MODUL TOPIK DALAM INFORMATION RETRIEVAL
(CMA 102)

MODUL PERTEMUAN 08
Boolean Retrieval

DISUSUN OLEH

Dr. Fransiskus Adikara, S.Kom, MMSI

UNIVERSITAS ESA UNGGUL

2019

INVERTED INDEX

A. Kemampuan Akhir Yang Diharapkan

After reading this session, you will be able to answer the following questions:

1. Index construction: how can we create inverted indexes for large collections?
2. How much space do we need for dictionary and index?
3. Index compression: how can we efficiently store and process indexes for large collections?
4. Ranked retrieval: what does the inverted index look like when we want the "best" answer?

B. Uraian dan Contoh

1.1. Introduction

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

The field of information retrieval covers supporting users in browsing or filtering document collections or further processing a set of retrieved documents. Given a set of documents, clustering is the task of coming up with a good grouping of the documents based on their contents. It is similar to arranging books on a bookshelf according to their topic. Given a set of topics, standing information needs, or other categories (such as suitability of texts for different age groups), classification is the task of deciding which class(es), if any, each of a set of documents belongs to. It is often approached by first manually classifying some documents and then hoping to be able to classify new documents automatically.

In this chapter we begin with a very simple example of an information retrieval problem, and introduce the idea of a term-document matrix (Section 1.2) and the central inverted index data structure (Section 1.3). We will then examine the Boolean retrieval model and how Boolean queries are processed (Sections 2).

1.2. An example information retrieval problem

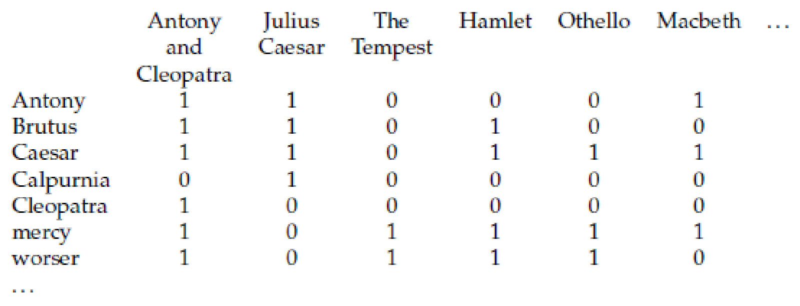
A fat book which many people own is Shakespeare's Collected Works. Suppose
You wanted to determine which plays of Shakespeare contain the words Brutus AND Caesar AND NOT Calpurnia. One way to do that is to start at the beginning and to read through all the text, noting for each play whether it contains Brutus and Caesar and excluding it from consideration if it contains Calpurnia. The simplest form of document retrieval is for a computer to do this sort of linear scan through documents. This process is commonly referred to as *grepping* through text, after the Unix command grep, which performs this process. Grepping through text can be a very effective process, especially given the speed of modern computers, and often allows useful possibilities for wildcard pattern matching through the use of regular expressions. With modern computers, for simple querying of modest collections (the size of Shakespeare's Collected Works is a bit under one million words of text in total), you really need nothing more.

But for many purposes, you do need more:

1) To process large document collections quickly. The amount of online data has grown at least as quickly as the speed of computers, and we would now like to be able to search collections that total in the order of billions to trillions of words.
2) To allow more flexible matching operations. For example, it is impractical to perform the query Romans NEAR countrymen with grep, where NEAR might be defined as “within 5 words” or “within the same sentence”.
3) To allow ranked retrieval: in many cases you want the best answer to an information need among many documents that contain certain words.

The way to avoid linearly scanning the texts for each query is to *index* the documents in advance. Let us stick with Shakespeare’s Collected Works, and use it to introduce the basics of the Boolean retrieval model. Suppose we record for each document – here a play of Shakespeare’s – whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000 different words). The result is a binary term-document *incidence* TERM *matrix*, as in Figure 1.1.

