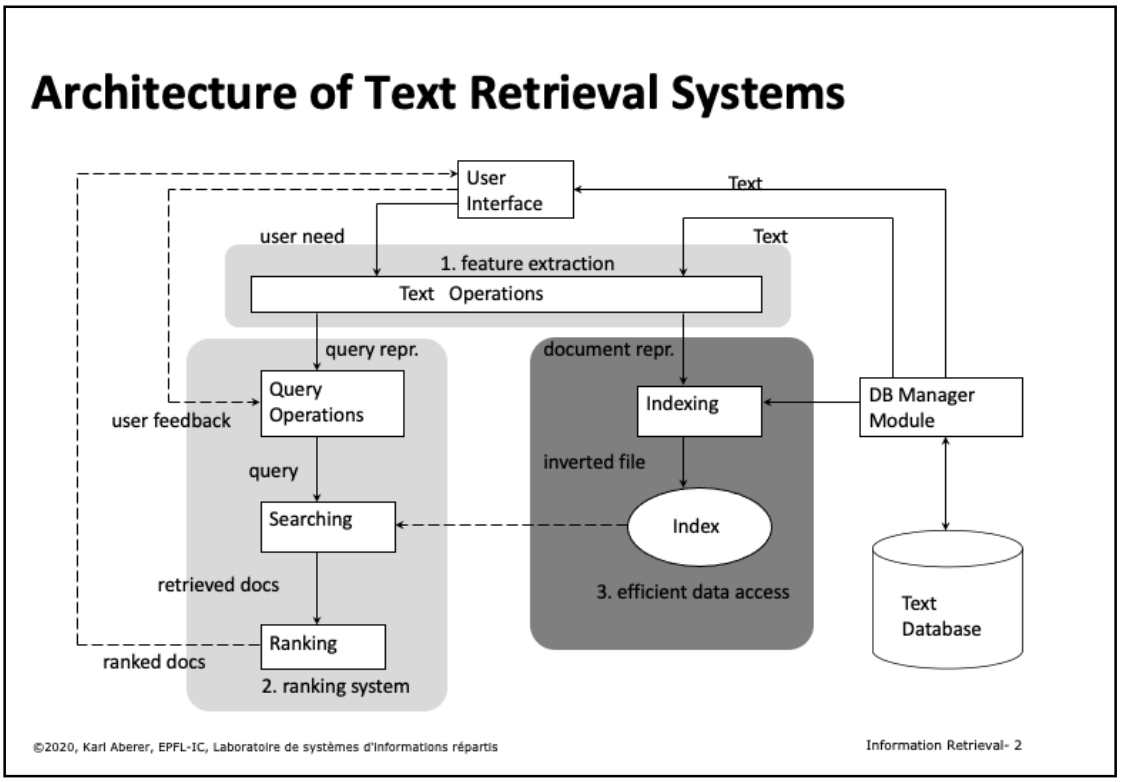
5. INDEXING FOR INFORMATION RETRIEVAL



This figure illustrates the basic architecture with the different functional components of a text retrieval system. We can distinguish three main groups of components:

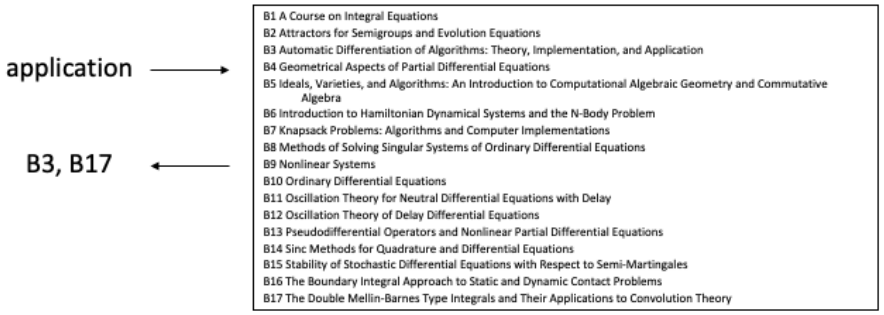
1. the feature extraction component: it performs text processing to turn queries and text documents into a keyword-based representation
2. the ranking system: it implements the retrieval model. In a first step user queries are potentially modified (in particular if user relevance feedback is used), then the documents required for producing the result are retrieved from the database and finally the similarity values are computed according to the retrieval model in order to compute the ranked result.
3. the data access system: it supports the ranking system by efficiently retrieving documents containing specific keywords from large document collections. The standard technique to implement this component is called **inverted files**.

In addition we recognize two components to interface the system to the user on the one hand, and to the data collection on the other hand.

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



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.

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 optimized for supporting search on relatively static text collections. For example, frequent updates are not supported with inverted files. This distinguishes inverted files from typical database indexing techniques, such as B+-Trees.

Inverted Files

Inverted list l_k for a term k

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

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

Inverted File: lexicographically ordered sequence of inverted lists

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

Inverted files are constructed by concatenating the inverted lists for all terms occurring in the document collection. Inverted lists enumerate all occurrences of the terms in documents, by keeping the document identifiers and the frequency of occurrence. Storing the frequency is useful for determining term frequency and inverse document frequency.

Example: Documents

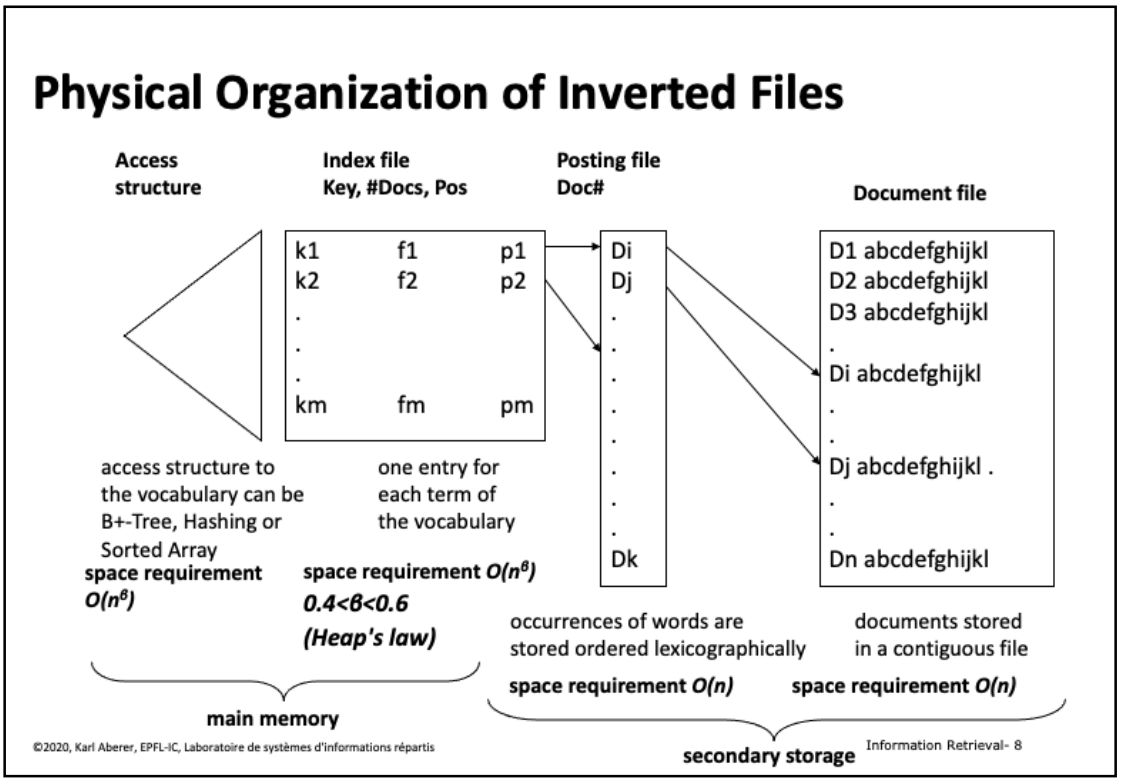
B1 A Course on Integral Equations
B2 Attractors for Semigroups and Evolution Equations
B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4 Geometrical Aspects of Partial Differential Equations
B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic 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 display the inverted list that is obtained for our example document collection.



