

Comparison of similarity search algorithms over inverted indexes

Project for MS&E239 Autumn 2010

Vanja Josifovski

September 27, 2010

1 Introduction

In this project you will examine the performance of two inverted indexing similarity search algorithms. The similarity search is between two textual documents and is therefore performed in a high dimensional space (approximately millions of dimensions). An example is content match, where the query is composed of the content of a web page and the objects that we are searching for are ads mapped into the same space. Here, each unique word represents a separate dimension. The similarity search function in this project will be a simple dot product. The task is to examine and compare two different algorithms: term-at-the-time (TAAT) and document-at-the-time (DAAT). Both algorithms implement a *top-k ranked retrieval*, i.e. given a query, retrieve the k closest ads sorted by their similarity score. In our case the *documents* are *ads* and we will use these two terms interchangeably in the following. Each document is also assigned a document ID (DID). Each document has multiple features with their associated weights. The query is also a set of features with associated weights. The similarity between document and the query is defined using a dot product:

$$sim(D, Q) = \sum_{fq \in q; fd \in d; fq.FID()=dq.FID()} fq.weight() * dq.weight()$$

where `weight()` is an accessor method to get a weight of a query or document. The sum is over the common features (i.e. intersection) of the document and the query. As the dataset weights are already normalized, the dot product in this case corresponds to the cosine of the angle between the query and the document in the space of their features.

2 Data

A dataset is provided where the document content is already broken into features (words, phrases, etc.) that are hashed into integer feature IDs (FIDs). The

document corpus is already *inverted* such that all occurrences of a feature across the corpus are grouped together into a single line. Each feature occurrence is also called *posting* and is conceptually a triplet $\langle DID, FID, weight \rangle$. Each line of the postings file **postingData.txt** represents one feature and all of its postings using an ASCII space separated format:

```
FID DID1 weigh1 DID2 weight2 ... 0 0
```

The query is also a set of features with associated weights: $Q = \langle FID, weight \rangle$. The query data file **queryData.txt** has one feature and weight per line. A query is terminated with a line containing "0 0":

```
FID1 weight1
FID2 weight2
...
0 0
```

Weights for both the documents and the query are integers in the range $[1 \dots 1000]$.

3 Resources

A good resource for the project is the presentation by Ronny Lempel et al [3]. The TAAT algorithm is described in [2]. The DAAT algorithm for similarity search was first reported in [1].

4 Index build

The first task in the project is to create the inverted index. As shown in Figure 1, the index has two parts.

- A dictionary hash table where given a feature you find a pointer to the beginning of the associated posting list. It is used at runtime to find the posting lists of the query features.
- A postings list file. This is a binary file where the posting lists are stored contiguously one one after another.

The dataset you get is already inverted: all the occurrences in of one feature are grouped together. The inverted index should be built by implementing the following steps for each feature:

1. Create the key array (discussed below) in a memory buffer.
2. Create the weight vector. All weights are fixed size, the vector is an array of all the weights.
3. Copy the key list and the weight vector into the file.

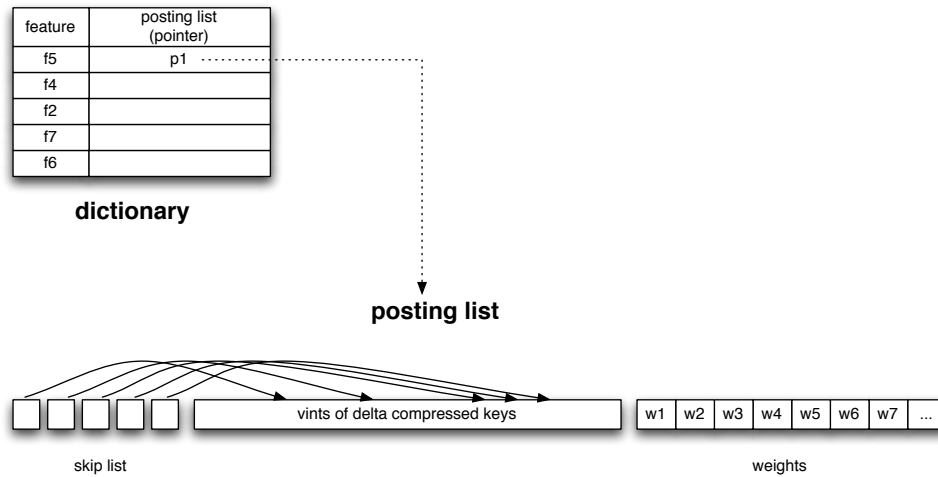


Figure 1: Inverted index structure

4. Insert into the dictionary hash the feature as a key and as a payload save the offset of the posting list and the maximum weight from the weight array.

After finishing all the features serialize the dictionary hash table to a second file. You need to implement methods to load the dictionary hash table and the posting file into memory.

