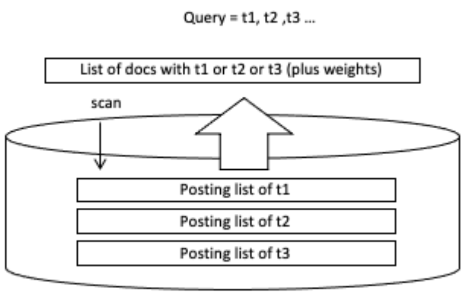
1. Exploiting locality with gap encoding may increase the size of an adjacency list
2. Exploiting similarity with reference lists may increase the size of an adjacency list
3. Both of the above is true
4. None of the above is true



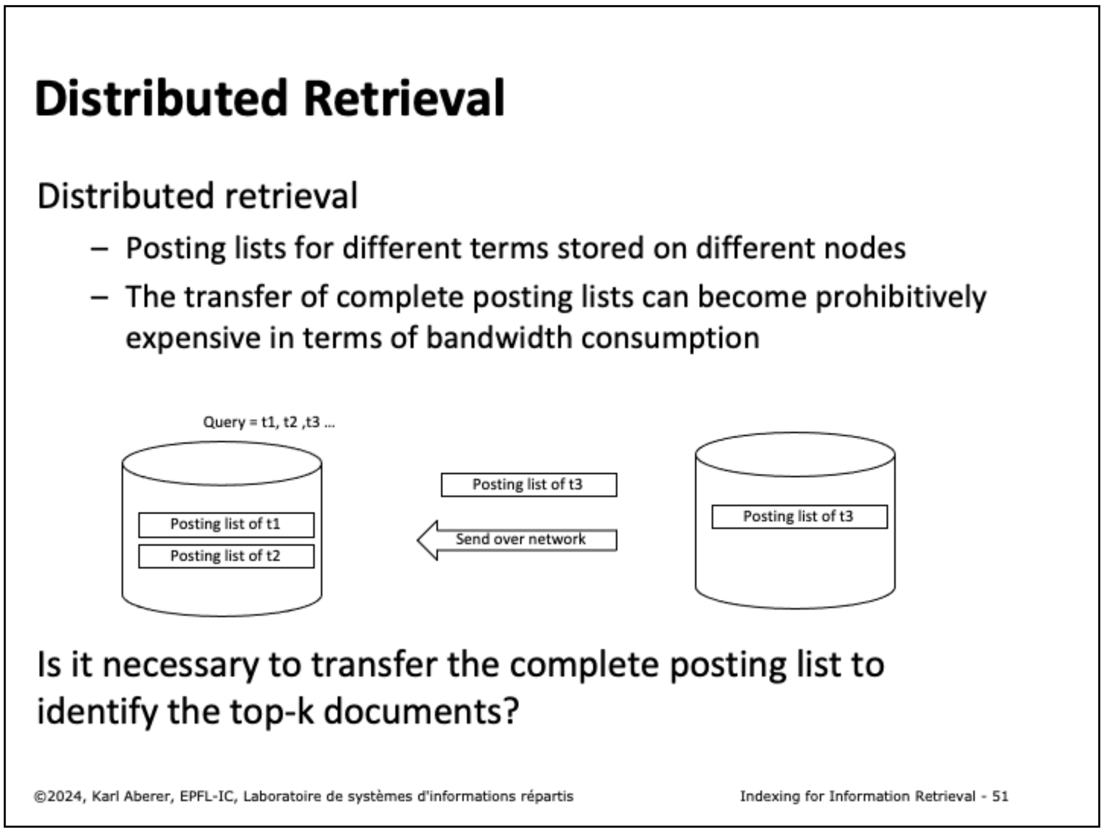
5.1.4 Distributed Retrieval

Centralized retrieval

- Aggregate the weights for ALL documents by scanning the posting lists of the query terms
- Scanning is relatively efficient
- Computationally quite expensive (memory, processing)



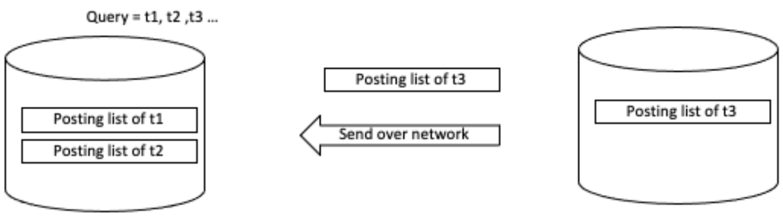
When using inverted files, a query involving multiple search terms requires the retrieval and scanning of the postings lists of all terms. In a centralized server this can be implemented relatively efficiently, though still resource-intensive, since scanning of disks is a comparably efficient operation.



Distributed Retrieval

Distributed retrieval

- Posting lists for different terms stored on different nodes
- The transfer of complete posting lists can become prohibitively expensive in terms of bandwidth consumption

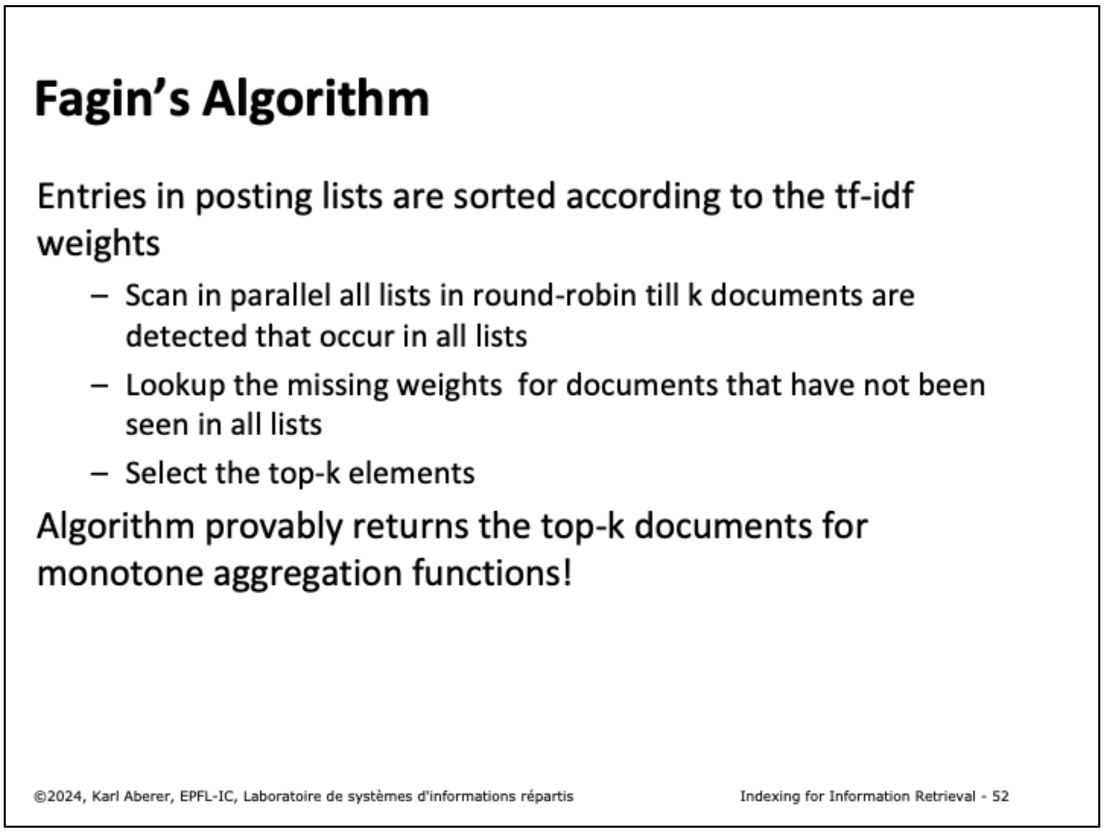


Is it necessary to transfer the complete posting list to identify the top-k documents?