Inverted files are a logical data structure, for which a physical storage organization needs to be designed. The physical organization has to take into account the quantitative characteristics of the inverted file structure. To that extent the 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 lower, 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. Index terms and their frequencies are stored in an index file that is kept in main memory. The access to this index file is supported by any suitable data access structure. Typically binary search, hash tables or tree-based structures, such as B+-Trees, or tries are used for that purpose. The posting files consist of the sequence of all term occurrences of the inverted file. The index file is related to the posting file by keeping for each index term a reference to the position in the posting file, where 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

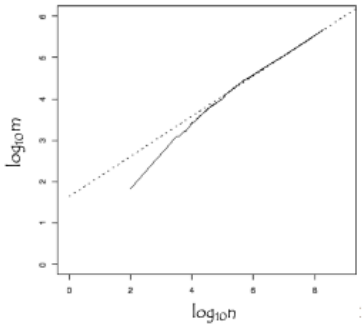
An empirical law that describes the relation between the size of a collection and the size of its vocabulary

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

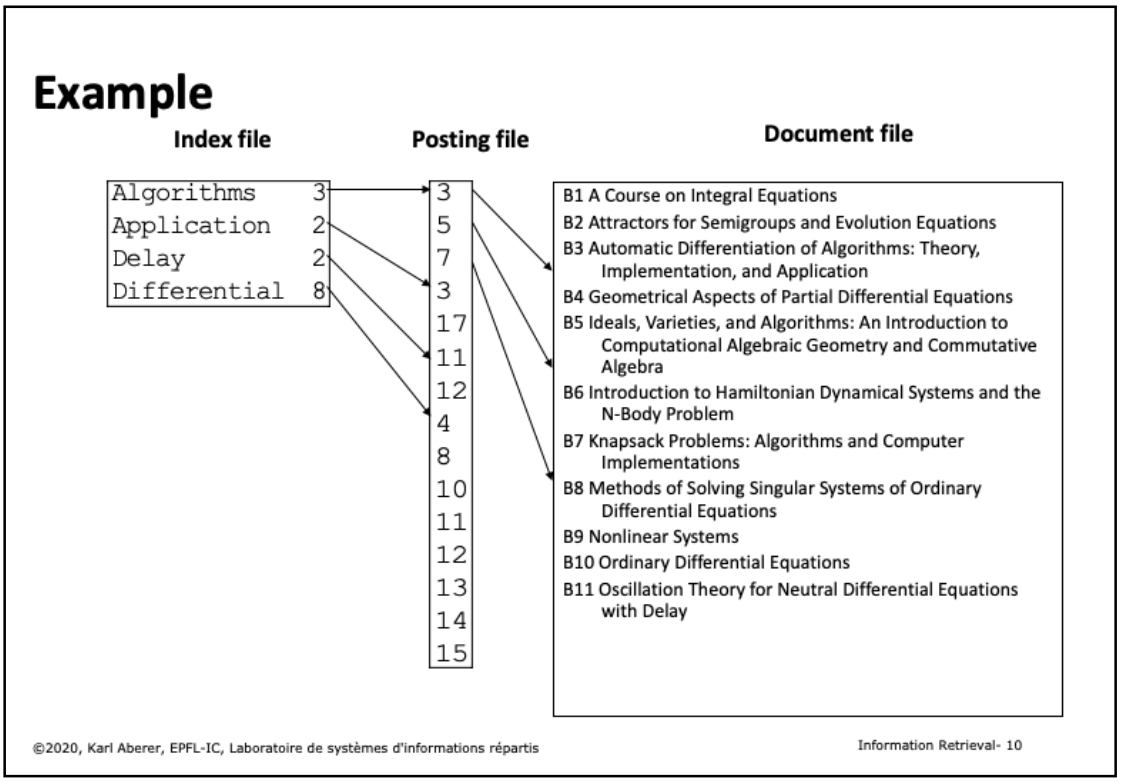
Typical values observed: $\beta \approx 0.5, 30 < k < 100$

Parameters depend on collection type and preprocessing

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



In fact, for a document collection of size n the number of occurrences of index terms is O(n), whereas 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 approximate m=100 * 10^3=10^5 index terms.



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

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).

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 it is not found, it is added to the vocabulary with an empty list of occurrences
- The word position is added to the end of its list of occurrences

The index construction is performed by first constructing dynamically a trie structure, in order to generate a sorted vocabulary and to collect the occurrences of index terms.

Construction of the Inverted File – Step 2

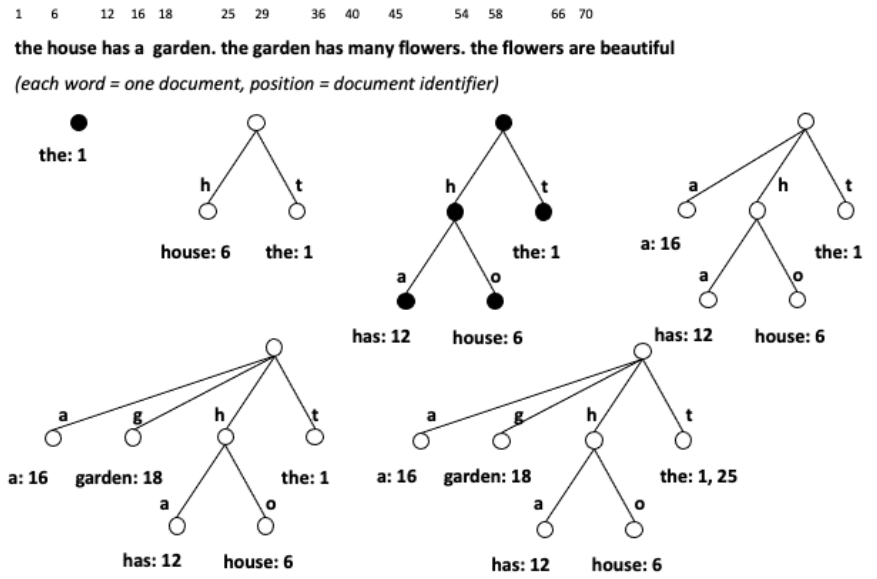
Step 2: Storage phase (once the text is exhausted)

