Faster and smaller Inverted Indices

Introduction

Web search engines face, as we saw in class, two great challenges:

1. The management of a huge amount of data that all the known websites represent
2. The provision of precise results in response to user queries, identifying a few relevant websites (documents) among increasingly large collections.

These are addressed via a two-stage ranking process. The first one is a simple filter that extract a few hundreds of relevant documents. The second one ranks them to obtain only the best results.

In most solutions to the presented challenges, ranked intersections are performed as Boolean Intersection (term is present or not), after which ranks are computed, while ranked unions are performed as approximative unions, using combinations of the scores of each term t of the query Q.

This article introduces us with a new representation of the inverted index. One that performs faster ranked union[1] and ranked intersections[2]. To do so, the two writers of this paper use **treaps**.

Refreshers

Treaps are a combination between two well-known data structures, that anyone practicing for coding interviews has seen before: **min-heaps** and **binary search trees.**

Binary Search Tree

A binary search tree is an ordered binary tree that allows fast look-up, addition and removal of elements, in $O(\log(numberOfElements))$.

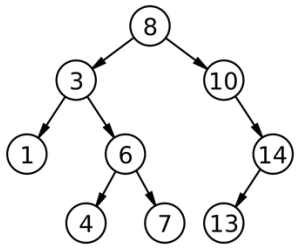

Figure 1. Example of binary tree

We can see that binary search trees are sorted **from left to right**, with, each node being bigger than his left child, and smaller than his right.

[1] Returning a set of the highest ranked documents containing some of the query terms
[2] Returning a set of the highest ranked documents containing all of the query terms

Min-Heap

A heap data structure is another form of binary tree. They are ordered **from top to bottom**, for the min-heap one.

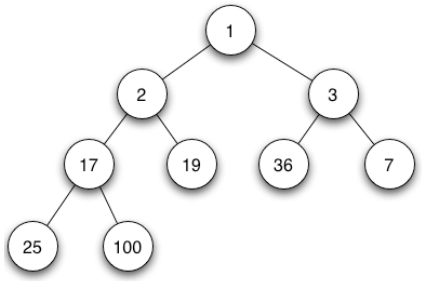

Figure 2. Example of a binary min heap

Treap

A treap is a combination of the two data structures detailed above. Each node stores two values, a key and a priority. It is a binary search tree for the first value, and a min-heap for the second. This means that it is sorted from left to right for the key, and from top to bottom for the priority.

Traditional Inverted Index

This article suggests a new method to **store the inverted index** of a collection of documents. Traditionally, an inverted index is an array of lists, where each entry is a term of the vocabulary, to which is associated a list of document identifiers *docid* and the weight of the term in each document.

When given a query Q, seen as a set of *q* terms, the score of each document *d* is computed like $score(Q, d) = \sum_{t \in Q} w(t, d)$ where $w(t, d)$ is the weight of term *t* in document *d*.

In the bag-of-words approach, we are given Q and an integer *k*, and asked to retrieve *k* documents *d* with the highest *scores (Q, d)*. It is not necessary that all terms are present, as long as the score is the highest possible. This is called ranked union. These cannot be solved through Boolean unions, as they would yield too many results, and not have decent time boundaries.

The lists associated to each term, or postings, were traditionally stored on disk, but with the availability of large amounts of main memory RAM, they are now increasingly stored on the RAM of a cluster of machines. In this case, documents are distributed across independent inverted indexes, each of which contributing to only a few of the *k* results.

Reducing the size of the inverted index representation is very important. On disk, access time is reduced. On RAM, cost is reduced.

Proposition

This article proposes a new data structure for the inverted index. It considers the posting list of each term as a sequence of sorted docids (keys) with their frequencies (priorities).
Given a positing list of n documents, the treap will contain n nodes. It is represented using the following isomorphism:

A fake root v_r is created. Its children become the nodes in the rightmost path of the treap, and each node is converted recursively. The treap root is the first child of v_r. The left child of any node v is its first child in the general tree. Therefore, an in-order traversal of the treap is a post order traversal of the general tree.