Terms are the indexed units; they are usually words, and for the moment you can think of them as words, but the information retrieval literature normally speaks of terms because some of them, such as perhaps I-9 or Hong Kong are not usually thought of as words. Now, depending on whether we look at the matrix rows or columns, we can have a vector for each term, which shows the documents it appears in, or a vector for each document, showing the terms that occur in it.



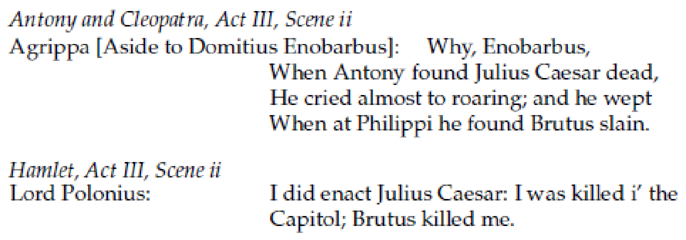
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

► **Figure 1.1** A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise AND:

110100 AND 110111 AND 101111 = 100100

The answers for this query are thus *Antony and Cleopatra* and *Hamlet* (Figure 1.2). The *Boolean retrieval model* is a model for information retrieval in which we can pose any query which is in the form of a Boolean expression of terms, that is, in which terms are combined with the operators AND, OR, and NOT. The model views each document as just a set of words.



Antony and Cleopatra, Act III, Scene ii
Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii
Lord Polonius: I did enact Julius Caesar: I was killed i’ the
Capitol; Brutus killed me.

► **Figure 1.2** Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia.

Let us now consider a more realistic scenario, simultaneously using the opportunity to introduce some terminology and notation. Suppose we have N = 1 million documents. By *documents* we mean whatever units we have decided to build a retrieval system over. They might be individual memos or chapters of a book. We will refer to the group of documents over which we perform retrieval as the (document) *collection*. It is sometimes also referred to as a *corpus* (a *body* of texts). Suppose each document is about 1000 words long (2–3 book pages). If we assume an average of 6 bytes per word including spaces and punctuation, then this is a document collection about 6 GB in size. Typically, there might be about M = 500,000 distinct terms in these documents. There is nothing special about the numbers we have chosen, and they might vary by an order of magnitude or more, but they give us some idea of the dimensions of the kinds of problems we need to handle.

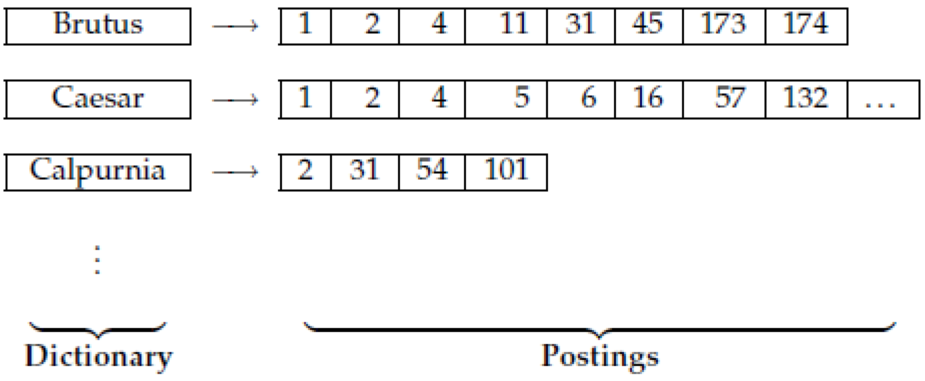
Our goal is to develop a system to address the *ad hoc retrieval* task. This is the most standard IR task. In it, a system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query. An *information need* is the topic about which the user desires to know more, and is differentiated from a *query*, which is what the user conveys to the computer in an attempt to communicate the information need. A document is *relevant* if it is one that the user perceives as containing information of value with respect to their personal information need. Our example above was rather artificial in that the information need was defined in terms of particular words, whereas usually a user is interested in a topic like “pipeline leaks” and would like to find relevant documents regardless of whether they precisely use those words or express the concept with other words such as pipeline rupture. To assess the *effectiveness* of an IR system (i.e., the quality of its search results), a user will usually want to know two key statistics about the system’s returned results for a query:

Precision: What fraction of the returned results are relevant to the information need?