- 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)

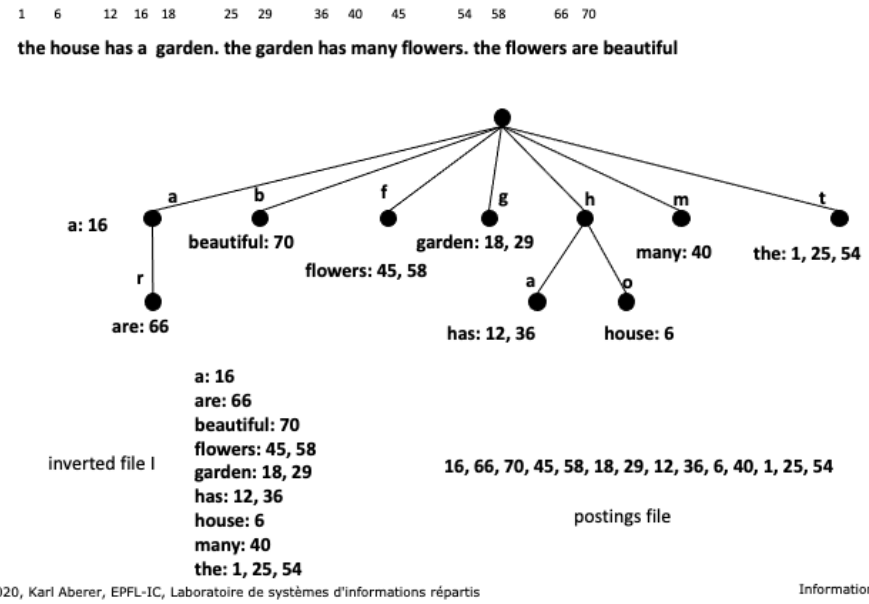
After the complete document collection has been traversed, the trie structure is sequentially traversed and the posting file is written to secondary storage. The trie structure itself can be used as a data access structure for the index file that is kept in main memory.

Example



In this example we consider each word of the text as a separate document identified by its position (for space limitations). 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



inverted file I

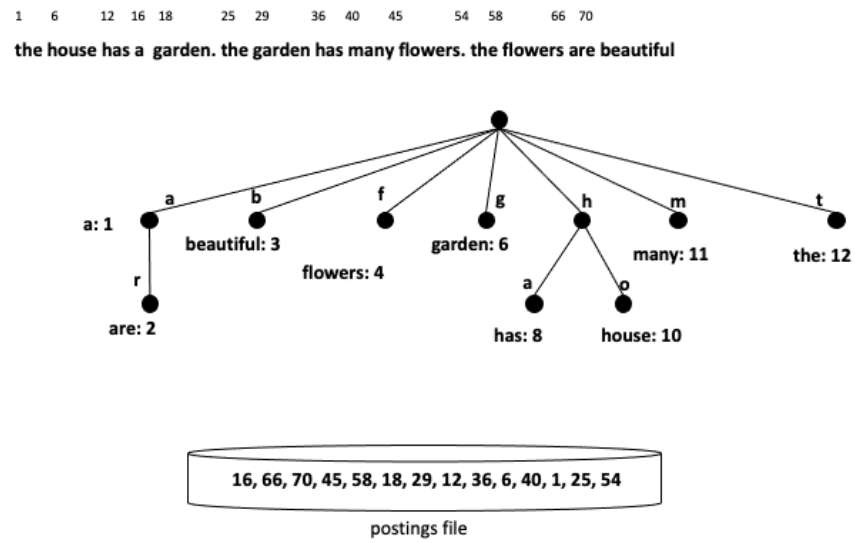
a: 16
are: 66
beautiful: 70
flowers: 45, 58
garden: 18, 29
has: 12, 36
house: 6
many: 40
the: 1, 25, 54

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

postings file

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. Analogously to the construction of the inverted file also the posting file can be derived.

Example



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 held in secondary storage.

Index Construction in Practice

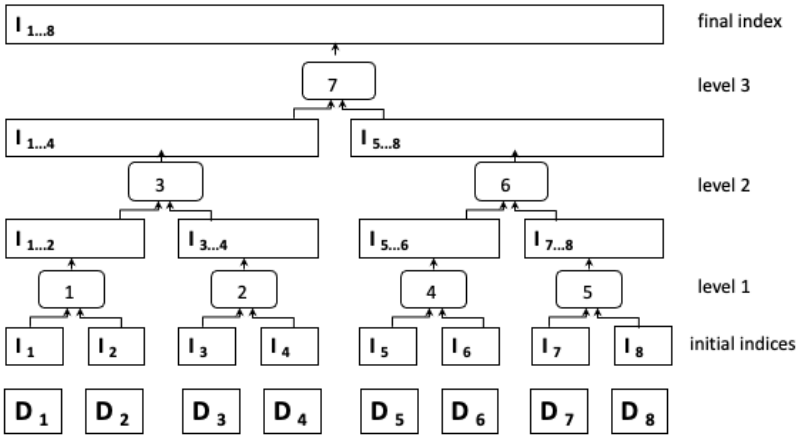
When using a single node not all index information can be kept in main memory → Index merging

