Information Retrieval and Extraction, 2005

Term Project

Date:2005/06/27

網媒博一 張原豪 D93944006

Table of Contents

1. System Environment

Developing language: Visual C++ 6.0

Platform: windows XP with Pentium IV 3.0GHz, 512MB DDR RAM

2. Files in This Zipped File

Directory **RelevanceJudgements**: where the source code is stored

d93944006_IR_TermProject_Report.pdf: the document you are reading now.

3. Overview

As the Internet becomes more and more popular, we can acquire huge amount of information in a short time. Therefore, how to retrieve the relevant documents that we want becomes an important issue.

However, to create a information retrieval model with high precision-recall is not an easy task. A lot of relevance model has been proposed to create a high precision-recall system, but the results of most proposed models are not satistiable.

In this term project, I design an information retrieval (IR) model to create a high relevance judgement system with the material from FBIS3 and FBIS4 of TREC6.

The following tale is the summarization of the IR model that I designed.

Metrics / Doc Set	Full Index Time		Incremental Time		Query (Searching) Time	Average Precision	Precision at R(30%)	Precision at 10 docs
	Parsing Time	Sorting Time	Parsing Time	Sorting Time				
FBIS3	73s	36s			1s	0.30	0.48	0.64
FBIS3+FBIS4	149s	107s	77s	111s	2s	0.30	0.52	0.56

We can find that this model has very good performance at query since I use **some special indexing techniques to reduce searching time** and **internal buffer for each file to reduce the number of performing disk I/O**. ***This is a tradeoff between indexing time and searching time. That is the model uses more indexing time to create more comprehensive indexes in order to reduce searching time at queries.*** However, the average precision in this model is comparatively low. I configure some different weighting parameters for title, narrative, description, and text sections, but the average precision is still very low. No matter how I configure these parameters, **the best average precision is around 30%.** And most of time, the average precision is around 18% to 25%. Due to the time limitation, I don't analyze how the ***trec_eval program*** judge a document is relevant or not relevant about the query topics 301, 302, 304, 306, and 307. Instead I follow this project's instruction to create index files and then generate my ranked answer list followed by using ***trec_eval program*** to judge my IR model's 11 standard recall levels.

All the details are discussed and explained in the following sections.

4. System Architecture

Figure 1 shows the procedure of parsing documents and creating indexes for later queries, while Figure 2 depicts the query procedure in my IR model.

