



**MODUL TOPIK DALAM INFORMATION RETRIEVAL
(CMA 102)**

**MODUL PERTEMUAN 12
The Term Vocabulary and Postings Lists (Part 3)**

**DISUSUN OLEH
Dr. Fransiskus Adikara, S.Kom, MMSI**

Universitas
Esa Unggul

**UNIVERSITAS ESA UNGGUL
2019**

FASTER POSTINGS LIST INTERSECTION VIA SKIP POINTERS

A. Kemampuan Akhir Yang Diharapkan

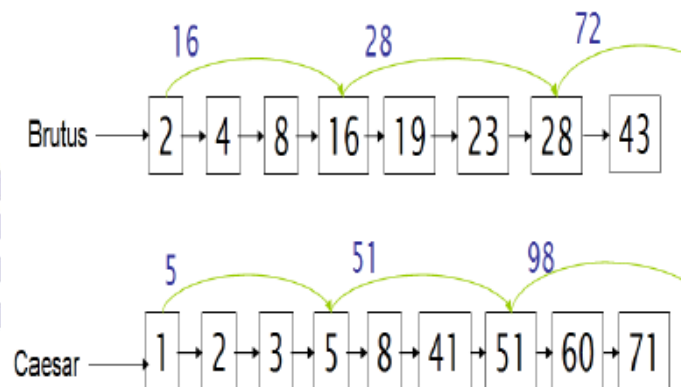
After reading this session, you will be able to answer the following questions:

1. Understanding of the basic unit of classical information retrieval systems: words and documents: What is a document, what is a term?
2. Tokenization: how to get from raw text to words (or tokens)
3. More complex indexes: skip pointers and phrases

B. Uraian dan Contoh

In the remainder of this chapter, we will discuss extensions to postings list data structures and ways to increase the efficiency of using postings lists. Recall the basic postings list intersection operation: we walk through the two postings lists simultaneously, in time linear in the total number of postings entries. If the list lengths are m and n , the intersection takes $O(m + n)$ operations. Can we do better than this? That is, empirically, can we usually process postings list intersection in sublinear time? We can, if the index isn't changing too fast.

SKIP LIST One way to do this is to use a *skip list* by augmenting postings lists with skip pointers (at indexing time), as shown in Figure 3.1. Skip pointers are effectively shortcuts that allow us to avoid processing parts of the postings list that will not figure in the search results. The two questions are then where to place skip pointers and how to do efficient merging using skip pointers.



► **Figure 3.1** Postings lists with skip pointers. The postings intersection can use a skip pointer when the end point is still less than the item on the other list.

Consider first efficient merging, with Figure 3.1 as an example. Suppose we've stepped through the lists in the figure until we have matched 8 on each list and moved it to the results list. We advance both pointers, giving us 16 on the upper list and 41 on the lower list. The smallest item is then the element 16 on the top list. Rather than simply advancing the upper pointer, we first check the skip list pointer and note that 28 is also less than 41. Hence we can follow the skip list pointer, and then we advance the upper pointer to 28. We thus avoid stepping to 19 and 23 on the upper list. A number of variant versions of postings list intersection with skip pointers is possible depending on when exactly you check

the skip pointer. One version is shown in Figure 3.2. Skip pointers will only be available for the original postings lists. For an intermediate result in a complex query, the call *hasSkip(p)* will always return false. Finally, note that the presence of skip pointers only helps for AND queries, not for OR queries.

```

INTERSECTWITHSKIPS(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 if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
9         then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
10            do p1 ← skip(p1)
11            else p1 ← next(p1)
12        else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13            then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
14                do p2 ← skip(p2)
15                else p2 ← next(p2)
16  return answer

```

► **Figure 3.2** Postings lists intersection with skip pointers.

Where do we place skips? There is a tradeoff. More skips means shorter skip spans, and that we are more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparisons, but then long skip spans which means that there will be fewer opportunities to skip. A simple heuristic for placing skips, which has been found to work well in practice, is that for a postings list of length P , use \sqrt{P} evenly-spaced skip pointers. This heuristic can be improved upon; it ignores any details of the distribution of query terms.