- 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

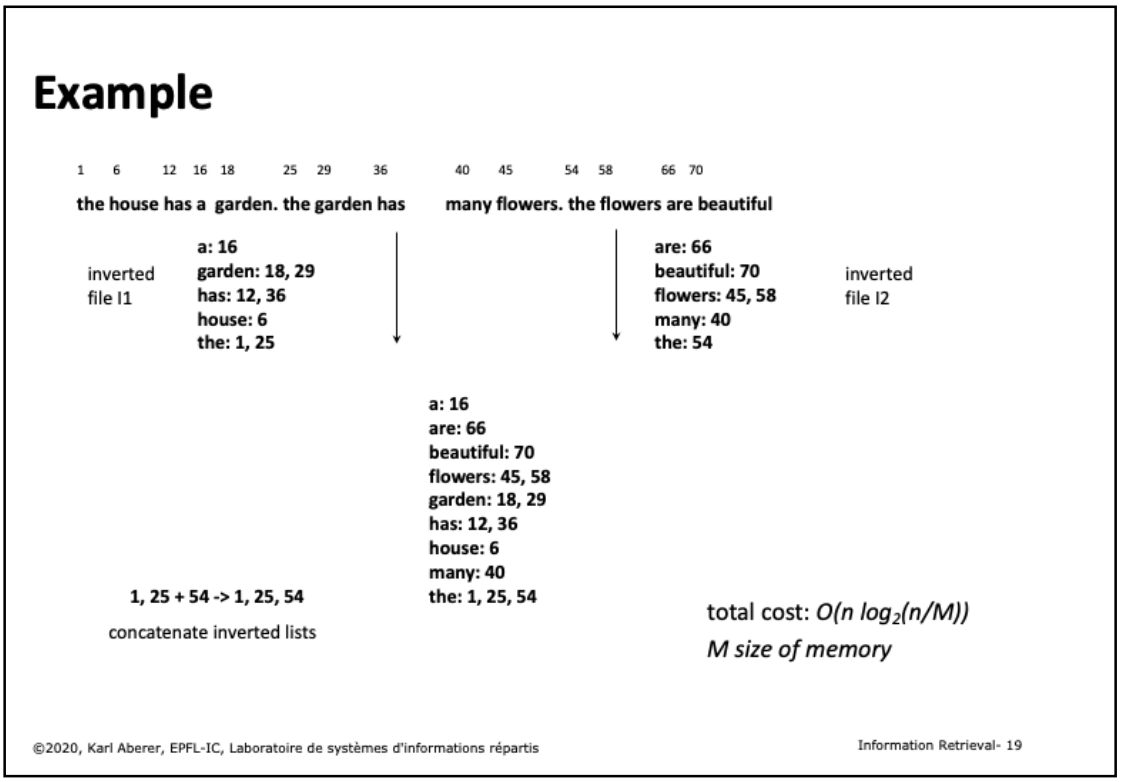
On a single node machine the index construction will be inefficient or impossible if the size of the trie structure with the associated posting lists exceeds the main memory space. Then the index construction process has to be partitioned in the following way: while the document collection is sequentially traversed, partial indices are written to the disk whenever the main memory is full. This results in a number 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 sequentially scanned on the disk, and while scanning the resulting index is written sequentially to the disk.



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 total 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 for example when the database is partitioned into two parts.

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

•Exact word position

•Occurrence within a document

•Occurrence within an arbitrary sized block = equally sized partitions of the document file spanning probably multiple documents

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.

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

$$l_k = \langle f_k : d_{i_1}, ..., d_{i_{fk}} \rangle \rightarrow$$

$$l_k' = \langle f_k : d_{i_1}, d_{i_2} - d_{i_1}, ..., d_{i_{fk}} - d_{i_{fk}-1} \rangle$$

- 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 large. 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.

Web-Scale Index Construction: 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 no longer 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 providers to scale up the 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

The map-reduce programming model is based on key-value pairs and lists of key value pairs (denoted by angle brackets here). The map function receives some input data (typically a piece of text to analyze or index), and produces a list of key-value pairs, that represent some partial results of the analysis (e.g. the counts of words in the text). 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.

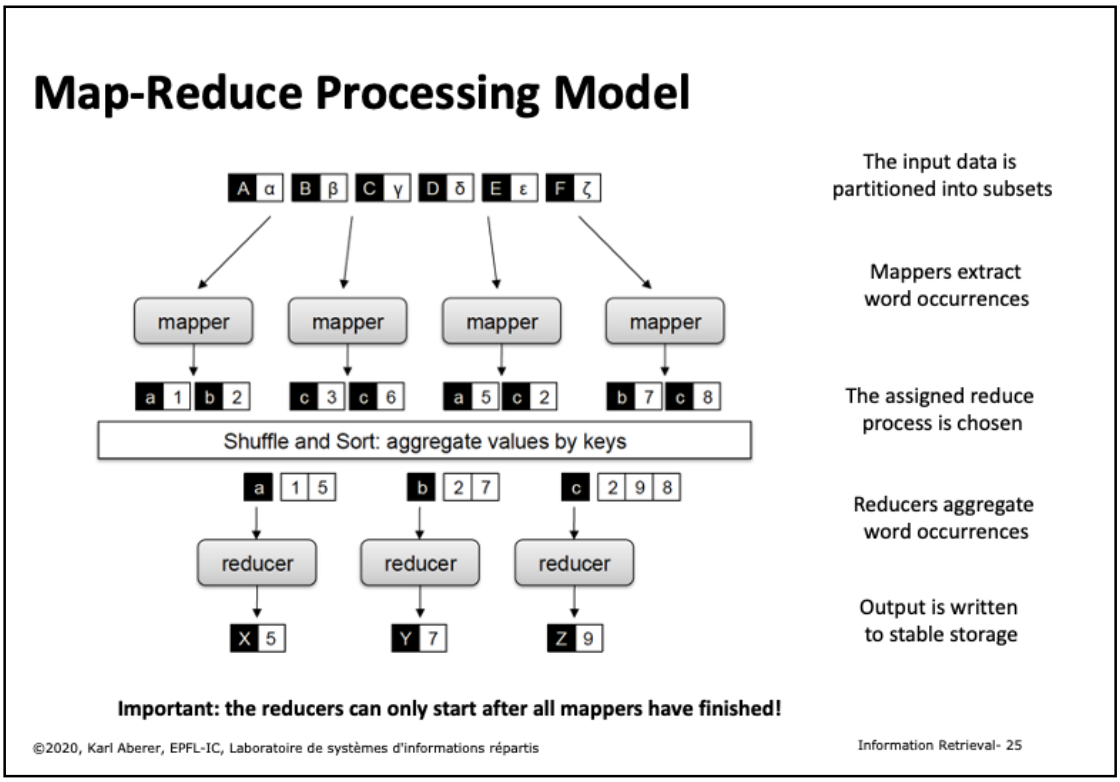
The reducer process receives as input all local results for a given key value, that have been computed by different mapper functions. It computes then an output value (e.g. the total count of words in the document corpus).

Example

Basic word counter program

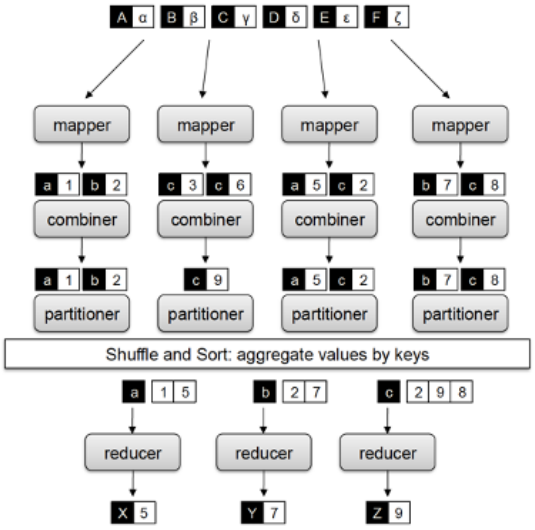
```
def mapper(document, line):
  for word in line.split(): output(word, 1)

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



This figure illustrates the basic steps of a map-reduce computation for the basic example of word counting. 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., a word, the mapper nodes send their local results for the word to the responsible reducer node. This can be controlled e.g. by hashngi the key values. The reducer nodes aggregate the statistics that they receive from all the mapper nodes. Once the reducer nodes have finalized 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.

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

What the Programmer Controls (and not)

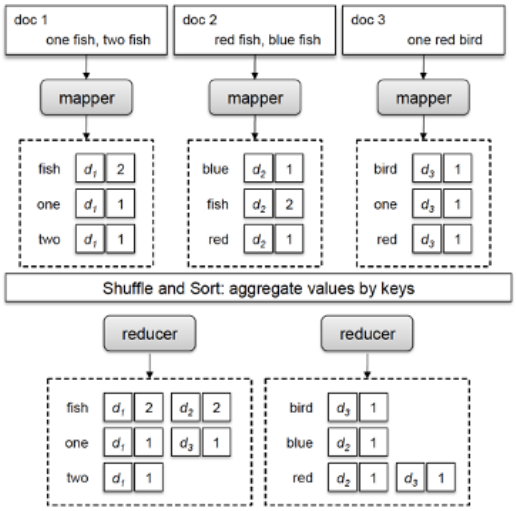
The programmer controls

- 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

Inverted File Construction Using Map-Reduce



Mappers extract postings from document

Postings are provided to reducers

Reducers aggregate posting lists

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)
```

