Lab Session 13: Designing a Search System II.

Cristina Aguilera, Jesús Antoñanzas, Alex Carrillo.

Information Retrieval, GCED, UPC - December 21st 2019

1 Improvements on already proposed functionalities

1.1 Finding communities and filtering topics

Our search system can be seen as a community network where diverse scientists and researchers share their works and studies. This network is said to have a community structure as the nodes of the network (*i.e.* the authors) can be easily grouped into sets of nodes (*i.e.* the data science community) such that each set of nodes is densely connected internally. Then, the goal is to find a community that a certain vertex belongs to. This information will be useful when categorizing papers into different topics and to allow individuals to filter documents.

Let $G = (V, E)$ be the network graph of the search system database, where the authors represent the V set of nodes and the citations of authors/collaborators in each paper represent the E set of edges. *Hierarchical clustering* is the most popular and widely used method to analyze social network data, like ours. In this method, nodes are compared with one another based on their similarity. Larger groups are built by joining groups of nodes based on their similarity. A criterion is introduced to compare nodes based on their relationship. Then, there are two types of hierarchical clustering approaches: 'divisive' or 'agglomerative'. The *divisive approach* (or *top-down*) is discarded due to the fact that it initially considers that all nodes belong to the same cluster (which it is not primarily our case) and its complexity grows in comparison. The *agglomerative approach* (or *bottom-up*) is more suitable instead. In this method, each node represents a single cluster at the beginning; eventually, nodes start merging based on their similarities. The algorithm is be as follows:

1. Compute the distance matrix between the input data points
2. Let each data point be a cluster
3. **Repeat**
   3.1. Merge the two closest clusters
   3.2. Update the distance matrix
4. **Until** only a single cluster remains

In our network, the distance or similarity measures are provided by the set of weighted edges between authors, that is, the weight value is computed from the cosine similarity between the two or more linked papers of that pair of authors. This means that every pair (v_i, v_j) of nodes will have $e_1, \ldots, e_k$ weighted edges, where k is the number of documents they have in common.

Regarding implementation, executing this algorithm for every new author found in the search system would not be feasible, as the already set up clusters would be forgotten and the procedure should start over. As it was stated in the previous report and following the same strategy that was proposed for the web crawlers, the execution of the hierarchical clustering should be only considered every certain period of time, say, every two weeks or every month.

1.2 Trending topics

In our search system we want to keep track of *Trending Topics*, that is, scientific topics and paper that are being substantially more visited than others at the moment. This way we get information of this events in real time from a stream of information that we could not store.

This problem is equivalent to solving the heavy hitters problem in a data stream. We want to design a small space data structure so that it can scan a stream of data items and at the end of the stream output the k-most frequent items that have occurred. We also want this data structure to have a small space complexity. For this purpose we will use *Space Saving* algorithm, which is one of the most efficient algorithms so far. The underlying idea is to maintain partial information of interest and update the counters in a way that accurately estimates the frequencies of the significant elements. The algorithm is

straightforward. When we observe an element in the stream, there are two possibilities: I) it has already been monitored so we just increment its counter or II) it's not being monitored so we replace the element that currently has the least estimated frequency with this new one, assigning the counter of the previous increased by 1.

This way, we are taking advantage on the fact that data is skewed and we expect a minority of the elements to have the highest frequencies, obtaining an error in the counters inversely proportional to their number. So, keeping a moderate number of counters will guarantee very small errors.

In addition, if the popular elements change over time, this algorithm adapts automatically. The elements that are growing more popular will gradually be pushed to the top of the list as they receive more hits. If one of the previous top elements loses its popularity, it will receive less hits, so its relative position will decline and eventually get dropped from the list.

2 Efficiency & scalability improvements

2.1 Consistent Hashing

(Could belong in 'new functionalities', too). The ability to swiftly respond the our search system's user's petitions is as important as the overall functionalities implemented. That is why we would like a mechanism with which to distribute cached files to our servers in an optimal manner, so that petition loads are balanced. Enter Consistent Hashing: a distribution scheme which does not depend directly on the number of servers, avoiding having to rellocate all pages to different servers every time a new cache is added. Thanks to consistent hashing we are able to dynamically adapt our server's responsiveness to demands in different geographical locations or of very popular papers. So, and because papers are not large files, we consider a situation when many of the most demanded papers at the time are cached in a single server (a hotspot), while other servers are not really being pushed. In this case consistent hashing would make it possible to know how to distribute those files in order to optimise responsiveness.

We are not going to discuss how the algorithm works, but regarding the implementation we would keep track of a binary tree, which would have hash values at the nodes and server values at the leaves. That would leave us with a very efficient implementation as all removal and addition of nodes in a binary tree is $O(logn)$ (where n is the number of servers) and hashing is $O(1)$. As per the reduction of variance and unbalancing, we would map each server to k 'virtual servers' (refer to slide 59 lesson 5) so petition distributions are balanced between servers. This is important as an upcoming business, because we would not be able to afford many physical servers thus requiring strict balance.

2.2 Non relational storage

But first, we have to consider where the cached files come from. A relational database system does not scale well at all, so when more and more documents are retrieved from the web and uploaded from new individuals our storage system can become unusable given the sheer amount of documents that need to be stored. So, we would like use a non-relational database system, running in relatively cheap and not failure-proof machines. Granted, the Hadoop ecosystem is a very good option for this case, providing tools for analysis, management, retrieval, upload amongst others. The files would be distributed amongst HDFS (Hadoop Distributed File System). As many documents in the web are representations of JSON files, we can take advantage of this with the AVRO format, which represents files in a schematic fashion and is optimized for row access (in our case, each row would be a document information about it of many kinds: author, labels, date, title, header...). This schema is very rich and aligns very well with our JSON structured documents. This data distribution strategy would allow us to have a scalability power that would otherwise be not possible.

3 Conclusion

Our search system is now complete. Bip-Bop.