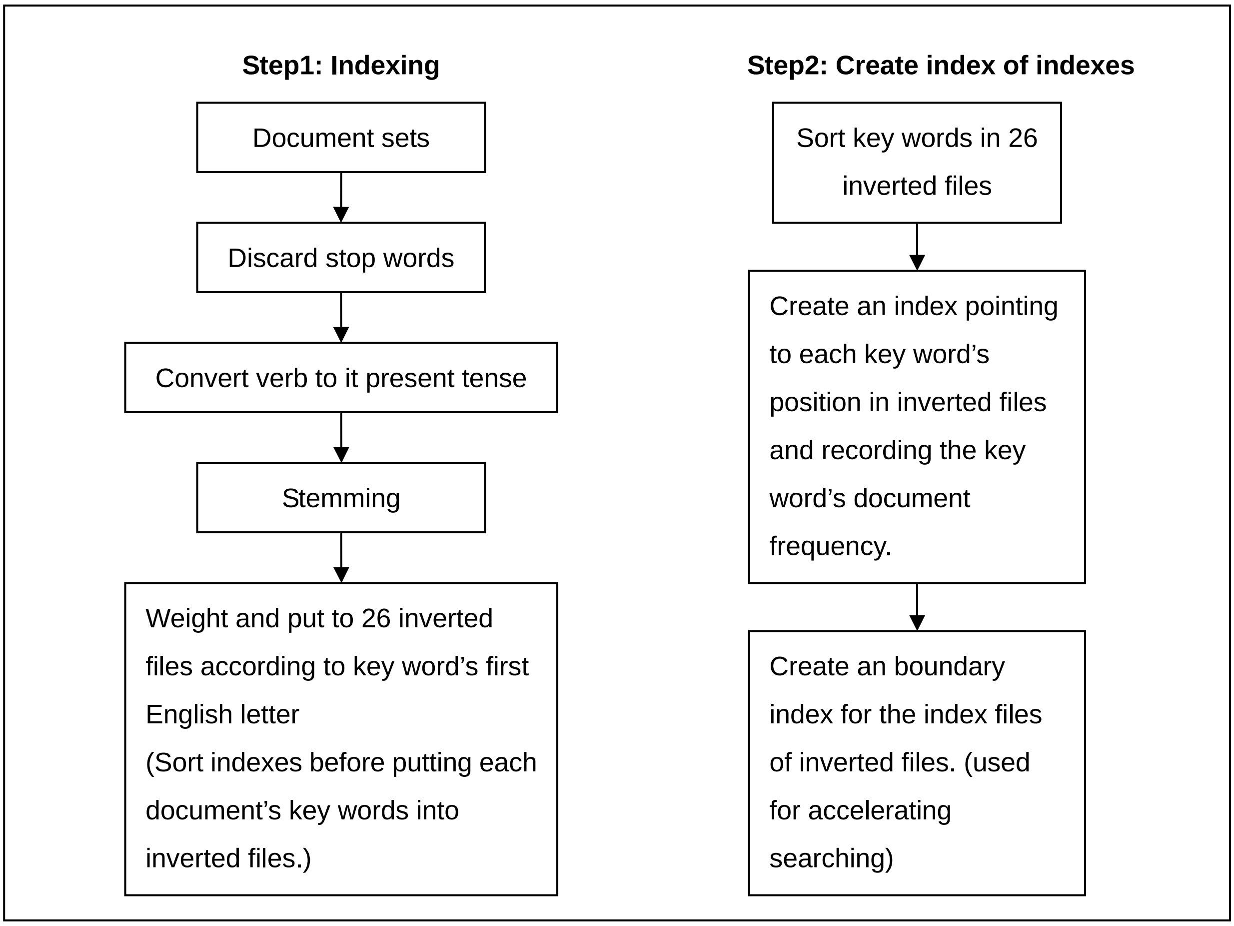


Figure 1: Indexing procedure

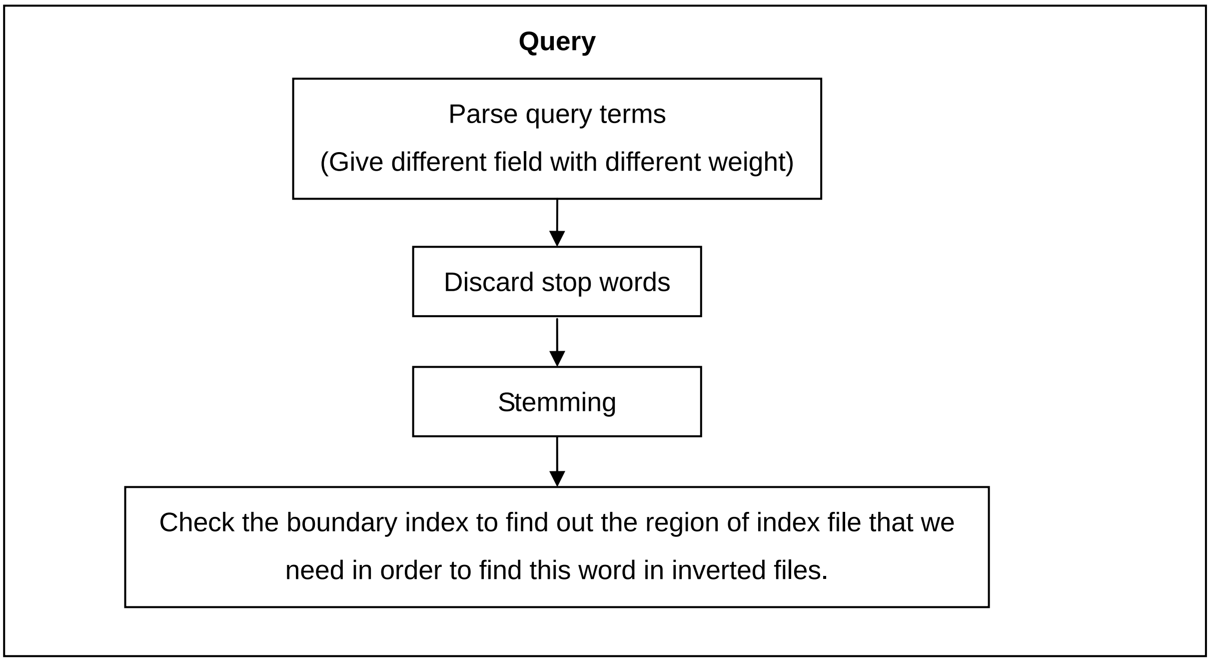


Figure 2: Query procedure

In order to accelerating searching time, we sort the inverted files in alphabetical order and create a two level index. For each query term, we check the **boundary index** according to this term's first English letter. The following examples shows that the query term is "dog", it's first English letter is **"d"** so the system checks the entry for "d" and find the range of index terms starting with "d" in **index of inverted files**. We use binary search to search the entry for "dog" in index of inverted files. When we find this entry, we can also retrieve this term's **offset** and **document** frequency in sorted inverted files. In this example, we can tell "dog" starts at **offset 2** and there are **3 documents having this word "dog"** in **inverted file for "d"**. Since the inverted files are sorted at indexing phase, we can just fetch the exact records from the corresponding inverted file starting from the exact offset position. In this example, we can tell the query term "dog" is relevant to FBIS3-2, FBIS3-9 and FBIS4-2 with <u>*tf-idf weight*</u> 6.12, 1.00 and 3.8 respectively. ***Through this indexing technique, we just use binary search for searching the query term in "index of inverted files," and don't need to search inverted files. This can save a lot of time on searching since this size of "index of inverted files" is much smaller than "inverted files." At the same time, we use binary file to store these index information in order to save the files size of indexing, and we also make each record in the same index file has the same size in order to accelerating read and write records in each file. Since each record's size is the same, we can easy to seek the starting address of this particular record to save the times of accessing disks or searches. The only drawback is that we 5 MB to store the" index of inverted files" to create indexes pointing out the exact term's position in inverted files.***

