4. PROBABILISTIC INFORMATION RETRIEVAL

Probabilistic Information Retrieval

The notion of similarity in the vector space model does not directly imply relevance

- The similarity values have no interpretation, they are just used to rank
- An information retrieval model deals with uncertainty on the users information needs
- Probability theory provides a principled approach to reason about this uncertainty

Probabilistic IR models attempt to directly model relevance as a probability

One of the key drawbacks of the vector space retrieval model is the lack of interpretability of the similarity values. This gave rise to the development of probabilistic retrieval models, that attempt to "compute" relevance as a probability.

Query Likelihood Model

Given query q, determine the probability $P(d|q)$ that document d is relevant to query q

Bayes Rule $P(d|q) = \frac{P(q|d)P(d)}{P(q)}$

Assumptions

- $P(d)$, the probability of a document occurring is uniform across a collection
- $P(q)$ is the same for all documents

Thus: $P(d|q)$ can be derived from $P(q|d)$

The problem of retrieval can be understood in a probabilistic setting as the problem of determining the probability of a document d being relevant, given a query q. We observe that the probability of a document to occur in a collection is constant (which makes sense assuming all documents are different), and the probability of a query to occur is the same for all documents. Thus, using Bayes rule, the problem of determining whether a document is relevant for a query is equivalent to the problem of determining whether a query is relevant to a document. The latter probability P(q|d) is also called the query likelihood.

Language Modeling

Query likelihood: determine $P(q|d)$

Assume each document d is generated by a Language Model M_d

- a language model is a mechanism that generates the words of the language

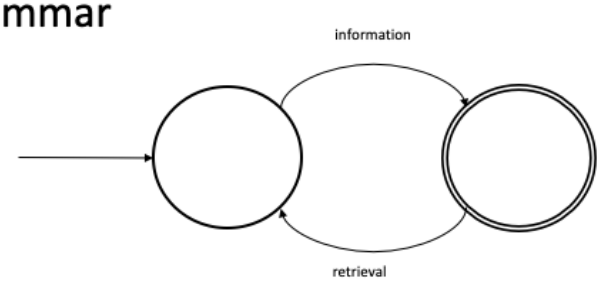
Then $P(q|d)$ can be interpreted as the probability that the query q was generated by the language model M_d

The notion of query likelihood gives now rise to the following approach to model relevance. We assume that documents are the result of language model. A language model is a (in general probabilistic) process that produces text, and a given document d is assumed to be produced by its specific language model M_d. Then the problem of retrieval can be viewed in the following way: if a query is relevant to a document, it should have been produced by the same language model as the document. Using this argument, the query likelihood corresponds to the probability that the query has been produced by the same language model as the document.

Let's have now a more detailed look in what a language model is and how we use it implement this intuitive model practically.

What is a Language Model?

Deterministic language model = automaton = grammar



This model can produce:
- information retrieval
- information retrieval information retrieval

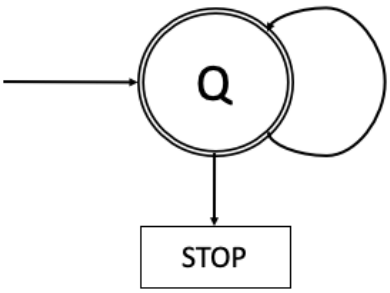
It cannot produce:
- retrieval information

In the simplest case a language model is a deterministic automaton. In theoretical computer science deterministic automatons are those that can recognize (or produce) regular languages.

Probabilistic Language Model

Unigram model: assign a probability to each term to appear

- More complex models can be used, e.g., bigrams



Model M_1		Model M_2	
STOP	0.2	STOP	0.25
the	0.2	the	0.15
a	0.1	a	0.12
frog	0.03	frog	0.0002
toad	0.03	toad	0.0001
said	0.02	said	0.01
likes	0.015	likes	0.01
dog	0.01	dog	0.04

Two different language models derived from 2 documents

Instead of using a deterministic automaton, we can also use a probabilistic state automaton, in other words, a Markov process. In the simplest case the automaton has a single state, and every state transition emits with a certain probability one term out of a vocabulary. In addition, the automaton can stop with a certain probability. The table captures the transition probabilities of two possible models M1 and M2. In the two models, the probability to stop is given as P(STOP|Q) = 0.2.

Probability to Create a Query

What is the probability that a query q has been generated by model M

Example: q = the frog said dog STOP

$P(q|M_1)$ = 0.2 * 0.03 * 0.02 * 0.01 * 0.2 = 0.00000024

So retrieval becomes the problem of computing for a query q the probability $P(q|M_d)$ for all the documents d

Given a language model for the generation of documents, we can now compute within that model the probability that a given query q has been generated by the model of a document d. We give one example showing such a computation. With this approach we are now ready to compute query likelihood for all documents of a document collection.

