MODUL TOPIK DALAM INFORMATION RETRIEVAL

(CMA 102)

MODUL PERTEMUAN 09

Boolean Retrieval (Part 2)

DISUSUN OLEH

Dr. Fransiskus Adikara, S.Kom, MMSI

UNIVERSITAS ESA UNGGUL

2019

INVERTED INDEX – QUERY PROCESSING

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

2.1. Processing Boolean queries

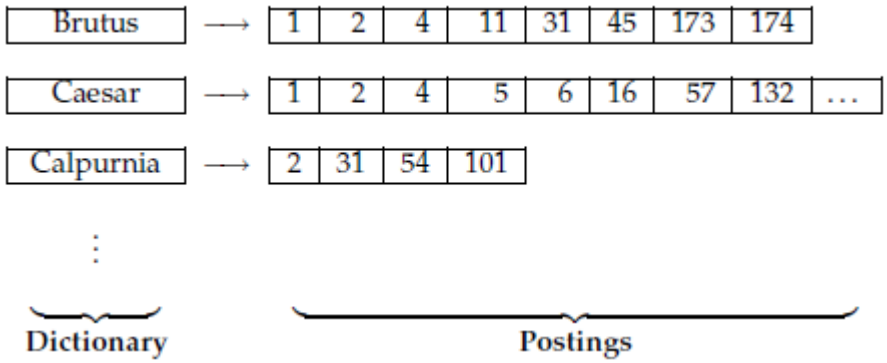
SIMPLE CONJUNCTIVE QUERIES (2.1)

How do we process a query using an inverted index and the basic Boolean retrieval model? Consider processing the *simple conjunctive query*:

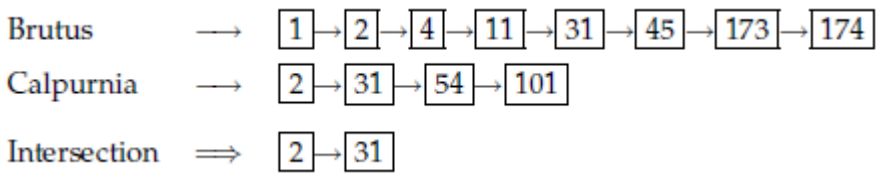
Brutus AND Calpurnia

over the inverted index partially shown in Figure 2.1. We:

1. Locate Brutus in the Dictionary
2. Retrieve its postings
3. Locate Calpurnia in the Dictionary
4. Retrieve its postings
5. Intersect the two postings lists, as shown in Figure 2.2.



► **Figure 2.1** 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.



► **Figure 2.2** Intersecting the postings lists for Brutus AND Calpurnia from Figure 2.1.

POSTINGS LIST INTERSECTION

POSTINGS MERGE

The *intersection* operation is the crucial one: we need to efficiently intersect postings lists so as to be able to quickly find documents that contain both terms. (This operation is sometimes referred to as *merging* postings lists: this slightly counterintuitive name reflects using the term *merge algorithm* for a general family of algorithms that combine multiple sorted lists by interleaved advancing of pointers through each; here we are merging the lists with a logical AND operation.)

There is a simple and effective method of intersecting postings lists using the merge algorithm (see Figure 2.3): we maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are *x* and *y*, the intersection takes $O(x + y)$ operations. Formally, the complexity of querying is $\Theta(N)$, where N is the number of documents in the collection.[6] Our indexing methods gain us just a constant, not a difference in Θ time complexity compared to a linear scan, but in practice the constant is huge. To use this algorithm, it is crucial that postings be sorted by a single global ordering. Using a numeric sort by docID is one simple way to achieve this.

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

► **Figure 2.3** Algorithm for the intersection of two postings lists p_1 dan p_2.

