Time Travel Search for Versioned Documents

By: Shahzaib Javed

Introduction

Versioned documents present a problem with what documents should be returned. It's not clear which of the versions should be returned and with what rank for each. Sometimes a user may want the most recent version, especially for current news related queries. Other time's the user may want the version that is most relevant to the query. Thus for versioned systems such as Wikipedia and Web Archives, the returned results may not always return results that the user was looking for.

In this paper, I propose a new interface for searching for versioned document. The interface allows users to search for the most current searches or most current to a given timestamp through special searches. Other than special commands, my interface also is a smart system that will learn from the users. The versions are ranked using a modified BM25 ranking function, which uses the learned information and ranks versions accordingly.

2. Background information

In this section I present some background information on inverted index, and Document At A Time Processing .

2.1 Inverted Index

An inverted index is used to store all the terms along with the term frequencies and document id's to which the terms map to. Since this structure might be very large and may not fit in main memory, it's usually stored on the disk and a lexicon is used to retrieve the inverted index for a particular term. The lexicon just holds the terms along with a pointer to the location within the file, so the inverted index can be looked up fairly quickly. An example of an inverted index along with the lexicon is shown in Figure 1.

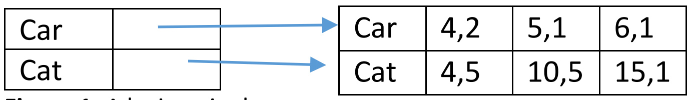


Figure 1: A lexicon is shown on the left side for terms and a file pointer is shown using blue pointers. The inverted index which is shown on the right holds all the documents (represented as Doc ids) that this term appears in along with the frequency.

In Figure 1, we represent the documents as document ids and have a separate table that maps the document ids back to documents. Note the lexicon and document table are small enough to be kept main memory.

2.2 Query Processing

Document At A Time (DAAT) is used for finding matching documents for a specific query. The documents are found using a conjunctive matching. DAAT is a simple algorithm and is efficient Figure 2 shows pseudocode for the algorithm.

```
For words in Query:
        list.add(openList(words));
did = -1;
// sort the list in increasing size
While did <= list[0].getMaxID:
        did = list[0].nextGEQ(did);
        int tmp = -1;
        for(i = 1; i<list.size; ++i):
            tmp = nextGEQ(did);
            if (tmp!=did)
               break;

        if (tmp == did):
           .. do ranking
        else:
          did=tmp;
```

Figure 2. Pseudocode for DAAT algorithm. Basically we first get the inverted list for each word in the query and store in the list. We then keep traversing through all the list and find all the did that occur in all the lists. We rank these documents using a ranking function. We do this until we reach the end of the first list.

First we loop through each of the words in the query and retrieve the inverted list using the function openList. We store the inverted list in our list variable. We then sort the lists in increasing order. We find all the document's that are common to all the documents and loop through until we have found a document id that is greater than the maximum document id of the shortest list. We use the nextGEQ function, which returns the first did that is greater than or equal to the did passed in for a particular inverted list. If nextGEQ returns the same did, we rank the document and push in a heap and find the next greatest did. If the nextGEQ returns a larger did, we change did and restart the search.

3. Versioned Search

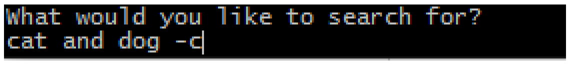
In this section I introduce my new search interface. First I introduce a modified inverted index, then I introduce how the user could do versioned search. Finally I end of the section with introducing a modified version for the ranking function.

3.1 Modified Version

My new system is just like any other system and the user can search for it any other query processor. The user can simply type in a query and top k results will be returned. I'll discuss in detail how these results are ranked later in the section. The user can also do special searches. These special searches consist of a current search or a time search. Before we discuss the versioned document search, I present the slightly modified inverted index that is used. The inverted index simply contains the document timestamp, along with the document id and frequency. The document time stamp is also stored in the document url table.

3.2 Current Search

Current search is a feature that allows the users to return the most recent documents. To activate current search, the user simply has to put a '-c' after the query and the versions with the most recent documents will be ranked higher. Figure 3 shows a current search.

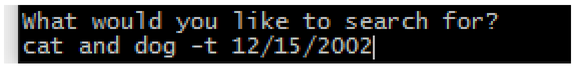


```
What would you like to search for?
cat and dog -c
```

Figure 3: Current Search. Notice the '-c' after the query. The system will recognize this is a current search and will return the most versions

3.3 Time Search

Time search is a feature that allows the users to return the most recent documents given, a particular date. To activate time search, the user simply has to put a '-t' after the query and give a date in the format: MM/DD/YYYY and the documents that have a timestamp most recent to this date will have a higher score thus will be ranked higher. Figure 4 shows a simple time search.



```
What would you like to search for?
cat and dog -t 12/15/2002
```

Figure 4: This is a time search. Notice the '-t 12/15/2002' after the query. The program will automatically detect this is a time search and get the versions most recent to the date 12/15/2002.