Learning and Using the Model

Learning the model: Maximum Likelihood Estimation (MLE) of probabilities under Unigram Model

$$\hat{P}_{mle}(t|M_d) = \frac{tf_{t,d}}{L_d}$$

where

- $tf_{t,d}$ is the number of occurrences of t in d (term frequency)
- L_d is the number of terms in the document (document length)

Using the model

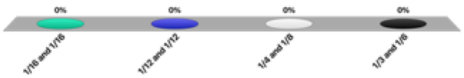
$$\hat{P}(q|M_d) = \prod_{t \epsilon q} \hat{P}_{mle}(t|M_d)$$

For applying the probabilistic retrieval method described before, we need first to learn the language model of each document. The learning is performed using Maximum Likelihood Estimation (MLE). In the case of the unigram model, this is a straightforward task. We just estimate the term probabilities by counting the document frequencies and normalizing by document length. When using the model for a query q, we then use those estimates, to estimate the relevance of a query for the document, as illustrated before.

Consider the document:
"Information retrieval is the task of finding the documents satisfying the information needs of the user"

Using MLE to estimate the unigram probability model, what is $P(\text{the}|M_d)$ and $P(\text{information}|M_d)$?

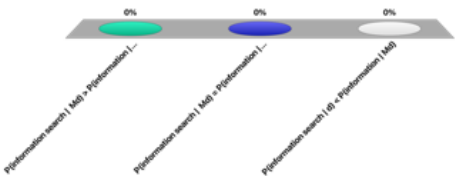
1. 1/16 and 1/16
2. 1/12 and 1/12
3. 1/4 and 1/8
4. 1/3 and 1/6



Consider the following document

d = “information retrieval and search”

1. P(information search | M_d) > P(information | M_d)
2. P(information search | M_d) = P(information | M_d)
3. P(information search | d) < P(information | M_d)



Issues with MLE Estimation

Problem 1: if query contains a term not occurring in the document $\hat{P}(q|M_d) = 0$!

Problem 2: this is an estimation! A term that occurs once, might have been "lucky", whereas another one with same probability to occur is not contained in the document

➢ need to give non-zero probability to unseen terms!

Applying the afore mentioned approach to estimate relevance of a document to a query has a practical problem: if the query contains a term not occurring in the document the estimated probability will be unavoidably zero, since one of the factors of the product computing that probability will be zero. In other words, the query cannot be generated by the document model, thus the document is not relevant to the query. This is not only impractical, but also not meaningful from a more theoretical perspective. Since we used MLE to generate the model, we were using the statistics of one specific document, that has been generated by a potentially complex model, that may contain other terms that just were not generated for this document.

Smoothing

Idea: add a small weight for non-occurring terms in a document, that is smaller than the normalized collection frequency

$$\hat{P}(t|M_c) \leq cf_t/T$$

where

- cf_t = number of times term t occurs in collection
- T = total number of terms in collection

Smoothed estimate

$$\hat{P}(t|d) = \lambda \hat{P}_{mle}(t|M_d) + (1-\lambda)\hat{P}_{mle}(t|M_c)$$

M_c = language model of the whole collection

λ = tuning parameter

To fix the aforementioned problem an approach called smoothing is applied. The basic idea is to assume that in fact every term potentially could occur in the document generated by its document model, including those that are not part of the actual document; only that the probability of terms not seen in the document is presumably less likely to occur as it would be expected to occur in the overall document collection. The smoothed estimate then combines the estimated likelihood to occur in the document according to the model generated from the document, with the estimated likelihood of a term occurring in the general document collection, modeled as a generic language model using the statistics from the document collection.

Probabilistic Retrieval

With smoothing the relevance is computed as

$$P(d|q) \propto P(d) \prod_{t \epsilon q} ((1-\lambda)P(t|M_c) + \lambda P(t|M_d))$$

From a technical perspective the probabilities are computed using term frequencies, thus same data used as in vector space retrieval

Probabilistically motivated models show generally better performance

- But parameter tuning (λ) is critical
- λ can be query-dependent, e.g., query size

Here we summarize the approach for probabilistic retrieval. From a more technical perspective computational cost of probabilistic retrieval is not very different from vector space retrieval. The computation of the likelihoods for the document models requires determination of term frequencies, so in that sense it is equivalent. For the collection models the global term frequencies need to be computed, which again is similar to computing inverse document frequencies in a document collection.

In practice, the fine tuning of the model parameters (in that case λ) is essential for that the model performs well. Different methods have been devised for that. It is also possible to make the parameters dependent on the query, in particular on the query size.

Example

Collection consisting of d_1 and d_2

d_1: Einstein was one of the greatest scientists

d_2: Albert Einstein received the Nobel prize

Query q: Albert Einstein

Using λ=1/2:

$P(q|d_1) = ½ * (0/7 + 1/13) * ½ * (1/7 + 2/13) \approx 0.0057$

$P(q|d_2) = ½ * (1/6 + 1/13) * ½ * (1/6 + 2/13) \approx 0.0195$

This is a simple example illustrating the use of probabilistic retrieval. Notate that the document lengths of d1 and d2 are 7 and 6, and that the collection length is 13.

Example: Comparing VS and PR

	Precision			
Rec.	tf-idf	LM	%chg	
0.0	0.7439	0.7590	+2.0	
0.1	0.4521	0.4910	+8.6	
0.2	0.3514	0.4045	+15.1	*
0.3	0.2761	0.3342	+21.0	*
0.4	0.2093	0.2572	+22.9	*
0.5	0.1558	0.2061	+32.3	*
0.6	0.1024	0.1405	+37.1	*
0.7	0.0451	0.0760	+68.7	*
0.8	0.0160	0.0432	+169.6	*
0.9	0.0033	0.0063	+89.3	
1.0	0.0028	0.0050	+76.9	
Ave	0.1868	0.2233	+19.55	*

Ponte & Croft, 1998

This is a result reported from comparing vector space retrieval with probabilistic retrieval. It shows that in this experiment probabilistic retrieval improves precision significantly, in particular for higher values of recall. (LM = language model).

