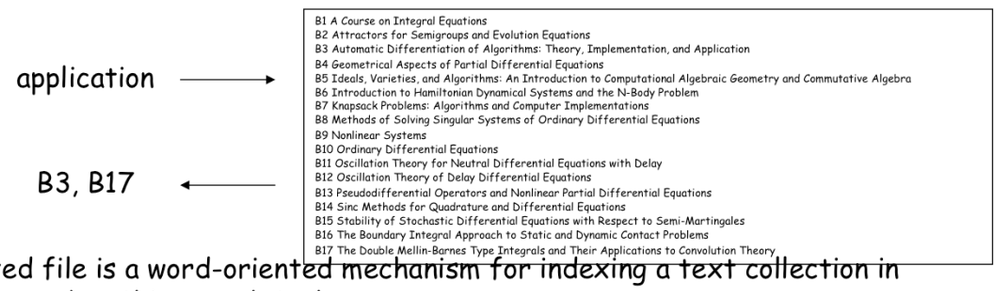
Information Retrieval and Data Mining
Part 1 – Information Retrieval

Today's Question

1. Information Retrieval
2. Text Retrieval Models
3. Latent Semantic Indexing & Relevance Feedback
4. Inverted Files
5. Web Information Retrieval
6. Distributed Information Retrieval

4. Inverted Files

- Problem: text retrieval algorithms need to find words in documents efficiently
  - Boolean retrieval, vector space retrieval
  - Given index term k_i, find document d_j



- An inverted file is a word-oriented mechanism for indexing a text collection in order to speed up this search task
  - Addressing of documents and word positions within documents
  - Most frequently used indexing technique for large text databases
  - Appropriate when the text collection is large and semi-static

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 file are optimized for supporting search on relatively static text collections. For example, database 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

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

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

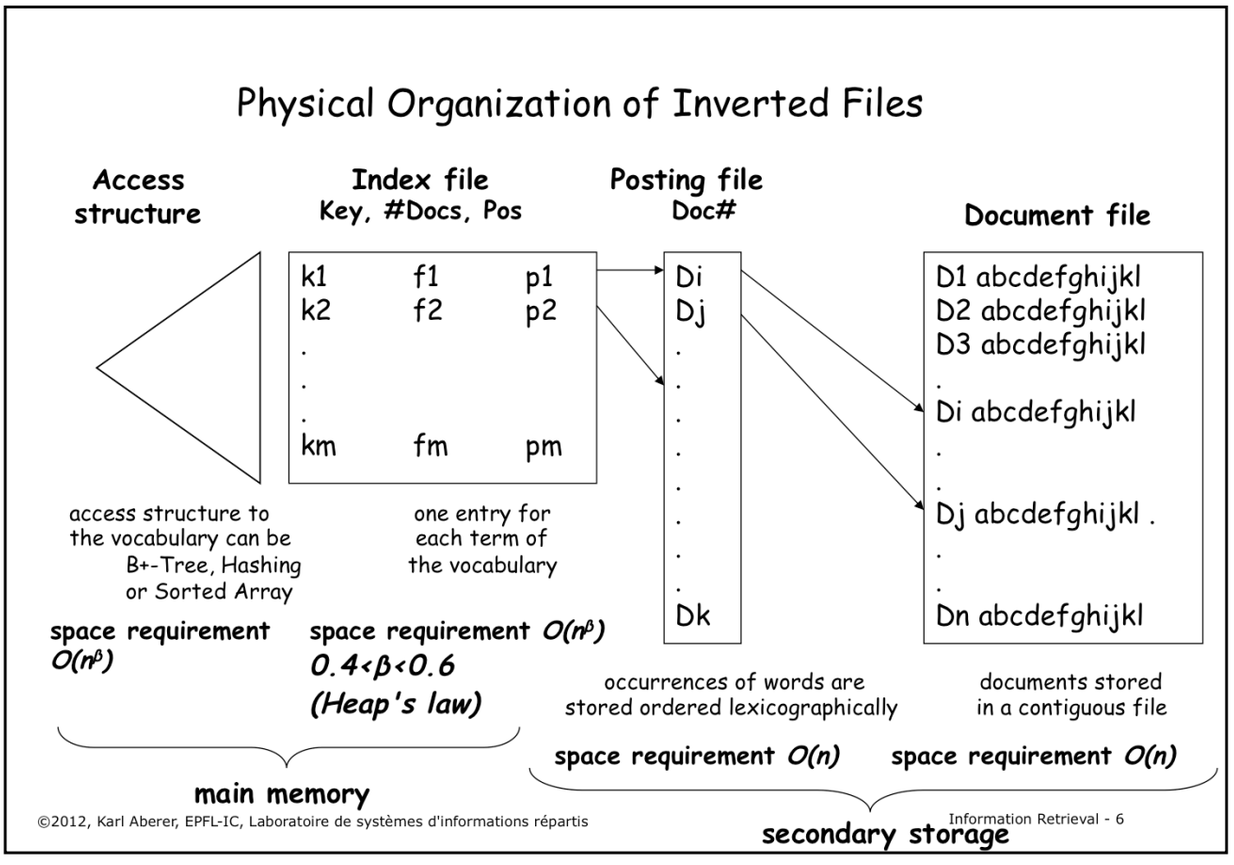
$$IF = \langle i, k_i, l_{k_i} \rangle, i = 1, \ldots, m$$

Inverted files are constructed by concatenating the inverted lists for all terms occurring in the document collection. The 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 inverse document frequency, for example.