We can extend the intersection operation to process more complicated queries like:

(2.2) (Brutus OR Caesar) AND NOT Calpurnia

QUERY OPTIMIZATION

Query optimization is the process of selecting how to organize the work of answering a query so that the least total amount of work needs to be done by the system. A major element of this for Boolean queries is the order in which postings lists are accessed. What is the best order for query processing? Consider a query that is an AND of *t* terms, for instance:

(2.3) Brutus AND Caesar AND Calpurnia

6. The notation $\Theta(\cdot)$, is used to express an asy asymptotically tight bound on the complexity of an algorithm. Informally, this is often written as $O(\cdot)$, but this notation really expresses an asymptotic upper bound, which need not be tight.

For each of the *t* terms, we need to get its postings, then AND them together. The standard heuristic is to process terms in order of increasing document frequency: if we start by intersecting the two smallest postings lists, then all intermediate results must be no bigger than the smallest postings list, and we are therefore likely to do the least amount of total work. So, for the postings lists in Figure 2.1, we execute the above query as:

(2.4) (Calpurnia AND Brutus) AND Caesar

This is a first justification for keeping the frequency of terms in the dictionary: it allows us to make this ordering decision based on in-memory data before accessing any postings list.

Consider now the optimization of more general queries, such as:

(2.5) (madding OR crowd) AND (ignoble OR strife) AND (killed OR slain)

As before, we will get the frequencies for all terms, and we can then (conservatively) estimate the size of each OR by the sum of the frequencies of its disjuncts. We can then process the query in increasing order of the size of each disjunctive term.

For arbitrary Boolean queries, we have to evaluate and temporarily store the answers for intermediate expressions in a complex expression. However, in many circumstances, either because of the nature of the query language, or just because this is the most common type of query that users submit, a query is purely conjunctive. In this case, rather than viewing merging postings lists as a function with two inputs and a distinct output, it is more efficient to intersect each retrieved postings list with the current intermediate result in memory, where we initialize the intermediate result by loading the postings list of the least frequent term. This algorithm is shown in Figure 2.4. The intersection operation is then asymmetric: the intermediate results list is in memory while the list it is being intersected with is being read from disk. Moreover the intermediate results list is always at least as short as the other list, and in many cases it is orders of magnitude shorter. The postings intersection can still be done by the algorithm in Figure 2.3, but when the difference between the list lengths is very large, opportunities to use alternative techniques open up. The intersection can be calculated in place by destructively modifying or marking invalid items in the intermediate results list. Or the intersection can be done as a sequence of binary searches in the long postings lists for each posting in the intermediate results list. Another possibility is to store the long postings list as a hash table, so that membership of an intermediate result item can be calculated in constant rather than linear or log time.

```
INTERSECT(⟨t_1, ..., t_n⟩)
1  terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
2  result ← postings(first(terms))
3  terms ← rest(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do result ← INTERSECT(result, postings(first(terms)))
6     terms ← rest(terms)
7  return result
```

► **Figure 2.4** Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms.

?

Exercise 2.1 [⋆]

For the queries below, can we still run through the intersection in time $O(x + y)$, where x and y are the lengths of the postings lists for Brutus and Caesar? If not, what can we achieve?

a. Brutus AND NOT Caesar

b. Brutus OR NOT Caesar

Exercise 2.2 [⋆]

Extend the postings merge algorithm to arbitrary Boolean query formulas. What is its time complexity? For instance, consider:

a. (Brutus OR Caesar) AND NOT (Antony OR Cleopatra)

Can we always merge in linear time? Linear in what? Can we do better than this?

Exercise 2.3 [⋆⋆]

We can use distributive laws for AND and OR to rewrite queries.

a. Show how to rewrite the query in Exercise 2.2 into disjunctive normal form using the distributive laws.

b. Would the resulting query be more or less efficiently evaluated than the original form of this query?

c. Is this result true in general or does it depend on the words and the contents of the document collection?

Exercise 2.4 [⋆]

Recommend a query processing order for

a. (tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes)

given the following postings list sizes:

Term	Postings size
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

Exercise 2.5 [⋆]

If the query is:

a. friends AND romans AND (NOT countrymen)

how could we use the frequency of countrymen in evaluating the best query evaluation order? In particular, propose a way of handling negation in determining the order of query processing.

