LAB 5: Indexing and Apache Lucene

1. **<u>Indexing</u>:** Finding relevant information such as words or sentences in all documents of a very large corpus may be a challenging task. Imagine that you want to find all documents with word "information" and your collection contains 1,000,000 documents. Performing a full-scan, i.e., searching throughout all documents, may take even hours, depending on the size of stored documents as well as on used hardware. Such **naïve** approach can be used only for very a small corpus. For large-scale collections, documents have to be **indexed**. An index is an auxiliary generated data that, in general (but not only!), stores information about locations of terms in documents. **Indexing** is a process by which an **index** is generated. When searching for a particular word, an index is used to quickly determine which documents contain such word. Consider the search results depicted in Figure 1. Finding 8,310,000,000 relevant documents in 0.52 second (January 2018) is indeed an impressive result. Without indexing, efficient search would not be possible. In this document two index structures are discussed: **inverted index, suffix tree**, and **suffix array**.

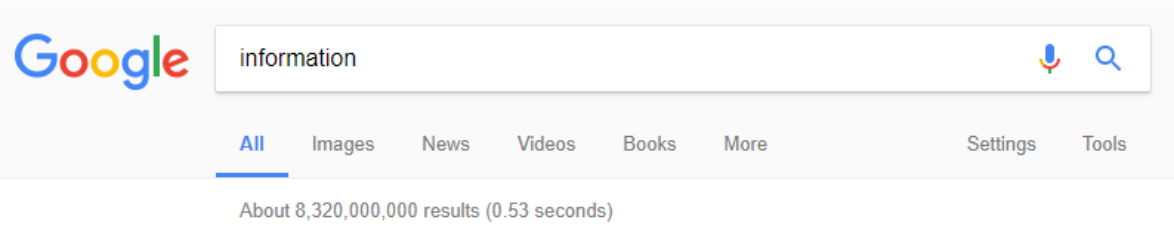

Figure 1: Google search results.

Inverted index is a simple yet very effective data structure that allows searching for terms throughout collection of documents. With each term there is an associated list of documents this term is contained in. Each document is represented by a unique serial number **docID (posting).** Similar, terms can be represented by **termID**. However, in this document only docIDs are used. For instance "information → [1 → 5 → 10]" means that the term "information" is contained in the 1st, 5th, and in the 10th document. The list "[1 → 5 → 10]" is referred to as a **posting list**. When using index structures data access is much faster at a cost of additional required memory. When constructing inverted index, some statistics are usually stored such as **document frequency** (the number of documents that contain this term) or document "quality" like, e.g., **page-rank** (Lab 10), or location of a term within documents. The last is referred to as **full inverted index**. An exemplary process of generation of inverted index is depicted in Figure 2. In this example four documents are considered. The process is divided into the following steps:

a) The collection is scanned and for each derived term a pair (term, posting) is generated.
b) The terms (pairs) are ordered alphabetically.
c) and d) The results are merged: the posting lists of the same terms are grouped and sorted. Some additional statistics may be included such as term location (**full inverted index**)

D1 = "new home sales top forecast

D2 = "home sales rise in july home"

D3 = "increase in home sales in july"

D4 = "july new home sales rise"