Building effective skip pointers is easy if an index is relatively static; it is harder if a postings list keeps changing because of updates. A malicious deletion strategy can render skip lists ineffective.

Choosing the optimal encoding for an inverted index is an ever-changing game for the system builder, because it is strongly dependent on underlying computer technologies and their relative speeds and sizes. Traditionally, CPUs were slow, and so highly compressed techniques were not optimal. Now CPUs are fast and disk is slow, so reducing disk postings list size dominates. However, if you're running a search engine with everything in memory then the equation changes again. We discuss the impact of hardware parameters on index construction and the impact of index size on system speed.

Exercise 3.1

[*]

Why are skip pointers not useful for queries of the form x OR y ?

Exercise 3.2

[*]

We have a two-word query. For one term the postings list consists of the following 16 entries:

[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]

and for the other it is the one entry postings list:

[47].

Work out how many comparisons would be done to intersect the two postings lists with the following two strategies. Briefly justify your answers:

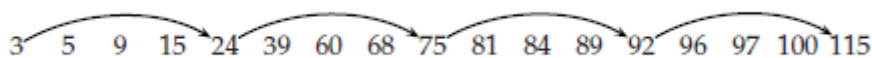
- Using standard postings lists
- Using postings lists stored with skip pointers, with a skip length of \sqrt{P} .

Exercise 3.3

[*]

Consider a postings intersection between this postings list, with skip pointers:

3 5 9 15 24 39 60 68 75 81 84 89 92 96 97 100 115



and the following intermediate result postings list (which hence has no skip pointers):

3 5 89 95 97 99 100 101

Trace through the postings intersection algorithm in Figure 3.2.

- How often is a skip pointer followed (i.e., p_1 is advanced to $skip(p_1)$)?
- How many postings comparisons will be made by this algorithm while intersecting the two lists?
- How many postings comparisons would be made if the postings lists are intersected without the use of skip pointers?

C. Latihan dan Jawaban

- We have a two-word query. For one term the postings list consists of the following 16 entries:

[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]

and for the other it is the one entry postings list:

[47].

Work out how many comparisons would be done to intersect the two postings lists with the following two strategies. Briefly justify your answers:

- Using standard postings lists
- Using postings lists stored with skip pointers, with a skip length of \sqrt{P} .

Solution :

- Applying MERGE on the standard postings list, comparisons will be made unless either of the postings list end i.e. till we reach 47 in the upper postings list, after which the lower list ends and no more processing needs to be done. Number of comparisons = 11.

b. Using skip pointers of length 4 for the longer list and of length 1 for the shortest list, the following comparisons will be made :

1. 4 & 47
2. 14 & 47
3. 22 & 47
4. 120 & 47
5. 81 & 47
6. 47 & 47

Number of comparisons = 6.

D. Daftar Pustaka

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



POSITIONAL POSTINGS AND PHRASE QUERIES

A. Kemampuan Akhir Yang Diharapkan

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

1. Understanding of the basic unit of classical information retrieval systems: words and documents: What is a document, what is a term?
2. Tokenization: how to get from raw text to words (or tokens)
3. More complex indexes: skip pointers and phrases

B. Uraian dan Contoh

PHRASE QUERIES

Many complex or technical concepts and many organization and product names are multiword compounds or phrases. We would like to be able to pose a query such as Stanford University by treating it as a phrase so that a sentence in a document like *The inventor Stanford Ovshinsky never went to university.* is not a match. Most recent search engines support a double quotes syntax (“stanford university”) for *phrase queries*, which has proven to be very easily understood and successfully used by users. As many as 10% of web queries are phrase queries, and many more are implicit phrase queries (such as person names), entered without use of double quotes. To be able to support such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms. In this section we consider two approaches to supporting phrase queries and their combination. A search engine should not only support phrase queries, but implement them efficiently. A related but distinct concept is term proximity weighting, where a document is preferred to the extent that the query terms appear close to each other in the text. This technique is covered in the context of ranked retrieval.