Exercise 2.6 [⋆⋆]

For a conjunctive query, is processing postings lists in order of size guaranteed to be optimal? Explain why it is, or give an example where it isn't.

Exercise 2.7 [⋆⋆]

Write out a postings merge algorithm, in the style of Figure 2.3, for an x OR y query.

Exercise 2.8 [⋆⋆]

How should the Boolean query x AND NOT y be handled? Why is naive evaluation of this query normally very expensive? Write out a postings merge algorithm that evaluates this query efficiently.

2.2. The extended Boolean model versus ranked retrieval

RANKED RETRIEVAL MODEL
FREE TEXT QUERIES

The Boolean retrieval model contrasts with *ranked retrieval models* such as the vector space model, in which users largely use *free text queries*, that is, just typing one or more words rather than using a precise language with operators for building up query expressions, and the system decides which documents best satisfy the query. Despite decades of academic research on the advantages of ranked retrieval, systems implementing the Boolean retrieval model were the main or only search option provided by large commercial information providers for three decades until the early 1990s (approximately the date of arrival of the World Wide Web). However, these systems did not have just the basic Boolean operations (AND, OR, and NOT) which we have presented so far. A strict Boolean expression over terms with an unordered results set is too limited for many of the information needs that people have, and these systems implemented extended Boolean retrieval models by incorporating additional operators such as term proximity operators. A *proximity operator* is a way of specifying that two terms in a query must occur close to each other in a document, where closeness may be measured by limiting the allowed number of intervening words or by reference to a structural unit such as a sentence or paragraph.

PROXIMITY OPERATOR

Example 1.1: Commercial Boolean searching: Westlaw.

Westlaw (http://www.westlaw.com/) is the largest commercial legal search service (in terms of the number of paying subscribers), with over half a million subscribers performing millions of searches a day over tens of terabytes of text data. The service was started in 1975. In 2005, Boolean search (called "Terms and Connectors" by Westlaw) was still the default, and used by a large percentage of users, although ranked free text querying (called "Natural Language" by Westlaw) was added in 1992.

Here are some example Boolean queries on Westlaw:

Information need: Information on the legal theories involved in preventing the disclosure of trade secrets by employees formerly employed by a competing company.
Query: "trade secret" /s disclos! /s prevent /s employe!

Information need: Requirements for disabled people to be able to access a workplace.
Query: disab! /p access! /s work-site work-place (employment /3 place)

Information need: Cases about a host's responsibility for drunk guests.
Query: host! /p (responsib! liab!) /p (intoxicat! drunk!) /p guest

Note the long, precise queries and the use of proximity operators, both uncommon in web search. Submitted queries average about ten words in length. Unlike web search conventions, a space between words represents disjunction (the tightest binding operator), & is AND and /s, /p, and /*k* ask for matches in the same sentence, same paragraph or within *k* words respectively. Double quotes give a *phrase search* (consecutive words). The exclamation mark (!) gives a trailing wildcard query; thus liab! matches all words starting with liab. Additionally work-site matches any of *worksite*, *work-site* or *work site*. Typical expert queries are usually carefully defined and incrementally developed until they obtain what look to be good results to the user.

Many users, particularly professionals, prefer Boolean query models. Boolean queries are precise: a document either matches the query or it does not. This offers the user greater control and transparency over what is retrieved. And some domains, such as legal materials, allow an effective means of document ranking within a Boolean model: Westlaw returns documents in reverse chronological order, which is in practice quite effective. In 2007, the majority of law librarians still seem to recommend terms and connectors for high recall searches, and the majority of legal users think they are getting greater control by using them. However, this does not mean that Boolean queries are more effective for professional searchers. Indeed, experimenting on a Westlaw subcollection, found that free text queries produced better results than Boolean queries prepared by Westlaw's own reference librarians for the majority of the information needs in his experiments. A general problem with Boolean search is that using AND operators tends to produce high precision but low recall searches, while using OR operators gives low precision but high recall searches, and it is difficult or impossible to find a satisfactory middle ground.