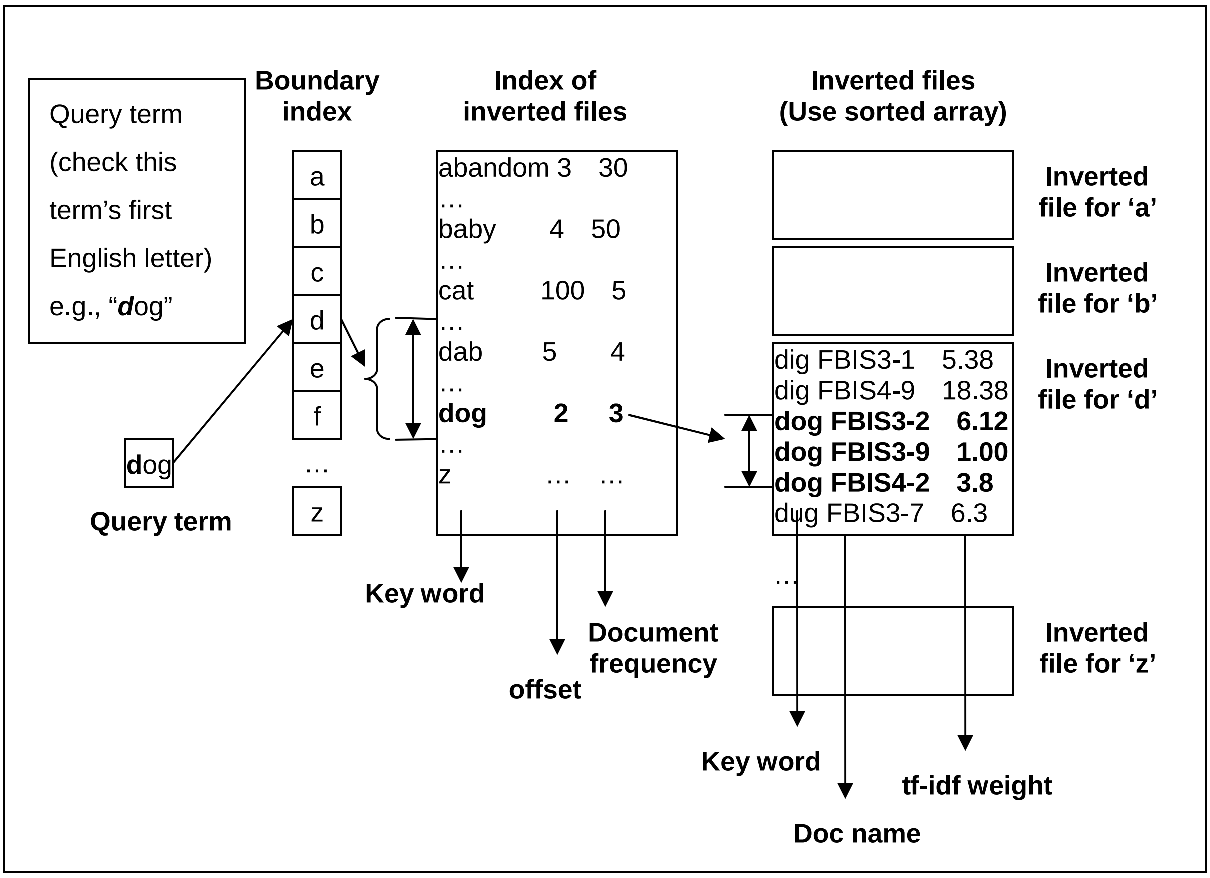


Figure 3: A query example

During indexing, sorting, or query, we always maintain an ***internal buffer*** for each file. With maintaining such internal buffers, we can save millions of time of accessing hard disk, which is extremely slow comparing to the access time of RAM.

During sorting phase, we use ***quick sort*** to save sorting time, and also use stemming to reduce the number of index terms and solve word's morphology. In total, the size of index files for FBIS3 and FBIS4 is around 500 MB.

4.1. Indexing

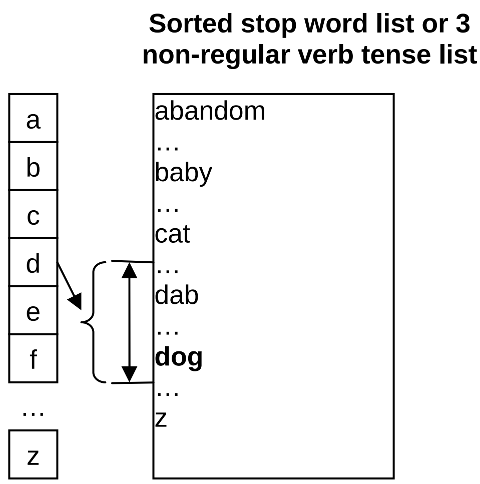
There are a lot of details that we have to take into consideration while creating indexes. First of all, we need to remove **stop words** since these words are meaningless but have high frequency of occurrence. Second, we have to consider **morphology** of English words, including ***converting verb's to its present tense*** and **stemming words**.

After considering stop words, verb tenses, and stemming issues, we can think about how to create indexes from training corpus, TREC6, especially that we want to create an indexing mechanism that can have a very efficient way for searching in order to make users acquire information with an reasonable response time.

The above issues will discuss in the following subsections.

4.1.1. Stop words, verb tenses

In fact, stop words and verb tenses have same problem. That is how to find out whether this term is a stop word, another tense of a verb, or different morphing of anther word. In order to accelerate matching a term from stop word list and verb tenses list, I sort each list and use each English word's first letter as index to classify them into 26 groups. Therefore, while matching a word, we can check this word's first English letter to find out the region and then use binary search to search this boundary. The following figure illustrates that we can use the first letter of dog "d" to find out the possible region of the word "dog" followed by performing binary search in this region to find out whether "dog" is in the list.



4.1.2. Hash table and Inverted Files

I take one document number as a parsing unit, such as FBIS3-1, FBIS3-2… etc. All the words retrieved from this document are temporarily stored in RAM. While retrieving a new word, we will search whether this word has been found in this document or not. If found, increase this word's weight by 1, or the weight of the field that this word appears. If this is the first time that this word appears in this document, create a new item to store it in RAM. After finishing scanning this document, classify all the accumulated weight words in RAM into 26 inverted files' output buffer by each word's first English letter. The data in output buffers of 26 inverted files will be flushed to file while the buffer is full or reaching end of parsing.

In order to find out whether this new retrieved word, called token, is already in temporal buffer in RAM, we create a hash table to increase the searching speed. We create a hash entry pointing the places where this word is stored. This hash table's hash function uses each word's first letter's 4-LSB and second's 4 LSBs as index to identify this word belonging to which hash entry class. The equation is listed in the following:

Word's hash entry = (word[0]&0x0f << 4) | (word[1] & 0x0f)

With this hash table, the average search time with this hash table is around 1/1000 of that of linear search.

After parsing all the documents, our 26 inverted files classified by the first English letter of words are also created. Each item in inverted files has three elements, which are word, document name, and word's weight (Illustrated in Figure 3). However, creating inverted files is not good enough, so we still need to add some features to enhance the performance of searching items from inverted files. That's why we need i***ndex of inverted files*** and ***boundary index***.

The size of inverted files for FBIS3 and FBIS4 is around 600MB.

4.1.3. Index of inverted files and boundary index

After creating inverted files, we need to sort these 26 inverted files in alphabetical order. However, personal computer's memory size is not big enough to store all inverted files in RAM. The size of inverted files for FBIS3 and FBIS4 is around 600MB. Therefore, it's impossible to load inverted files into RAM and then sort them. That's why I separated them into 26 inverted files according to each word's first English letter. In this way, the size of the biggest inverted file is around 35 MB which is acceptable. Therefore, I load one inverted file into RAM and adopt **Quick Sort** to sort this inverted file and then store it back to file in one go. ***After sorting, we scan the whole sorted inverted file to collect information about each word's offset and document frequency in the inverted file. We create one item in index of inverted files for each word in inverted files. In this way, once we find the item in index of inverted files, we can fix the offset and number of occurrence of this particular word in inverted files. Since we scan inverted files after inverted files have been sorted. Therefore, the index of inverted files is also sorted in alphabetical order. However, we still need to search the query word in the index of inverted files. In order to reduce the search time, I create a boundary index indicating the upper bound and lower bound of one enough English letter.***

The operation mechanism is introduced in section **3.System Architecture**.

4.1.4. Indexing training documents

In this project, there are three requirement of creating indexes. They are “index FBIS3”, “index FBIS3+FBIS4,” and “index FBIS3 + increment FBIS4.”