Example

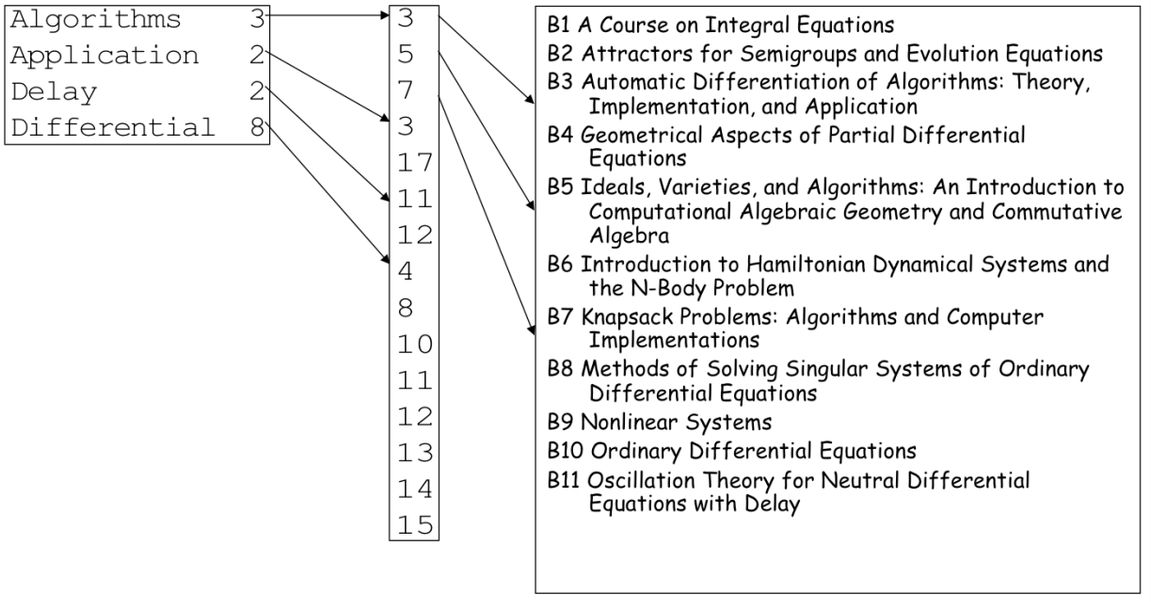
1	Algorithms	3	:	3	5	7						
2	Application	2	:	3	17							
3	Delay	2	:	11	12							
4	Differential	8	:	4	8	10	11	12	13	14	15	
5	Equations	10	:	1	2	4	8	10	11	12	13	14	15
6	Implementation	2	:	3	7							
7	Integral	2	:	16	17							
8	Introduction	2	:	5	6							
9	Methods	2	:	8	14							
10	Nonlinear	2	:	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 as introduced before are a logical data structure, for which a physical storage organization needs to be provided. This physical organization has to take into account the quantitative characteristics of the inverted file structure. To that extent the key observation is: 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^β), 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 consists 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.

Example



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

Questions

- 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

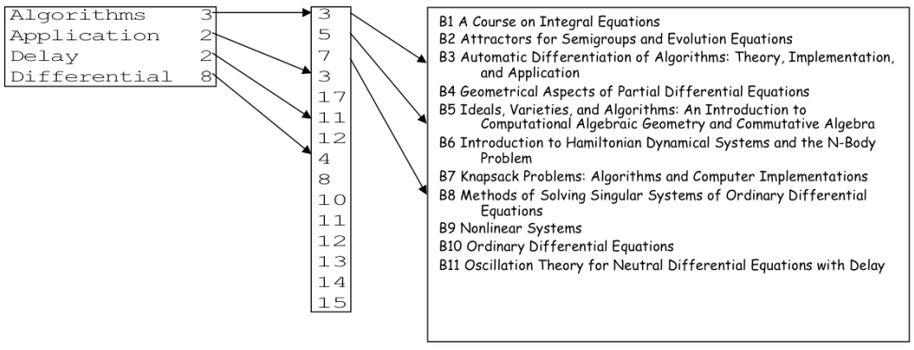
- When indexing a document collection using an inverted file, the main space requirement is implied by
  1. The access structure
  2. The inverted lists
  3. The index file
  4. The postings file

Searching the Inverted File

Step 1: Vocabulary search
- the words present in the query are searched in the index file

Step 2: Retrieval of occurrences
- the lists of the occurrences of all words found are retrieved from the posting file

Step 3: Manipulation of occurrences
- the occurrences are processed in the document file to process the query



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

Construction of the Inverted File

Step 1: Search phase

- The vocabulary is kept in a trie data structure 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

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)