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

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 Correct
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 Correct

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 Correct

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

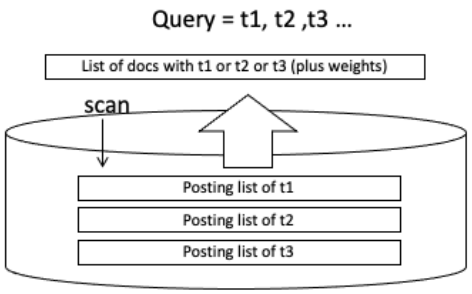
1. in the index merging approach for single node machines Correct
2. in the map-reduce approach for parallel clusters
3. in both
4. in neither of the two

6. DISTRIBUTED RETRIEVAL

Retrieval Processing

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)

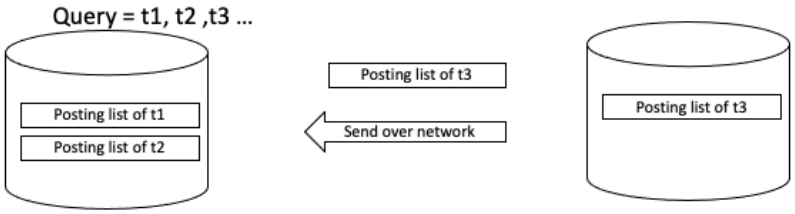


When using inverted files, a query involving multiple search terms requires the scanning of the postings lists of all terms. Typically in this process the term frequencies are computed for ALL documents in the document collection containing any of the query terms. In a centralized server this can be 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 have to be transferred over the network. Assuming that these postings lists can contain up to millions of entries, data in the order of megabytes needs to be transferred 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) for the results of a query, avoiding complete scans of posting lists.

Remark: in the following we will use k to indicate the number of results retrieved, despite the fact that we have used earlier k to denote the size of the vocabulary. The terminology top-k is so well established today, that it would be confusing to deviate here for notational consistency.

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!

One approach to deal with this problem is Fagin's algorithm. It has been originally developed 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 considered 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. In that case we would consider the tf-idf weights as the scores. Note that this assumption implies that an additional cost is occurred for sorting the posting lists (once). 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 is continued till k elements are detected that appear in all lists.

Phase 2: By then many other documents also may have been detected, but not in all lists. Thus in a next step the missing scores are retrieved from the lists. This requires random (and not scanning) access, e.g., 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 that have been identified in the Phase 1 as those k elements that occur in all lists. 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

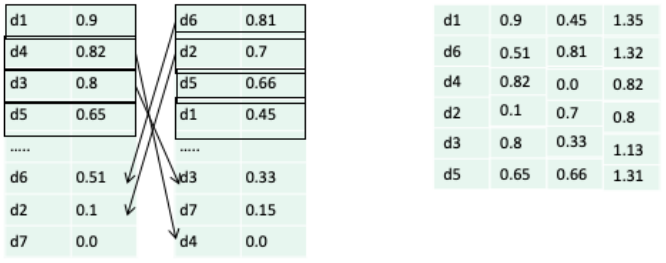
d6	0.81
d2	0.7
d5	0.66
d1	0.45
.....	
d3	0.33
d7	0.15
d4	0.0

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



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

Useful for many applications

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

Other Variants: threshold algorithm(s)

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).

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), sensor data processing, but also in text document filtering. Also alternative algorithms for solving the same problem have been proposed. They are known under the name of threshold algorithms. They work in a similar fashion, but have slightly different performance characteristics.

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

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 Correct
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

7. QUERY EXPANSION

Motivation

If the user query does not contain any relevant term, a corresponding relevant document will not show up in the result

Example: query “car” will not return “automobile”

How to add such documents (increase recall)?

Idea: System adds query terms to user query!

Users cannot predict or imagine all possible ways of how the concepts they are interested to find in their search can be expressed in natural language. This may have as a consequence, even under the vector space retrieval model, that relevant results are missed. This is, for the example, the case when there exist different synonyms (different terms with the same meaning). In the following we will see one possible approach to deal with this problem, namely extending the user query automatically by the system with additional terms.

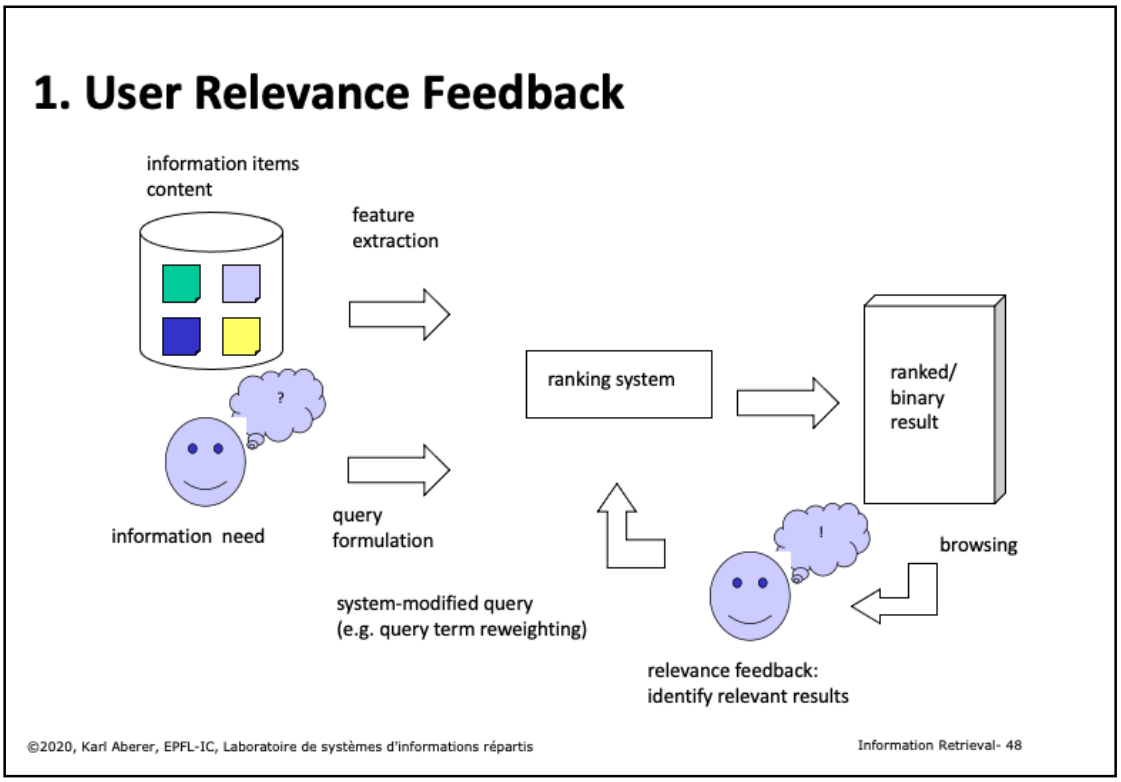
In a situation where the user has not fully specified the information need, also an automated approach such as LSI or word embedding will not be able to guess the users’s information need.

