Impact-Layered Multi-Tier Index for Improving Search Efficiency

Alexander Gao, Nikhil Nar

Final Project
CS-GY 6913: Web Search Engines, Fall 2019
New York University

Background & Motivation

The baseline objective of any search engine architecture is to deliver relevant and high-quality results given any query. Moreover, queries must be executed and results delivered within a reasonable time. The more efficiently a single query can be completed, the better the experience will be for a user of our search engine system, and the less computational cost we will incur on the back end. This is an especially important design consideration for commercial-scale search engines which receive a huge volume of queries per second.

We explore a multi-tiered, impact-layered index in order to significantly increase query efficiency while still achieving a high degree of precision in returned results. Rather than sorting postings by *Document ID* for a given term, we sort postings by an *impact score*, which is based on the individual contribution the posting would make to a BM25 score for a document, given some query. This is described in more specific detail in following sections. We have also designed a simple fall-through algorithm that determines which tiers to route a query to prior to executing the query.

Documents / Data

We use a collection of approximately 275,000 cooking recipe documents available on the web. Our initial goal was to create a tiered-index search engine which would be specific to a cooking recipe domain. However, the work we have done is easily generalized to a larger collection of documents, which do not necessarily need to be related to cooking, or any specific domain.

Running The Query Program

In the main directory 'MultiTierSearchEngine', there is a JAR file that can be run using the following commands.

```
$ cd <path to 'MultiTierSearchEngine' directory>
$ java -cp MultiTierSearchEngine.jar wse_project.MainQueryProgram
```

The query program will run in the command line, and will provide a series of prompts that are self-explanatory.
These prompts include:

- Tier 1 Split Percentage (choose 10% or 20%)
- Target # of results to return for each query
- Batch query or manually-input single queries
    - If Batch Query, # of queries to be executed in batch (up to 60,000)
    - If manually-entered queries, simply enter queries.

Queries will be executed on both the Single-Tier and Multi-Tier systems. As our primary interest is to compare these two systems in terms of average precision and performance, individual results are not displayed as snippets. Instead, information regarding the metrics of interest are displayed in the following format:

```
QUERY: ramen custard sugar

* 0 documents in Tier 1 contain all query terms. *
( EXECUTING QUERY IN TIER 1 ONLY. )

Mutli-Tier Query Time = 0.006 s
        | # Results Returned: 50

Single-Tier Query Time = 0.025 s
        | # Results Returned: 50

Average Precision (AP): 0.9834238
```

Initial Exploration & Ideation

We considered multiple possible strategies for partitioning our inverted index into tiers. *Ding and Suel, 2011* [2] provide a helpful overview of some of the most widely-used indexing strategies, which we use as a starting point for designing our system.

The strategies we considered included (A) classifying documents into tiers of quality based on features extracted from the documents, (B) assigning terms to tiers based on their Document Frequency, and (C) sorting term postings-lists according to an "impact score" of our design, and partitioning the list for every term according to an experimentally determined percentage split. Each of these architectures has potential merits and tradeoffs.

In our exploration of (A), we considered using the following as features to classify cooking documents into bins of quality: document pagerank, ingredients used, number of reviews, recipe average rating. For this particular approach on this corpus of recipe documents, consistency of features proved to be a significant problem. Across the entire collection of documents, the large majority had missing values for these features, and thus we were not able to engineer a robust feature set. Another problem was the lack of ground truth "labels" to be able to use as training data to train a classifier that would be able to predict the quality of previously unseen documents. This issue could potentially be circumvented by creating a heuristic "recipe" to assign a score to each document, and then simply assigning the top k% of documents to tier 1, and so on for the subsequent tiers. Ultimately though, we decided that more robust features would be necessary for this approach to be the most effective.

We then considered approach (B). The logic of assigning terms to tiers based on their Document Frequency seems promising. After all, the most commonly seen terms across the corpus of documents would be assigned to an earlier tier, and the less frequently occurring terms would be assigned to the later tiers. In general though, we would like for Tier 1 to be significantly smaller than Tier 2, **and** for **most** queries to only need to be routed to Tier 1 while maintaining a high amount of precision. However, if we used a 20%/80% split between Tier 1 and Tier 2, it seems unlikely that Tier 1, containing 20% of terms, would deliver high-precision results for the majority of potential queries. On the other hand, if we had access to the most common queries (e.g. the top 1M cooking recipe queries), then we could certainly use this to our advantage, and include all terms within those queries in Tier 1. Since we do not have access to such data at this time, we consider one more architecture.