4.1.4.1. Index FBIS3

The indexing procedure was described in previous sections. Therefore, we just show the performance of indexing FBIS3 in Figure 4. We can tell the system spends 73 seconds in indexing creating inverted files while spending 36 seconds in sorting inverted files. Due to the special designed hash functions and indexing techniques, we spend less than 2 minutes to create a very good architecture of inverted files for queries.

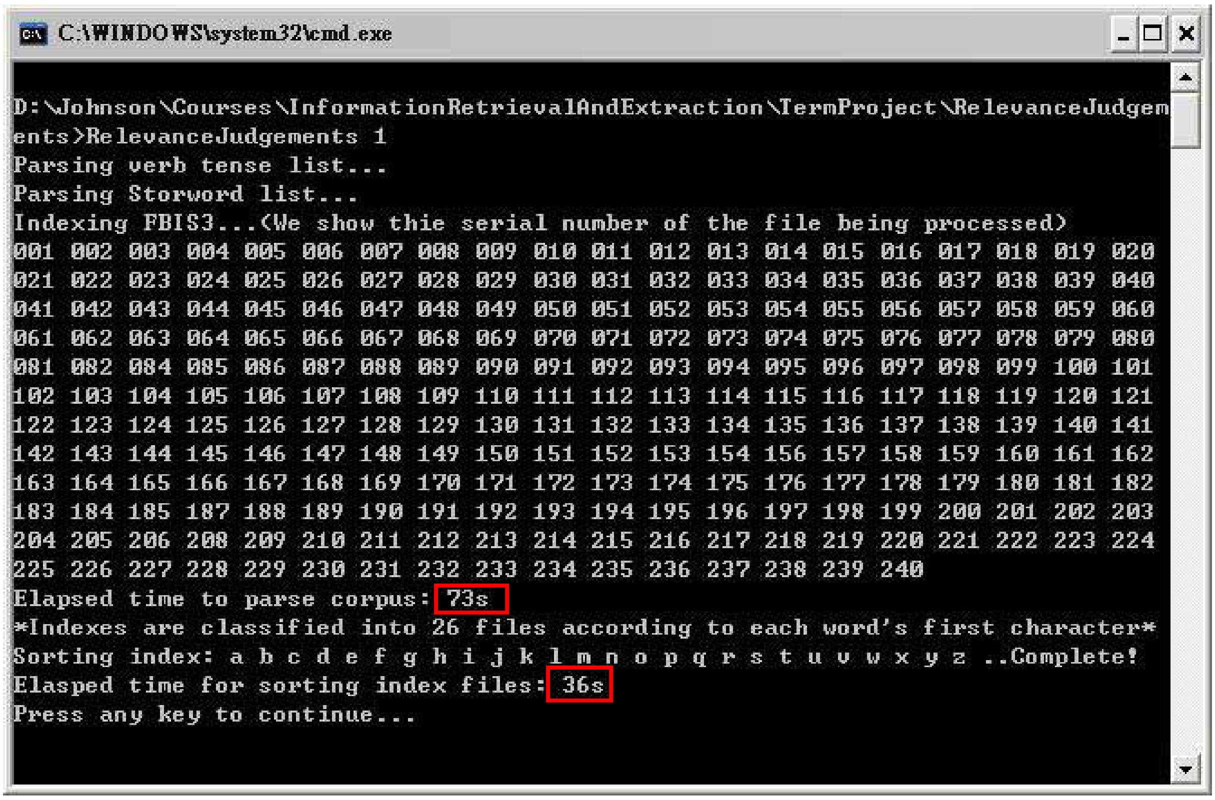


Figure 4: Indexing FBIS3

4.1.4.2. Index FBIS3+FBIS4

The elapsed time for creating inverted files for FBIS3 and FBIS4 is 149 seconds, but the elapsed time for sorting index files is 107 seconds, which is more than 2 times of that for sorting FBIS3 only. The reason is that quick sort is an **O(n*logn)** algorithm.