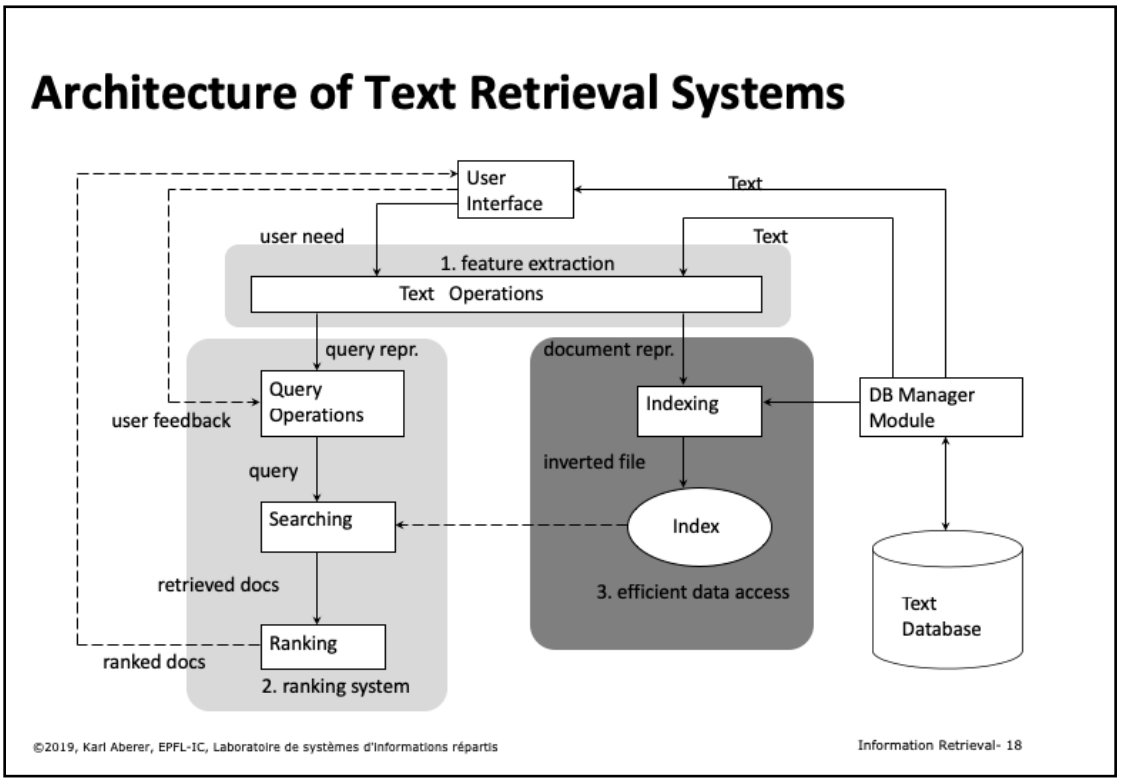
Overview of Retrieval Model Properties

	Vector Space Model	Language Model	BM25 (another prob. Model)
Model	geometric	probabilistic	probabilistic
Length normalization	Requires extensions (pivot normalization)	Inherent to model	Tuning parameters
Inverse document frequency	Used directly	Smoothing and collection frequency has similar effect	Used directly
Multiple term occurrences	Taken into account	Taken into account	Ignored
Simplicity	No tuning required	Tuning essential	Tuning essential

Here we compare the characteristics of the vector space model with the probabilistic retrieval model based on language models, and BM25 another model based on a probabilistic approach, that is today considered as one of the most performant retrieval models.

One aspect that is taken implicitly care off in the probabilistic retrieval model based on language models is normalization for document length. For vector space retrieval specific extensions have been developed, that modify the weighting parameters with the document length. For collections with widely varying document lengths this proved to be a useful improvement. In general, the vector space model is preferred when a quick and simple solution is sought. For probabilistic models better performance can be achieved, but this depends on careful parameter tuning which requires specialized expertise.

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:

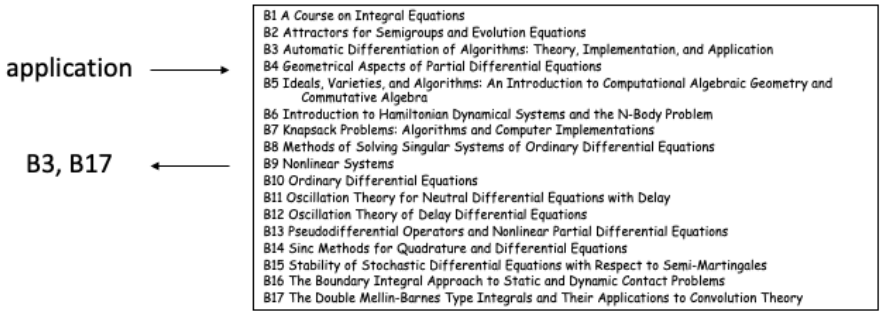
1. the feature extraction component: it performs text processing to turn queries and text documents into a keyword-based representation
2. the ranking system: it implements the retrieval model. In a first step user queries are potentially modified (in particular if user relevance feedback is used), then 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 retrieval, 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