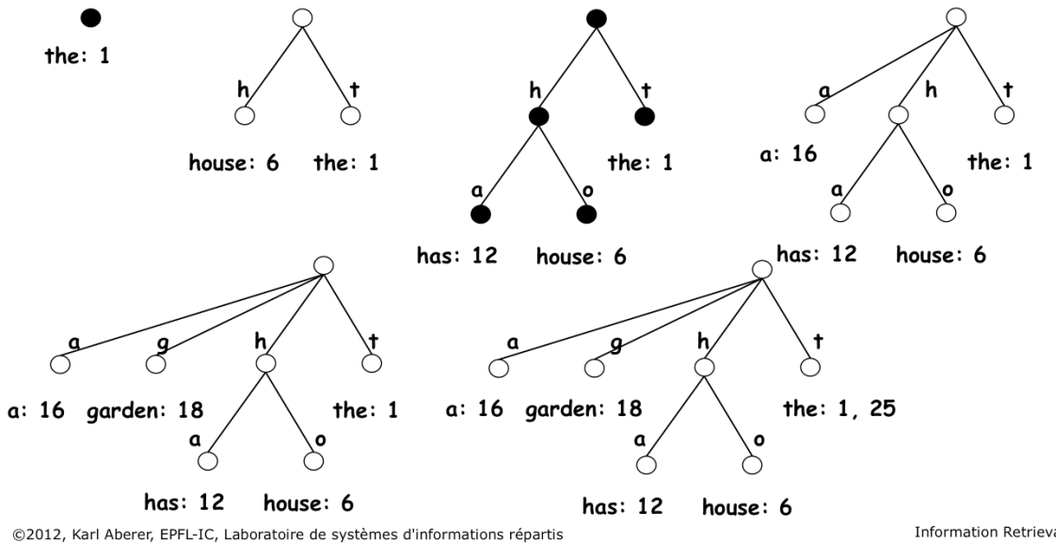
The index construction is performed by first constructing dynamically a trie structure, in order to generate a sorted vocabulary and to collect the occurrences of index terms. After the complete document collection has been traversed the trie structure is sequentially traversed and the posting file is 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

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

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

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

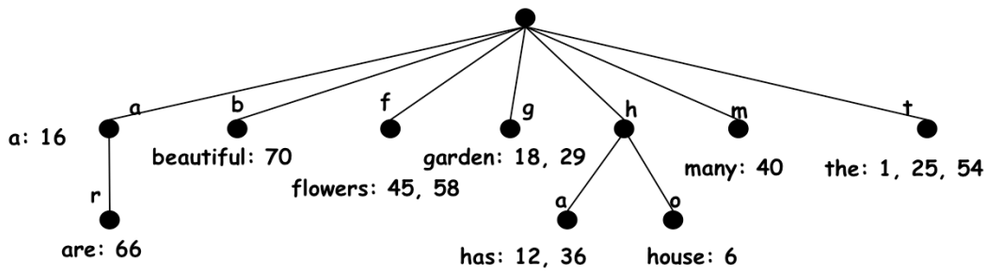


In this example we consider each word of the text as a separate document identified by its position (for space limitations). We demonstrate the initial steps of constructing the trie structure and adding to it the occurrences of index terms. The changes to the trie structure are highlighted for each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

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

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



inverted file I

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

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

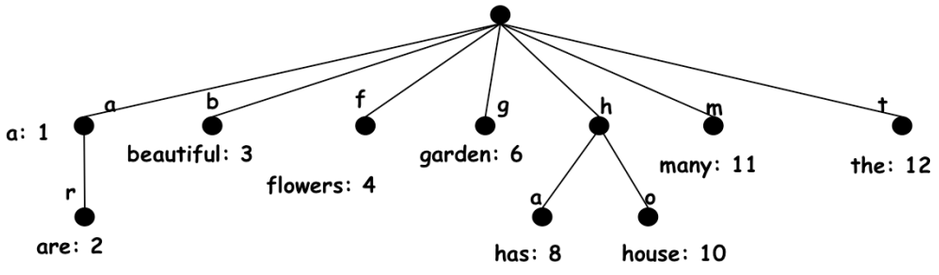
postings file

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

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

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



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

postings file

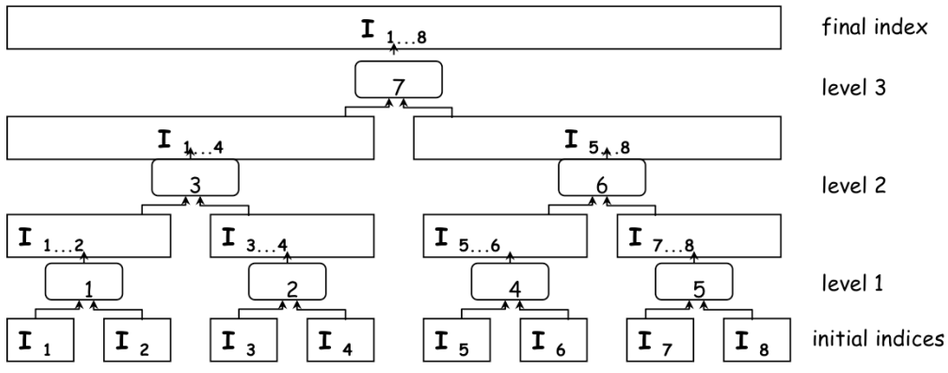
The resulting physical organization of the inverted file is shown here. The trie structure can be used as an access structure to the index file in main memory. Thus the entries of the index files occur as leaves (or internal nodes) of the trie. Each entry has a reference to the position of the postings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