In a distributed setting the picture changes quite significantly. Assuming that posting lists for different terms are stored on different nodes, complete posting lists need to be transferred over the network. For frequent terms,

these postings lists can contain very large numbers of entries, and significant amounts of data need to be transferred over network in order to compute the query result, which results in a prohibitively high network bandwidth consumption. So, the question is, whether there exist more efficient ways to determine the top ranked (top-k) elements of the result of a query, without the need to inspect complete posting lists.

Remark: in the following we will use k to indicate the number of results retrieved, even though we have used earlier k to denote the size of the vocabulary. The terminology top-k is so well established, that it would be confusing to deviate from it.



Fagin's Algorithm

Entries in posting lists are sorted according to the tf-idf weights

- Scan in parallel all lists in round-robin till k documents are detected that occur in all lists
- Lookup the missing weights for documents that have not been seen in all lists
- Select the top-k elements

Algorithm provably returns the top-k documents for monotone aggregation functions!

Fagin's algorithm is an approach for efficient distributed retrieval. It has been originally devised for multimedia queries, where multiple features of an object (e.g., an image) need to be combined to determine the most similar ones. The algorithm tries to minimize the number of objects (in our case documents) that need to be inspected in that process.

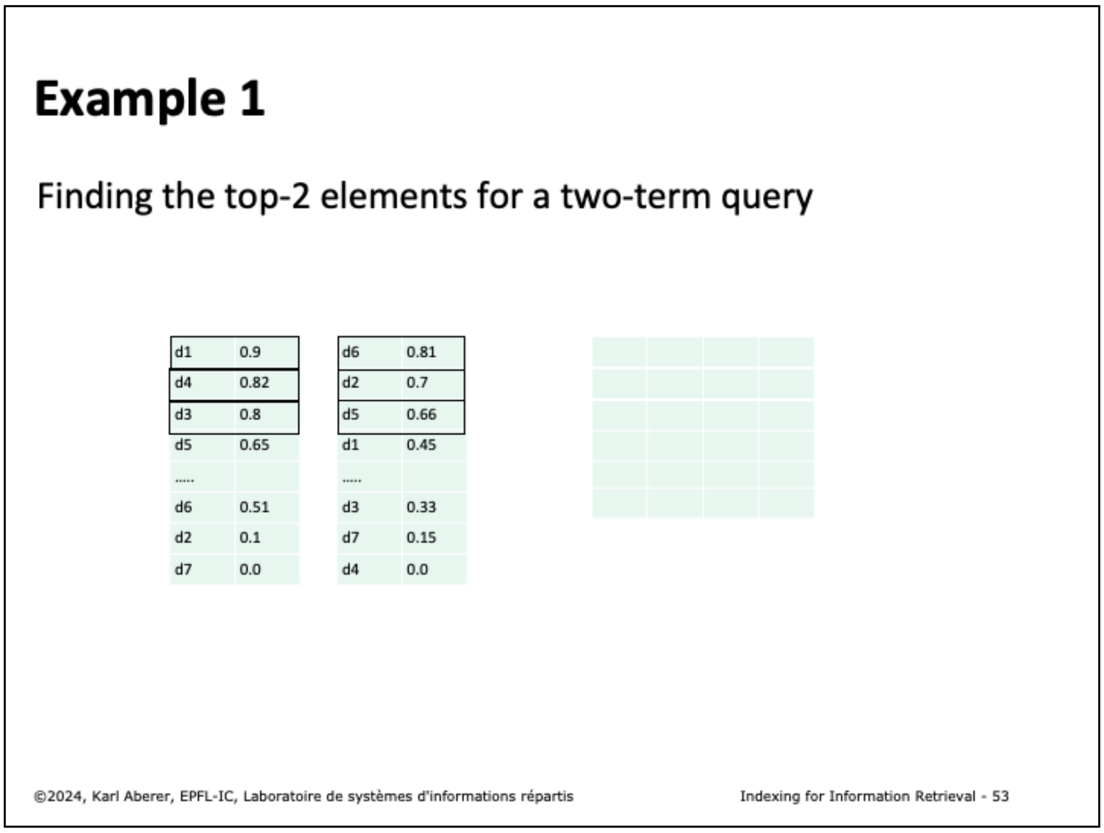
An important assumption that is made in Fagin's algorithm, is that the elements in a posting list are ordered according to the scores of the documents and not by document identifiers. In the context of text retrieval we would use tf-idf weights as the scores. Note that this assumption implies that an additional one-time cost occurs for sorting the posting lists. The algorithm proceeds as follows:

Phase 1: The algorithm scans in a round-robin fashion the elements of the posting lists starting from those with the highest score. Whenever an element is encountered in multiple lists, their scores are combined (e.g., added). This processing is continued till k elements are detected that appear in all lists.

Phase 2: By then, many other documents also may have been detected, that do not occur in all lists. Therefore, in a next step the missing scores are retrieved from the lists. This requires random access, supported by an index. This constitutes the most expensive part of the algorithm.

Phase 3: Finally, the k elements with the highest scores are returned. These are not necessarily corresponding to those elements that have been identified in the Phase 1. They also might include elements for which additional scores have been retrieved in Phase 2.

The algorithm returns provably always the k elements with the highest combined score.