Recall: What fraction of the relevant documents in the collection were returned by the system?

We now cannot build a term-document matrix in a naive way. A 500K ×1M matrix has half-a-trillion 0’s and 1’s – too many to fit in a computer’s memory. But the crucial observation is that the matrix is extremely sparse, that is, it has few non-zero entries. Because each document is 1000 words long, the matrix has no more than one billion 1’s, so a minimum of 99.8% of the cells are zero. A much better representation is to record only the things that do occur, that is, the 1 positions.

This idea is central to the first major concept in information retrieval, the *inverted index*. The name is actually redundant: an index always maps back from terms to the parts of a document where they occur. Nevertheless, *inverted index*, or sometimes *inverted file*, has become the standard term in information retrieval. The basic idea of an inverted index is shown in Figure 1.3. We keep a *dictionary* of terms (sometimes also referred to as a *vocabulary* or *lexicon*; in this book, we use *dictionary* for the data structure and *vocabulary* for the set of terms). Then for each term, we have a list that records which documents the term occurs in. Each item in the list – which records that a term appeared in a document (and, later, often, the positions in the document) – is conventionally called a *posting*. The list is then called a *postings list* (or inverted list), and all the postings lists taken together are referred to as the *postings*. The dictionary in Figure 1.3 has been sorted alphabetically and each postings list is sorted by document ID. We will see why this is useful in Section 1.3, below, but later we will also consider alternatives to doing this.



► **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

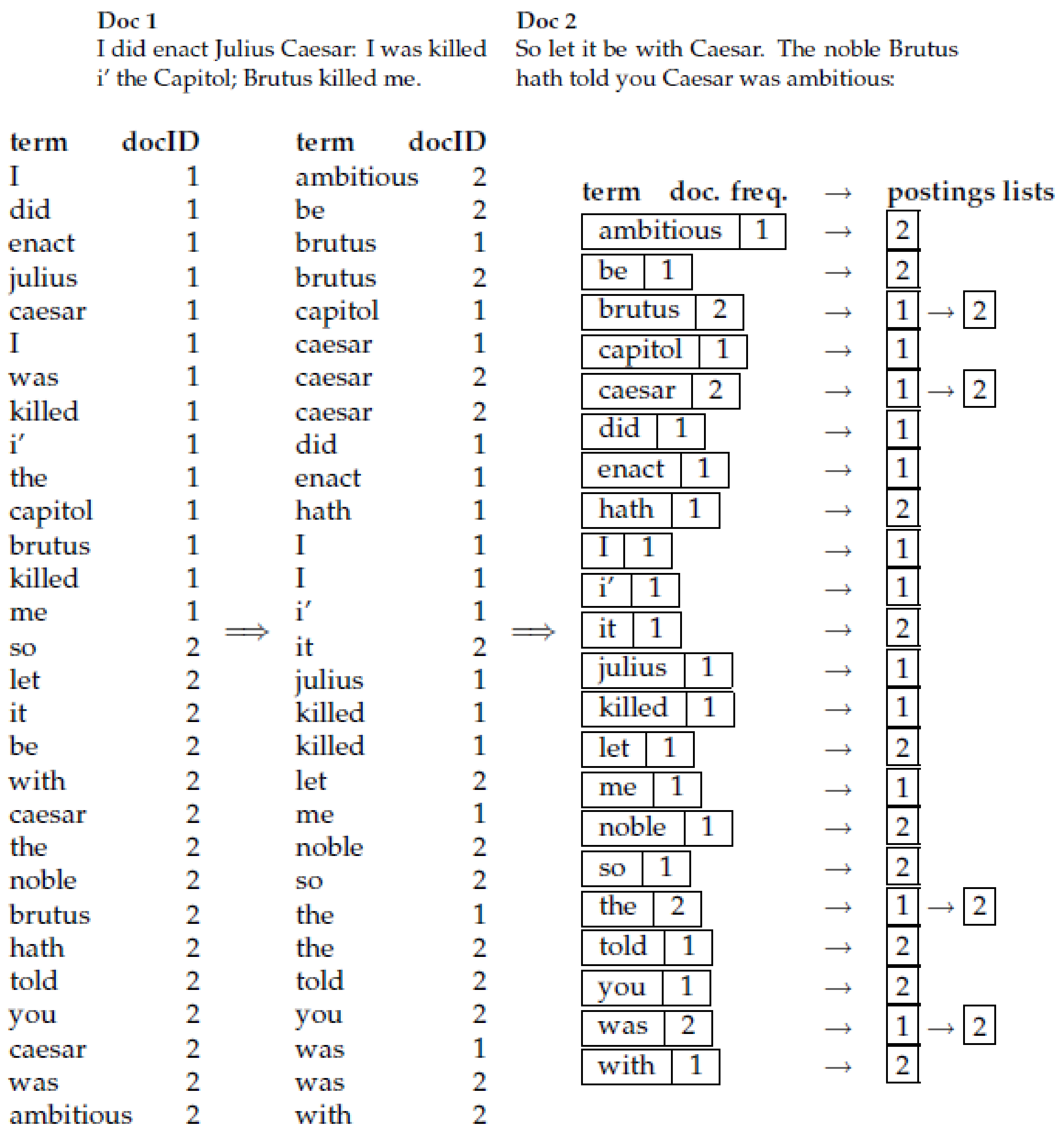
1.3. A First take at building an inverted index