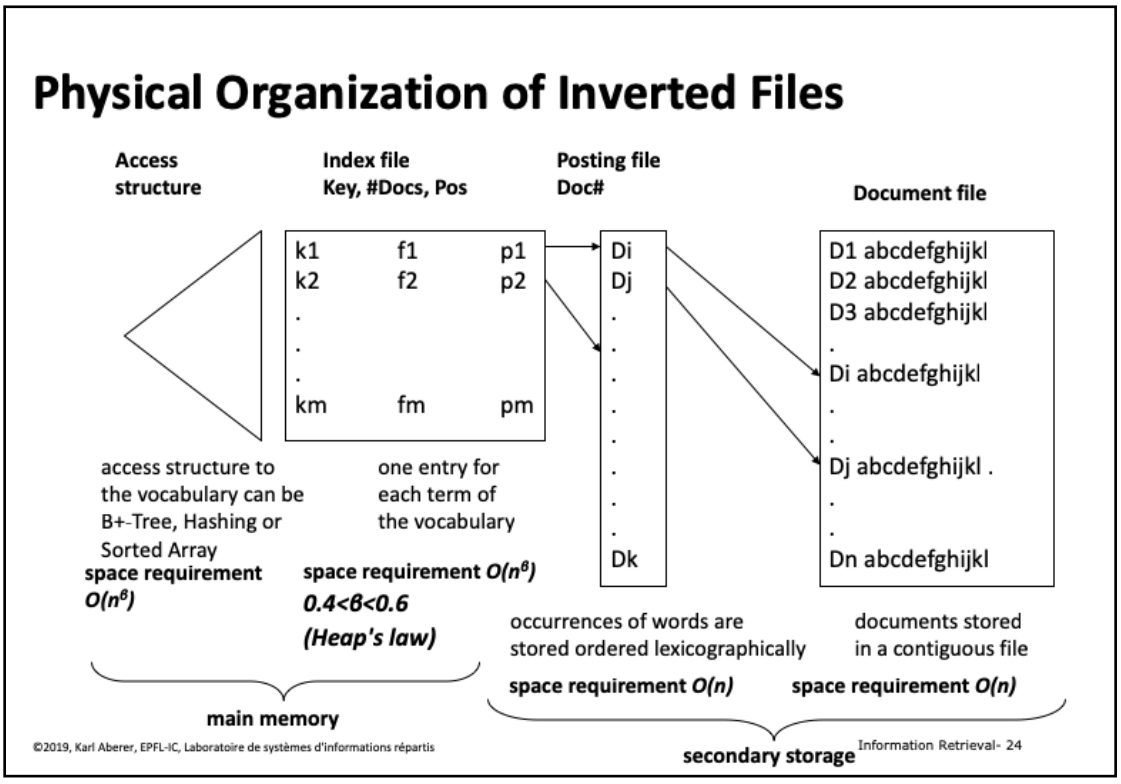
B1 A Course on Integral Equations
B2 Attractors for Semigroups and Evolution Equations
B3 Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4 Geometrical Aspects of Partial Differential Equations
B5 Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic 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. 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 $O(n^{\beta})$, where β is roughly 0.5 (Heap's law). For example, a document collection of size $n= 10^6$ would have approximate $m=10^3$ index terms. Therefore 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.