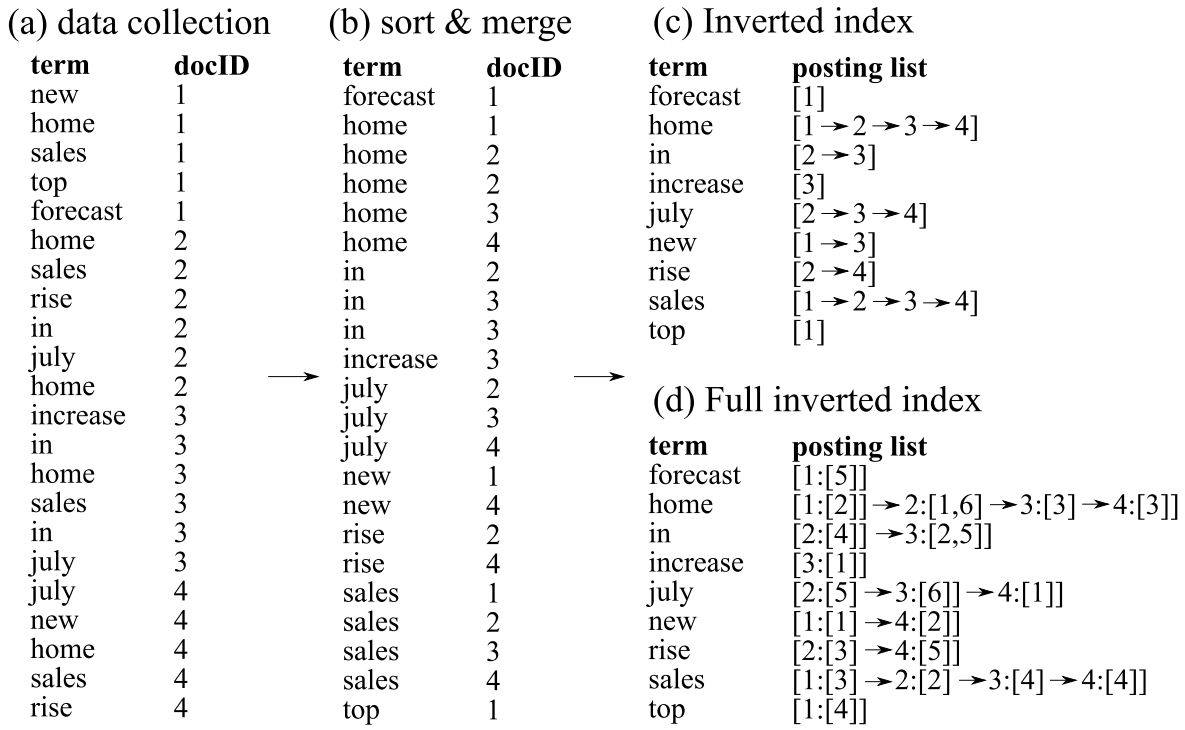

Figure 2: Process of generation of inverted index.

The list of terms usually requires much less space than posting lists and thus is kept in **memory**. A **hash table** may be used o get a quick access to corresponding posting lists resulting in O(1) search complexity. Since terms may be contained in multiple documents and documents may contain many different words (terms) posting lists are very large and should be stored on disks.

2. **Apache Lucene** is an open source search engine library written in Java. The main features of **Apache Lucene** are:
   - It is a high-performance search engine written in Java: low RAM requirements, fast incremental indexing, and small indexes.
   - Provides tools for ranked and fielded searching. Documents are sorted according to relevance or any field (e.g., “title” or “author”).
   - **Lucene** supports various complex types of query, e.g., phrase queries, wildcard queries, proximity queries, or range queries.
   - **Lucene** supports multiple-index searching and the results can be merged.
   - **Lucene** provides advanced tools for aiding users, such as faceting, highlighting, joins and result grouping.
   - **Lucene** can update database and allows searching at the same time.
   - **Lucene** ranking models can be configured/tuned by using plugins.

Exercises

Your task is to build a simple Lucene application. This exercise is divided into two sub-tasks:

- Firstly, you will build a simple index using Apache Lucene, based on the provided HTML documents.
- Then, you will make several queries and search the collection (index) for the most relevant documents.

Before going further:

- Download **Indexer.java, Searcher.java, Constant.java** and **pages.zip** from http://www.cs.put.poznan.pl/alabijak/ezi/lab5/. Create java project using any IDE and these files to the project (unzip pages folder!).
- From http://www.cs.put.poznan.pl/alabijak/ezi/lab5/, download **libraries.zip** and unzip it in the project's directory. Add the following jars to your project: **lucene-core-7.2.1.jar**, **lucene-queryparser-7.2.1.jar, lucene-backward-codecs-7.2.1.jar** and **tika-app-1.17.jar**.

Exercise 1. Firstly, use Apache Lucene to build construct an index from the given collection:

1) Go to indexer.java. This class is responsible for generating a Lucene index.
2) Seek for **indexDocuments()** method. There are several TODOs that must be completed.
3) Firstly, consider the method **getHTMLDocuments()**. It should load the files from the collection and construct Document (Apache Lucene class) objects. You may go to **getHTMLDocuments()** method and see that it iterates over the files in "pages" folder. This method is completed. You should go to **getHTMLDocument()** method that construct a Document object for a single File.
4) Read carefully the provided comments. Focus on the Document <-> Field relation. What is the Field? What does STORED and INDEXED mean? Complete this method (as you may notice, apache Tika is involved in content extraction ☺).
5) Go back to **indexDocuments()** method. Now, having a list of Documents, you should create an index using IndexWriter. Follow the provided comments.
6) If you completed all the TODOs, run Indexer.java. You should see that "index" folder is generated and it contains an index.

Exercise 2. Let's search for some documents.

7) Go to Searcher.java. It is suggested to start with **printResultsForQuery()** method.
8) This method is invoked after the Query object is constructed. As you may notice, IndexSearcherobject is passed as an argument. This object provides a method for seeking through the index for the first N documents that are the most relevant according to the query. Follow the provided comments and complete this method. The documents should be printed in the following form (each in a separate line):

   SCORE: FILENAME (Id=…ID) (Content=CONTENT) (Size=SIZE**b**).

   Why CONTENT will be null☺?

9) Now, go to **main()**. Follow the provided comments to generate several queries:
   a. TermQuery – seeking for documents that contain some term,
   b. BooleanQuery – boolean query based on terms, i.e., "medicine AND drug",
   c. RangeQuery – seeking for documents which values (according to some field) are in the provided range, e.g., size of the document,
   d. PrefixQuery – like TermQuery, but the prefix of a term is provided instead of a whole string (i.e., "antelope" and "ant" match "**ant"** prefix),
   e. WildcardQuery – You can use ?to indicate any single letter (e.g., "cats" matches "cat?") or use * to indicate multiple letters (e.g., "antelope" matches "ant*"),
   f. FuzzyQuery – Seeking for terms that are similar to the provided term, e.g., "mammal" matches "mamml".
   g. **Use** QueryParser to parse human-entered query strings.