Question

- 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

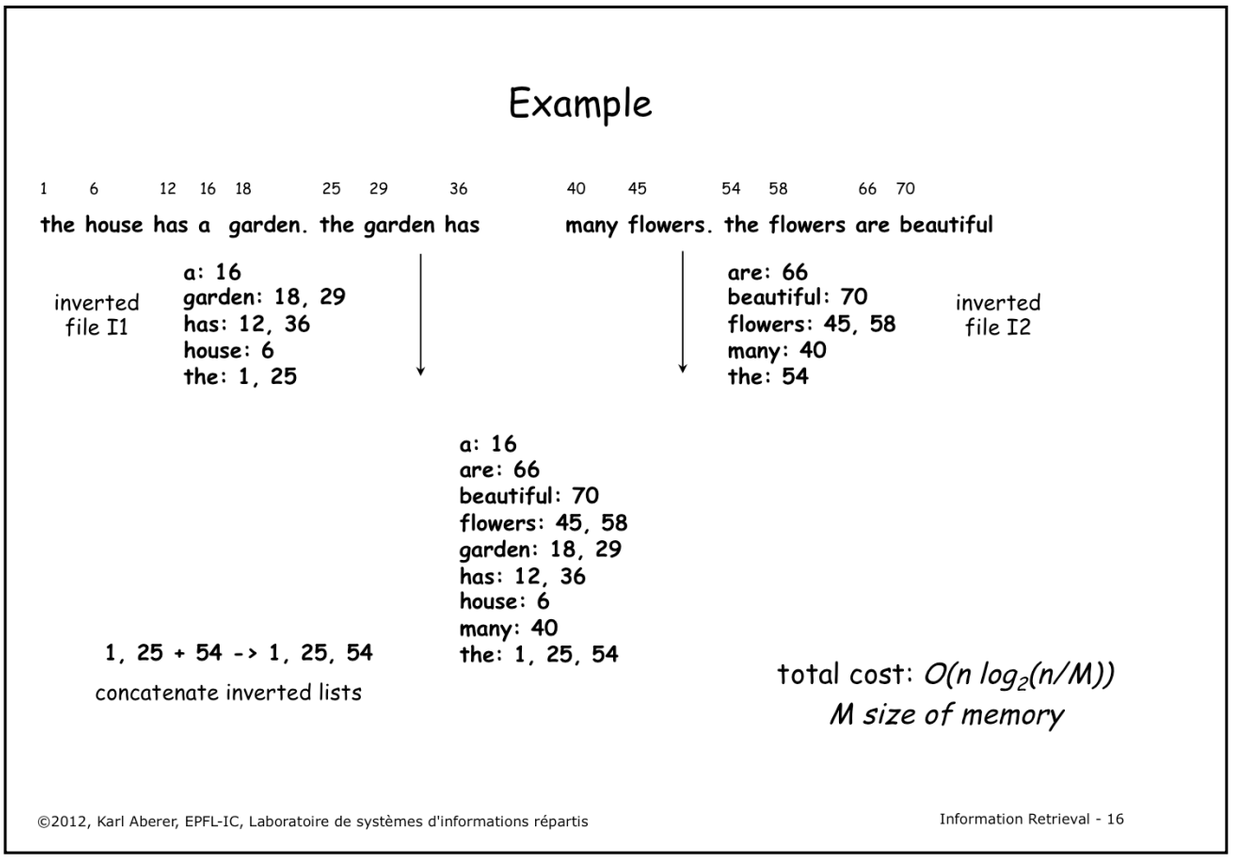
Index Construction in Practice

- 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 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.

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.



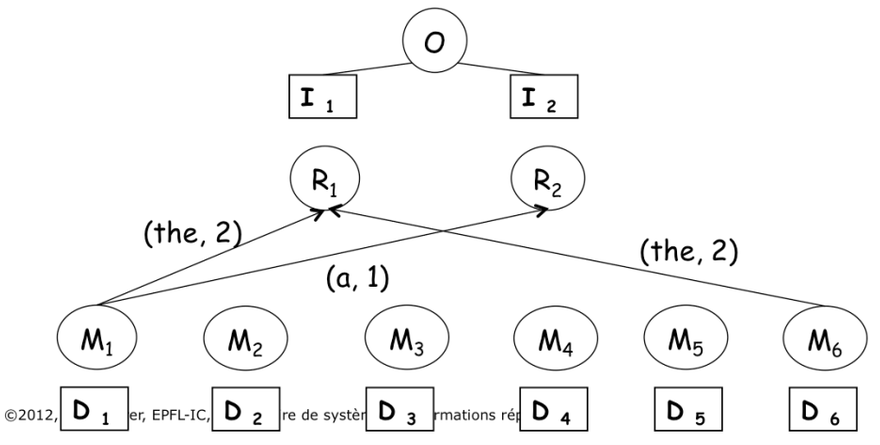
Merging the indices requires first merging the vocabularies. As we know, the vocabularies are comparably small and thus the merging of the vocabularies can take place in main memory. In case a vocabulary term occurs in both partial indices, their list of occurrences from the posting file need to be combined. 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.

Index Construction Today

- Map-Reduce programming model
  - The input data is partitioned into subsets
  - Mapper processes scan the input and produce lists of (key, value) pairs
  - The (key, value) pairs are send to (multiple) reduce processes
  - The reduce process is e.g. chosen by hashing the keys
  - The reduce processes aggregate all arriving data pertaining to a specific key



Output writer writes results to file system

Reducers aggregate word occurences

Mappers extract word occurences

On modern computing clusters, a different approach can be taken by completely parallelizing the process of index construction. This approach has become known as map-reduce programming model. The idea is that the document collection is partitioned and processed in parallel by so-called mappers. These extract the word statistics. Then they send the results to so-called reducers that aggregate the results for a sub-set of the keys, i.e. the words to be indexed. The partitioning of the work is automatically performed by the system, e.g. by using hash partitioning on the key space. Once the reducers have finalized generating the partial indices for their key space, the result is written to the file system.