4.1. Biword indexes

BIWORD INDEX

One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase. For example, the text *Friends, Romans, Countrymen* would generate the *biwords*:

friends romans
romans countrymen

In this model, we treat each of these biwords as a vocabulary term. Being able to process two-word phrase queries is immediate. Longer phrases can be processed by breaking them down. The query stanford university palo alto can be broken into the Boolean query on biwords:

“stanford university” AND “university palo” AND “palo alto”

This query could be expected to work fairly well in practice, but there can and will be occasional false positives. Without examining the documents, we cannot verify that the documents matching the above Boolean query do actually contain the original 4 word phrase.

Among possible queries, nouns and noun phrases have a special status in describing the concepts people are interested in searching for. But related nouns can often be divided from each other by various function words, in phrases such as *the abolition of slavery* or *renegotiation of the constitution*. These needs can be incorporated into the biword indexing model in the following way. First, we tokenize the text and perform part-of-speech-tagging. We can then group terms into nouns, including proper nouns, (N) and function words, including articles and prepositions, (X), among other classes. Now deem any string of terms of the form NX*N to be an extended biword. Each such extended biword is made a term in the vocabulary. For example:

renegotiation	of	the	constitution
N	X	X	N

To process a query using such an extended biword index, we need to also parse it into N's and X's, and then segment the query into extended biwords, which can be looked up in the index.

This algorithm does not always work in an intuitively optimal manner when parsing longer queries into Boolean queries. Using the above algorithm, the query

cost overruns on a power plant

is parsed into

“cost overruns” AND “overruns power” AND “power plant”

whereas it might seem a better query to omit the middle biword. Better results can be obtained by using more precise part-of-speech patterns that define which extended biwords should be indexed.

The concept of a biword index can be extended to longer sequences of words, and if the index includes variable length word sequences, it is generally referred to as a *phrase index*. Indeed, searches for a single term are not naturally handled in a biword index (you would need to scan the dictionary for all biwords containing the term), and so we also need to have an index of single-word terms. While there is always a chance of false positive matches, the chance of a false positive match on indexed phrases of length 3 or more becomes very small indeed. But on the other hand, storing longer phrases has the potential to greatly expand the vocabulary size. Maintaining exhaustive phrase indexes for phrases of length greater than two is a daunting prospect, and even use of an exhaustive biword dictionary greatly expands the size of the vocabulary. However, towards the end of this section we discuss the utility of the strategy of using a partial phrase index in a compound indexing scheme.

PHRASE INDEX

4.2. Positional indexes

For the reasons given, a biword index is not the standard solution. Rather, a *positional index* is most commonly employed. Here, for each term in the vocabulary, we store postings of the form docID: (position1, position2, . . .), as shown in Figure 4.1, where each position is a token index in the document. Each posting will also usually record the term frequency.

POSITIONAL INDEX

```

to, 993427:
  < 1, 6: <7, 18, 33, 72, 86, 231>;
    2, 5: <1, 17, 74, 222, 255>;
    4, 5: <8, 16, 190, 429, 433>;
    5, 2: <363, 367>;
    7, 3: <13, 23, 191>;...>

```

```

be, 178239:
  < 1, 2: <17, 25>;
    4, 5: <17, 191, 291, 430, 434>;
    5, 3: <14, 19, 101>;...>

```

- **Figure 4.1** Positional index example. The word *to* has a document frequency 993,477, and occurs 6 times in document 1 at positions 7, 18, 33, etc.

To process a phrase query, you still need to access the inverted index entries for each distinct term. As before, you would start with the least frequent term and then work to further restrict the list of possible candidates. In the merge operation, the same general technique is used as before, but rather than simply checking that both terms are in a document, you also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. This requires working out offsets between the words.