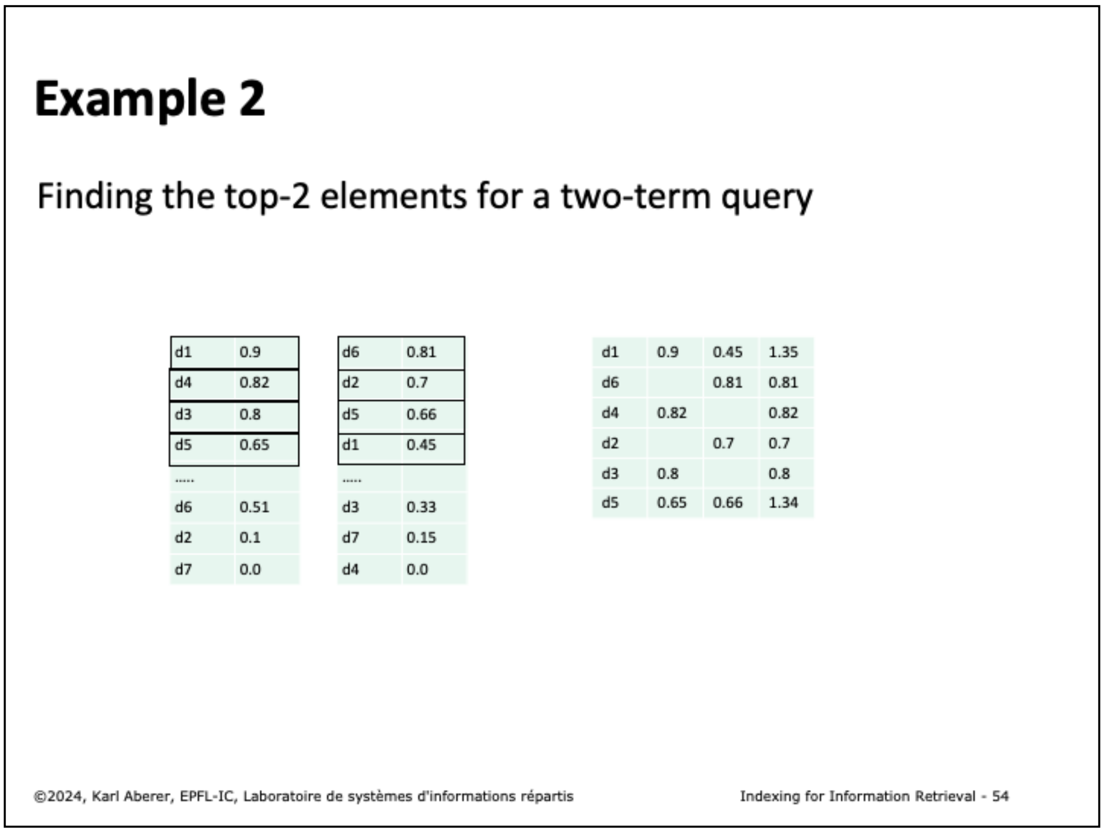
Example 1

Finding the top-2 elements for a two-term query

d1	0.9
d4	0.82
d3	0.8
d5	0.65
.....	
d6	0.51
d2	0.1
d7	0.0

d6	0.81
d2	0.7
d5	0.66
d1	0.45
.....	
d3	0.33
d7	0.15
d4	0.0

The example illustrates a case where two lists are searched, i.e., processing a query with two terms. First 6 new different documents are detected in phase 1 and their scores are recorded.



Example 2

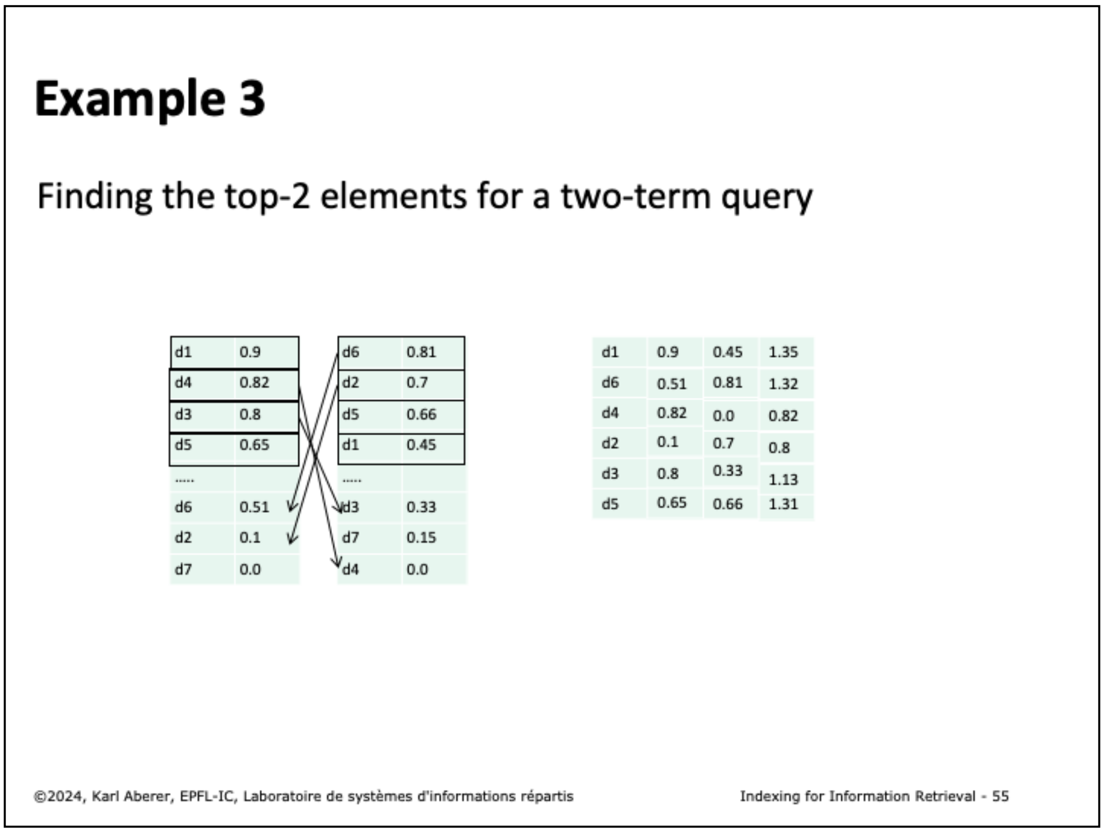
Finding the top-2 elements for a two-term query

d1	0.9
d4	0.82
d3	0.8
d5	0.65
.....	
d6	0.51
d2	0.1
d7	0.0

d6	0.81
d2	0.7
d5	0.66
d1	0.45
.....	
d3	0.33
d7	0.15
d4	0.0

d1	0.9	0.45	1.35
d6		0.81	0.81
d4	0.82		0.82
d2		0.7	0.7
d3	0.8		0.8
d5	0.65	0.66	1.34

In the next step we are detecting two documents, d1 and d5, that are occurring in both posting lists. Thus, we finish phase 1 of the algorithm, as we are now sure that the top-2 elements will be found in the documents detected so far.



