Exploring Size-Speed Trade-Offs in Static Index Pruning

Juan Rodriguez
Computer Science and Engineering
New York University
Brooklyn, New York USA
jcr365@nyu.edu

Torsten Suel
Computer Science and Engineering
New York University
Brooklyn, New York USA
torsten.suel@nyu.edu

***Abstract*—Static index pruning techniques remove postings from inverted index structures in order to decrease index size and query processing cost, while minimizing the resulting loss in result quality. A number of authors have proposed pruning techniques that use basic properties of postings as well as results of past queries to decide what postings should be kept. However, many open questions remain, and our goal is to address some of them using a machine learning based approach that tries to predict the usefulness of a posting. In this paper, we explore the following questions: (1) How much does an approach that learns from a rich set of features outperform previous work that uses heuristic approaches or just a few features? (2) What is the relationship between index size and query processing speed in static index pruning? We show that an approach that prunes postings using a rich set of features including post-hits and doc-hits can significantly outperform previous approaches, and that there is a very pronounced trade-off between index size and query processing speed for static index pruning that has not been previously explored.**

***Index Terms*—static index pruning, web search engine, search engine performance, search optimization**

I. Introduction

Large search engines receive billions of queries per day that are evaluated over many billions of documents, leading to significant hardware and energy costs for executing queries. Current engines are based on inverted index structures, and storing and accessing these indexes accounts for a significant fraction of the overall query processing costs. This has motivated a lot of research on improving the efficiency of inverted indexes. In this paper, we focus on one of many proposed approaches, called *static index pruning*.

The basic idea in static index pruning is simple: remove index entries (postings) from the inverted index that are unlikely to be useful, thus obtaining a smaller index that can still achieve a result quality close to that of a full index. A smaller index then implies less memory needed to hold the index in main memory, and faster traversal of the now much shorter inverted lists for query terms. As argued, e.g., in [15], even with billions of queries per day, the vast majority of index postings never lead to a top-10 result in a given month; thus, many postings could potentially be pruned if we can predict which postings are unlikely to be useful for real user queries.

Static index pruning was first proposed in [14], and since then a number of authors have proposed methods for pruning [1], [5], [6], [8]–[15], [17], [18], [21], [24]. This includes various heuristics that use basic properties of the postings or documents, approaches that use observations from past queries to identify particularly useful postings or documents [1], [4], plus a few other methods. While overall a lot of progress has been made, with significant improvements in the trade-off between index size and result quality, there are still many open problems and possibilities for improvements. Our goal here is to address several of these issues.

Our main contributions are as follows:

- We approach pruning as a learning problem, where the goal is to predict for each posting how likely it is to lead to top results on future queries. This allows us to combine many basic and advanced features for improved pruning decisions. We show that by including features based on query language models and based on past occurrences of postings and documents in top results, we can significantly outperform previous work that mostly uses heuristics based on only one or two features.
- We study the problem of how to prune an index to minimize query processing cost. Previous work has focused on minimizing index size, with the expectation that a smaller index also reduces query processing cost. We study how to optimize for speed using an approach based on linear programming, and explore the trade-off between size and speed. Our experimental results show that focusing only on index size often gives only limited reductions in query processing costs, but that query processing costs can be significantly reduced if we allow a moderate increase in index size over the minimum size case.

This paper is organized as follows. Section II gives some technical background and discusses related work. Section III describes the basic setup for our approach. Section IV details our experimental setup, and section V explains the results of our work. We present our closing remarks in Section VI.

II. Background and Related Work

A. Inverted Indexes

Suppose we have a set D of n documents $d_0, d_1, \ldots d_{n-1}$ where each d_i is a sequence of words (terms). Then an *inverted index* for D contains one *inverted list* L_t for each

distinct term t that occurs anywhere in D. Each inverted list L_t is a sequence of postings, where each posting p contains information about the occurrences of term t in some document d_i.

We assume that postings are of the form (d, f), where d is a document ID, and f is the frequency of the term in that document. However, the approach does not require this exact format, and we could have additional data such as impact scores or position data in the postings.

We also assume that queries are run as disjunctive top-k queries, where k is typically between 10 and 1000. While the basic approach can be also applied to conjunctive queries, some changes would be needed to get best results. (In particular, the query cost model would have to be changed to account for the conjunctive case.)

We run our experiments using BM25 as a ranking function, other functions could also be used. An end-to-end evaluation of static pruning under complex ranking functions (with top-k disjunction as an initial filter) is deferred to future work.