To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in this are:

1. Collect the documents to be indexed:
   $\boxed{\text{Friends, Romans, countrymen.}}\ \boxed{\text{So let it be with Caesar}}\ \ldots$
2. Tokenize the text, turning each document into
   $\boxed{\text{Friends}}\ \boxed{\text{Romans}}\ \boxed{\text{countrymen}}\ \boxed{\text{So}}\ \ldots$
3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms:
   $\boxed{\text{friend}}\ \boxed{\text{roman}}\ \boxed{\text{countryman}}\ \boxed{\text{so}}\ \ldots$
4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index by sort-based indexing.

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (*docID*). During index construction, we can simply assign successive integers to each new document when it is first encountered. The input to indexing is a list of normalized tokens for each document, which we can equally think of as a list of pairs of term and docID, as in Figure 1.4. The core indexing step is *sorting* this list so that the terms are alphabetical, giving us the representation in the middle column of Figure 1.4. Multiple occurrences of the same term from the same document are then merged. Instances of the same term are then grouped, and the result is split into a *dictionary* and *postings*, as shown in the right column of Figure 1.4. Since a term generally occurs in a number of documents, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the number of documents which contain each term (the *document frequency*, which is here also the length of each postings list). This information is not vital for a basic Boolean search engine, but it allows us to improve the efficiency of the search engine at query time, and it is a statistic later used in many ranked retrieval models. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. This inverted index structure is essentially without rivals as the most efficient structure for supporting ad hoc text search.



Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

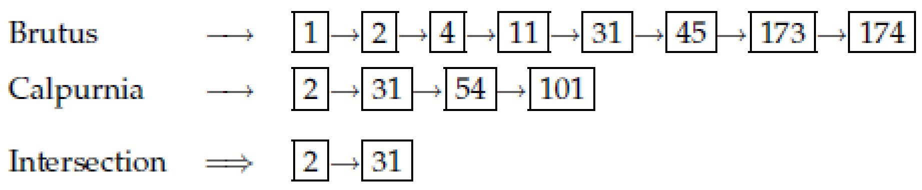
term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