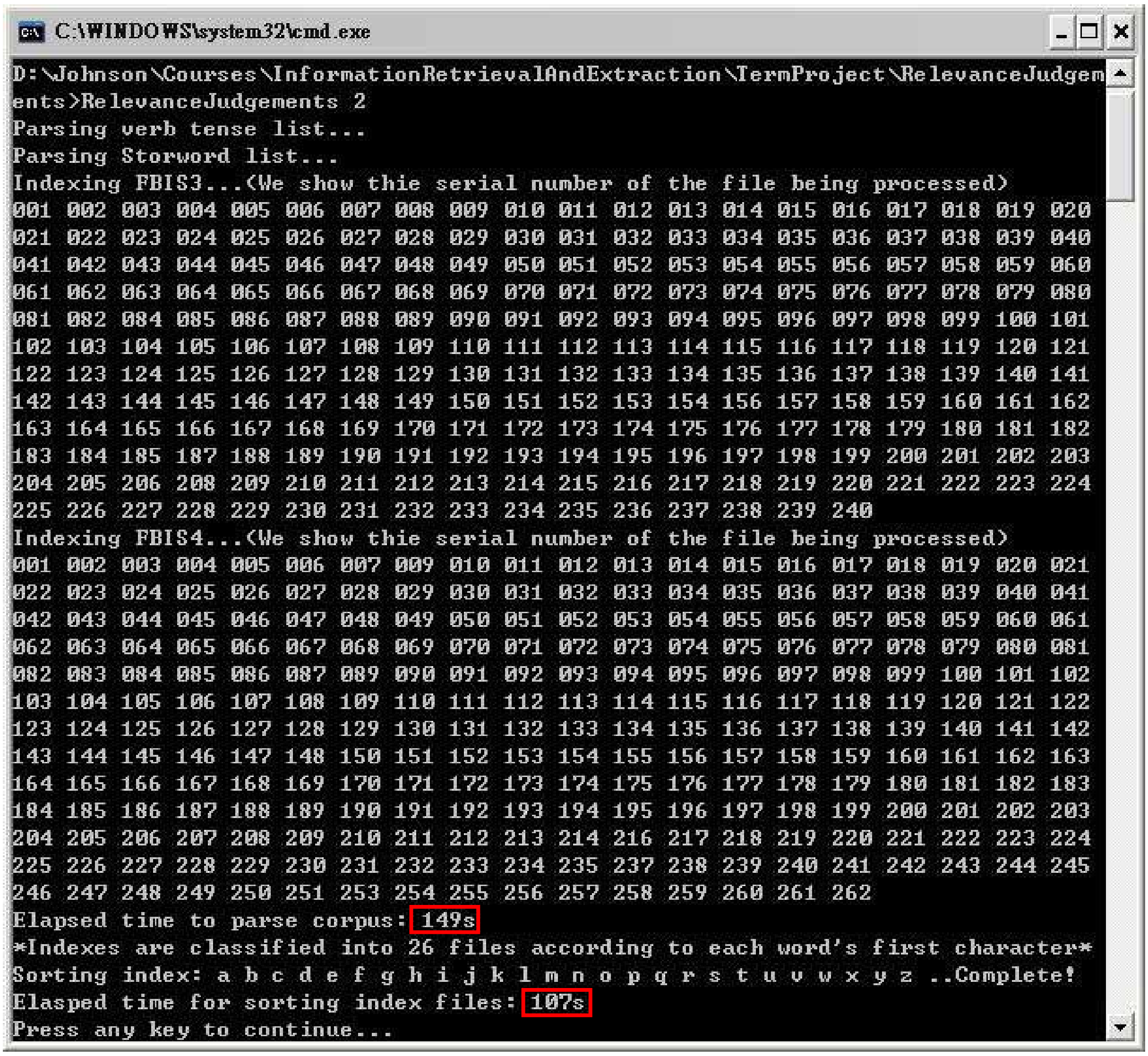


Figure 5: Indexing FBIS3+FBIS4

4.1.4.3. FBIS4 increment

In addition to create index in one go, we need to increment new index items while new document coming into our corpus. Therefore, performing incremental inverted files is necessary. In this project, the requirement is that we create inverted files for FBIS3 followed incremental FBIS4. ***My solution for incremental inverted files is to parse the new documents coming from FBIS4 and append to the existing inverted files. After finishing parsing FBIS4, we sort these incremented inverted files and also create new index of inverted files and index boundaries. Therefore, the content of index boundary, index of inverted files, and inverted files is the same of indexing FBIS3 and FBIS4 directly.*** Since we sort the whole inverted files including both FBIS3 and FBIS4, the sorting time is similar to sorting FBIS3+ FBIS4. That's the drawback of my solution. In other words, my solution spend some unnecessary on sorting. Therefore, there is still room for improvement. However, it's a workable solution.

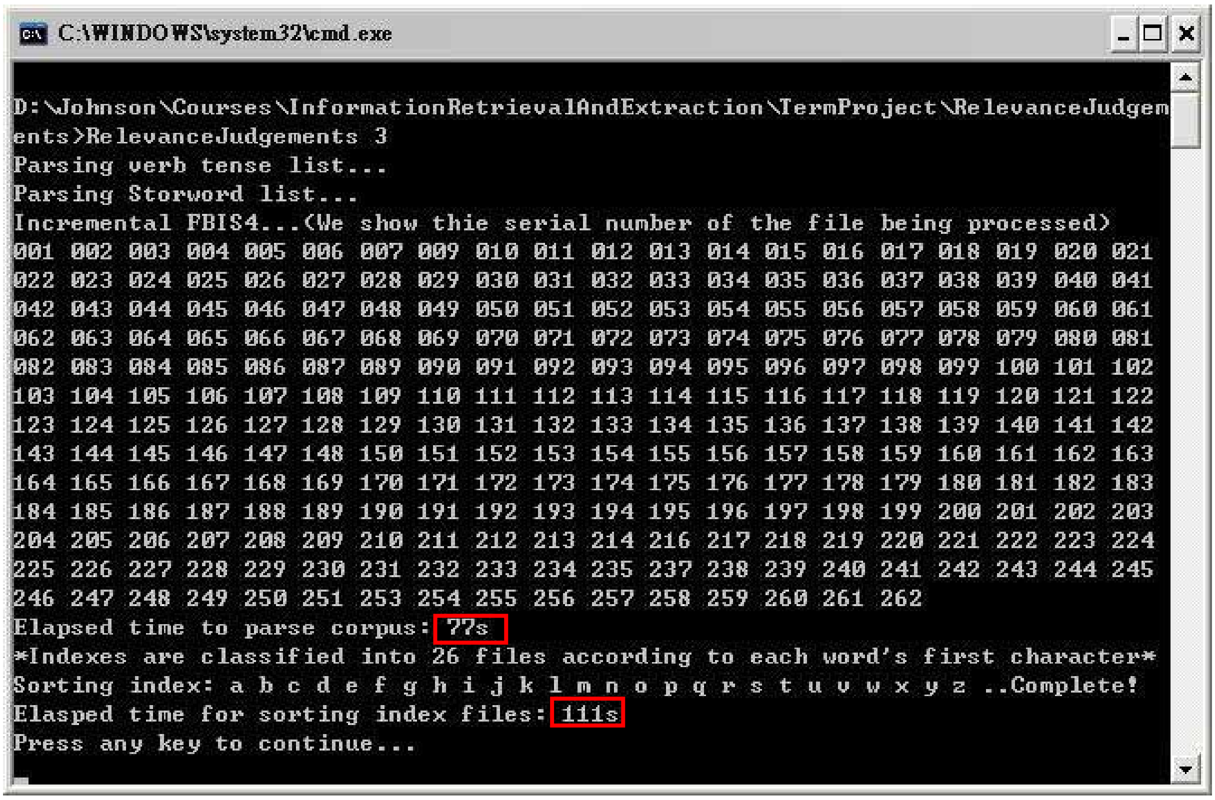


Figure 6: Incremental FBIS4

4.2. Query

In this project, the query topics are 301, 302, 304, 306, and 307. ***I report the top 2000 ranked document to be evaluated by trec_eval program.*** The query performance is shown in the following:

4.2.1. Query FBIS3

We can find that the query time is only 1 second. The performance is quite impressive.