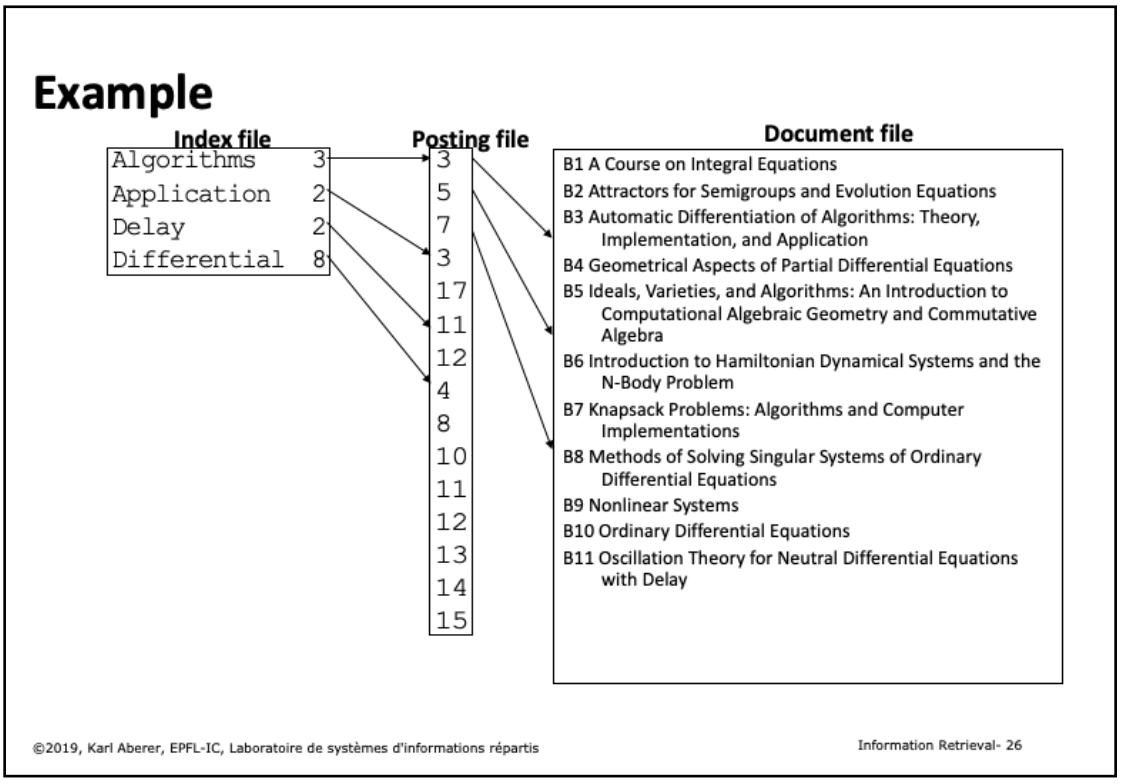
Searching the Inverted File

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

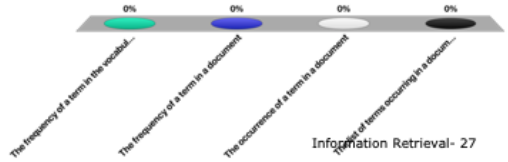
Search in an inverted file is a straightforward process. Using the data access structure, first the index terms occurring in the query are searched in the index file. Then the occurrences can be sequentially retrieved from the postings file. Afterwards the corresponding document portions are accessed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



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

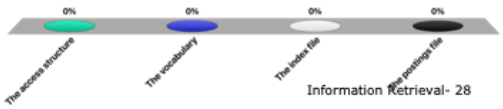
A posting indicates…

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



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

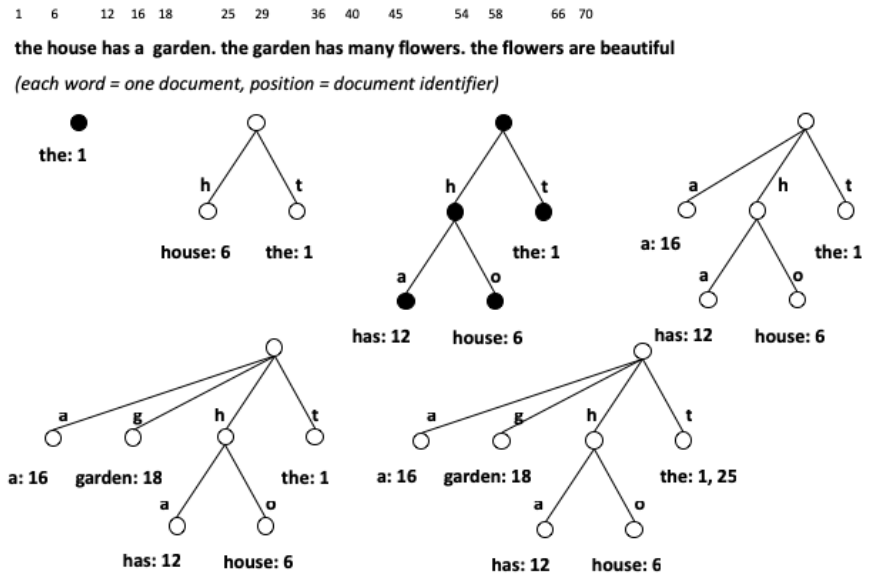
Step 2: Storage phase (once the text is exhausted)

- 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)

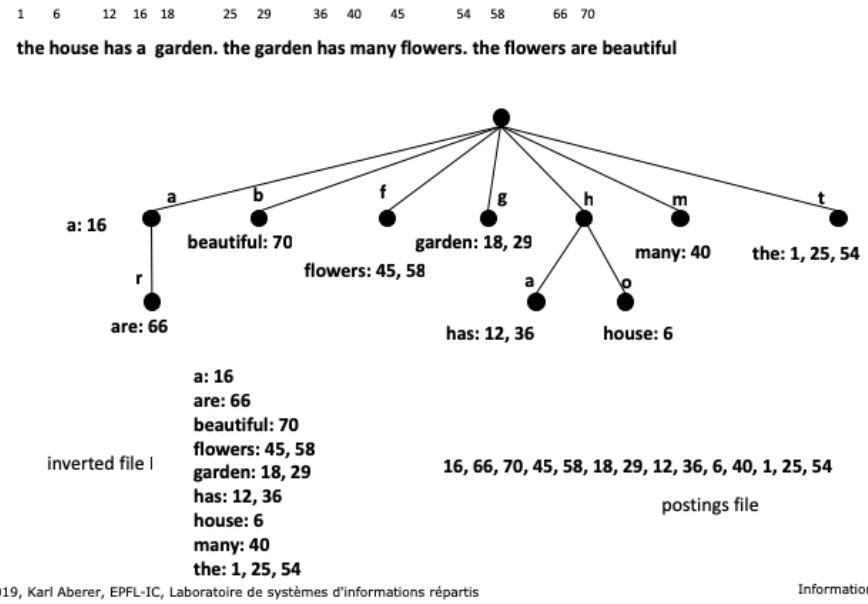
After the complete document collection has been traversed, the trie structure is sequentially traversed and the posting file is written to secondary storage. The trie structure itself can be used as a data access structure for the index file that is kept in main memory.