Map Reduce Programming Model

- 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 (cheap, but unreliable), commodity network
  - Automatic fault-tolerance (fewer admins)
  - Easy to use (fewer programmers)
- Programming Model
  - Data type: key-value records
  - Map function: (K_{in}, V_{in}) ➔ list(K_{inter}, V_{inter})
  - Reduce function: (K_{inter}, list(V_{inter})) ➔ list(K_{out}, V_{out})

The map-reduce programming model has been key in the ability of Google and later other web providers to scale upt the applications. It actually led to a novel distributed programming paradigm and systems approach, that is tuned towards cost-efficiency and simplicity of programming.

Sample Map-Reduce program

Simple word count program

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

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

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

Combiner locally aggregate results
-> smaller messages to reducers

This is a small sample program for computing the word count over a document collection. As a further optimization local results produced by a mapper can be pre-aggregated to reduce the amount of data that is transmitted over the network.

Question

- Partial index files maintain lexicographic order
  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

Addressing Granularity

- Documents can be addressed at coarser and finer 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 for 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 postprocessing 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

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

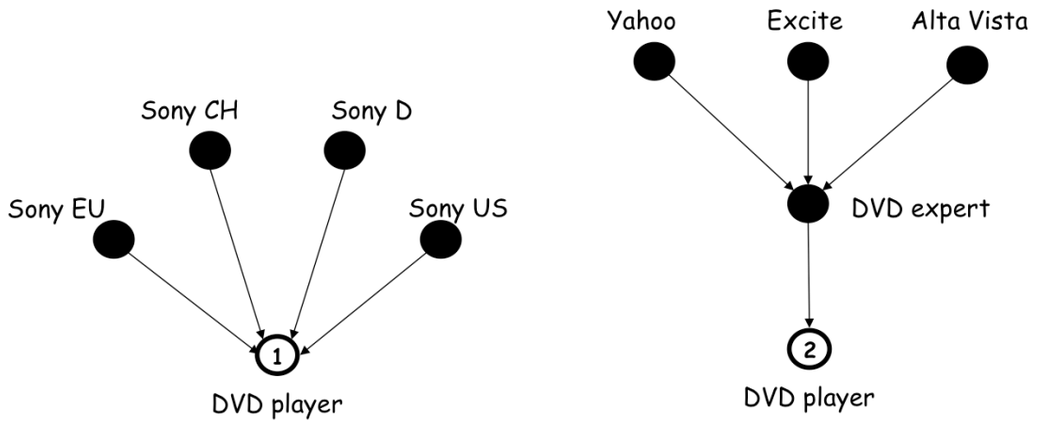
- Use of varying length compression further reduces space requirement
- In practice index is reduced to 10- 15% of database size

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.

5. Web Information Retrieval



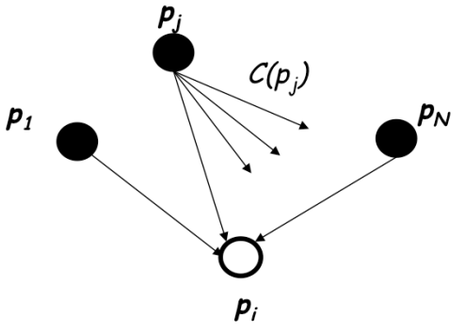
- Full text retrieval result with equal ranking; which page is more relevant ?
  - relevance related to number of referrals (incoming links)
  - relevance related to number of referrals with high relevance

When retrieving documents from the Web, the link structure bears important information on the relevance of documents. Generally speaking, a document that is referred more often by other documents by Web links, is likely to be of higher interest and therefore relevance. So a possibility to rank documents is considering the number of incoming links. Doing this allows to distinguish documents that otherwise would be ranked equally or similarly when relying on text retrieval alone.

However, when doing this, also the importance of documents having a link to the document to be ranked may be different. Therefore not only counting then number of incoming links, but also weighting the links by the relevance of documents that contain these links appears to be appropriate. The same reasoning of course again applies then for evaluating the relevance of documents pointing to the document and so forth.

Random Walker Model

- Assumption: If a random walker visits a page more often it is more relevant: takes into account the number of referrals AND the relevance of referrals



N is the number of Web pages
$C(p)$ is the number of outgoing links of page p
$P(p_i)$ probability to visit page p_i, where page p_i is pointed to by pages p_1 to p_N = **relevance**

$$P(p_i) = \sum_{p_j | p_j \to p_i} \frac{P(p_j)}{C(p_j)}$$

In order to capture the process of evaluating the relevance of documents by considering incoming links, assuming the relevance of documents is known (which is a recursive procedure) the random walker model is used. The intuitive idea is that if a random walker on the web graph visits a document more often the document is more relevant. This intuition is captured mathematically in a recursive equation characterizing the probability of visiting a specific page by a random walker. The assumption on the random walker used in this mathematical formulation of the process is that it follows every link of document with equal probability.

Transition Matrix for Random Walker

- The definition of $P(p_i)$ can be reformulated as matrix equation

$$R_{ij} = \begin{cases} \frac{1}{C(p_j)}, if\ p_j \rightarrow p_i \\ 0, otherwise \end{cases}$$

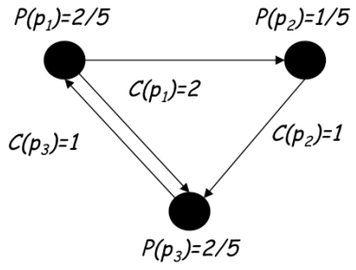
$$\vec{p} = (P(p_1), ..., P(p_n))$$

$$\vec{p} = R.\vec{p}, \quad \left\|\vec{p}\right\|_1 = \sum_{i=1}^{n} p_i = 1$$

- The vector of relevance of pages is an Eigenvector of the matrix R

If we consider the matrix of transition probabilities for the random walker, the recursive definition of the probability to visit a Web page can be formulated as matrix equation. More precisely the resulting equation shows that the vector of probabilities for visiting the Web pages is an Eigenvector of the transition matrix. In fact, it is the one corresponding to the largest Eigenvalue.