Two Methods for Extending Queries

1. Local Approach:
- Use information from **current query results**: *user relevance feedback*

2. Global Approach:
- Use information from a **document collection**: *query expansion*

In the following we will present two types of approaches to query extension, which are distinguished by the source of information used to identify new additional query terms. In the local approach the source of information is the current user query, respectively results produced by answering the user query. In the global approach the source of information is a existing document collection, either the documents that make up the corpus that is being queried by the user, or another, external collection of documents.

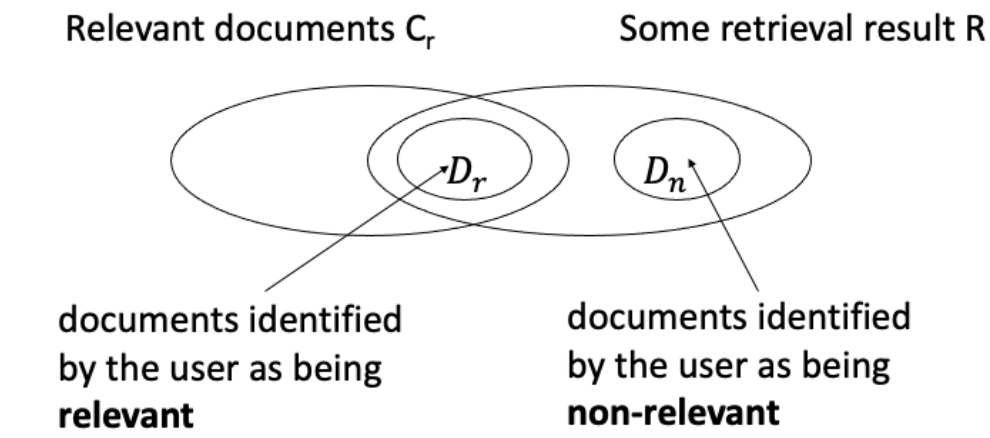


In general, a user does not necessarily know what is his information need and how to appropriately formulate a query. BUT usually a user can well identify relevant documents. Therefore the idea of user relevance feedback is to reformulate a query by taking into account feedback of the user on the relevance of already retrieved documents.

The advantages of such an approach are the following:

- The user is not involved in query formulation, but just points to interesting data items.
- The search task can be split up in smaller steps.
- The search task becomes a process converging to the desired result.

Feedback from Users



The general situation when receiving feedback from users can be depicted as follows: the retrieval system returns some result set R that is presented to the user. This result set overlaps with the set of relevant documents (C_r). The user can the identify within the result set both documents that are relevant and non-relevant. This gives the two feedback sets D_r and D_n.

Rocchio Algorithm

Rocchio algorithm: find a query that optimally separates relevant from non-relevant documents

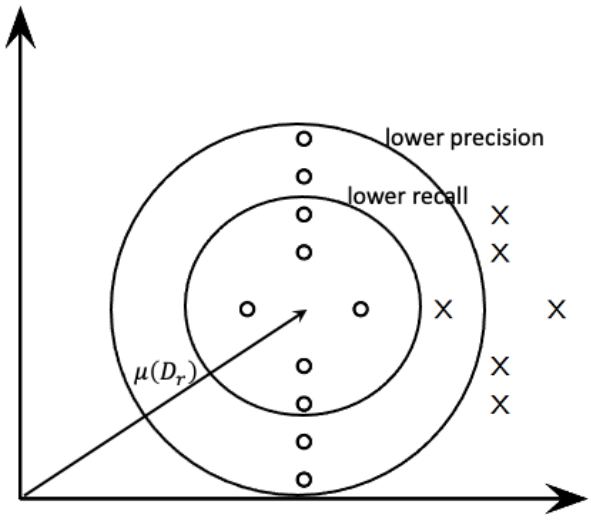
$$\vec{q}_{opt} = \arg\max_{\vec{q}} \left[\text{sim}\left(\vec{q}, \mu(D_r)\right) - \text{sim}\left(\vec{q}, \mu(D_n)\right)\right]$$

Centroid of a document set

$$\mu(D) = \frac{1}{|D|} \sum_{d \in D} \vec{d}$$

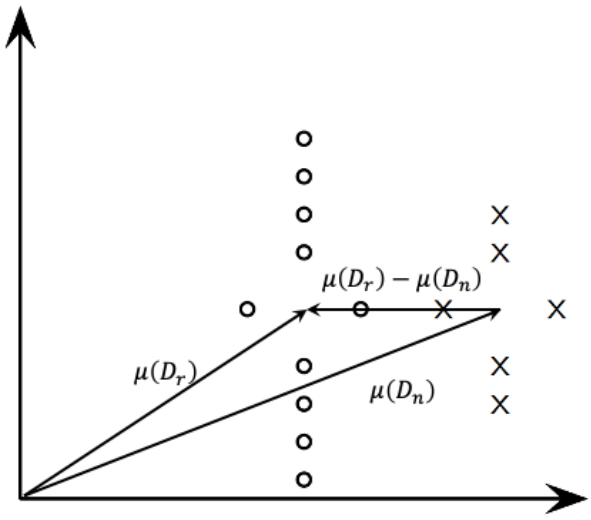
The basic idea for user relevance feedback was introduced by Rocchio. It is based on the observation, that the centroid of all document vectors of a document set D can be considered as the most characteristic representation of the document set. Then one could attempt to construct a query q_{opt} that optimally separates relevant from non-relevant documents. In order to achieve this, the query to be constructed has to have maximal similarity with the set of relevant documents, respectively its centroid, and maximal dissimilarity with the set of non-relevant documents, respectively its centroid. This can be achieved by finding a query that maximizes the difference among these two similarity values.