In this chapter, we have looked at the structure and construction of a basic inverted index, comprising a dictionary and postings lists. We introduced the Boolean retrieval model, and examined how to do efficient retrieval via linear time merges and simple query optimization. Here we just mention a few of the main additional things we would like to be able to do:

1. We would like to better determine the set of terms in the dictionary and to provide retrieval that is tolerant to spelling mistakes and inconsistent choice of words.

2. It is often useful to search for compounds or phrases that denote a concept such as “operating system”. As the Westlaw examples show, we might also wish to do proximity queries such as Gates NEAR Microsoft. To answer such queries, the index has to be augmented to capture the proximities of terms in documents.
3. A Boolean model only records term presence or absence, but often we would like to accumulate evidence, giving more weight to documents that have a term several times as opposed to ones that contain it only once. To be able to do this we need *term frequency* information (the number of times a term occurs in a document) in postings lists.

TERM FREQUENCY

4. Boolean queries just retrieve a set of matching documents, but commonly we wish to have an effective method to order (or “rank”) the returned results. This requires having a mechanism for determining a document score which encapsulates how good a match a document is for a query.

With these additional ideas, we will have seen most of the basic technology that supports ad hoc searching over unstructured information. Ad hoc searching over documents has recently conquered the world, powering not only web search engines but the kind of unstructured search that lies behind the large eCommerce websites. Although the main web search engines differ by emphasizing free text querying, most of the basic issues and technologies of indexing and querying remain the same, as we will see in later chapters. Moreover, over time, web search engines have added at least partial implementations of some of the most popular operators from extended Boolean models: phrase search is especially popular and most have a very partial implementation of Boolean operators. Nevertheless, while these options are liked by expert searchers, they are little used by most people and are not the main focus in work on trying to improve web search engine performance.

? **Exercise 2.9** [⋆]

Write a query using Westlaw syntax which would find any of the words professor, teacher, or lecturer in the same sentence as a form of the verb explain.

Exercise 2.10 [⋆]

Try using the Boolean search features on a couple of major web search engines. For instance, choose a word, such as burglar, and submit the queries (i) burglar, (ii) burglar AND burglar, and (iii) burglar OR burglar. Look at the estimated number of results and top hits. Do they make sense in terms of Boolean logic? Often they haven’t for major search engines. Can you make sense of what is going on? What about if you try different words? For example, query for (i) knight, (ii) conquer, and then (iii) knight OR conquer. What bound should the number of results from the first two queries place on the third query? Is this bound observed?

2.3. References and further reading

The practical pursuit of computerized information retrieval began in the late 1940s (Cleverdon 1991, Liddy 2005). A great increase in the production of scientific literature, much in the form of less formal technical reports rather than traditional journal articles, coupled with the availability of computers, led to interest in automatic document retrieval. However, in those days, document retrieval was always based on author, title, and keywords; full-text search came much later.

The article of Bush (1945) provided lasting inspiration for the new field:

> "Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, 'memex' will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that itmay be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory."

The term *Information Retrieval* was coined by Calvin Mooers in 1948/1950 (Mooers 1950).

In 1958, much newspaper attention was paid to demonstrations at a conference (see Taube and Wooster 1958) of IBM "auto-indexing" machines, based primarily on the work of H. P. Luhn. Commercial interest quickly gravitated towards Boolean retrieval systems, but the early years saw a heady debate over various disparate technologies for retrieval systems. For example Mooers (1961) dissented:

> "It is a common fallacy, underwritten at this date by the investment of several million dollars in a variety of retrieval hardware, that the algebra of George Boole (1847) is the appropriate formalism for retrieval system design. This view is as widely and uncritically accepted as it is wrong."