Example



Links from p_1

$$L = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Links to p_1

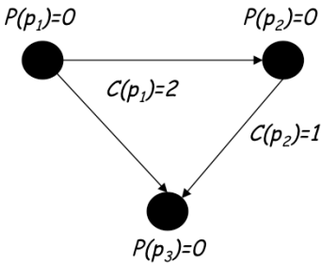
Link Matrix

$$R = \begin{pmatrix} 0 & 0 & 1 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & 0 \end{pmatrix}, \quad \vec{p} = \begin{pmatrix} \frac{2}{5} \\ \frac{1}{5} \\ \frac{2}{5} \end{pmatrix}$$

Ranking p_1, p_3 > p_2

This example illustrates the computation of the probabilities for visiting a specific Web page. The values $C(p_i)$ correspond to the transition probabilities. They can be derived from the link matrix, which is the matrix with entries 1 at (i,j) if there exists a link from p_i to p_j, by dividing the values in the columns by the sum of the values found in the column. The probability of a random walker being in a node is then obtained from the Eigenvector of this matrix.

Modified Example



Links from p_1

$$L = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Links to p_1

Link Matrix

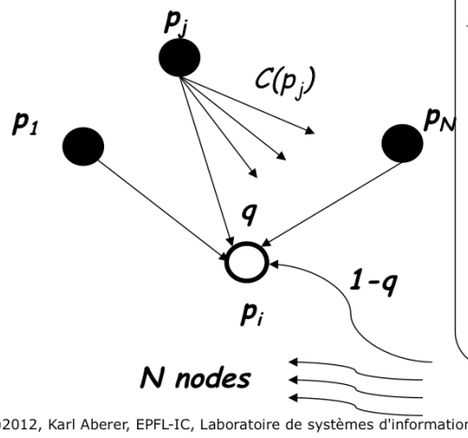
$$R = \begin{pmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & 0 \end{pmatrix}, \quad \vec{p} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

No Ranking

The approach described so far has however a problem. When looking at the modified example we see that there exists a node p_3 that is a "sink of rank". Any random walk ends up in this sink, and therefore the other nodes do not receive any ranking weight. Consequently also the rank of sink does not. Therefore the only solution to the equation p=Rp is the zero vector.

Source of Rank

- Assumption: random walker jumps with probability 1-q to an arbitrary node: thus nodes without incoming links are reached
- PageRank method



$$P(p_i) = c(\frac{(1-q)}{N} + q \sum_{p_j | p_j \to p_i} \frac{P(p_j)}{C(p_j)}), c \le 1$$

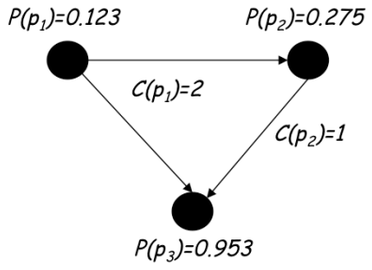
$$\vec{p} = c((qR + (1-q)E)).\vec{p}, \quad E = \left[\frac{1}{N}\right]_{NxN}$$

$$\vec{p} = c(qR.\vec{p} + \frac{(1-q)}{N}\vec{e}), \quad \vec{e} = (1,...,1)$$

To avoid the previously described problem, we add a "source of rank". The idea is that a random walker in each step can, rather than following a link, jump to any page with probability 1-q. Therefore also pages without incoming links can be reached by the random walk. In the mathematical formulation of the random walk this means that a term for the source of rank is added. Since at each step the random walker makes a jump with probability 1-q and any of the N pages is reached with the same probability the additional term is (1-q)/N. Reformulating this again as matrix equation means adding a NxN Matrix E with all entries being 1/N. This is equivalent to saying that with probability 1/N transitions among any pairs of nodes (including transition from a node to itself) are performed. Since the vector p has norm 1, i.e., the sum of the components is exactly 1, E.p=e, where e is the unit vector, and the matrix equation can be reformulated in the second form shown below. The method described is called PageRank and is used by Google.

By modifying the values of the matrix E also a priori knowledge about the relative importance of pages can be added to the algorithm.

Modified Example



$$R = \begin{pmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & 0 \end{pmatrix}, E = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix}, q = 0.9$$

$$qR + (1-q)E = \begin{pmatrix} \frac{1}{30} & \frac{1}{30} & \frac{1}{30} \\ \frac{29}{60} & \frac{1}{30} & \frac{1}{30} \\ \frac{29}{60} & \frac{14}{15} & \frac{1}{30} \end{pmatrix} \quad \vec{p} = \begin{pmatrix} 0.123 \\ 0.275 \\ 0.953 \end{pmatrix}$$

Ranking $p_3 > p_2 > p_1$

With the modification of rank computation using a source of rank, we obtain for our example again a non-trivial ranking which appears to be appropriate.

Question

- A positive random jump value for exactly one node implies that
  1. a random walker can leave the node even without outgoing edges
  2. a random walker can reach the node multiple times even without outgoing edges
  3. a random walker can reach the node even without incoming edges
  4. none of the above

Practical Computation of PageRank

- Iterative computation

ε termination criterion
s arbitrary vector, e.g. s=e

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

For the practical determination of the PageRank ranking an iterative computation is used. It is derived from the second form of the formulation of the visiting probabilities of the random walker that we have given. The vector e used to add a source of rank not necessarily has to assign uniform weights to all pages, but might reflect itself a ranking of Web pages.