Approach (C) is distinct from the previous strategies, in that rather than assigning distinct terms or documents to a single (exclusive) tier, we split the lists for each term across multiple tiers. Thus, each term will be found in each tier, and each document will likely be found in multiple term lists, in multiple tiers. This is known as *Impact-Layering*. We will assign each document within each term list a score based on the BM25 contribution for the term-document. In this project, we evaluate this technique. We also consider the problem of routing incoming queries to the appropriate tier(s) in order to maximize precision while minimizing query execution time.

General Indexing Architecture and Auxiliary Data Structures

We have a dataset of 275,000 cooking recipes extracted from multiple websites and stored in a json file. We divided the process of index development in three phases. In first phase we created postings. The postings are created by parsing the JSON file records one by one and extracting terms from title, instructions and ingredients fields in the json file. The url present in the JSON record is assigned a document id. A postings.gz file is created which stores the term, document id and frequency of each term in that document. Another file called url_to_doc_mapping.gz is created which stores the document id, url and the total number of terms in the document. The total number of terms are used to calculate the BM25 score. The second phase includes the sorting of the postings. A shell script is written to sort the postings by term. A unix merge sort is used to sort the postings by term. A sorted.gz file is created as an output of the merge sort. The third phase includes creating the lexicon files and inverted index for each tiers. The sorted.gz file is read and is parsed term by term. A tier percentage parameter is set to find out the total number of documents that should be part of tier 1 and tier 2. A list is created for each term which contains frequency followed by document id. The frequency and document id is stored using var byte compression. A lexicon file is created for each tier. The lexicon file contains the term, starting byte of the term in the tier and the total number of bytes (including both Frequencies and Document IDs). Upon running the query program, both the Lexicon and Inverted Index files are loaded into main memory to retrieve the results of the queries.

Sorting Postings by BM25 Contribution (Impact Score)

For a given term in the inverted index, the documents contained in that term's list are ordered by *impact score*. We chose a simple scoring function that is based on the individual contribution a document would make to a query that contains that term. The general BM25 formula:

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

$$K = k_1 \times \left((1-b) + b \times \frac{|d|}{|d|_{avg}}\right)$$

(Credit: Search Engines, Lecture 4 Slides, Prof. Torsten Suel, NYU Fall 2019)

As the BM25 score for a document given query terms is defined as the sum of individual contributions to that document's score from each of the individual terms. Therefore if we precompute the BM25 contributions for all term-documents, then sorting documents in a term list by their BM25 contribution makes sense: instead of generating a large list of candidate documents, computing the BM25 score during query time, and returning documents in order of decreasing BM25 score, documents will already be sorted, and we can take advantage of early termination algorithms to improve efficiency of querying.

Tier Assignment / Partitioning Strategy

Now that we have established our method for sorting postings based on impact score, we need a strategy for partitioning term lists into tiers. In order to maximize efficiency, we would like for Tier 1 to be as small as possible while still delivering high average precision in the average case. Thus, we decide to assign a percentage of each term list to Tier 1 and Tier 2. In our evaluation, we tested 10/90 and 20/80 splits between Tier 1 and Tier 2. The empirical results reflect what seems intuitive: if Tier 1 is smaller, then the performance savings are greater, but precision is negatively impacted. Ultimately, using a 20/80 split seemed to deliver a good balance between these two objectives.

Tradeoffs

There are tradeoffs about our design that are worth considering.

Compression

Indexes that are sorted strictly by Document ID have the benefit of term lists being in sequential order. This is highly conducive to compression techniques such as gap encoding. Sorting by impact score, however, presents a challenge, in that corresponding Document IDs are not in a predictable order, and therefore gap encoding is likely to be inefficient. In our implementation, we use Var-Byte compression. For extremely large datasets, this is a significant tradeoff of using impact-sorted indexes.

Performance vs. Precision

This tradeoff is primarily related to the percentage of postings split between Tier 1 and Tier 2. The smaller Tier 1 is in relation to Tier 2, the better performance will be, as there are guaranteed to be equal (but even more likely) fewer random lookups required to reach the stopping condition of the Threshold Algorithm. The more documents are available in Tier 1, the more lookups will be required, but this will also lead to equal or greater average precision.