Example



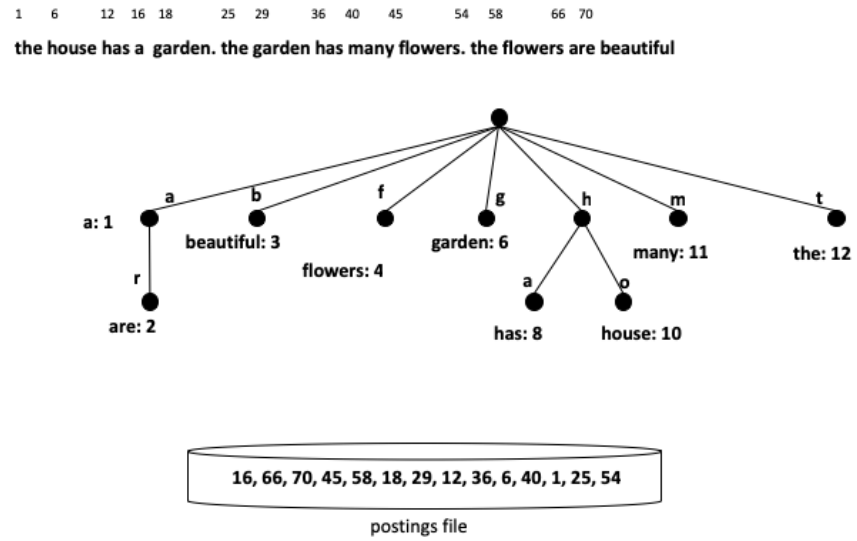
In this example we consider each word of the text as a separate document identified by its position (for space limitations). We demonstrate the initial steps of constructing the trie structure and adding to it the occurrences of index terms. The changes to the trie structure are highlighted for each 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



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.

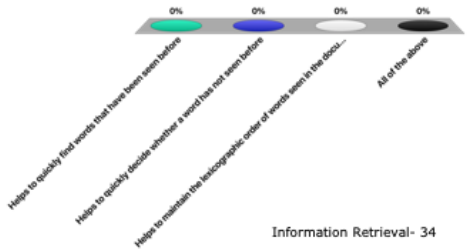
Example



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.