Example 3

Finding the top-2 elements for a two-term query

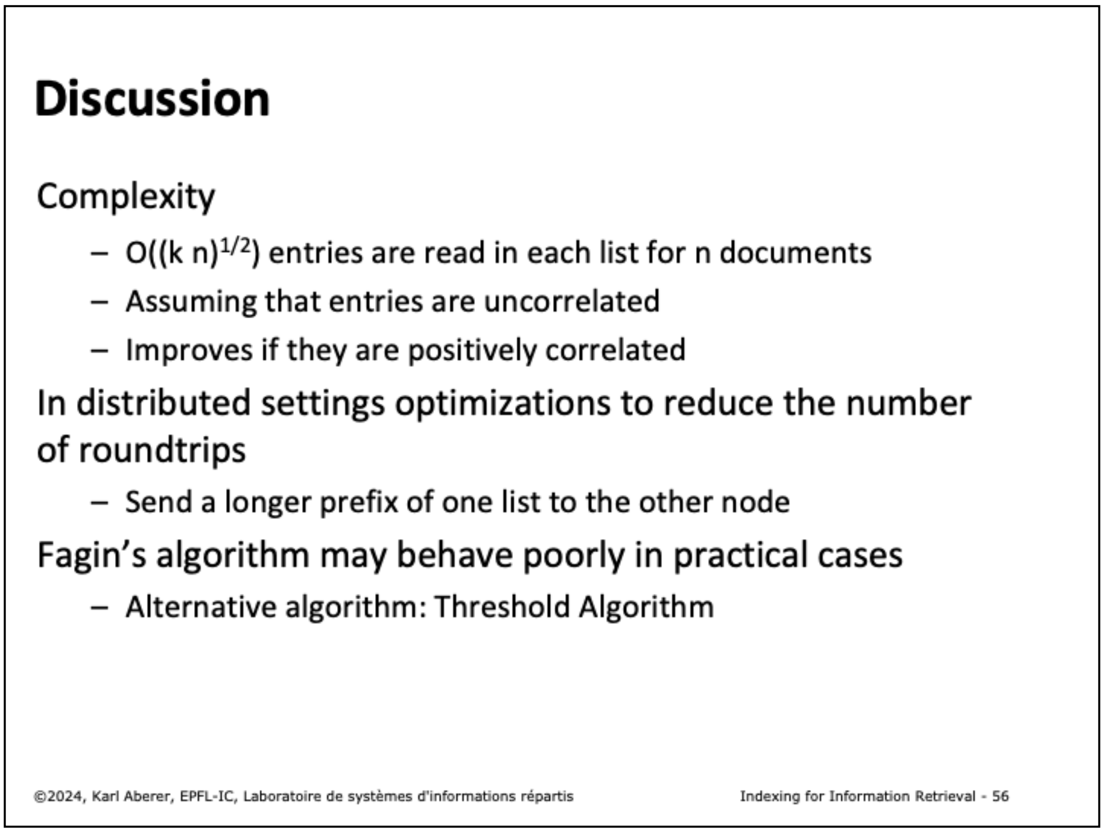
d1	0.9		d6	0.81
d4	0.82		d2	0.7
d3	0.8		d5	0.66
d5	0.65		d1	0.45
.....			.....	
d6	0.51		d3	0.33
d2	0.1		d7	0.15
d7	0.0		d4	0.0

d1	0.9	0.45	1.35
d6	0.51	0.81	1.32
d4	0.82	0.0	0.82
d2	0.1	0.7	0.8
d3	0.8	0.33	1.13
d5	0.65	0.66	1.31

©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

Indexing for Information Retrieval - 55

In phase 2, the missing scores of the other documents are retrieved using random access. Once they have been obtained, the top 2 documents are returned. In this example these are documents d1 and d6. Note that these are not the 2 documents that have been first discovered to occur in both lists, which where d1 and d5.



Discussion

Complexity

- $O((k\ n)^{1/2})$ entries are read in each list for n documents
- Assuming that entries are uncorrelated
- Improves if they are positively correlated

In distributed settings optimizations to reduce the number of roundtrips

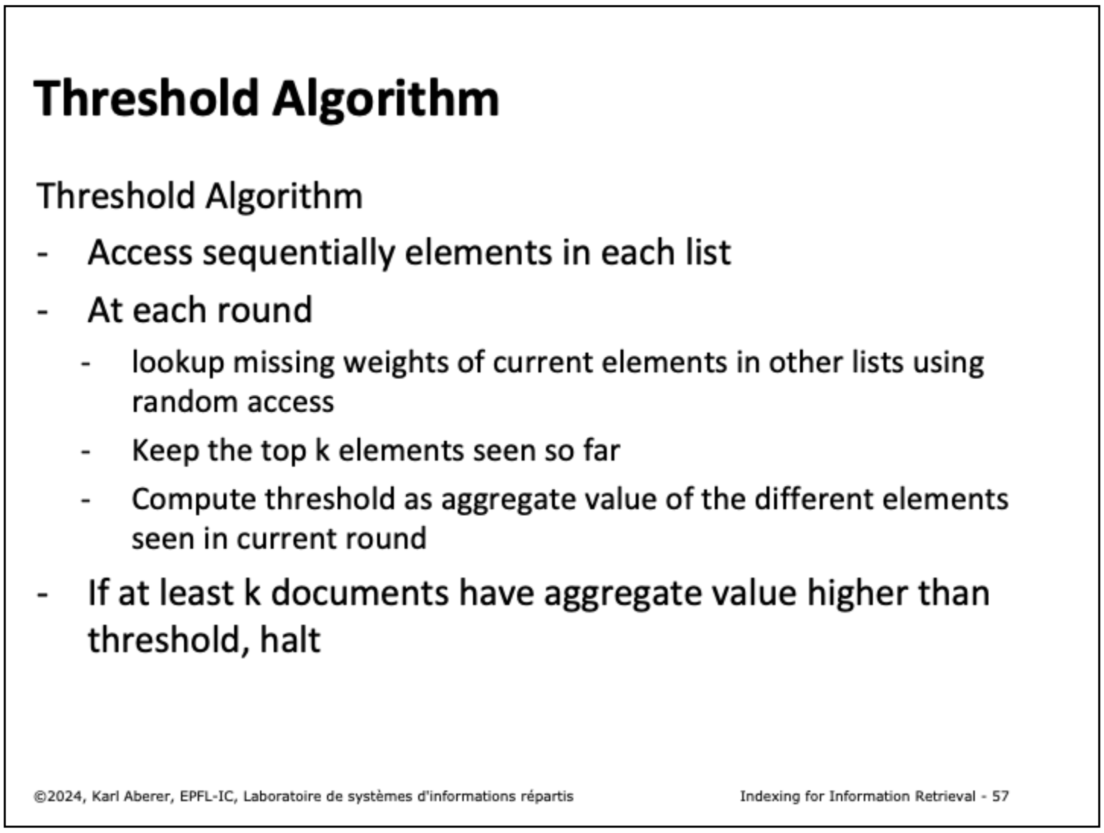
- Send a longer prefix of one list to the other node

Fagin's algorithm may behave poorly in practical cases

- Alternative algorithm: Threshold Algorithm

It can be shown that the complexity of the Fagin algorithm in the case of two lists is $O((k\ n)^{1/2})$ for the number of entries that are read from each list, where n is the number of documents in the document collection. This is significantly smaller than reading the complete lists and reduces further if the entries are positively correlated (i.e., if a document is highly ranked in one list, then it has also higher probability to be highly ranked in the other list), which is likely to be the case. The results generalizes to the case of multiple lists.

In a distributed setting applying Fagin's algorithm directly is still not very practical, since for every element retrieved from a list a message would have to be exchanged with another node. To avoid this, variants of this algorithm have been proposed, where larger chunks of the list from one node are sent to the other. In the ideal case one node "guesses" how many entries from its list would have to be read and transmits this set of entries to the other node(s).