Example 4.1: Satisfying phrase queries. Suppose the postings lists for *to* and *be* are as in Figure 4.1, and the query is “to be or not to be”. The postings lists to access are: *to*, *be*, *or*, *not*. We will examine intersecting the postings lists for *to* and *be*. We first look for documents that contain both terms. Then, we look for places in the lists where there is an occurrence of *be* with a token index one higher than a position of *to*, and then we look for another occurrence of each word with token index 4 higher than the first occurrence. In the above lists, the pattern of occurrences that is a possible match is:

```

to: <...; 4: <... ,429,433>; ...>
be: <...; 4: <... ,430,434>; ...>

```

The same general method is applied for within k word proximity searches:

employment /3 place

Here, $/k$ means “within k words of (on either side)”. Clearly, positional indexes can be used for such queries; biword indexes cannot. We show in Figure 4.2 an algorithm for satisfying within k word proximity searches; it is further discussed in Exercise 4.5.

```

POSITIONALINTERSECT( $p_1, p_2, k$ )
1  answer  $\leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4     then  $l \leftarrow \langle \rangle$ 
5          $pp_1 \leftarrow \text{positions}(p_1)$ 
6          $pp_2 \leftarrow \text{positions}(p_2)$ 
7         while  $pp_1 \neq \text{NIL}$ 
8         do while  $pp_2 \neq \text{NIL}$ 
9             do if  $|\text{pos}(pp_1) - \text{pos}(pp_2)| \leq k$ 
10                then ADD( $l, \text{pos}(pp_2)$ )
11                else if  $\text{pos}(pp_2) > \text{pos}(pp_1)$ 
12                   then break

```



```

13          $pp_2 \leftarrow next(pp_2)$ 
14     while  $l \neq \langle \rangle$  and  $|l[0] - pos(pp_1)| > k$ 
15     do DELETE( $l[0]$ )
16     for each  $ps \in l$ 
17     do ADD( $answer, (docID(p_1), pos(pp_1), ps)$ )
18      $pp_1 \leftarrow next(pp_1)$ 
19      $p_1 \leftarrow next(p_1)$ 
20      $p_2 \leftarrow next(p_2)$ 
21     else if  $docID(p_1) < docID(p_2)$ 
22     then  $p_1 \leftarrow next(p_1)$ 
23     else  $p_2 \leftarrow next(p_2)$ 
24 return  $answer$ 

```

Figure 4.2 An algorithm for proximity intersection of postings lists p_1 and p_2 . The algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in p_1 and p_2 .

Positional index size. Adopting a positional index expands required postings storage significantly, even if we compress position values/offsets. Indeed, moving to a positional index also changes the asymptotic complexity of a postings intersection operation, because the number of items to check is now bounded not by the number of documents but by the total number of tokens in the document collection T . That is, the complexity of a Boolean query is $\Theta(T)$ rather than $\Theta(M)$. However, most applications have little choice but to accept this, since most users now expect to have the functionality of phrase and proximity searches.

Let's examine the space implications of having a positional index. A posting now needs an entry for each occurrence of a term. The index size thus depends on the average document size. The average web page has less than 1000 terms, but documents like SEC stock filings, books, and even some epic poems easily reach 100,000 terms. Consider a term with frequency 1 in 1000 terms on average. The result is that large documents cause an increase of two orders of magnitude in the space required to store the postings list:

Document size	Expected postings	Expected entries in positional posting
1000	1	1
100,000	1	100

While the exact numbers depend on the type of documents and the language being indexed, some rough rules of thumb are to expect a positional index to be 2 to 4 times as large as a non-positional index, and to expect a compressed positional index to be about one third to one half the size of the raw text (after removal of markup, etc.) of the original uncompressed documents.

4.3. Combination schemes

The strategies of biword indexes and positional indexes can be fruitfully combined. If users commonly query on particular phrases, such as Michael Jackson, it is quite inefficient to keep merging positional postings lists. A combination strategy uses a phrase index, or just a biword index, for certain queries and uses a positional index for other phrase queries. Good queries to

include in the phrase index are ones known to be common based on recent querying behavior. But this is not the only criterion: the most expensive phrase queries to evaluate are ones where the individual words are common but the desired phrase is comparatively rare. Adding *Britney Spears* as a phrase index entry may only give a speedup factor to that query of about 3, since most documents that mention either word are valid results, whereas adding *The Who* as a phrase index entry may speed up that query by a factor of 1000. Hence, having the latter is more desirable, even if it is a relatively less common query.