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



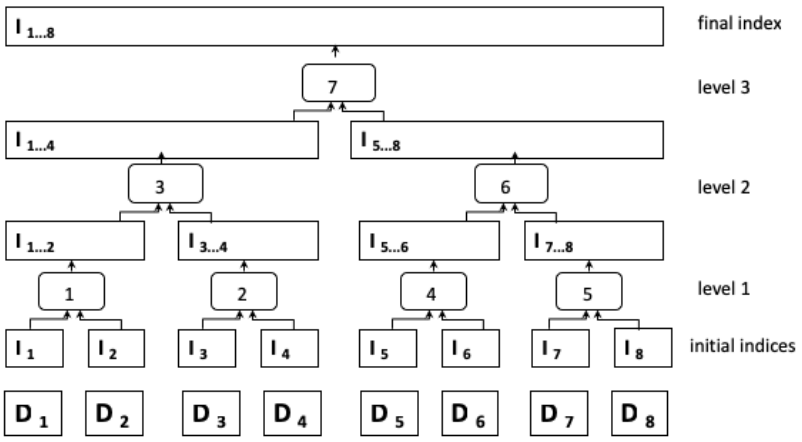
Index Construction in Practice

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

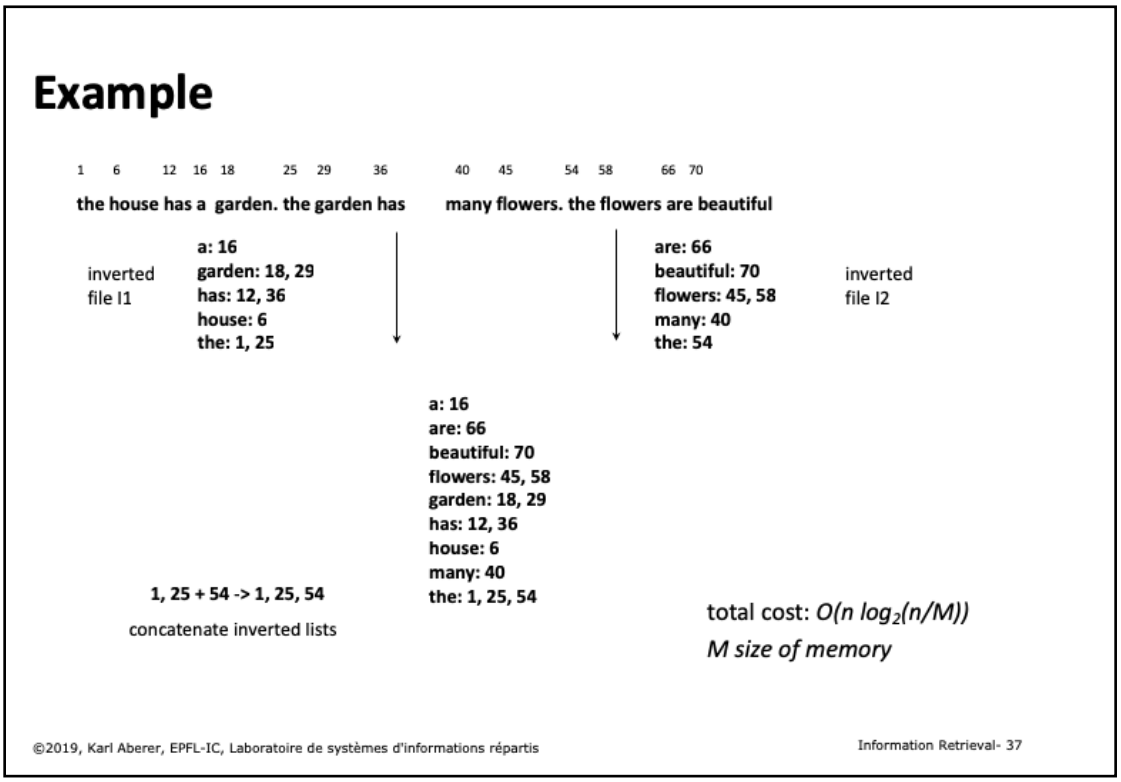
- When no more memory is available, a partial index I_i is written to disk
- The main memory is erased before continuing with the rest of the text
- Once the text is exhausted, a number of partial indices I_i exist on disk
- The partial indices are merged to obtain the final index

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



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

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

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

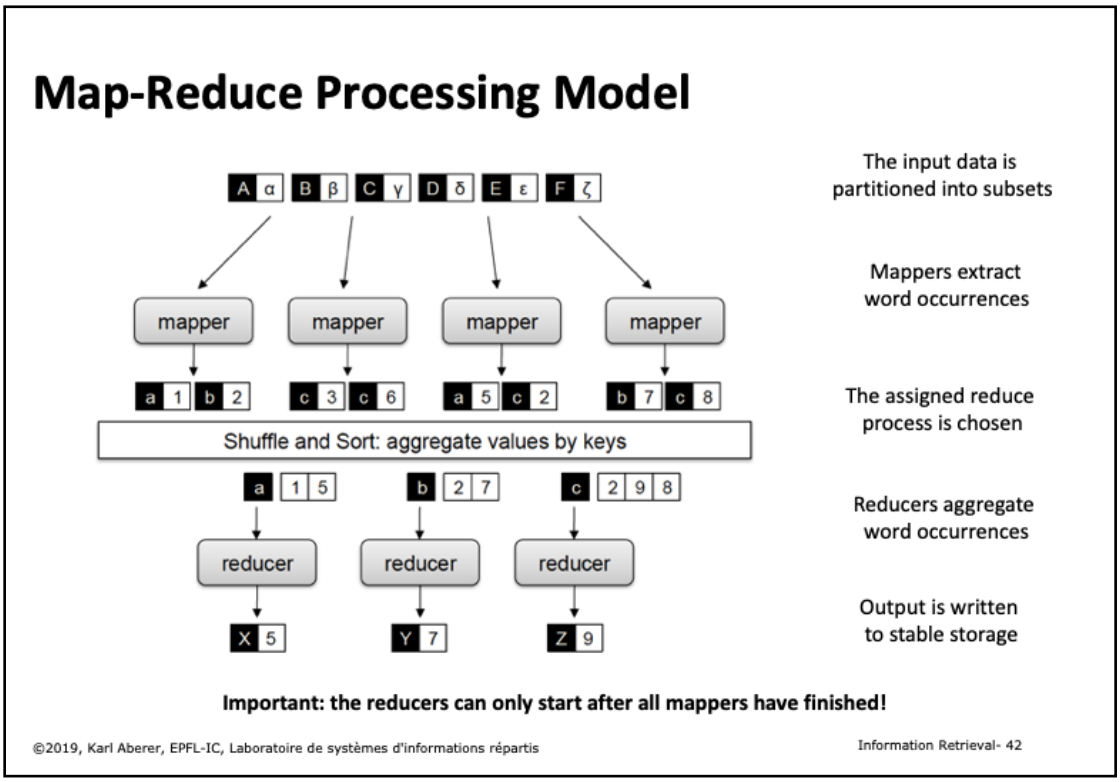
Example: basic word counter program

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

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

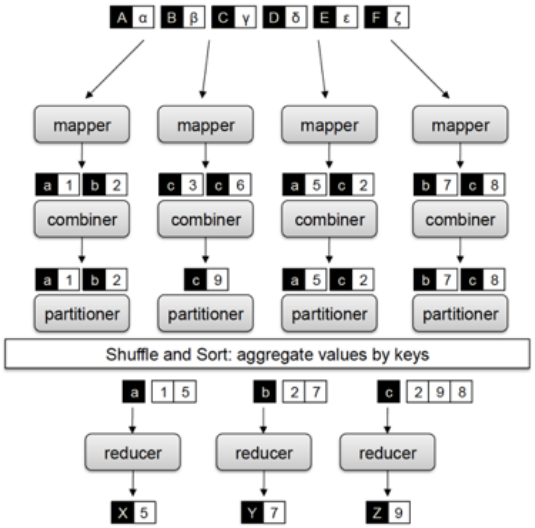

The map-reduce programming model is based on key-value pairs and lists of key value pairs (denoted by angle brackets here). The map function receives some input data (typically a piece of text to analyze or index), and produces a list of key-value pairs, that represent some partial results of the analysis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