Threshold Algorithm

Threshold Algorithm

- Access sequentially elements in each list
- At each round
  - lookup missing weights of current elements in other lists using random access
  - Keep the top k elements seen so far
  - Compute threshold as aggregate value of the different elements seen in current round
- If at least k documents have aggregate value higher than threshold, halt

©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

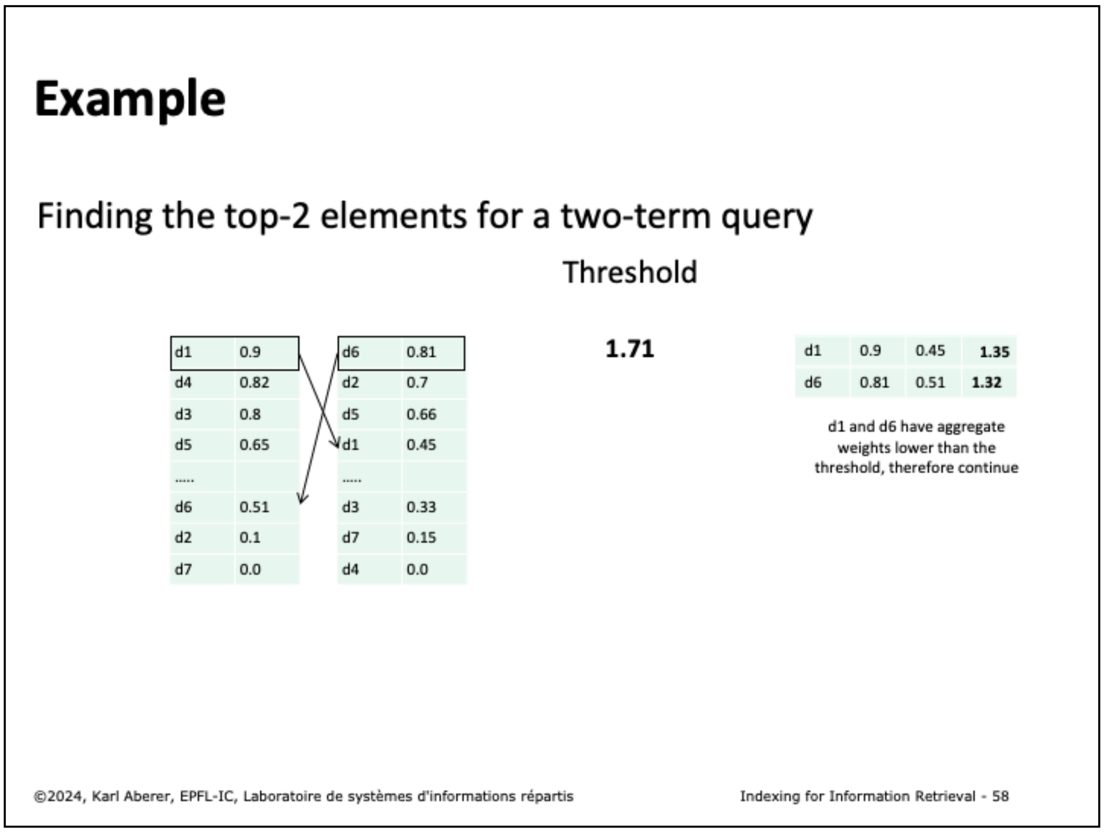
Indexing for Information Retrieval - 57

The threshold algorithm is an alternative to Fagin's algorithm. It processes also elements in the lists in a round robin fashion.

At each round it computes a threshold, as the aggregate value of the weights of the different elements accessed in the current round. Since the lists are sorted, the threshold value will continuously decrease. For the elements considered in the current round, the missing values from the other lists are obtained using random access, to obtain the aggregate score of those elements. Then the k elements with the highest scores are retained.

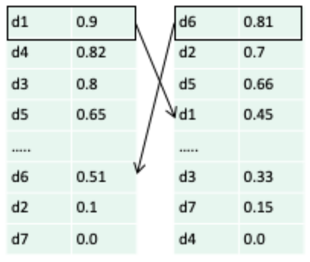
The algorithm stops once all retained elements have higher score than the current threshold value.

The algorithm returns provably always the k elements with the highest combined score in case of monotone aggregation functions.



Example

Finding the top-2 elements for a two-term query



d1	0.9		d6	0.81
d4	0.82		d2	0.7
d3	0.8		d5	0.66
d5	0.65		d1	0.45
…..			…..	
d6	0.51		d3	0.33
d2	0.1		d7	0.15
d7	0.0		d4	0.0

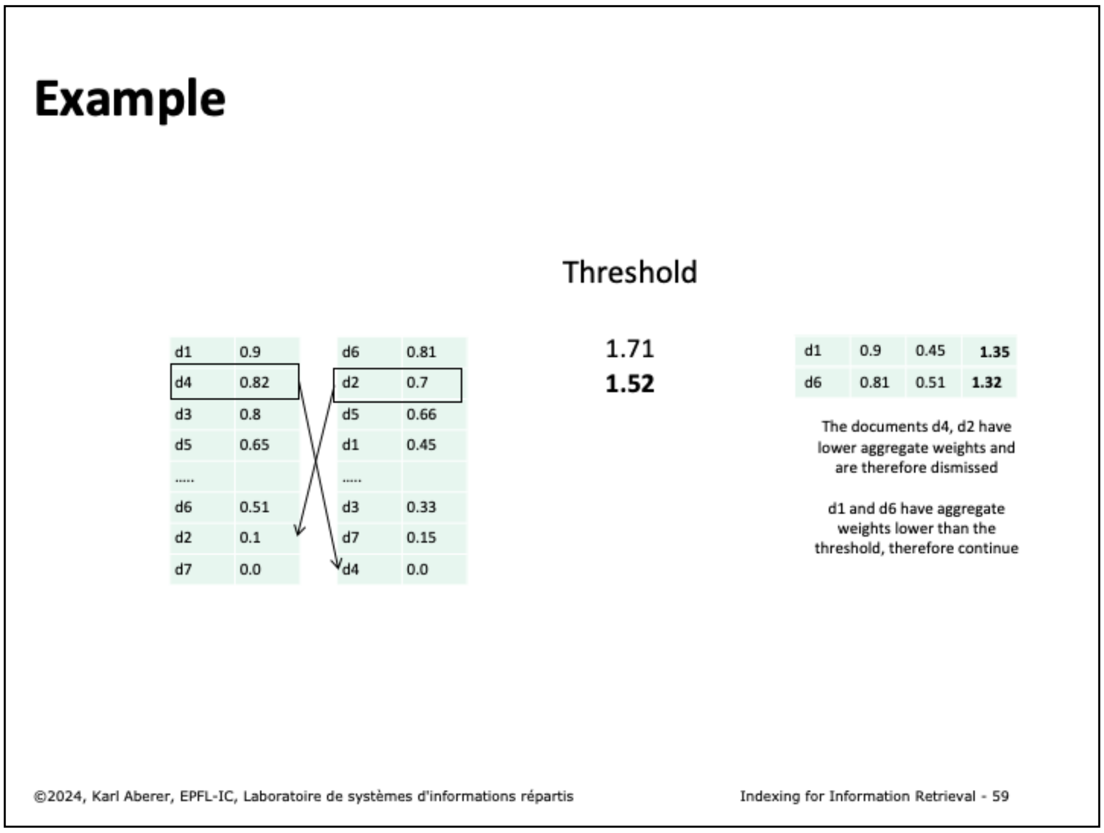
Threshold

1.71

d1	0.9	0.45	**1.35**
d6	0.81	0.51	**1.32**

d1 and d6 have aggregate weights lower than the threshold, therefore continue

We execute the threshold algorithm for the same example, as we did for Fagin's Algorithm.