NEXT WORD
INDEX

Williams et al. (2004) evaluate an even more sophisticated scheme which employs indexes of both these sorts and additionally a partial next word index as a halfway house between the first two strategies. For each term, a *next word index* records terms that follow it in a document. They conclude that such a strategy allows a typical mixture of web phrase queries to be completed in one quarter of the time taken by use of a positional index alone, while taking up 26% more space than use of a positional index alone.

Exercise 4.1 [*]

Assume a biword index. Give an example of a document which will be returned for a query of New York University but is actually a false positive which should not be returned.

Exercise 4.2 [*]

Shown below is a portion of a positional index in the format: term: doc1: (position1, position2, . . .); doc2: (position1, position2, . . .); etc.

```
angels: 2: <36,174,252,651>; 4: <12,22,102,432>; 7: <17>;
fools: 2: <1,17,74,222>; 4: <8,78,108,458>; 7: <3,13,23,193>;
fear: 2: <87,704,722,901>; 4: <13,43,113,433>; 7: <18,328,528>;
in: 2: <3,37,76,444,851>; 4: <10,20,110,470,500>; 7: <5,15,25,195>;
rush: 2: <2,66,194,321,702>; 4: <9,69,149,429,569>; 7: <4,14,404>;
to: 2: <47,86,234,999>; 4: <14,24,774,944>; 7: <199,319,599,709>;
tread: 2: <57,94,333>; 4: <15,35,155>; 7: <20,320>;
where: 2: <67,124,393,1001>; 4: <11,41,101,421,431>; 7: <16,36,736>;
```

Which document(s) if any match each of the following queries, where each expression within quotes is a phrase query?

- a. "fools rush in"
- b. "fools rush in" AND "angels fear to tread"

Exercise 4.3 [*]

Consider the following fragment of a positional index with the format:

```
word: document: <position, position, ...>; document: <position, ...>
...
```

```
Gates: 1: <3>; 2: <6>; 3: <2,17>; 4: <1>;
IBM: 4: <3>; 7: <14>;
Microsoft: 1: <1>; 2: <1,21>; 3: <3>; 5: <16,22,51>;
```

The $/k$ operator, $\text{word1} /k \text{word2}$ finds occurrences of word1 within k words of word2 (on either side), where k is a positive integer argument. Thus $k = 1$ demands that word1 be adjacent to word2 .

- a. Describe the set of documents that satisfy the query $\text{Gates} /2 \text{Microsoft}$.
- b. Describe each set of values for k for which the query $\text{Gates} /k \text{Microsoft}$ returns a different set of documents as the answer.

Exercise 4.4

[**]

Consider the general procedure for merging two positional postings lists for a given document, to determine the document positions where a document satisfies a $/k$ clause (in general there can be multiple positions at which each term occurs in a single document). We begin with a pointer to the position of occurrence of each term and move each pointer along the list of occurrences in the document, checking as we do so whether we have a hit for $/k$. Each move of either pointer counts as a step. Let L denote the total number of occurrences of the two terms in the document. What is the big-O complexity of the merge procedure, if we wish to have postings including positions in the result?

Exercise 4.5

[**]

Suppose we wish to use a postings intersection procedure to determine simply the list of documents that satisfy a $/k$ clause, rather than returning the list of positions, as in Figure 4.2. For simplicity, assume $k \geq 2$. Let L denote the total number of occurrences of the two terms in the document collection (i.e., the sum of their collection frequencies). Which of the following is true? Justify your answer.