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)

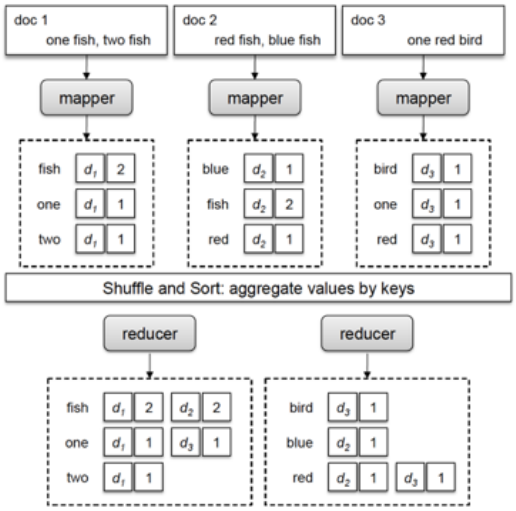
The programmer controls

- Key-value data structures (can be complex)
- Maintain 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)
```

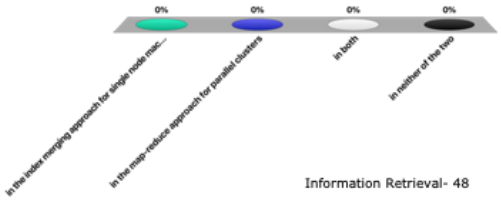
Other Applications of Map-Reduce

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

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

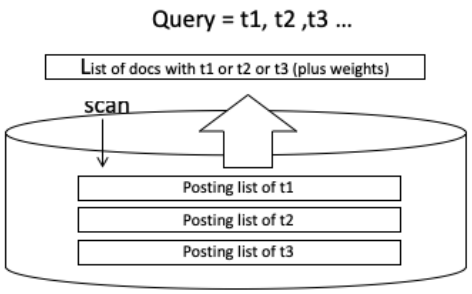


6. DISTRIBUTED RETRIEVAL

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)

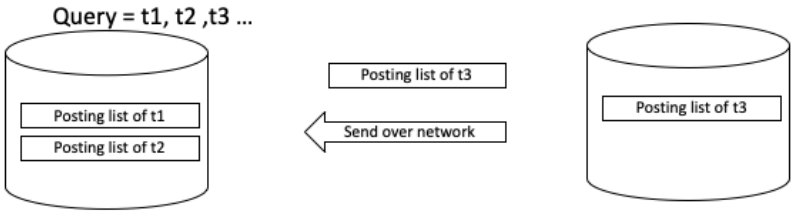


When using inverted files, a query involving multiple search terms requires the scanning of the postings lists of all terms. Typically in this process the term frequencies are computed for ALL documents in the document collection containing any of the query terms. In a centralized server this can be 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

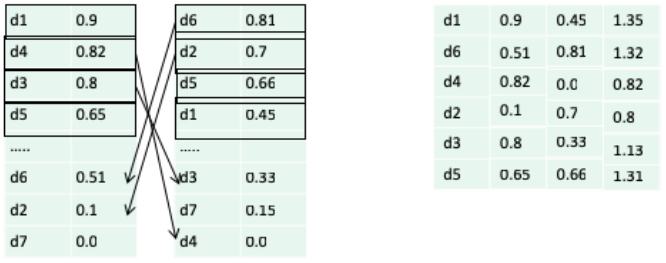
d6	0.81
d2	0.7
d5	0.66
d1	0.45
.....	
d3	0.33
d7	0.15
d4	0.0

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)

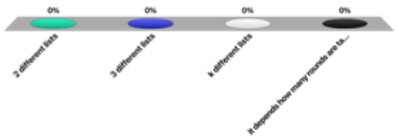
It can be shown that the complexity of the Fagin algorithm in the case of two lists is $O((k\ n)^{1/2})$ for the number of entries that are read from each list, where n is the number of documents in the document collection. This is significantly smaller than reading the complete lists, and reduces further 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.

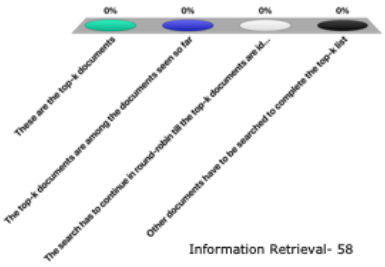
When applying Fagin's algorithm for a query with three different terms for finding the k top documents, the algorithm will scan ...

1. 2 different lists
2. 3 different lists
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
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

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.

Paper

- Fagin, R., Lotem, A., & Naor, M. (2003). Optimal aggregation algorithms for middleware. Journal of computer and system sciences, 66(4), 614-656.