Illustration of Rocchio Algorithm

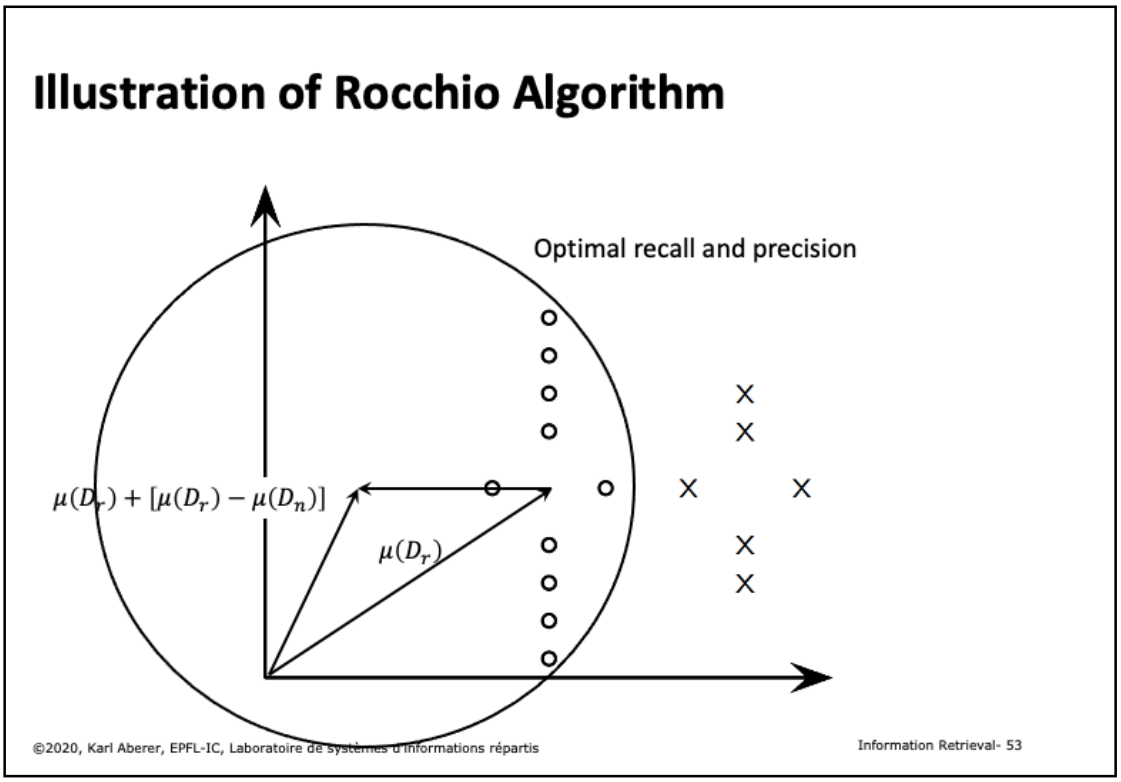


We now motivate of how the optimal query vector can be found with an illustration. Assume that the relevant documents are marked by circles, and the non-relevant documents are marked by crosses, and that the vector space has (only) 2 dimensions. When we consider the centroid of the relevant documents (which could be a potential query based on user relevance feedback) as a potential search query, then we see that we cannot achieve optimal precision and recall at the same time.

Illustration of Rocchio Algorithm



We therefore consider also the centroid of the non-relevant documents as part of the user relevance feedback. We compute the difference vector among the two centroids, and we will use this difference vector to "move away" the query from the non-relevant documents.



We add the difference vector to the centroid for the relevant documents. The resulting optimal query vector now can include all relevant documents in its result, without including non-relevant ones. In practice, such a clear separation will not always be possible, but it has been shown that under some additional assumptions, this method is the optimal way to constructing the optimal query vector.

Identifying Relevant Documents

Following the previous reasoning the optimal query is

$$\vec{q}_{opt} = \mu(D_r) + [\mu(D_r) - \mu(D_n)]$$

$\vec{q}_{opt} = [\mu(D_r) - \mu(D_n)]$ (under cosine similarity)

Practical issues

- User relevance feedback is not complete
- Users do not necessarily identify non-relevant documents
- Original query should continue to be considered

Constructing an optimal query vector as described before is only theoretically possible, since the complete information on relevant and non-relevant documents is lacking in practice. Therefore, the theoretical considerations put forward so far, serve as an intuition to devise a practical scheme, that is **approximating** the theoretical construction of an optimal query vector.

SMART: Practical Relevance Feedback

Approximation scheme for the theoretically optimal query vector

If users identify some relevant documents D_r from the result set R of a retrieval query q

- Assume all elements in $R \setminus D_r$ are not relevant, i.e., $D_n = R \setminus D_r$
- Modify the query to approximate theoretically optimal query

$$\vec{q}_{approx} = \alpha\vec{q} + \frac{\beta}{|D_r|}\sum_{\vec{d}_j \in D_r}\vec{d}_j - \frac{\gamma}{|R \setminus D_r|}\sum_{\vec{d}_j \notin D_r}\vec{d}_j$$

- α, β, γ are tuning parameters, $\alpha, \beta, \gamma \geq 0$
- Example: $\alpha = 1$, $\beta = 0.75$, $\gamma = 0.25$

The approximation scheme for user relevance feedback is called SMART. It starts from the assumption that users have identified some relevant documents. Then the scheme assumes that all other documents should be considered as non-relevant. This results in a modification of the original query that is controlled by 3 tuning parameters.

Since this is of course not correct, two mechanisms are used to moderate the impact of this wrong assumption:

1. The original query vector is maintained, in order not to drift away too dramatically from the original user query.
2. The weight given for the modification using the centroid of non-relevant documents is generally kept lower than the weight for the centroid of the relevant documents, as their non-relevance is just an assumption made, and not based on real user relevance feedback.

Example

Query q= "application theory"

Result

0.77: B17 The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory
0.68: B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
0.23: B11 Oscillation Theory for Neutral Differential Equations with Delay
0.23: B12 Oscillation Theory of Delay Differential Equations

Query reformulation

$$\vec{q}_{approx} = \frac{1}{4}\vec{q} + \frac{1}{4}\vec{d}_3 - \frac{1}{12}(\vec{d}_{17} + \vec{d}_{12} + \vec{d}_{11}), \alpha = \beta = \gamma = \frac{1}{4}$$

Result for reformulated query

0.87: B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
0.61: B17 The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory
0.29: B7 Knapsack Problems: Algorithms and Computer Implementations
0.23: B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra

This example shows how the query reformulation works. By identifying document B3 as being relevant and modifying the query vector it turns out that new documents (B5 and B7) become relevant. The reason is that those new documents share terms with document B3, and these terms are newly considered in the reformulated query.

Discussion

Underlying assumptions of SMART algorithm

1. Original query contains sufficient number of relevant terms
2. Results contain new relevant terms that co-occur with original query terms
3. Relevant documents form a single cluster
4. Users are willing to provide feedback (!)