$\Longrightarrow$

term	doc. freq.	$\rightarrow$	postings lists
ambitious	1	$\rightarrow$	2
be	1	$\rightarrow$	2
brutus	2	$\rightarrow$	1 $\rightarrow$ 2
capitol	1	$\rightarrow$	1
caesar	2	$\rightarrow$	1 $\rightarrow$ 2
did	1	$\rightarrow$	1
enact	1	$\rightarrow$	1
hath	1	$\rightarrow$	2
I	1	$\rightarrow$	1
i'	1	$\rightarrow$	1
it	1	$\rightarrow$	2
julius	1	$\rightarrow$	1
killed	1	$\rightarrow$	1
let	1	$\rightarrow$	2
me	1	$\rightarrow$	1
noble	1	$\rightarrow$	2
so	1	$\rightarrow$	2
the	2	$\rightarrow$	1 $\rightarrow$ 2
told	1	$\rightarrow$	2
you	1	$\rightarrow$	2
was	2	$\rightarrow$	1 $\rightarrow$ 2
with	1	$\rightarrow$	2

► **Figure 1.4** Building an index by sorting and grouping. The sequence of terms in each document, tagged by their documentID (left) is sorted alphabetically (middle). Instances of the same term are then grouped by word and then by documentID. The terms and documentIDs are then separated out (right). The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as, here, the document frequency of each term. We use this information for improving query time efficiency and, later, for weighting in ranked retrieval models. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency (the frequency of each term in each document) or the position(s) of the term in each document.

In the resulting index, we pay for storage of both the dictionary and the postings lists. The latter are much larger, but the dictionary is commonly kept in memory, while postings lists are normally kept on disk, so the size of each is important. What data structure should be used for a postings list? A fixed length array would be wasteful as some words occur in many documents, and others in very few. For an in-memory postings list, two good alternatives are singly linked lists or variable length arrays. Singly linked lists allow cheap insertion of documents into postings lists (following updates, such as when recrawling the web for updated documents), and naturally extend to more advanced indexing strategies such as skip lists, which require additional pointers. Variable length arrays win in space requirements by avoiding the overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches. Extra pointers can in practice be encoded into the lists as offsets. If updates are relatively infrequent, variable length arrays will be more compact and faster to traverse. We can also use a hybrid scheme with a linked list of fixed length arrays for each term. When postings lists are stored on disk, they are stored (perhaps compressed) as a contiguous run of postings without explicit pointers (as in Figure 1.3), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.



Brutus → 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174
Calpurnia → 2 → 31 → 54 → 101
Intersection ⇒ 2 → 31

► **Figure 1.5** Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3.

C. Latihan dan Jawaban

1) How good are the retrieved docs?

   - Precision : Fraction of retrieved docs that are relevant to user's information need.
   - Recall : Fraction of relevant docs in collection that are retrieved.
   - More precise definitions and measurements to follow in another lecture on evaluation.

2) Example of Input Collection :

Doc 1	=	English tutorial and fast track
Doc 2	=	learning latent semantic indexing
Doc 3	=	Book on semantic indexing
Doc 4	=	Advance in structure and semantic indexing
Doc 5	=	Analysis of latent structures

   Query Problem : advance and structure AND NOT analysis

First we build the term-document incidence matrix which represents a list of all the distinct terms and their presence on each document (incidence vector). If the document contains the term than incidence vector is 1 otherwise 0.

Doc **Terms**	**Doc1**	**Doc2**	**Doc3**	**Doc4**	**Doc5**
English	1	0	0	0	0
Tutorial	1	0	0	0	0
Fast	1	0	0	0	0
Track	1	0	0	0	0
Books	0	0	1	0	0
Semantic	0	1	1	1	0
Analysis	0	0	0	0	1
Learning	0	1	0	0	0
Latent	0	1	0	0	1
Indexing	0	1	1	1	0
Advance	0	0	0	1	0
Structures	0	0	0	1	1

So now we have 0/1 vector for each term. To answer the query we take the vectors for **advance, structure,** and **analysis,** complement the last, and the do a bitwise AND.

Doc1	**Doc2**	**Doc3**	**Doc4**	**Doc5**	
0	0	0	1	0	
0	0	0	1	1	(AND)
0	0	0	1	0	
1	1	1	1	0	(NOT analysis)
0	0	0	1	0	

$\downarrow$

Doc4

Hence Doc4 is retrieved here.

D. Daftar Pustaka

1. Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.