Example

Threshold

d1	0.9
d4	0.82
d3	0.8
d5	0.65
.....	
d6	0.51
d2	0.1
d7	0.0

d6	0.81
d2	0.7
d5	0.66
d1	0.45
.....	
d3	0.33
d7	0.15
d4	0.0

1.71
1.52

d1	0.9	0.45	**1.35**
d6	0.81	0.51	**1.32**

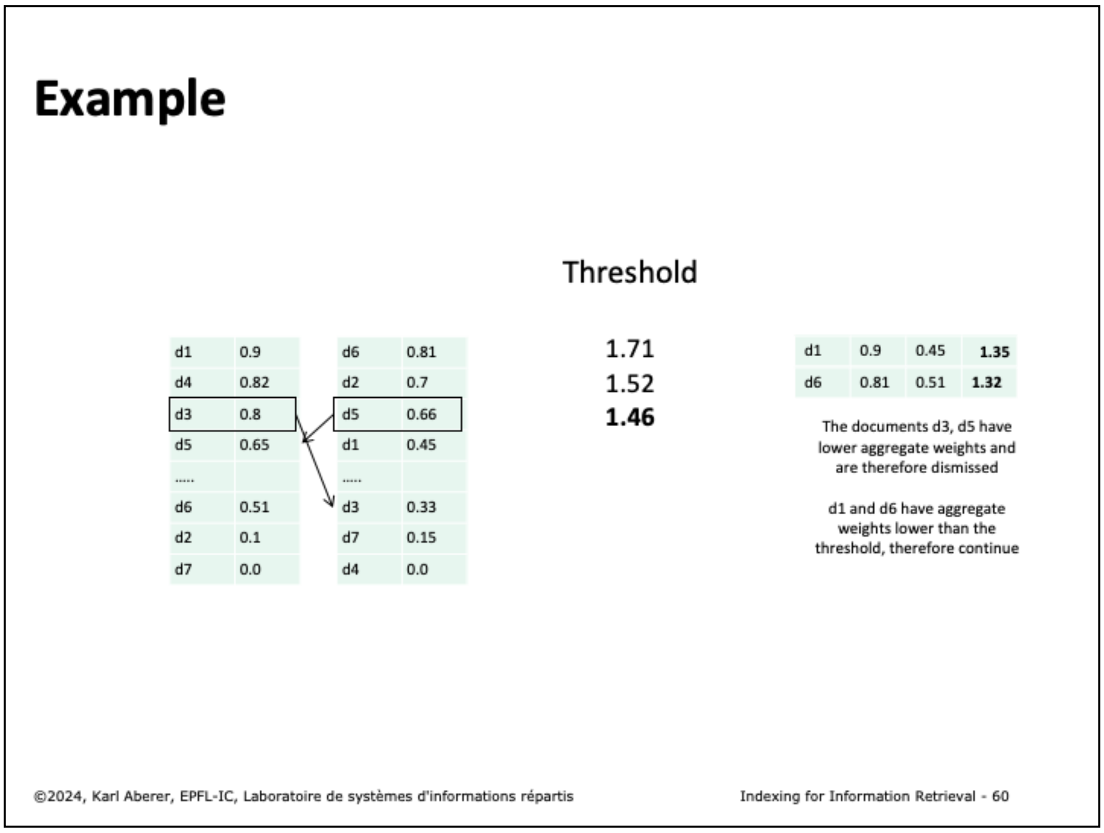
The documents d4, d2 have lower aggregate weights and are therefore dismissed

d1 and d6 have aggregate weights lower than the threshold, therefore continue

©2024, Karl Aberer, EPFL-IC, Laboratoire de systèmes d'informations répartis

Indexing for Information Retrieval - 59

Note that the threshold is always the aggregate value of the scores of the currently considered elements, in this case d4 and d2. In this round the weight of d4 is 0.82 and the wieght of d2 is 0.8. Both are lower than the weight of the current top-2 elements, d1 and d6. Therefore, d4 and d2 are dismissed.



Example

d1	0.9	d6	0.81
d4	0.82	d2	0.7
d3	0.8	d5	0.66
d5	0.65	d1	0.45
…..		…..	
d6	0.51	d3	0.33
d2	0.1	d7	0.15
d7	0.0	d4	0.0

Threshold

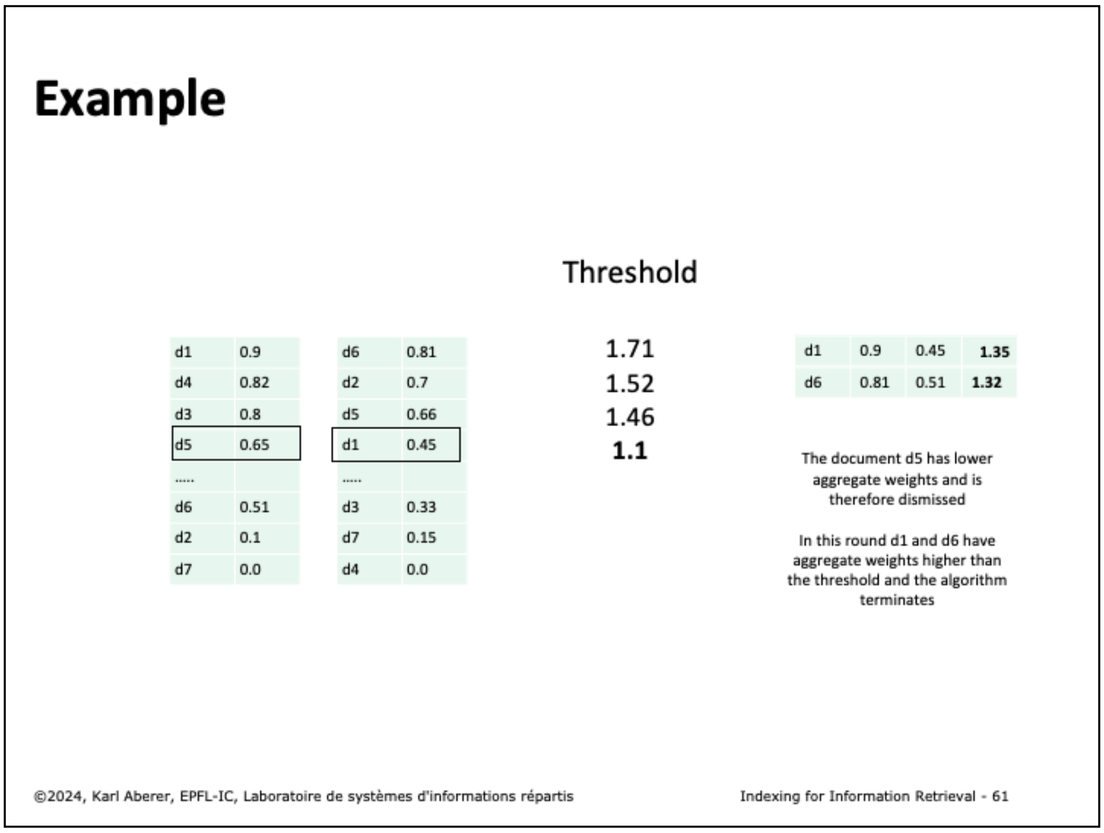
1.71
1.52
1.46

d1	0.9	0.45	**1.35**
d6	0.81	0.51	**1.32**

The documents d3, d5 have lower aggregate weights and are therefore dismissed

d1 and d6 have aggregate weights lower than the threshold, therefore continue

Also d3 and d5 can be dismissed in this round. Since the aggregate weights of the current top-2 elements are lower than the threshold, the algorithm continues.