3.4 Modified Ranking function

In section I introduce how the documents are ranked with a normal search. Majority of the users will not want to enter the added extra options and still would expect the most relevant documents to be returned. To counteract this, I modified the BM25 ranking function such that it learns from the user's searches. The ranking function learns in two ways. The first way is, that we keep track of the number of times a document is clicked for a particular search query. This is used when ranking the versions for the top k results. This helps when the versions that are the most relevant to the user are not ranked high enough. With enough users clicking for a version, that version will be ranked highe. We keep another table that keeps track of all the queries searched and all the results a user clicked, along with the number of times clicked. The original BM25 function is shown in Figure 3.

$$BM25 = \sum \log\left(\frac{N - f_t + .5}{f_t + .5}\right) * \frac{(k_1 + 1) f_{d,t}}{(k_1 * \left((1 - b) + b * \frac{|d|}{|d|_{avg}}\right) + f_{d,t}}$$

Figure 3: This is the Original BM25 function that is popular in ranking documents.

N is the total number of documents, f_t is the number of documents that contain the term t. $f_{d,t}$ is the term frequency in document d, $|d|$ is the length of document d and $|d|_{avg}$ is the average length of a document. k_1 *and* b are constants and are set to 1.2 and .75, respectively.

Now I present the modified BM25 ranking function, which is shown in Figure 5.

$$BM25 = (\sum \log\left(\frac{N - f_t + .5}{f_t + .5}\right) * \frac{(k_1+1) f_{d,t}}{(k_1 * \left((1-b) + b * \frac{|d|}{|d|_{avg}}\right) + f_{d,t}}) + k_1 * \left((1 - b) + b * \frac{C}{C_T}\right)$$

Figure 5: This is the Modified Version of the BM25 ranking function. Note the extra term we add on to the original BM25 score.

Note we add an extra term to the original BM25 score. *C* is the number of times this link was clicked and C_T is the total click count for all the documents combined. The reasoning behind this is that if a lot of users clicked on particular version, that version must be a lot more relevant and should be benefitted, which is exactly what the second term added on does. However, there shouldn't be any sudden jumps in the scoring and this new ranking function takes care of that.

The second way the system learns is from current events search. Sometimes users will want to search for current events whether that be politics, news related or anything that may be trending. The system will learn which topics are trending and will return the most recent versions based on that those queries. Another table is used to store which searches has been searched for, along with the frequency and a timestamp with last time this query was searched for. A threshold time value which we will call t in this paper, a click count value which we will call c are also defined. Every time a user searches a query, the system will check if the last given search time for this query is within t, if so we will increment the counter and update the timestamp to the most recent time. If the last given search for this query is not within t, we will reset the counter and continue.

During the query processing stage, we will check if the query is in the current rank table. If not, we will insert it. If the query is in the current rank table and the count is greater than c, then this must be a current event search and the most recent versions should be returned.

4. Experiments

I conducted simple experiments using about 20 versions for a topic downloaded from Wikipedia. The interface was successful and very robust. With enough searches, the interface was able to detect the correct versions and rank them higher up in the results list. These tests were done on a small scale and to get the full scale of interface, a larger audience is needed.

5. Conclusion

In the paper I described a new system for searching for versioned documents. The new system had a slightly modified inverted index and had a timestamp associated with it. Next, I introduced the two special search options my system offered; Current search and Time search. Rather than entering extra commands to activate special search, my system will also adapt and learn from user's searches. The system will keep track of all the search queries and keep a tally for which documents were selected. The new modified ranking function that I introduced will take into account these counts and slightly boost the scores for those versions. Next my system will also learn when something is trending, whether that maybe politics, news, celebrities or anything that's trending at the moment and will return the most recent documents for those queries. Furthermore, time travel search engine is a way to enhance search for current versioned sets, such as Wikipedia, where it may not be trivial to what should be returned and in what order.

6. Future work

In this paper I presented a few ideas to improve search for versioned sets. Now I will discuss further idea’s that may also improve this system further but are not within the scope of this paper and project.

The system uses additional tables to keep track of what is trending and which documents were selected based on the query. These tables may get very large and thus may not be able to completely fit in main memory. Thus there should be an efficient way to quickly store and retrieve this information from disk, without slowing down the program. A simple approach maybe to imitate the structure of the lexicon and inverted index. The lexicon which will hold all the queries that were searched for and a file pointer to the location to where the document click list is for this query. Then, we can quickly retrieve list for any query, without much overhead.

We can also rank the versions using a modified version of HITS. The idea behind this is that if good pages link to a particular version over the rest, then that page must be much more valuable for the given query. Thus this particular version should be benefitted.