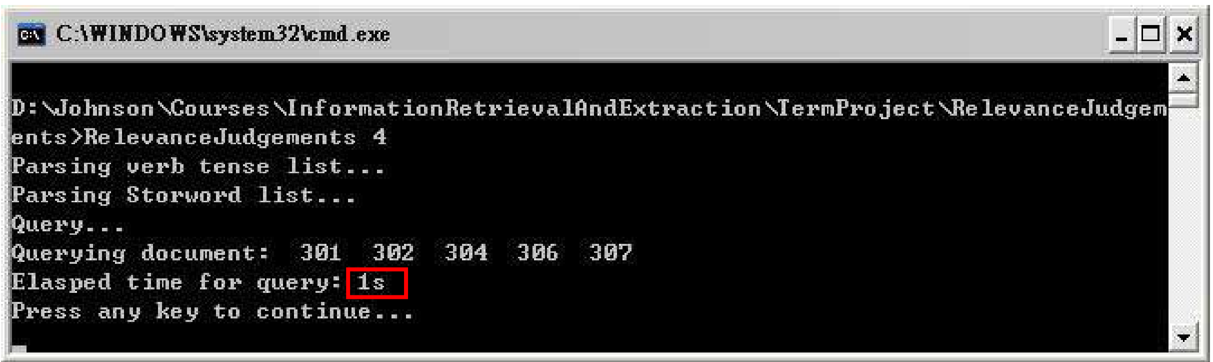


Figure 7: Query FBIS3

4.2.2. Query FBIS3+FBIS4

The query time for FBIS3+FBIS4 is 2 seconds. Again, the performance is extremely good. The reason is due to my special indexing architecture.

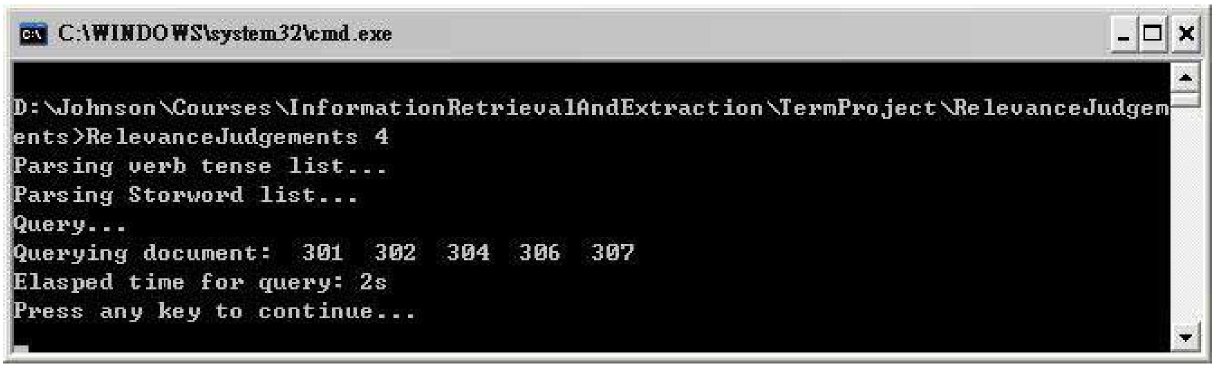


Figure 8: Query FBIS3+FBIS4

4.3. Evaluation

In this project, ***trec_eval program*** is the tool to evaluate our system's performance. The evaluation result is as follows:

4.3.1. Evaluate FBIS3

The following is the result evaluated by trec_eval program with FBIS3 as the training data. The average precision is only around 30%.