Example

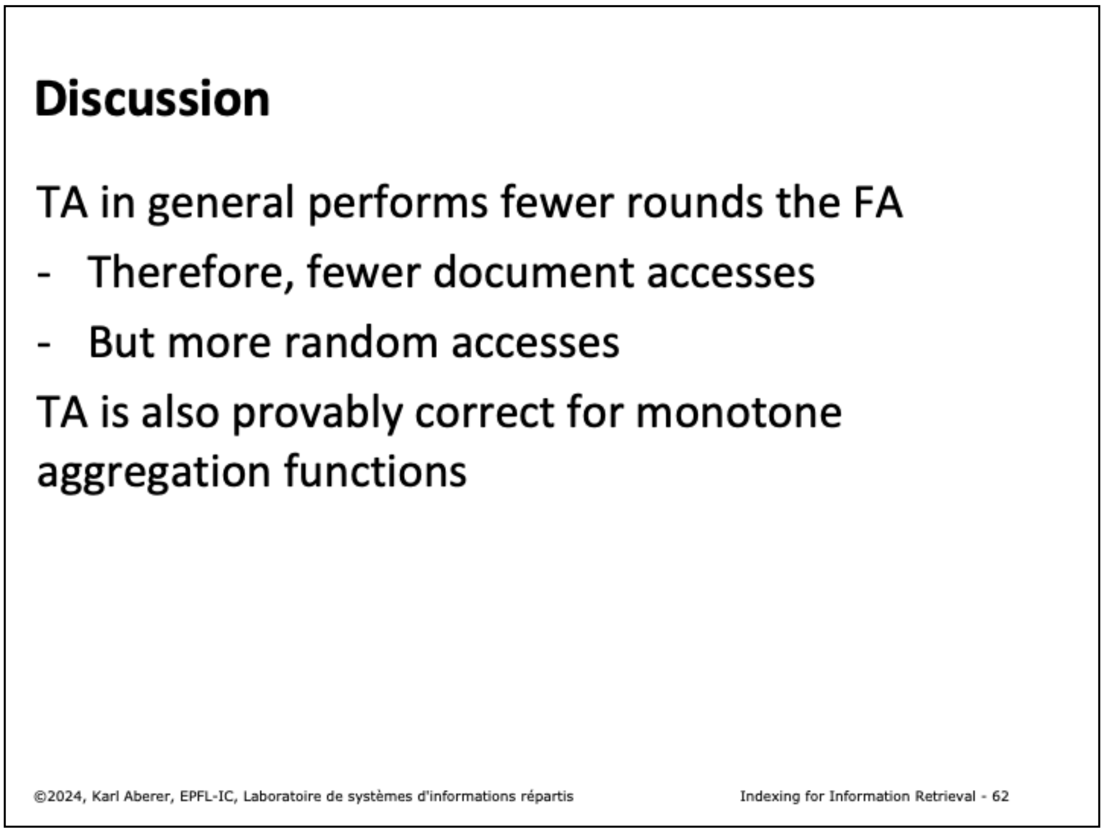
						Threshold
d1	0.9		d6	0.81		1.71
d4	0.82		d2	0.7		1.52
d3	0.8		d5	0.66		1.46
d5	0.65		d1	0.45		**1.1**
.....			.....			
d6	0.51		d3	0.33		
d2	0.1		d7	0.15		
d7	0.0		d4	0.0		

d1	0.9	0.45	**1.35**
d6	0.81	0.51	**1.32**

The document d5 has lower aggregate weights and is therefore dismissed

In this round d1 and d6 have aggregate weights higher than the threshold and the algorithm terminates

Finally the threshold has dropped below the weights of d1 and d6, and so the algorithm stops.



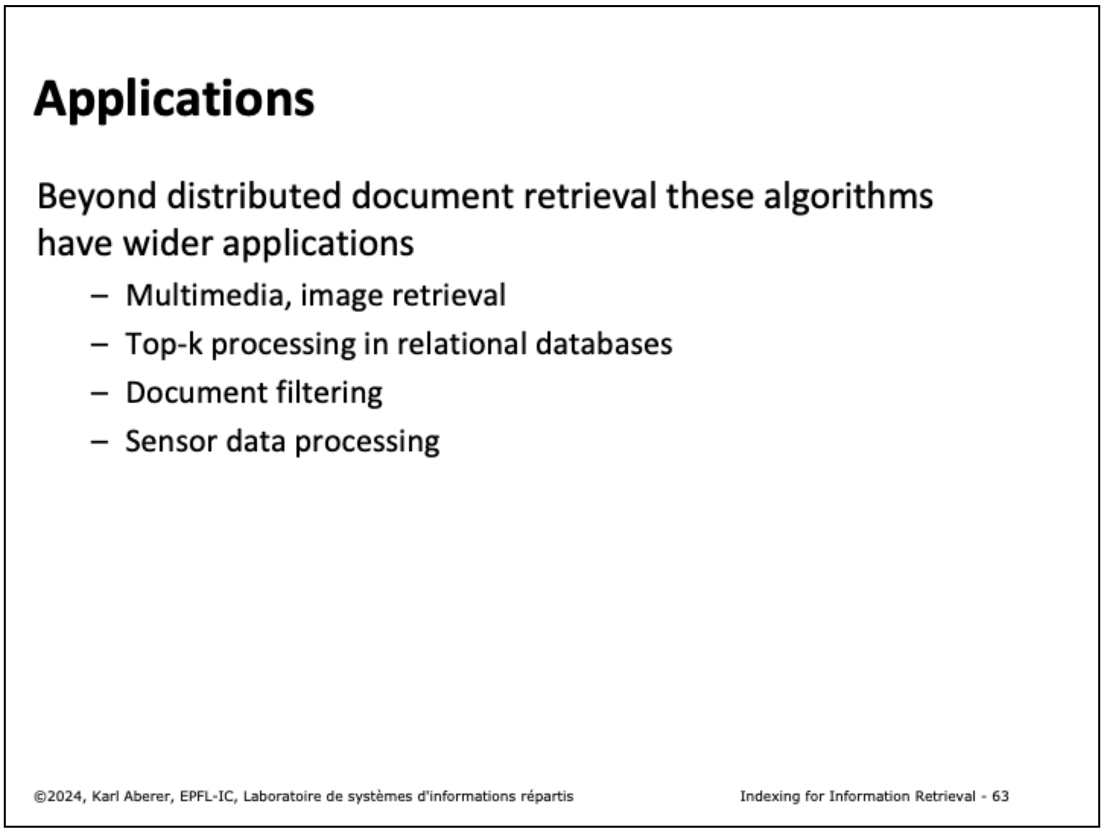
Discussion

TA in general performs fewer rounds the FA

- Therefore, fewer document accesses
- But more random accesses

TA is also provably correct for monotone aggregation functions

The threshold algorithm terminates faster than FA in general, at the expense of performing more random accesses.