- a. The merge can be accomplished in a number of steps linear in L and independent of k , and we can ensure that each pointer moves only to the right.
- b. The merge can be accomplished in a number of steps linear in L and independent of k , but a pointer may be forced to move non-monotonically (i.e., to sometimes back up)
- c. The merge can require kL steps in some cases.

Exercise 4.6

[**]

How could an IR system combine use of a positional index and use of stop words? What is the potential problem, and how could it be handled?

4.4. References and further reading

EAST ASIAN
LANGUAGES

Exhaustive discussion of the character-level processing of East Asian languages can be found in [Lunde \(1998\)](#). Character bigram indexes are perhaps the most standard approach to indexing Chinese, although some systems use word segmentation. Due to differences in the language and writing system, word segmentation is most usual for Japanese ([Luk and Kwok 2002](#), [Kishida et al. 2005](#)). The structure of a character k -gram index over unsegmented text differs: there the k -gram dictionary points to postings lists of entries in the regular dictionary, whereas here it points directly to document postings lists. For further discussion of Chinese word segmentation, see [Sproat](#)

et al. (1996), Sproat and Emerson (2003), Tseng et al.(2005), and Gao et al. (2005).

Lita et al. (2003) present a method for truecasing. Natural language processing work on computational morphology is presented in (Sproat 1992, Beesley and Karttunen 2003).

Language identification was perhaps first explored in cryptography; for example, Konheim (1981) presents a character-level k -gram language identification algorithm. While other methods such as looking for particular distinctive function words and letter combinations have been used, with the advent of widespread digital text, many people have explored the character n -gram technique, and found it to be highly successful (Beesley 1998, Dunning 1994, Cavnar and Trenkle 1994). Written language identification is regarded as a fairly easy problem, while spoken language identification remains more difficult; see Hughes et al. (2006) for a recent survey.

Experiments on and discussion of the positive and negative impact of stemming in English can be found in the following works: Salton (1989), Harman (1991), Krovetz (1995), Hull (1996). Hollink et al. (2004) provide detailed results for the effectiveness of language-specific methods on 8 European languages. In terms of percent change in mean average precision over a baseline system, diacritic removal gains up to 23% (being especially helpful for Finnish, French, and Swedish). Stemming helped markedly for Finnish (30% improvement) and Spanish (10% improvement), but for most languages, including English, the gain from stemming was in the range 0 – 5%, and results from a lemmatizer were poorer still. Compound splitting gained 25% for Swedish and 15% for German, but only 4% for Dutch. Rather than language-particular methods, indexing character k -grams (as we suggested for Chinese) could often give as good or better results: using within word character 4-grams rather than words gave gains of 37% in Finnish, 27% in Swedish, and 20% in German, while even being slightly positive for other languages, such as Dutch, Spanish, and English. Tomlinson (2003) presents broadly similar results. Bar-Ilan and Gutman (2005) suggest that, at the time of their study (2003), the major commercial web search engines suffered from lacking decent language-particular processing; for example, a query on www.google.fr for l'électricité did not separate off the article *l'* but only matched pages with precisely this string of article+noun.

SKIP LIST The classic presentation of skip pointers for IR can be found in Moffat and Zobel (1996). Extended techniques are discussed in Boldi and Vigna (2005). The main paper in the algorithms literature is Pugh (1990), which uses multilevel skip pointers to give expected $O(\log P)$ list access (the same expected efficiency as using a tree data structure) with less implementational complexity. In practice, the effectiveness of using skip pointers depends on various system parameters. Moffat and Zobel (1996) report conjunctive queries running about five times faster with the use of skip pointers, but Bahle et al.(2002, p. 217) report that, with modern CPUs, using skip lists instead slows down search because it expands the size of the postings list (i.e., disk I/O dominates performance). In contrast, Strohman and Croft (2007) again show good performance gains from skipping, in a system architecture designed to optimize for the large memory spaces and multiple cores of recent CPUs.

Johnson et al. (2006) report that 11.7% of all queries in two 2002 web query logs contained phrase queries, though Kammenhuber et al. (2006) report only 3% phrase queries for a different data set. Silverstein et al. (1999) note that many queries without explicit phrase operators are actually implicit phrase searches.