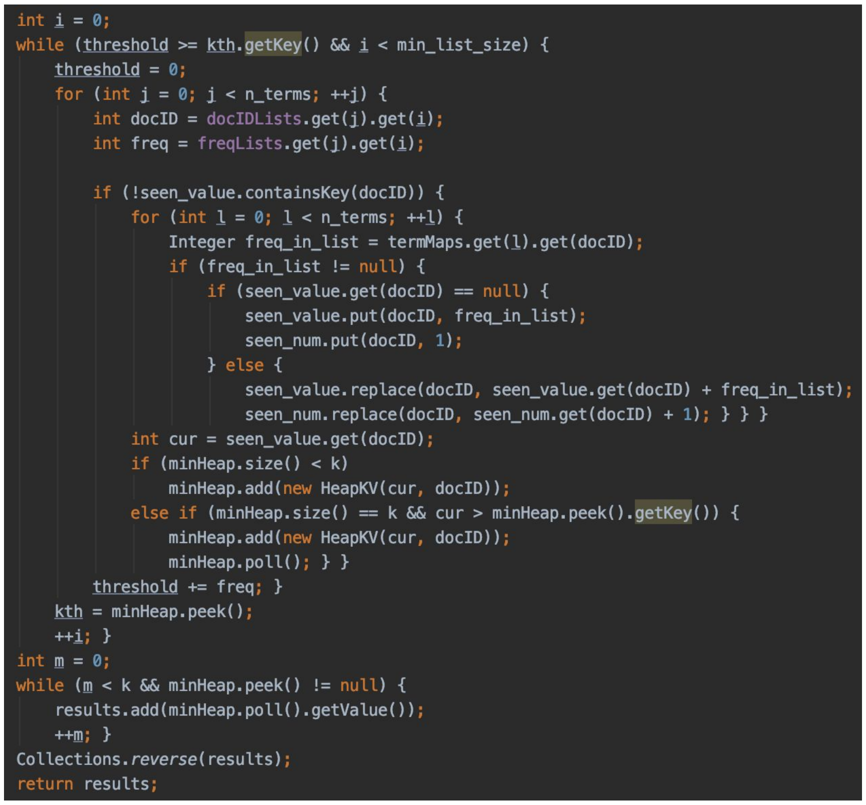
Generating Queries

To evaluate our tiered system, we need to generate a high volume of test queries, which will be used to compare performance of the multi-tier architecture against the single-tier architecture. Having a large volume of test queries will also help us to tune performance parameters such as split percentage between Tier 1 and Tier 2.

We took a very simple probabilistic approach to generating queries based on the frequency of terms across the corpus of documents. The idea is to read the sorting.gz file (intermediate file generated from merge sorting the postings) and count the frequency of each unique term. The sorting.gz file stores term, Document ID and frequency. A hashmap is created in memory to hold the count (cumulative frequency) of each term -- the key of the hashmap is the term and the value is the total frequency across all the documents for that specific term. The next step is to find the probability of the term across all the documents by dividing the term frequency by the total number of tokens in the entire document collection. We then generate a random number between 0 and 1 from a Uniform distribution, and that number is mapped to a corresponding term. Ultimately, we generated 60,000 random queries, each with three terms, where each query is unique from any other query, and no terms are repeated within a given query.