This is the basic flow of the implementation, you are free however to modify the process if you feel you can improve/make it easier to obtain the target data structures. Note that there are different ways to store the key list, in increasing complexity. You are free to use any approach to start (e.g. using a simple fixed size entry array). We will do some fancier structures later.

The operations that you need supported on these data structures are written below in c++ style pseudo-code. In the implementation you may use any language you want, but c++ or Java would be a good choice.

```

typedef unsigned long FID;
typedef unsigned int DID;
typedef unsigned int Weight;
Class Dictionary {
    Dictionary(string fileName); //Constructor
    PostingList* getPostingList(FID f);
    int getMaxWeight(FID f);
    void WriteToFile(string fileName);
};

Class PostingList {

```

```

PostingList(int offset, FILE* fileName); //can use c++ streams for this
reset(); //position at the first doc in the posting list
DID next(); //move to the next doc
DID current(); //return the current DID
DID goTo(DID did); //move to or after a given DID
Weight getWeight();
};

```

Note that the correspondence between the key array and the weight array should be positional. That is, the weight for key at position X is found in the weight array at position X. So you can just simply index into the weight array. Once you build these data structures and the interface, you have your first inverted index!

4.1 Compressed key array

The total size of the index impacts the performance in several ways. First if the index fits in physical memory we can avoid accessing disk. This generalizes to the higher levels of caching as the L1 and L2 processor caches. One way to reduce the size of the index and improve cache performance is to perform compression. The major portion of the data is usually in the posting lists, so these are the prime target in the index compression.

In this project we will try to test the impact of compression on the algorithm performance and index size. We will change the basic posting list into a compressed posting list. This will impact the cost of the *next()* and *goto()* operations: we would expect less cache misses at a price of added decompression cost. Figure 1 shows the basic structure of the compressed posting list.

While there are number of different compression techniques, we will examine only one with the following choices:

- **Delta compression:** Instead of storing the actual key value in the posting list, we will store the difference (delta) from the previous key. As the posting lists are sorted in ascending order, this delta is always positive. Furthermore, for dense posting lists, the deltas are likely to be small.
- **Variable size integers:** As now we expect to have smaller key values, we should be able to encode small numbers more efficiently. For this we will use a simple variable integer encoding. Some languages provide Vint data types and it is fine to use them as long as you can fit those in the posting list. If not you can implement the following strategy. An integer (key) is represented by a sequence of bytes. In each byte, the 7 lower bits will be used to represent the number, while the top bit (bit 8) will indicate if there are more bytes that are part of this number. Decoding can be simple done by masking bit 8 adding the value to a register and shifting the register by 7 places left before the next addition. Once we encounter a byte with the 8th bit 0, the process stops.

- **Skip lists:** In order to obtain the value of the key at position k in the scheme above we have to decompress the posting list from the start. To avoid this, we introduce a skip list: a list of pointers into the posting list with the associated values. There is one pointer for each m entries in the posting list. For example if $m = 200$ and the posting list has 773 entries, the skip list will have 3 entries, pointing at the 201th, 401th, and 601th entry and the key value of the entry preceding the one that is being pointed at.

As we need to know the position of the key in the posting list (its index) to retrieve the payload, we should organize the skip list so that we preserve the positional information. For this, we can have a skip list entry for every n keys in the posting list (e.g. 200). The skip list entry contains now the value of the key and the first byte of the next entry in the posting list. To do goto, we do two level search: 1) find the last skip list entry that is smaller than the given DID; and 2) decompress the posting list (using the value in the skip list as a starting point for the additions) until we find the DID or find a posting with a larger key. Note that if the entry that we are looking for is in the posting list, we don't need to do any decompression. Also note that during the decompression we need to keep updating the index so we can retrieve the payload if needed.

Please try varying the parameter m of the skip list and see the impact on the performance and index size.

5 Term-at-the-time algorithm

The TAAT algorithm merges one term posting list at the time. We keep an intermediate result of already merged lists, pick the next one and merge it with the intermediate result. When merging a posting list with the intermediate result, if the DID is already present, we add to the score from the new feature. (note that with the dot product, the score contribution of the features are additive). Otherwise we add a new entry to the intermediate result with a score that is the product of the feature and the document weight. During the merging we will save only the best k results. This can be implemented using a heap to be able to delete the ad with the minimum score and insert new ads instead of it. In fact, we can keep $k + 1$ ads to be able to do early termination (explained next).

Using the max weight stored in the dictionary we can estimate the maximum contributions of the posting lists that we have not processed so far. So after merging each of the lists, we can see if the upper bound of the contribution of the remaining lists can bring the $(k + 1)$ th ad score over the k th score. An upper bound would assume that all of the remaining posting lists will have a posting for the $(k + 1)$ th ad while the k th ad will get none. Here we check if the remaining posting lists have a chance of changing the top- k set of results. If not, the processing can stop. This is called *early termination*.