Example: ETHZ Page Rank

Doc_ID	Rank_Value	URL
1	0.002536	http://www.ethz.ch/
146	0.002292	http://www.ethz.ch/r_amb/
10	0.000654	http://www.ethz.ch/default_de.asp
35	0.000511	http://www.rereth.ethz.ch/
376124	0.000503	http://computing.ee.ethz.ch/sepp/matlab-5.2-to/helpdesk.html
67378	0.000497	http://computing.ee.ethz.ch/sepp/
59887	0.000485	http://www.computing.ee.ethz.ch/sepp/
89307	0.000485	http://www.isg.inf.ethz.ch/docu/documents/java/jdk1.2.2ref/docs/api/overview-summary.html
216716	0.000485	http://www.isg.inf.ethz.ch/docu/documents/java/jdk1.2/api/overview-summary.html
147932	0.000484	http://isg.inf.ethz.ch/docu/documents/java/jdk1.2ref/docs/api/overview-summary.html
175544	0.000484	http://www.isg.inf.ethz.ch/docu/documents/java/jdk1.2ref/docs/api/overview-summary.html
186766	0.000478	http://isg.inf.ethz.ch/docu/documents/java/jdk1.2/api/overview-summary.html
228634	0.000477	http://isg.inf.ethz.ch/docu/documents/java/jdk1.2.1ref/docs/api/overview-summary.html
228421	0.000464	http://isg.inf.ethz.ch/docu/documents/java/jdk1.2.2ref/docs/api/overview-summary.html
3161	0.00045	http://www.ethz.ch/r_amb/reto_ambuehler.html
215673	0.000447	http://www.vision.ee.ethz.ch/computing/statlinks/sepp.sun/vxl-1.2b-mo/files.html
259672	0.000447	http://www.vision.ee.ethz.ch/computing/statlinks/sepp.sun/vxl-1.2b-mo/globals.html
259671	0.000447	http://www.vision.ee.ethz.ch/computing/statlinks/sepp.sun/vxl-1.2b-mo/functions.html
259670	0.000447	http://www.vision.ee.ethz.ch/computing/statlinks/sepp.sun/vxl-1.2b-mo/annotated.html
259669	0.000447	http://www.vision.ee.ethz.ch/computing/statlinks/sepp.sun/vxl-1.2b-mo/classes.html

Figure 1: Top 20 of ETH Zurich Web Documents

These are the top documents from the PageRank ranking of all Web pages at ETHZ (Data from 2001). It is interesting to see that documents related to Java documentation receive high ranking values. This is related to the fact that these documents have many internal cross-references.

Practical Usage of PageRank

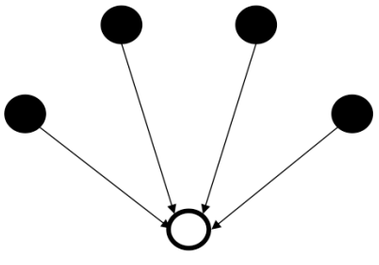
- PageRank is part of the ranking method used by Google
  - Compute the global PageRank for all Web pages
  - Given a keyword-based query retrieve a ranked set of documents using standard text retrieval methods
  - Merge the ranking with the result of PageRank to both achieve high precision (text retrieval) and high quality (PageRank)
  - Google uses also other methods to improve ranking (trade secret)
- Main problems
  - Crawling the Web
  - Efficient computation of Page Rank for large link databases
  - Combination with other ranking methods (text)

PageRank is used as one criterion to rank result documents in Google. Essentially Google uses text retrieval methods to retrieve relevant documents and then applies PageRank to create a more appropriate ranking. Google uses also other methods to improve ranking, e.g., by giving different weights to different parts of Web documents. For example, title elements are given higher weight. The details of the ranking methods are trade secrets of the Web search engine providers.

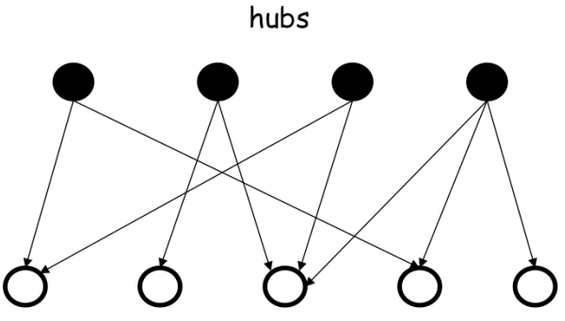
Other issues that Web search engines have to deal with, are crawling the Web, which requires techniques that can explore the Web without revisiting pages too frequently. Also the enormous size of the document and link database poses implementation challenges in order to keep the ranking computations scalable. One of the outcomes of solving these challenges are the recent “cloud computing” infrastructures, which are large-scale computing clusters constructed from commodity hardware.

Hub-Authority Ranking

- Key idea: identify not only authorative pages (like with PageRank) but also hubs
  - Hubs are pages that point to many/relevant authorities
  - Authorities are pages that are pointed to by many/relevant hubs
- Originally devised to postprocess query results



page with large in-degree



authorities

Hub-Authority ranking identifies not only pages that have a high authority, as measured by the number of incoming links, but also pages that have a substantial "referential" value, having many outgoing links (to pages of high importance). In difference to the PageRank algorithm this technique has been proposed to post-process query results (rather than to rank pages from the complete Web graph). It can be used as a add-on to existing search engines, but as well as an alternative to the PageRank method.