The representation is based on a balanced parentheses representation of the tree. Reaching a node is opening a parenthesis, leaving it is closing the parenthesis.
All of this is illustrated below.

Furthermore, to reduce the size of the storing of the treap, both keys and priorities are stored differentially. For each node v, let $id(v)$ be its docid and $f(v)$ its frequency (for the given term). Let's note u its parent node. If v is the left child of u, then we store $id(u) - id(v)$ instead of $id(v)$. Let v' be the right child of u. We store $id(v) - id(v')$ instead of $id(v')$.
We also store $f(u) - f(v)$ (or $f(v) - f(v')$).
These numbers go smaller as we go down the treap.

Inside the nodes are the docids, outside are the frequencies. Here is the former tree, before the isomorphism is applied:

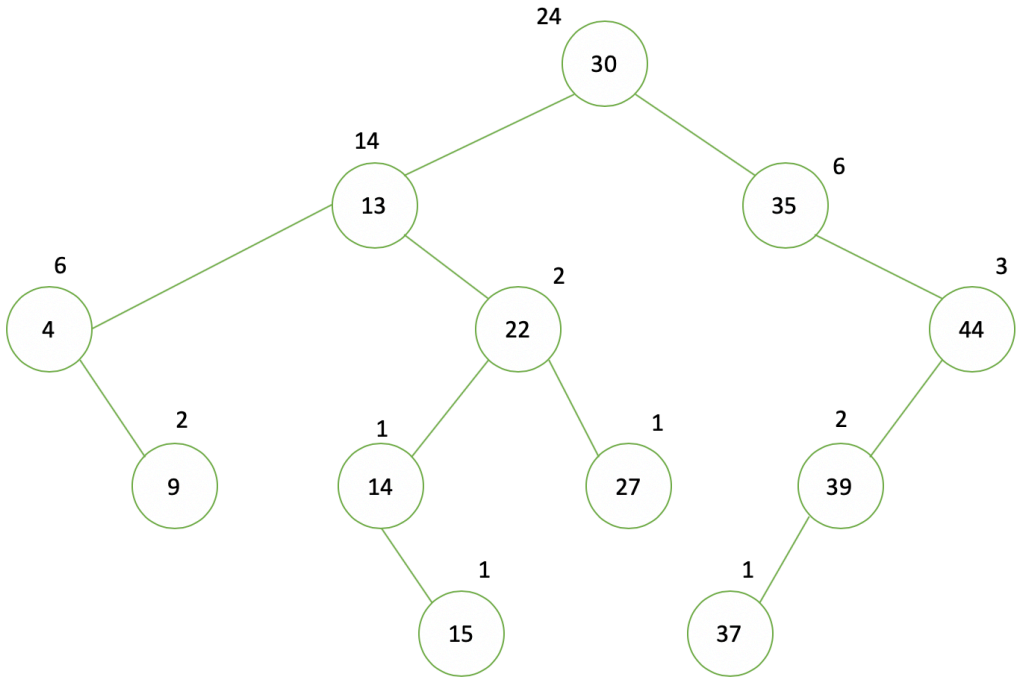

Figure 3. Treap with docids inside the nodes, and term frequencies outside

Remember that this treap is the posting list for a certain term, the key of the dictionary that is the inverted index.

We apply the isomorphism to get this treap:

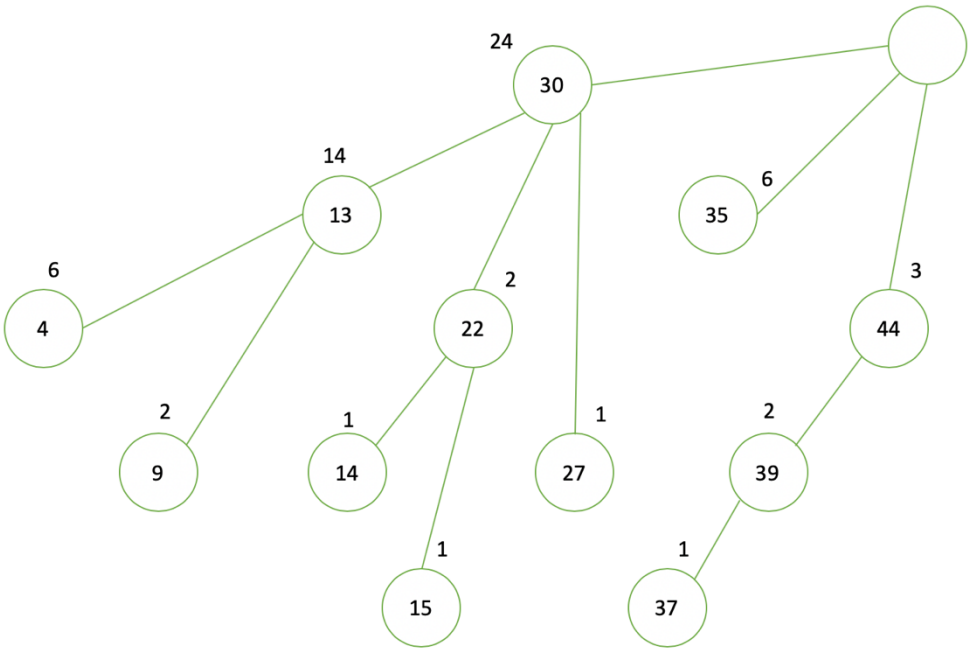

Figure 4. Isomorphismed treap

From left to right, this is the parentheses representation of this treap:
((( ) ( )) ( ) ( )) ( )) ( ) ((( ))))
Then we compress the docids and frequencies as described earlier:

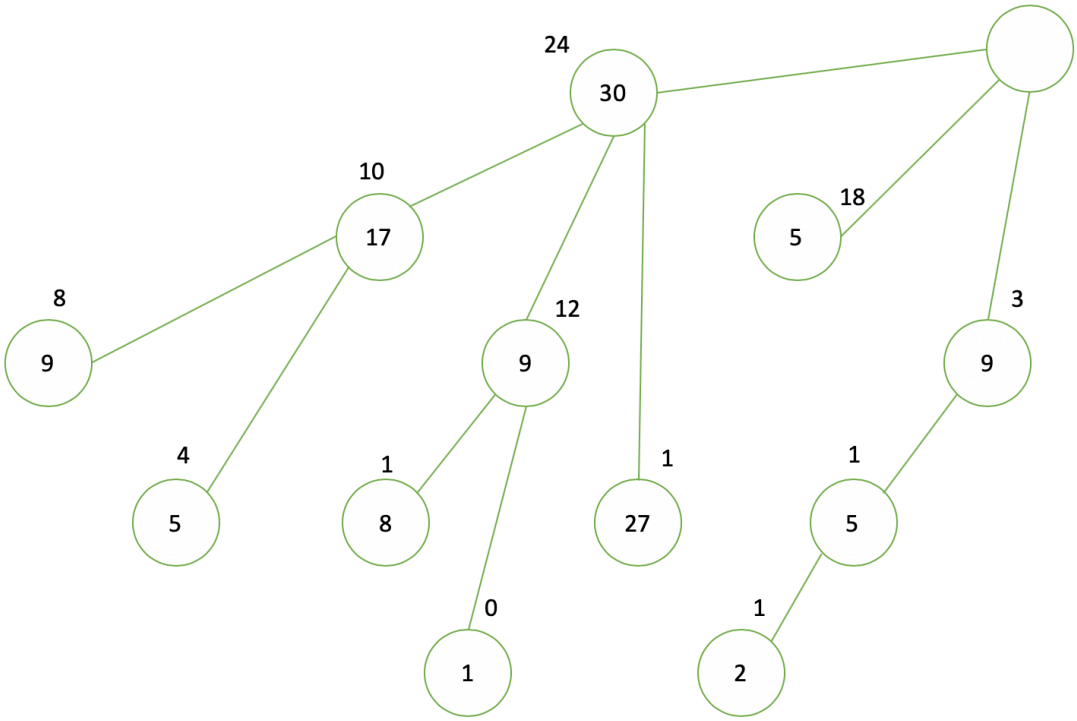

Figure 5. Final Treap

This schema can be once again improved. According to Zipf's law, many terms have very low frequencies, even below a certain frequency f_0. Whenever they are, we can cut these words, and the one below on the treap, from the treap, and store them in a separate posting list. They can here be differentially encoded in classical sequential form, the most efficient way to do so.
Here is what it looks like with $f_0 = 2$. We would make the treap with the four nodes left, and link to the correct nodes the beginning of the posting lists corresponding to the red parts.

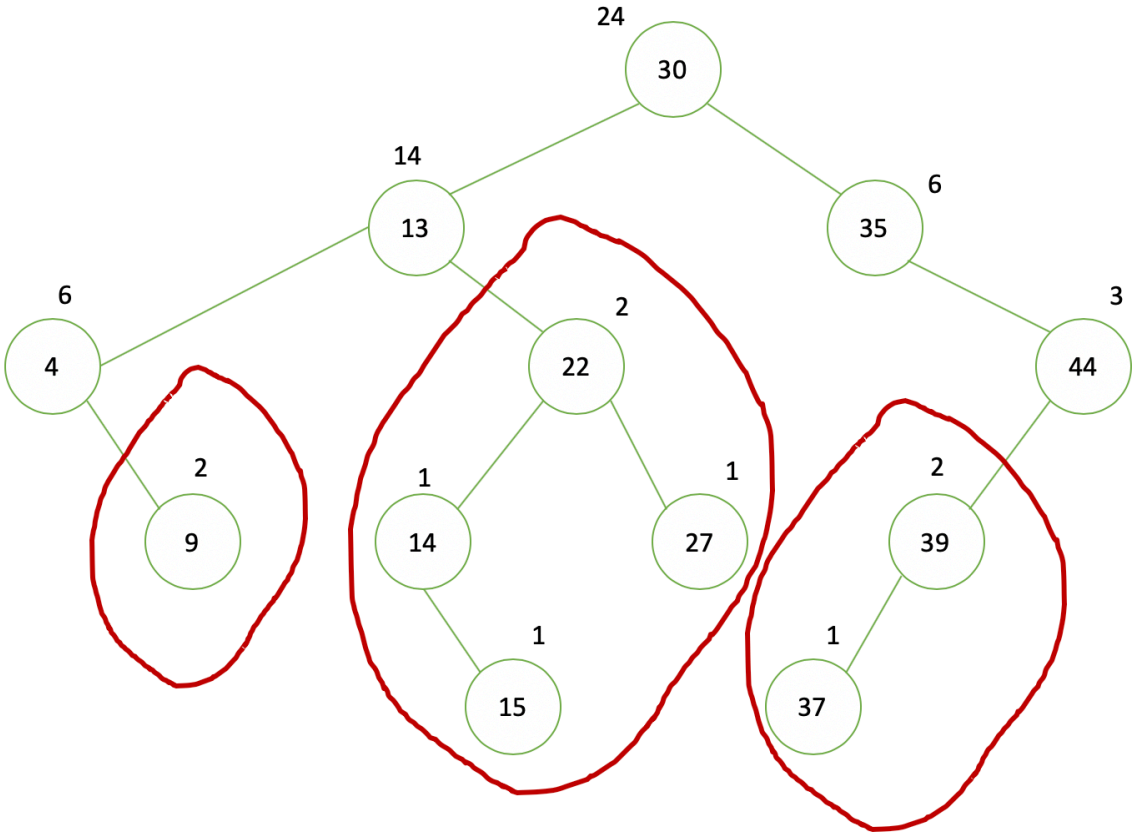

Figure 6. Improved tree leading to smaller treap

Query processing

For a query Q made of *q* terms *t*, we obtain the *k* best results from the union, we proceed one document at a time for each posting list *q*. (One term and it's associated posting list). We traverse the *q* posting lists in synchronization, finding the documents that contain some of the terms and computing the weights into $score(Q, d) = \sum_{t \in Q} w(t, d)$.