The order in which the lists are processed is important for both the processing time and the size of the intermediate result. From the processing time perspective, it is best to start with the posting lists with the biggest potential impact and hope that we can do early termination. Furthermore, not all the intermediate results have a chance to make it in the top-k set. Using the upper bound estimate we can prune the intermediate result after each merge (or during the merge to save a pass over the list).

Here is a version of the algorithm in pseudo code (assume the array indexes start at 0):

```

DID array taat(PostingList array) {
  resultHeap == heap of the best top k results
  input == temp array
  output == temp array

  sort postingLists by maxWeight;
  copy postingList[0] into input;
  populate resultHeap with the top k+1 results

  //Calculate the initial upper bound: sum over posting lists 1..n
  upperBound = 0;
  for(i=1;i<postingList.size();i++){
    upperBound += postingList[i].getMaxWeight();
  }

  for(mergeindex = 1;mergeIndex < postinList.size(); mergeIndex++){

    if(the score difference between the min element and the
      next one is larger than upperBound){
      break;
    }

    merge input and postingList[mergeIndex] into
      output, while swapping elements in the heap if they have lower score;
    swap input and output;
    upperBound -= postingList[mergeIndex].getMaxWeight();

  }
  return the results in the heap;

```

Here is an example where the postings in the lists are in the format *DID; weight*. The max weight is given after the posting list label in parenthesis. The query is “A B C D” and the query weights are all 1. So in practice we are summing up the weights from the ads. We will be looking for the top 2 results in the example

Start:

A(9)	B(7)	C(4)	D(1)
1;3	1;5	3;4	1;1
4;5	2;1	7;3	7;1
7;3	4;7		9;1
10;2			
13;4			

Second iteration:

A+B
1;8
2;1
4;12
7;3
10;2
13;4

After the second iteration (the first iteration simply copies A into the input, the second one calculates $A + B$), the upper bound of what can still be added is $4+1=5$. The top two documents so far are 1 and 4 with scores 8 and 12 respectively. The next document by score is 13 with score 4. If 13 would have both C and D, it could have a score of 9, which would replace 1 in the top-2. Thus we have to make another iteration and merge the next posting list (C). We also note that documents 2, 7 and 10 cannot make the cut even if they get the maximum remaining score. Thus we can merge those out. Furthermore we can also notice that new documents (that are not already in the intermediate result) would also not make the cut and we can stop adding those as well. After pruning the list we get:

A+B
1;8
4;12
13;4

And after merging with C the result stay the same. At this point the upper bound of the remaining score is 1. Thus we can conclude that 13 cannot make the cut and we do not need to process the posting list D. Note also that when pruning the intermediate result, an alternative stopping condition is when we do not consider new entries and the intermediate result has a size of k .

6 Document-at-the-time algorithm

The DAAT algorithm considers one document at the time. We simultaneously traverse more than one posting list. At each point we evaluate one document versus the current candidate set. If the document has higher score we substitute the candidate with the minimum score with it. A simple version of the DAAT

algorithm can simply merge the posting lists of the query terms and examine all the documents in the union. A merge can be done as in the TAAT algorithm by moving in the posting list with the smallest DID, except here we consider multiple lists. The WAND algorithm proposed in [1] proposes a more efficient way where, based on upper bound estimates of the document score, we can skip over the posting lists and avoid evaluation of all the documents. For the this project you need to read the paper and implement the algorithm.

7 Correctness

Similarity search algorithms are difficult to test as the output is not a single number. Please implement one-document-at-a-time scoring that will score all of the documents (not indexed) and give back a result. A document is composed of all the postings in the dataset that has the same DID. This is a slow process so for starters you can run it for only a subset of queries. Compare the results of your algorithms to the result of this brute force scoring.

8 Experiments

1. Basic comparison of the performance of TAAT and DAAT
2. TAAT: In the case of early termination in TAAT, we know that the top-k set cannot change. Can the remaining lists change the order of the results? Provide an example if so. Compare the scores and the order with the version without early termination.
3. TAAT: what is the impact of early termination on performance? In how many cases this happens?
4. TAAT: what is the impact of pruning on performance? What about on memory savings?
5. DAAT: What is the average skip in the posting lists? What percentage of postings do we examine?
6. DAAT and TAAT: Suppose we change the max weights to be the second or third or 10th greatest. How much does the result set change? Describe why some of the best results are missed.
7. Provide an analysis of when would one algorithm be better than the other.
8. TBD: impact of compression on performance.

References

- [1] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM '03: Proc. of the twelfth intl. conf. on Information and knowledge management*, pages 426–434, New York, NY, 2003. ACM.
- [2] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, pages 97–110, 1985.
- [3] R. Lempel. Term-at-a-time and document-at-a-time evaluation. In *Course on Introduction to Search Engine Technology*, Technion, Israel, <http://webcourse.cs.technion.ac.il/236620/Winter2006-2007/ho/WCFfiles/lec5-evaluation.pdf>, 2009.