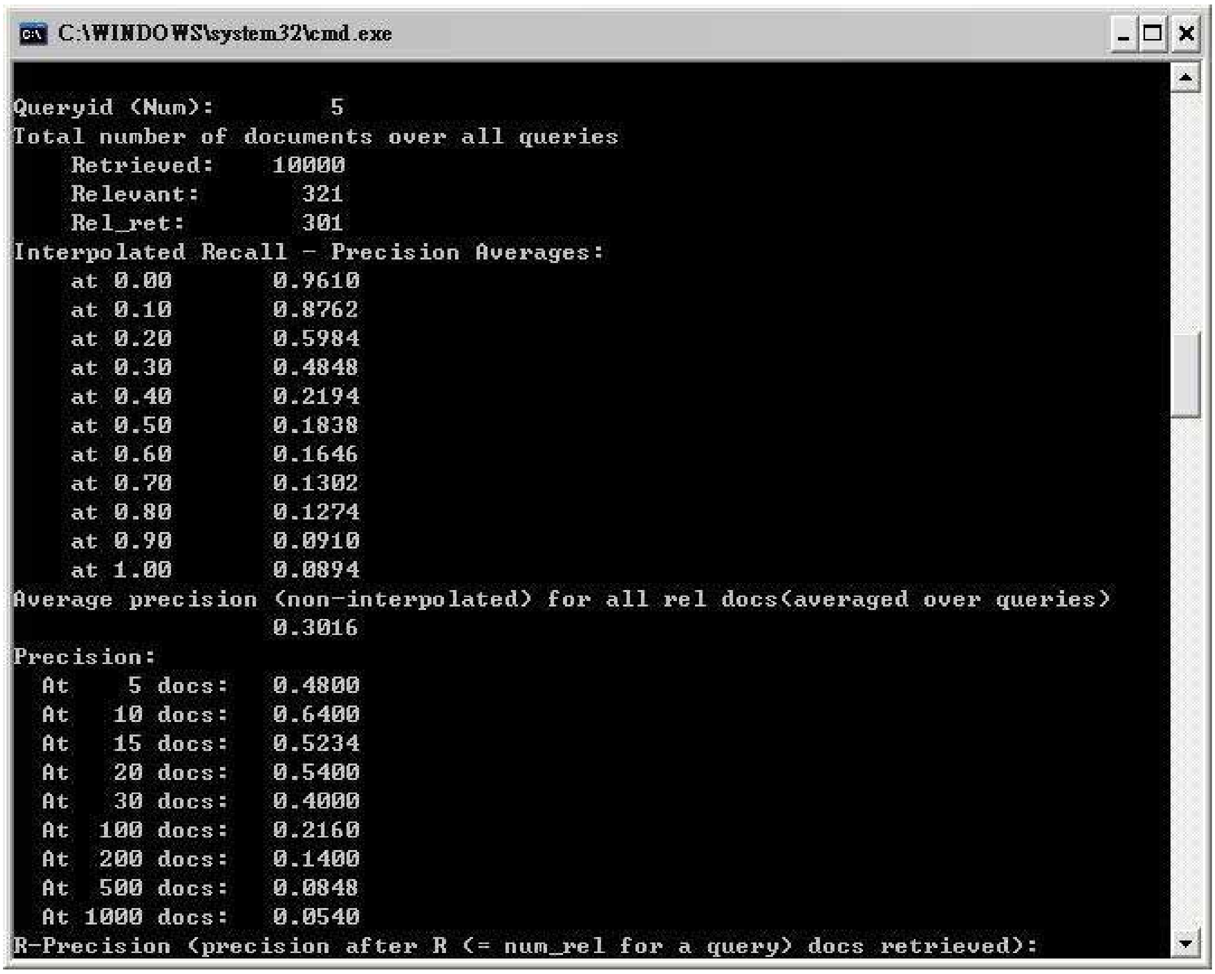
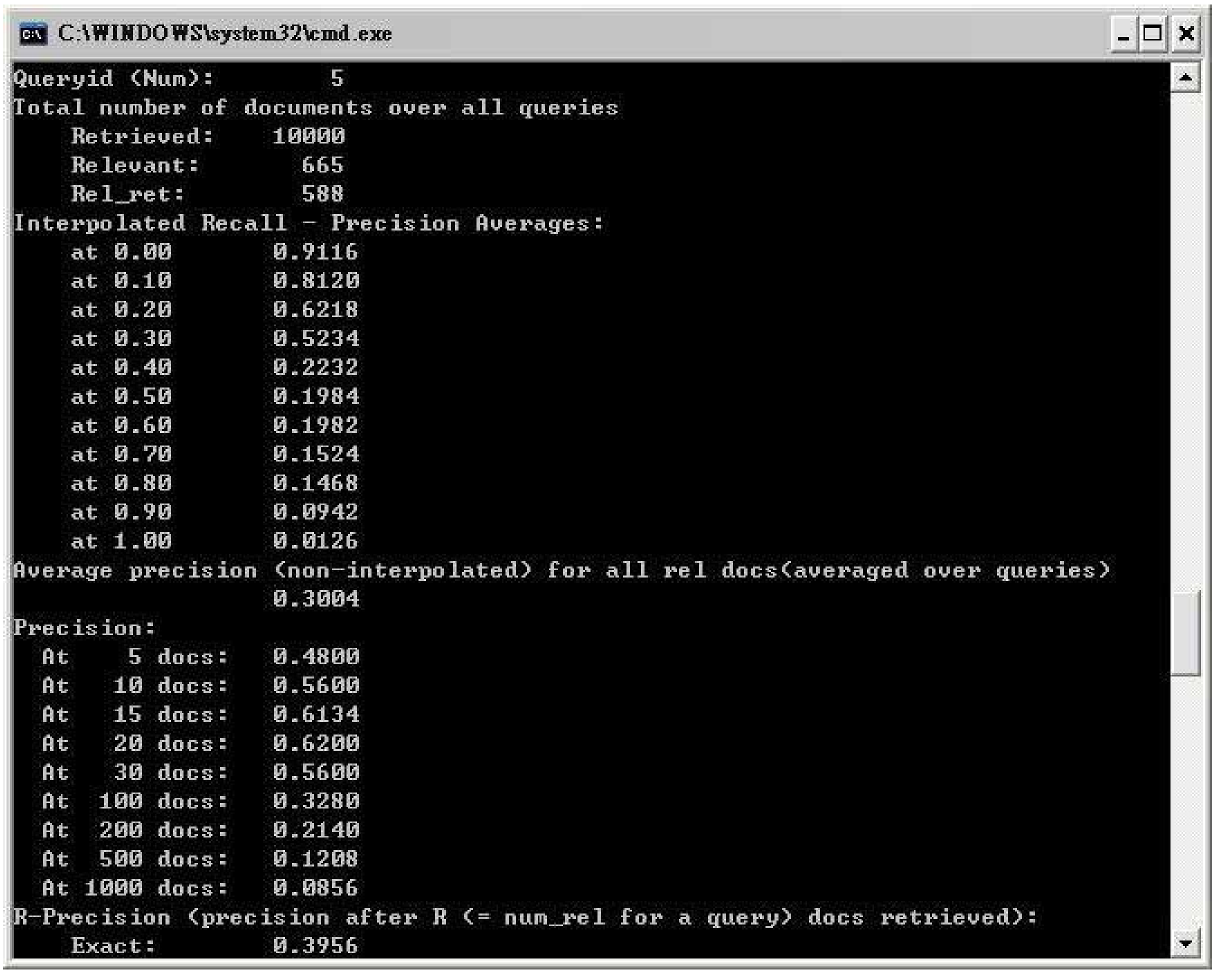


Figure 9: Evaluate FBIS3

4.3.2. Evaluate FBIS3+FBIS4

The following is the result evaluated by trec_eval program with FBIS3+FBIS4 as the training data. The average precision is only around 30%.



```
C:\WINDOWS\system32\cmd.exe
Queryid (Num):        5
Total number of documents over all queries
    Retrieved:    10000
    Relevant:       665
    Rel_ret:        588
Interpolated Recall - Precision Averages:
    at 0.00       0.9116
    at 0.10       0.8120
    at 0.20       0.6218
    at 0.30       0.5234
    at 0.40       0.2232
    at 0.50       0.1984
    at 0.60       0.1982
    at 0.70       0.1524
    at 0.80       0.1468
    at 0.90       0.0942
    at 1.00       0.0126
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.3004
Precision:
  At    5 docs:   0.4800
  At   10 docs:   0.5600
  At   15 docs:   0.6134
  At   20 docs:   0.6200
  At   30 docs:   0.5600
  At  100 docs:   0.3280
  At  200 docs:   0.2140
  At  500 docs:   0.1208
  At 1000 docs:   0.0856
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.3956
```

5. Notes for Users

If you want to run this system, remember put the FBIS3 and FBIS4 documents under ***directory FBIS***, which should be located in the same directory as ***directory RelevanceJudgements***.

6. Experiment Result

In this experiment, I trained my system with FBIS3 and BFIS4 document set from TREC6. After the system has been trained, the query topics 301, 302, 304, 306, and 307 are used to evaluate our system, followed by using **trec_eval program** to verify my system performance.

I test my system with two metrics. The first is 11 standard recall levels, and the relation between number of documents and precision. ***Figure 11*** depicts the system's 11 standard recall levels. We can tell that the system's precision is up to 90% while the recall level is very low. However, the system's precision decades seriously as the recall level increases.

The other metric is measuring system's precision related to the number of documents. We can tell that the system's precision decades dramatically while the retrieved number of documents increases. In addition, an interesting phenomenon is that the system's precision with retrieved 5 documents is higher than that with 10 documents.