They are inserted into a minimum priority queue, limited to *k* elements. We keep score of the minimum of the stack, note it L, and pop the last one every time a new document gets a better score.

Intersection

Let *d* be the smallest docid not yet considered. All treap have their stack. They advance toward the node representing *d*, while **skipping node using the current minimum of the stack L**. Since the frequencies are decreasing in the treaps, we can compute an Upper Bound **U** to the score of a

document **d** by using the frequency of the node we have our cursor on rather than the frequency of **d.**
If this bound is inferior than $score(Q, d)$, it means that there **is a valid set of *k* documents where *d* does not participate**, meaning we can completely discard *d*, and **all the documents under it.**
If not, in some treap chosen with a certain scheme not detailed here, we advance toward d. If we have reached *d* in all *q* treaps, *d* becomes part of the intersection, which may increase the boundary L.

Remember that must also use the lists computed elsewhere for elements with $f < f_0$ to adjust the scores.

Unions

Ranked unions are computed in a likely manner, with a few variations mentioned in the article.

Evaluation and results

Experiments were run on the TREC GOV2 collection, containing 25.2 million documents and 32.8 million terms in the vocabulary. This article's solution is compared to the Block Max solution, which was the current standard for best solution.
Treap based approach is a 18% disk usage gain over this solution.
Treaps are 11% of that usage, the list with documents for $f < f_0$ take 41% of the disk.

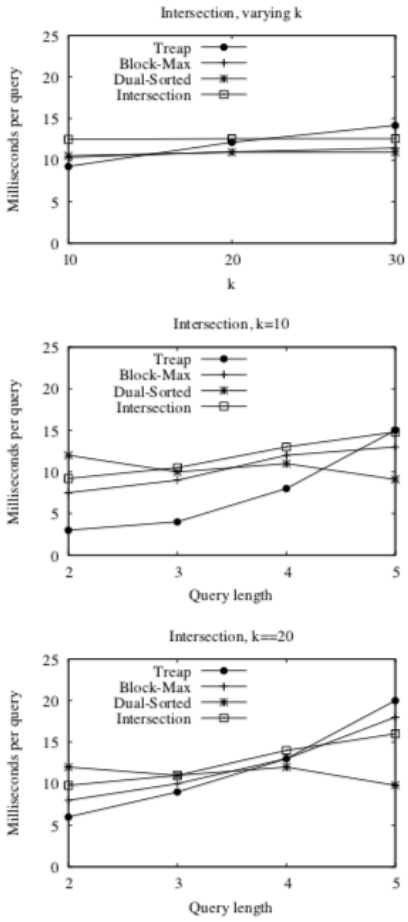

Figure 6: Time performance for ranked intersections. On top, for all the queries and increasing k. The other two discriminate by number of words in the query and use fixed $k = 10$ and $k = 20$.

Here we copy the results for ranked intersection. We can see that treaps start to perform much better than their competition as we want more results (*k* increasing) and as the query gets more complex (*q* increasing).

Critical Analysis

We found the article very convincing and well written. It is hard to understand and sometimes skip over the difficulties.
On the treap representations, we have had to trust the graphs and change the formulas for differential encoding of docids and frequencies for the rightmost node. There was a difference between the results and what was shown on the graphs.
This summary contains our revised version of the formulas.

It is also set upon a couple other scientific articles, especially one on Block Max, which we have had to read diagonally to understand the concerned paragraph.

It is especially interesting, and this is why we selected this article amongst the others, to see an entire article on the improvement of the inverted index after spending hours encoding our own inverted index. What we did was quite simple, with no real questioning about the space our index take, but here we see how these questions are crucial.