The observation of AND vs. OR giving you opposite extremes in a precision/ recall tradeoff, but not the middle ground comes from (Lee and Fox 1988).

The book (Witten et al. 1999) is the standard reference for an in-depth comparison of the space and time efficiency of the inverted index versus other possible data structures; a more succinct and up-to-date presentation appears in Zobel and Moffat (2006).

REGULAR EXPRESSIONS

Friedl (2006) covers the practical usage of *regular expressions* for searching. The underlying computer science appears in (Hopcroft et al. 2000).

C. Latihan dan Jawaban

1. Perhatikan beberapa dokumen berikut :

- Doc1

Algoritma	Genetik	dapat	digunakan	untuk	Optimasi	Fuzzy
1	2	3	4	5	6	7

- Doc2

Optimasi	fungsi	keanggotaan	pada	Fuzzy
1	2	3	4	5

- Doc3

Algoritma	Genetik	merupakan	algoritma	Learning
1	2	3	4	5

Boolean Query Retrieval : **Fuzzy OR NOT (Genetik AND Learning)**

- Set vocabulary :
  {algoritma, genetik, dapat, digunakan, untuk, optimasi, fuzzy, fungsi, keanggotaan, pada, merupakan, learning}

- Inverted Index sederhana :

Term	Inverted List
algoritma	Doc1, Doc3
genetik	Doc1, Doc3
dapat	Doc1
digunakan	Doc1
untuk	Doc1
optimasi	Doc1, Doc2
fuzzy	Doc1, Doc2
fungsi	Doc2
keanggotaan	Doc2
pada	Doc2
merupakan	Doc3
learning	Doc3

- Bentuk kompleks dari Inverted Index :

Term	Inverted List
algoritma	<Doc1, 1, [1]>, <Doc3, 2, [1, 4]>
genetik	<Doc1, 1, [2]>, <Doc3, 1, [2]>
dapat	<Doc1, 1, [3]>
digunakan	<Doc1, 1, [4]>
untuk	<Doc1, 1, [5]>
optimasi	<Doc1, 1, [6]>, <Doc2, 1, [1]>
fuzzy	<Doc1, 1, [7]>, <Doc2, 1, [5]>
fungsi	<Doc2, 1, [2]>
keanggotaan	<Doc2, 1, [3]>
pada	<Doc2, 1, [4]>
merupakan	<Doc3, 1, [3]>
learning	<Doc3, 1, [5]>

➢ Tentukan biner dari Boolean Query Retrieval : **Fuzzy OR NOT (Genetik AND Learning)**

- TF_{biner}(Fuzzy) = 110
- TF_{biner}(Genetik) = 101
- TF_{biner}(Learning) = 001

Term	Inverted List
genetik	<Doc1, 1, [2]>, <Doc3, 1, [2]>
fuzzy	<Doc1, 1, [7]>, <Doc2, 1, [5]>
learning	<Doc3, 1, [5]>

➢ **Fuzzy OR NOT (Genetik AND Learning)**

= 110 OR NOT (101 AND 001)
= 110 OR NOT (001)
= 110 OR 110
= **110**

➢ Jadi, hasil Boolean Query Retrieval : **Fuzzy OR NOT (Genetik AND Learning)** adalah Dokumen 1 dan Dokumen 2.

2. Term postings :

white → [1, 2, 7, 19, 174, 210, 331, 2046]
black → [2, 3, 7, 11, 94, 210]
blue → [2, 7, 24, 2001]
red → [8, 19, 94]

Query :

(white AND black AND blue) OR (white AND red) OR (black AND red)

➢ Conjuction postings :

(white AND black AND blue) → [2, 7]
(white AND red) → [19]
(black AND red) → [94]

➢ OR merges :

First = (white AND red) OR (black AND red) → [19, 94]
Second= (white AND black AND blue) OR result-First → [2, 7, 19, 94]

➢ Intersection ➔ [2, 7, 19, 94]

D. Daftar Pustaka

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