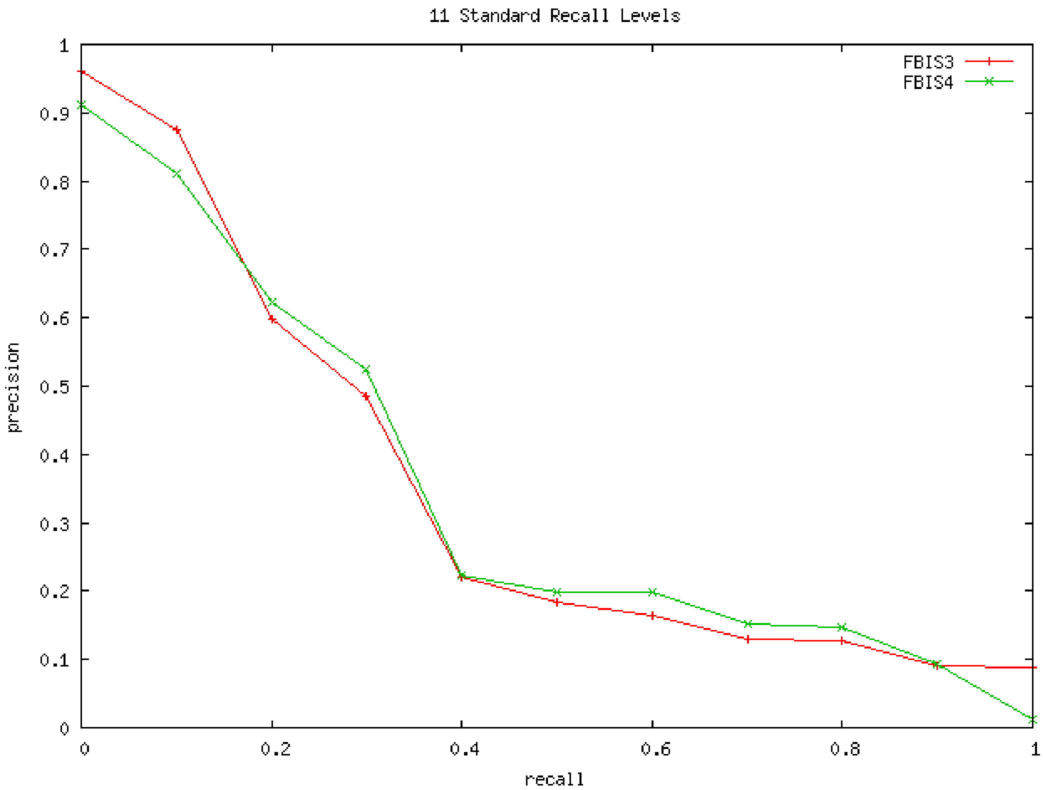


Figure 11: 11 standard recall levels

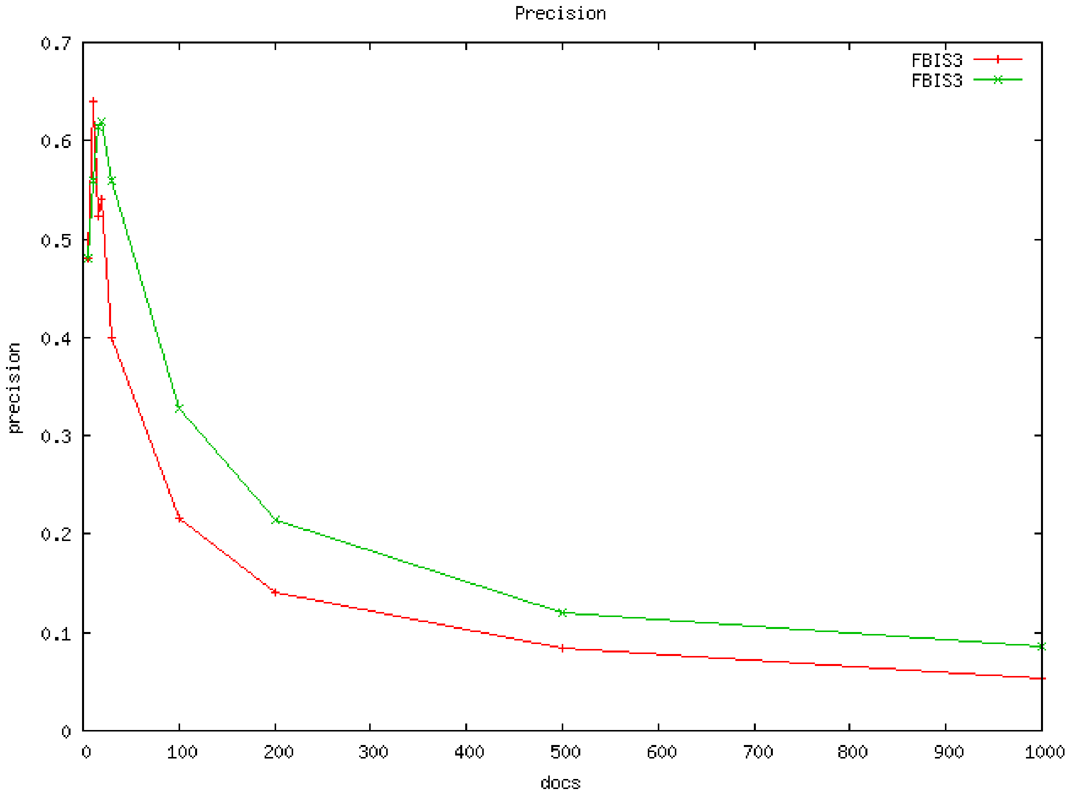


Figure 12: Precision

7. Conclusion

In conclusion, although this designed IR model doesn't perform good performance in terms of precision and precision-recall level. However, it demonstrates a good IR model to create indexes in an efficient way, and have extremely good performance at query, especially searching time.

In this project, I learn that using **tf-idf** is not an ideal way to design IR model, since a document with high query term frequencies doesn't mean that it's more relevant than other documents.

Another lesson I've learned through this project is that I learn how to improve system performance while handling with a huge amount of files including reading and writing data, searching a specific item, and how to create an efficient indexing architecture. A good searching and indexing model can save hundreds of times while comparing to a poor searching and indexing model.