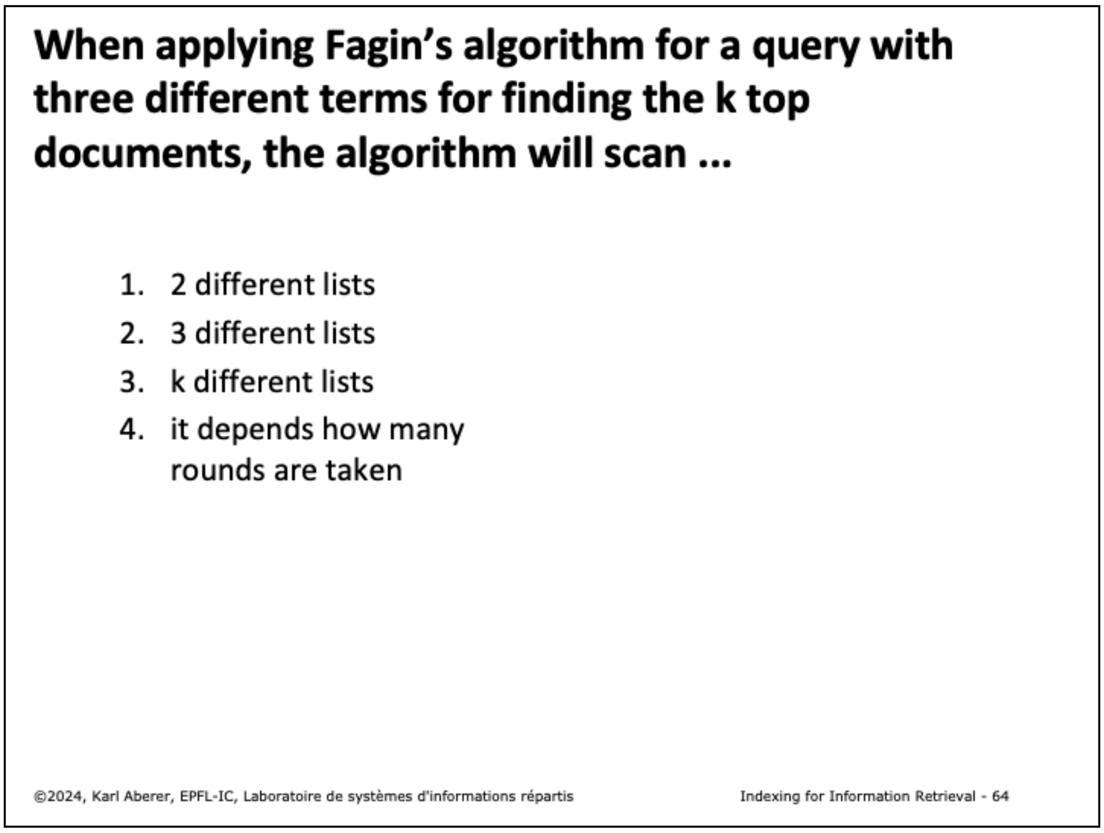


Applications

Beyond distributed document retrieval these algorithms have wider applications

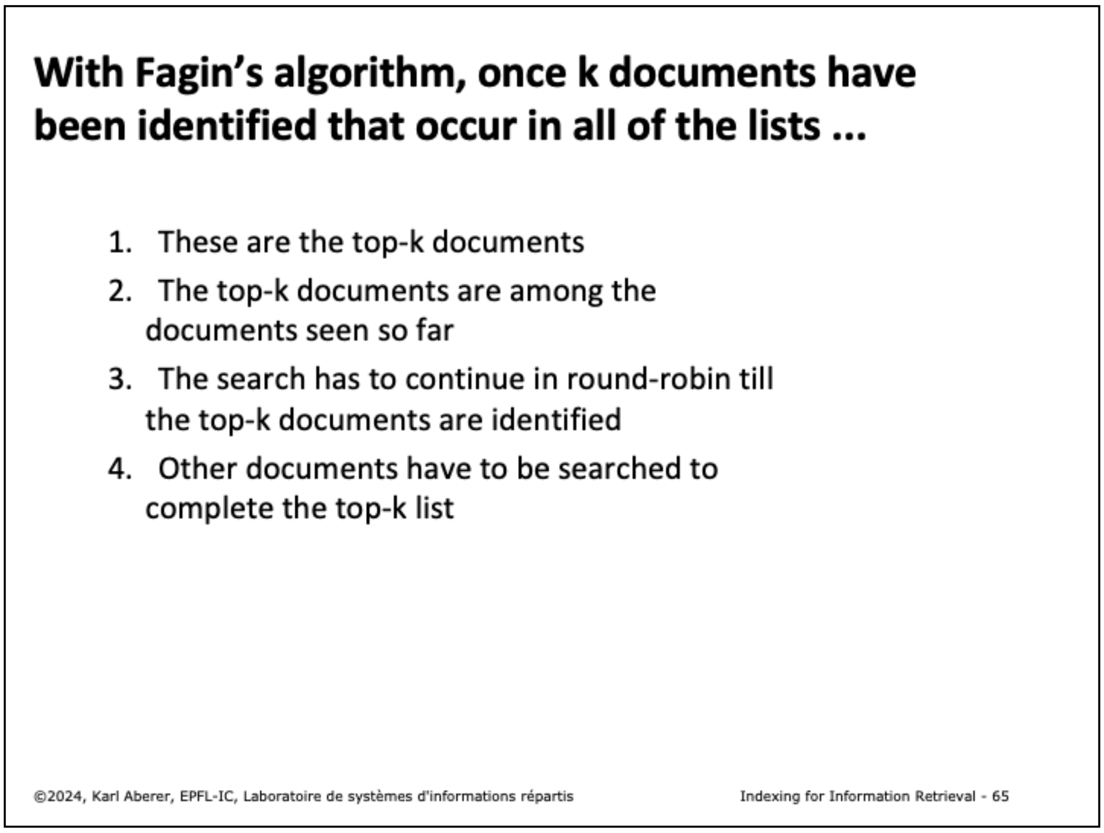
- Multimedia, image retrieval
- Top-k processing in relational databases
- Document filtering
- Sensor data processing

Fagin’s algorithm has found many applications apart from distributed retrieval. It is being used in multimedia retrieval (it’s original application), but also in processing data from relational databases (e.g., finding tuples with a highest combined value for multiple attributes), or sensor data processing.



When applying Fagin’s algorithm for a query with three different terms for finding the k top documents, the algorithm will scan ...

1. 2 different lists
2. 3 different lists
3. k different lists
4. it depends how many rounds are taken



With Fagin's algorithm, once k documents have been identified that occur in all of the lists ...

1. These are the top-k documents
2. The top-k documents are among the documents seen so far
3. The search has to continue in round-robin till the top-k documents are identified
4. Other documents have to be searched to complete the top-k list

References

Lin, J., Dyer, C. (2010). Inverted Indexing for Text Retrieval. In: Data-Intensive Text Processing with MapReduce. Synthesis Lectures on Human Language Technologies. Springer, Cham.

Ronald Fagin, Amnon Lotem, Moni Naor. Optimal aggregation algorithms for middleware, PODS 2001.

References

Course material based on

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008 (http://www-nlp.stanford.edu/IR-book/)

Relevant articles

- Boldi, Paolo, and Sebastiano Vigna. "The webgraph framework I: compression techniques." Proceedings of the 13th international conference on World Wide Web. ACM, 2004.
- Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce, Morgan & Claypool Synthesis, 2011.
- Lin, J., Dyer, C. (2010). Inverted Indexing for Text Retrieval. In: Data-Intensive Text Processing with MapReduce. Synthesis Lectures on Human Language Technologies. Springer, Cham.
- Ronald Fagin, Amnon Lotem, Moni Naor. Optimal aggregation algorithms for middleware, PODS 2001.