Executing Queries using Threshold Algorithm

Once term lists have been read from disk and loaded into memory, we perform queries using the Threshold Algorithm, developed by Lotem et al, 2011 **[3]**. Threshold Algorithm (TA) is an early termination algorithm that achieves optimality, and is ideal for our use case. Since our term lists are already sorted by impact score, there is no need to continue considering documents once we have met the stopping condition presented the Threshold Algorithm. This is in contrast to algorithms that execute queries on document-ID-sorted lists, which must consider the entire lists. Our implementation of the Threshold Algorithm (in Java language):



```java
int i = 0;
while (threshold >= kth.getKey() && i < min_list_size) {
    threshold = 0;
    for (int j = 0; j < n_terms; ++j) {
        int docID = docIDLists.get(j).get(i);
        int freq = freqLists.get(j).get(i);

        if (!seen_value.containsKey(docID)) {
            for (int l = 0; l < n_terms; ++l) {
                Integer freq_in_list = termMaps.get(l).get(docID);
                if (freq_in_list != null) {
                    if (seen_value.get(docID) == null) {
                        seen_value.put(docID, freq_in_list);
                        seen_num.put(docID, 1);
                    } else {
                        seen_value.replace(docID, seen_value.get(docID) + freq_in_list);
                        seen_num.replace(docID, seen_num.get(docID) + 1); } } }
            int cur = seen_value.get(docID);
            if (minHeap.size() < k)
                minHeap.add(new HeapKV(cur, docID));
            else if (minHeap.size() == k && cur > minHeap.peek().getKey()) {
                minHeap.add(new HeapKV(cur, docID));
                minHeap.poll(); } }
        threshold += freq; }
    kth = minHeap.peek();
    ++i; }
int m = 0;
while (m < k && minHeap.peek() != null) {
    results.add(minHeap.poll().getValue());
    ++m; }
Collections.reverse(results);
return results;
```

Simple Fallthrough Algorithm

We designed a simple *Fall Through Algorithm* that determines at query time whether to route the query to Tier 1 only, or to both tiers. The pseudocode follows:

```
1   k = # of results to be returned
2   for each term in query:
3       load term list from Tier 1
4   num_intersect = # of documents that appear in all term lists
5   if num_intersect != 0 and num_intersect <= k/10:
6       for each term in query:
7           load term list from Tier 2
8   run Threshold-Algorithm
```

"num_intersect" uses a simple intersection of sets algorithm to determine the number of documents that appear in all term lists loaded from Tier 1. We then arrive at the heuristic condition in line 5 using the following logic: if there are exactly zero documents that appear across all term lists, then it is unlikely that loading the lists from Tier 2 will find more documents that are contained in all terms' lists. Even if more documents were to be found, it is unlikely that these documents would be highly relevant to the query, given that the term contributions to the document BM25 score is in the bottom 80%. Also, if the number of documents found in all Tier 1 term lists for the query is greater than some threshold (we choose a threshold of k/10), then we hypothesize that enough of the relevant documents have been found in Tier 1, and we are unlikely to gain much by routing the query to Tier 2. The fall through case -- when we determine that

Evaluation (Multi-Tier Architecture versus Single-Tier Architecture)

We compare the performance of our multi-tiered architecture against the performance of a single-tiered architecture, with all other factors being identical (i.e. documents, ordering of term lists), and determine that our multi-tiered system delivers increased efficiency, with a tradeoff between precision of returned results, and computational cost savings, depending on the percentage of postings included in Tier 1 vs. Tier 2.

Average Precision

The main metric we use to evaluate the performance of the multi-tier system against the performance of a single-tier system is **average precision** -- the average of **precision@k**, for k == 1 to the max number of results returned. A simple example illustrates this. Suppose we have the following results returned from each system:

Single-Tier results:	Multi-Tier results:
{8, 12, 3, 27, 19}	{8, 7, 27, 14, 3}

Then, the **average precision** is calculated as follows:

Precision @ (k = 1): 1
Precision @ (k = 2): ½
Precision @ (k = 3): ⅓
Precision @ (k = 4): ½
Precision @ (k = 5): ⅗

Average Precision = (1 + ½ + ⅓ + ½ + ⅗) / 5 = **0.587**

Additionally, there are instances when the multi-tier system returns fewer results than does the single-tier system (and implicitly fewer results than the target # of results). In this case, we normalize the average precision by the ratio of (# results returned from multi-tier) / (# results returned from single-tier). If both systems return the same number of results, then the average precision is not affected. However, consider the following scenario, in which our target number of results is 8:

Single-Tier results:	Multi-Tier results:
{8, 12, 3, 27, 19, 31, 23, 18}	{8, 7, 27, 14, 3}

The multi-tier system has returned only 5 results, while the single-tier system has returned 8 results. Thus, we normalize the average precision by a factor of ⅝.

Average precision = 0.587 * (5/8) = **0.367**

Mean Average Precision

Mean average precision simply takes the mean of the average precision over many queries. Based on general statistics principles (i.e. Law of Large Numbers), as we increase the number of samples (individual average precision measurements), the empirical mean will converge to the true mean. Therefore we are fairly confident that if we execute >1000 queries, the mean average precision is a reliable estimator for the average precision for a single query.