C. Latihan dan Jawaban

1. How could an IR system combine use of a positional index and use of stop words? What is the potential problem, and how could it be handled?

Solution :

Is the problem referred to in this question is the problem faced in constructing the positional index after removal of stop words as this preprocessing changes the positions of terms in the original text? As far as the first part of the question is concerned, can you give a hint of what kind of use is the question looking for? I am assuming the answer of the question is not the following; Phrasal queries can handled using both of them. For any query, remove the stop-words and merge the positional indexes of the remaining terms looking for exact phrasal match by determining relative positions.

2. Penerapan *Case-Folding*, *Tokenisasi*, *Filtering*, *Stemming*, dan *Biword*.

Input : Dalam setahun belakangan ini, pengaksesan KRS diganti ke SIAM (sebelumnya menggunakan SINERGI). Saat menggunakan SINERGI, fitur serta kecepatan akses sangat handal dan nyaman. Tapi setelah diganti menggunakan SIAM, keadaan berbalik menjadi buruk (lambat dan bahkan sampai keluar dengan sendirinya). *KRS tidak hanya berpengaruh bagi mahasiswa semester muda, tapi juga keseluruhan mahasiswa.

Output : ...

Dokumen

Dalam setahun belakangan ini, pengaksesan KRS diganti ke SIAM (sebelumnya menggunakan SINERGI). Saat menggunakan SINERGI, fitur serta kecepatan akses sangat handal dan nyaman. Tapi setelah diganti menggunakan SIAM, keadaan berbalik menjadi buruk (lambat dan bahkan sampai keluar dengan sendirinya). *KRS tidak hanya berpengaruh bagi mahasiswa semester muda, tapi juga keseluruhan mahasiswa.
--



Case-Folding

dalam setahun belakangan ini, pengaksesan krs diganti ke siam (sebelumnya menggunakan sinergi). saat menggunakan sinergi, fitur serta kecepatan akses sangat handal dan nyaman. tapi setelah diganti menggunakan siam, keadaan berbalik menjadi buruk (lambat dan bahkan sampai keluar dengan sendirinya). *krs tidak hanya berpengaruh bagi mahasiswa semester muda, tapi juga keseluruhan mahasiswa.



Tokenisasi

dalam setahun belakangan ini pengaksesan krs diganti ke siam sebelumnya menggunakan sinergi saat menggunakan sinergi fitur serta kecepatan akses sangat handal dan nyaman tapi setelah diganti menggunakan siam keadaan berbalik menjadi buruk lambat dan bahkan sampai keluar dengan sendirinya krs tidak hanya berpengaruh bagi mahasiswa semester muda tapi juga keseluruhan mahasiswa



Filtering

setahun belakangan pengaksesan krs diganti siam sinergi sinergi fitur kecepatan akses handal nyaman diganti siam keadaan berbalik buruk lambat sendirinya krs berpengaruh mahasiswa semester muda keseluruhan mahasiswa



Stemming

tahun belakang akses krs ganti siam sinergi sinergi fitur cepat akses handal nyaman ganti siam ada balik buruk lambat sendiri krs pengaruh mahasiswa semester muda luruh mahasiswa



Biword

[0] = tahun belakang	[11] = handal nyaman
[1] = belakang akses	[12] = nyaman ganti
[2] = akses krs	[13] = ganti siam
[3] = krs ganti	[14] = siam ada
[4] = ganti siam	[15] = ada balik
[5] = siam sinergi	[16] = balik buruk
[6] = sinergi sinergi	[17] = buruk lambat
[7] = sinergi fitur	[18] = lambat sendiri
[8] = fitur cepat	[19] = sendiri krs
[9] = cepat akses	[20] = krs pengaruh
[10] = akses handal	[21] = pengaruh mahasiswa

[22] = mahasiswa semester
[23] = semester muda
[24] = muda luruh
[25] = luruh mahasiswa

D. Daftar Pustaka

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