Definition of Hubs and Authorities

- Direct application of the definition

$$H(p_i) = \sum_{p_j \in N | p_i \to p_j} A(p_j) \qquad A(p_i) = \sum_{p_j \in N | p_j \to p_i} H(p_j)$$

- If we define $\vec{h} = (H(p_1), ..., H(p_N)) \quad \vec{a} = (A(p_1), ..., A(p_N))$ the hub-authority vectors are satisfying the above equation and normalized to 1

$$\overrightarrow{\|a\|}_1 = \sum_{i=1}^{n} a_i = 1, \overrightarrow{\|h\|}_1 = \sum_{i=1}^{n} h_i = 1,$$

- Matrix formulation

$$\vec{h} = L^t \cdot \vec{a}, \quad \vec{a} = L \cdot \vec{h} \qquad L_{ij} = \begin{cases} 1, if\ p_j \to p_i \\ 0, otherwise \end{cases}$$

respectively

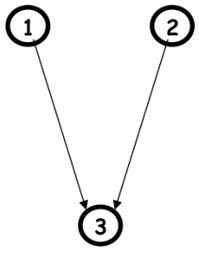
$$\vec{h} = L^t L \cdot \vec{h}, \quad \vec{a} = L L^t \cdot \vec{a}$$

- Iterative computation analogous to PageRank

If we formulate the hub-authority ranking as equations and transform into a matrix representation we realize that the hub-authority values are nothing else then Eigenvectors of two matrices derived from the link matrix L, namely the matrices L^tL and LL^t. The computation of the values can be performed then analogous as for PageRank.

Question

- Given the graph below and an initial hub vector of (1,1,1). The hub-authority ranking will result in the following
  1. authority vector (0,0,1); hub vector (1,1,0)
  2. authority vector (0,0,2); hub vector (2,2,0)
  3. authority vector (0,0,1); hub vector ($\frac{1}{2}$,$\frac{1}{2}$,0)
  4. authority vector (0,0,2); hub vector (1,1,0)



6. Distributed Information Retrieval

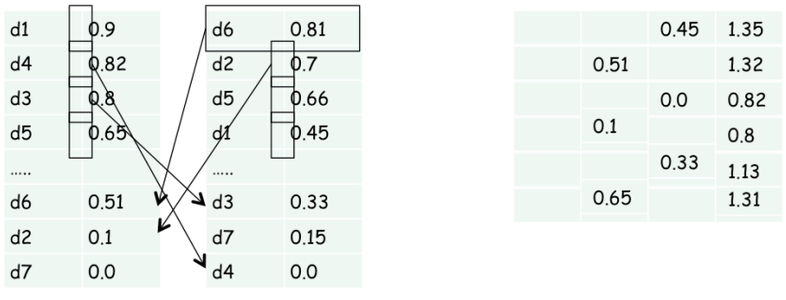
- Typical processing in centralized retrieval system
  - Aggregate the weights for ALL documents by scanning the posting lists of the query terms
  - Scanning relatively efficient
  - Computationally quite expensive (memory, processing)
- Distributed retrieval
  - Posting lists for different terms stored on different machines
  - In a distributed retrieval system 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?

As we have seen earlier, 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. In a centralized server this can be implemented relatively efficiently, though still resource-intensive, since scanning of disks is a comparably efficient operation. In a distributed setting the picture changes quite significantly. Assuming that posting lists for different terms are stored on different machines, the consequences of such an approach would be that complete posting lists would 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 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.

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
- Example: finding the top-2 elements for a two-term query



One approach to deal with this problem is based on 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).

Phase 1: The algorithm proceeds by scanning in a round-robin fashion the elements of the 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 elements 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, which is 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 the 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.

The example illustrates a case where two lists are searched, i.e. processing a query with two terms.

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 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 directly applying Fagin's algorithm 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. The latter application is similar to XML filtering which we have discussed earlier, with the difference that the user profiles are expressed as text queries that are evaluated against text documents.

Also alternative algorithms for solving the same problems have been proposed, that are known under the name of threshold algorithms. They work in a similar fashion, but have slightly different performance characteristics.

Question

- 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

Summary

- Which aspect of information retrieval is addressed by inverted files ?
- How do inverted files compare to other database indexing approaches ?
- How is an inverted list organized and which are methods to compress it ?
- Which is the file organization of an inverted file ?
- Why are different addressing granularities used in inverted files ?
- Which problem occurs in the construction of an inverted file and how is it addressed ?
- Which additional source of information can be used to rank Web pages ?
- What is the informal idea underlying PageRank ?
- Why is a source of rank used in PageRank and what can it be used for ?
- How is PageRank practically computed ?
- Why can distributed retrieval become expensive ?
- How does Fagin's algorithm work?

References

- Course material based on
  - Ricardo Baeza-Yates, Berthier Ribeiro-Neto, Modern Information Retrieval (ACM Press Series), Addison Wesley, 1999.
- Relevant articles
  - Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2, Article 6 (July 2006).
  - Sergey Brin , Lawrence Page, The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN Systems, v.30 n.1-7, p.107-117, April 1, 1998.
  - Jon M. Kleinberg: Authoritative Sources in a Hyperlinked Environment. JACM 46(5): 604-632 (1999)
  - Ronald Fagin. 2002. Combining fuzzy information: an overview. *SIGMOD Rec.* 31, 2 (June 2002), 109-118.