All assumptions can be violated in practice

Practical considerations

- Modified queries are complex → expensive processing
- Relevance Feedback consumes user time → could be used in other ways

Concerning the first assumption, if the initial query of the user does not contain sufficient information to retrieve a sufficient number of documents that are relevant to the true interest of the user (i.e. have sufficient recall), the relevance feedback system will not be able to produce sufficient relevant documents with additional terms.

Concerning the second assumption, new terms can only be included as part of the modified query, if they co-occur at least in some documents together with original query terms. Otherwise, these terms could never be part of relevant documents in the result of the original query (why?).

Concerning the third assumption, implicitly the SMART algorithm assumes that all relevant documents are part of one cluster in the vector space. If they form multiple clusters, it is not able to correctly produce a query that can retrieve the relevant documents.

Can documents which do not contain any keywords of the original query receive a positive similarity coefficient after relevance feedback?

1. No
2. Yes, independent of the values β and γ
3. Yes, but only if $\beta > 0$ Correct
4. Yes, but only if $\gamma > 0$

Which year Rocchio published his work on relevance feedback?

A. 1965 Correct

B. 1975

C. 1985

D. 1995

Pseudo-Relevance Feedback

If users do not give feedback, automate the process

- Choose the top-k documents as the relevant ones
- Apply the SMART algorithm

Works often well

- But can fail horribly: query drift

2. Query Expansion

Query is expanded using a global, *query-independent* resource

- Manually edited thesaurus
- Automatically extracted thesaurus, using term co-occurrence
- Query logs

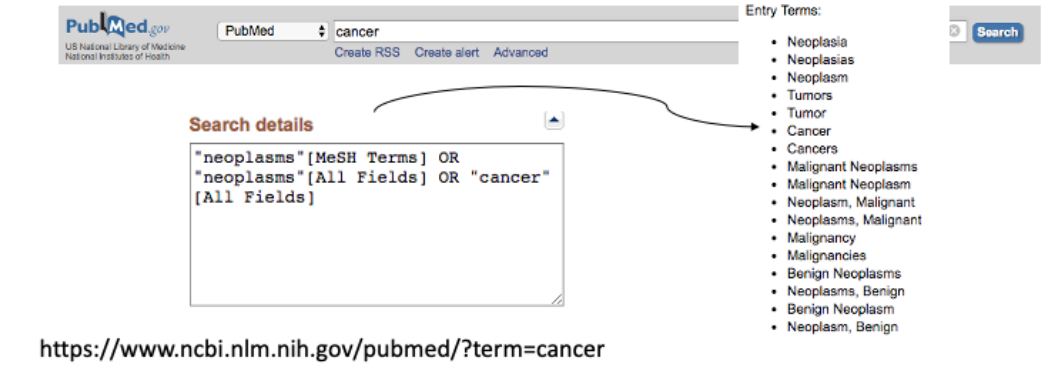
Global methods for expanding user queries can rely on a variety of resources. These may include thesauri (a thesaurus is a database that contains (near-) synonyms) that are manually constructed or automatically derived, or the automated analysis of query logs.

Manually Created Thesaurus

Expensive to create and maintain

- Used mainly in science and engineering

Example: Pubmed



https://www.ncbi.nlm.nih.gov/pubmed/?term=cancer

Performing query expansion using a manually thesaurus requires the (expensive) effort of creating and maintaining such a thesaurus. This task is mainly performed in highly specialized technical fields in science and engineering. One prominent example of such a Thesaurus is maintained by Pubmed, the biggest publication database for medical literature maintained by the NIH, the National Institute of Health in the US. When using its search engine, you will find a window ”Search details” that shows how the user query is automatically expanded using the Pubmed thesaurus. In this example we see that the search system identifies that “cancer” is an entry on the concept “neoplasms”, and thus extends the query with all entries that it finds associated in the thesaurus (e.g. it would also search for “tumor”).

Automatic Thesaurus Generation

Attempt to generate a thesaurus automatically by analyzing the distribution of words in documents

Fundamental notion: *similarity between two words*

Definition 1:

Two words are similar if they **co-occur** with similar words. "switzerland" ≈ "austria" because both occur with words such as "national", "election", "soccer" etc., so they must be similar.

Definition 2:

Two words are similar if they occur in a given **grammatical relation** with the same words. "live in *", "travel to *", "size of *" are all phrases in which both "switzerland" or "austria" can occur

In order to avoid the effort of manually creating a thesaurus one can attempt to create it automatically by studying large numbers of documents and the distribution of words in those. This leads to the concept of word similarity. There exists two basic methods to study this similarity, either purely statistically, by observing which words occur together in documents, or in a more accurate way by identifying whether the words occur in the same grammatical relationships.

For the first approach we study so-called "word embeddings". For the second approach we will learn about methods of "information extraction".

Example

Terms related to “cat”

Name	Literals	Categories	Similarity
keyword	cat		1.000
keyword	cats		0.799
dog	en: "dog", "hound"		0.769
keyword	kitten		0.755
pet	en: "pet"		0.733
keyword	kitty		0.688
keyword	dogs		0.677
keyword	puppy		0.629
animal	en: "animal", "beast", "brute", "creature",	animal	0.626
keyword	kittens		0.622
keyword	pets		0.610
rabbit	en: "hare", "lapin", "rabbit"		0.602
bird	en: "bird", "birdie", "fowl"	animal	0.595

This example illustrates of how such automatic thesaurus generation based on statistical analysis looks in practice. A statistical analysis would allow to compute similarity of words. With such a similarity measure it is possible to search for related words (just like searching for documents with an information retrieval system). As the example shows, such a search reveals immediately many terms directly related with the original word, which was “cat”.

Epxansion using Query Logs

Main source of query expansion at search engines

- Exploit correlations in user sessions

Example 1: users extend query

- After searching “Obama”, users search “Obama president”
- Therefore, ”president” might be a good expansion

Example 2: users refer to same result

- User A accesses URL epfl.ch after searching “Aebischer”
- User B accesses URL epfl.ch after searching “Vetterli”
- “Vetterli” might be a potential expansion for the query “Aebischer” (and vice versa)

Query logs contain potentially rich information for query expansion. There a numerous ways of how such knowledge can be exploited. We show here two possible examples. Other methods rely on mining query logs using various techniques, including clustering and association rule mining, that we will encounter in the later part on data mining.

References

Course material based on

- Ricardo Baeza-Yates, Berthier Ribeiro-Neto, Modern Information Retrieval (ACM Press Series), Addison Wesley, 1999.
- Lin, J., & Dyer, C. (2010). Data-intensive text processing with MapReduce. Synthesis Lectures on Human Language Technologies, 3(1), 1-177.

Papers

- Fagin, R., Lotem, A., & Naor, M. (2003). Optimal aggregation algorithms for middleware. Journal of computer and system sciences, 66(4), 614-656.
- Ponte, Jay Michael, and W. Bruce Croft. "A language modeling approach to information retrieval." PhD diss., University of Massachusetts at Amherst, 1998.