Performance

Based on our empirical results, the 20% Tier 1 delivers higher M.A.P. than the 10% Tier 1, with the tradeoff being execution time. Additionally, as we increase the number of results to be returned, the M.A.P. trends downward, and the efficiency of queries also trends downward.

Overall, the results are promising, and we conclude that the multi-tier architecture that we have designed and implemented delivers an acceptable level of average precision while significantly reducing query time. We discuss possible future improvements and optimizations in the following sections.

Empirical Results

Tier 1 Split	Target # Results	Mean Average Precision	Multi vs. Single Tier Query Execution Time Ratio	# Unique Queries Executed
10 %	1	**0.810**	0.117	1000
10 %	3	**0.907**	0.112	1000
10 %	5	**0.924**	0.117	1000
10 %	10	**0.909**	0.166	1000
10 %	25	**0.822**	0.189	1000
10 %	50	0.749	0.206	1000
10 %	100	0.610	0.222	1000
20 %	1	**0.860**	0.226	1000
20 %	3	**0.953**	0.228	1000
20%	5	**0.966**	0.234	1000
20 %	10	**0.973**	0.265	1000
20 %	25	**0.927**	0.315	1000
20 %	50	**0.855***	0.357	1000
20 %	100	0.792	0.389	1000
20 %	50	**0.845** * empirical results for 5000 queries are approximately consistent with results for 1000 samples	0.368	50000

Bolded := MAP greater than 0.8

In practice, BM25 works well as a ***rough*** scoring function -- that is, the BM25 score is useful for narrowing down the potential result candidates on a coarse level. Ideally, we would use even more sophisticated scoring functions to rank the top 10, or top 1 results. Therefore, in our opinion, the most salient # of results is the 50-100 range. This range most realistically reflects the fineness that the BM25 scoring function can deliver.

Conclusion

We conclude that the multi-tier system we have designed accomplishes our goal: to improve query efficiency by a non-trivial factor while maintaining a high average precision. By sorting postings by their individual contribution to the BM25 score of a document given a query, we are able to partition tiers in a manner that takes advantage of early termination algorithms for retrieving the top-k documents. While average precision is not perfect (i.e. 100%), it is high enough that it seems sufficient to satisfy general search engine use cases. In a production search engine environment, we believe the architecture presented here might serve as a useful first step of a cascaded system: the multi-tiered system could provide a large number of documents, with document scores being equivalent to their BM25 score, and more refined/expensive ranking algorithms could then be run on the results returned from our system.

Future Work

There are many opportunities to follow up on the work we have done in this project -- in order to further optimize, evaluate, and expand the work presented here. These include:

- Evaluating with more diverse queries (not limited to only three terms).
- Generating queries with a more sophisticated language model, to better represent realistic English-language queries. Currently, we generate query terms based on a simple probabilistic model, which does well in generating diverse queries. However, we would like queries to more accurately represent the true distribution.
- If we had access to query logs, then we would want to take those most frequent queries into account in our initial document scoring function. This would likely have a major impact on improving both search precision and performance.
- Evaluating our system on much larger datasets. Currently, we have only evaluated our two-tier system on a collection of 275,000 documents. In the future, ideally we could evaluate our system on a collection of 25M+ documents.
- Using more than two tiers. For the sake of keeping our exploration focused, we limited the current implementation to two tiers. However, we could likely optimize performance by using more tiers.

References

[1] Knut Magne Risvik, Yngve Aasheim, Mathias Lidal. Multi-tier Architecture for Web Search Engines. In Proceedings of the First Latin American Web Congress, 2003.

[2] Shuai Ding, Torsten Suel. Faster Top-k Document Retrieval Using Block-Max Indexes. *SIGIR'11*, Beijing, China, July 2011.

[3] Ronald Fagin, Amnon Lotem, Moni Naor. Optimal aggregation algorithms for middleware. In Proceedings of the 20th ACM Symposium on Principles of Database Systems, pp. 102-113, 2001.

[4] Aris Anagnostopoulos, Luca Becchetti, Ilaria Bordino, Stefano Leonardi, Ida Mele, Piotr Sankowski. Stochastic Query Covering for Fast Approximate Document Retrieval. In ACM Transactions on Information Systems, Vol. 33, No. 3, Article 11, February 2015.

[5] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze. Introduction to Information Retrieval, Cambridge University Press. 2008.