B. Previous Work on Static Index Pruning

Static index pruning was first proposed in [14] in 2001, and since then a number of authors have studied the problem [1], [3], [5], [6], [8]–[13], [15], [17], [18], [21], [22], [24]. The focus of previous work is on achieving the best possible result quality given a limit on index size. While some papers also report running times for queries on the pruned index, they do not attempt to directly model and optimize for speed.

Previous work can be divided into several groups. Posting-oriented pruning techniques decide whether to keep individual postings, document-oriented pruning techniques keep or delete complete documents, and list-oriented pruning techniques keep or remove entire inverted lists. We can also group methods by their approach as follows:

Impact-Based Methods: Several papers consider posting pruning rules based on the impact or within-list rank of a posting according to a given ranking function, e.g., BM25 or a Cosine measure. Rules are designed to preserve as many of the top results obtained by that ranking function on the full index. Carmel et al. proposed two methods, UP, which uses a global cut-off on the impact scores of postings, and TCP, which selects the highest impact postings from each inverted list. Büttcher et al. [9] also evaluated TCP as part of TREC 2006, with promising results. Work by Chen and Lee [10] revisited the UP method, providing a theoretical foundation. Other work by Chen et al. [11] improves on [10] by refining the mathematical foundation and optimization objective. Nguyen [18] showed how to improve pruning by combining features to determine which postings should be kept.

Retrieval Quality-Oriented Methods: Another set of approaches try to select postings such that retrieval quality, measured in terms of measures such as P@10 or MAP, is optimized. An example is the work by Büttcher and Clarke [8], which selects postings based on KL-divergence, where the postings are selected based on their likelihood to result in the document being highly relevant under a query. Other approaches by Blanco and Barreiro [6], de Moura et al., and Thota and Carterette [21] use language models to select postings. Yet other approaches try to remove entire documents [22], [24] or inverted lists [5], [20] that are unlikely to be useful from the index.

Using Query Traces: Another approach is to use past queries to decide which postings should be kept. Work by Lam et al. [17] combines the impact-based approach in [14] with a query-based approach that looks at how often a posting appears in the result set. Altingovde et al. [1] introduce an approach called QueryView (QV) that marks and keeps postings that were part of top results in past queries. Another version combines this rule with the TCP method in [14]. Work by Anagnostopoulos et al. [3] applies this idea to documents, by keeping all postings in documents that were often returned as top results in the past.

Hybrid Methods: Recent work by Jiang et al. [15] implements a posting selection model, UPP, based on combinations of language modeling, impact-based models, and query-views to derive the *promise* of a posting as the basis for selection. UPP was shown to outperform previous methods.

We compare our results in Section IV to TCP, UP, UPP.

C. Related Work

There are many techniques for improving the performance of query processing algorithms on inverted indexes, including early termination algorithms (also called dynamic pruning) that choose which postings to consider at query time, index tiering techniques, index compression methods, and index reordering approaches. Static index pruning is an example of an unsafe early termination technique, i.e., a technique that does not guarantee to return the same results as an exhaustive algorithm. Current large search engines commonly apply a number of such techniques in conjunction.

Among these techniques, index tiering may be the most closely related. In this technique, the index is divided into two or more subsets called tiers, and queries are first routed to the first tier, and only evaluated on further tiers if results from lower tiers are considered insufficient. Index pruning could be considered as a special case of tiering where we only keep the first tier and queries are never sent to additional tiers.

Finally, there has been some amount of recent work on predicting query execution costs that is relevant to our work. We note here that complex models such as those in [23] would be too unwieldy to use in making pruning decisions. For disjunctive queries, which are the focus in this paper, a simple model uses the sum of the inverted list lengths of the query terms as an estimate of query cost. We use this model, which allows us to assign an expected query processing cost to each posting that is determined by the query language model. We note that this model primarily applies to a scenario where the index is kept completely in memory, or where enough index data is cached so that disk access is not the main performance bottleneck.

III. Problem Definition and Approach

A. Problem Definition

Static index pruning aims to reduce index size by removing postings that are unlikely to lead to top results for likely queries. We now formalize this in several definitions that capture the goals of small index size, fast query processing speed, and the trade-off between size and speed.

Setup: We assume that we have a full inverted index I that we need to prune, and a query distribution Q that models how likely a query is to occur in the future. In practice, during the pruning process, we only have an estimate of Q based on a language model built from a set of past queries, and then evaluate based on a distinct set of testing queries. In fact, our approach only uses unigram estimates during pruning: for any possible query term t we need an estimate of the likelihood $p(t)$ of t occurring in a random query. Similarly, we need to model index size, query processing cost, and result quality, as discussed further below. Given this, we now define the following problems:

Problem 1 – Optimizing Quality Given Index Size: Given a bound S on index size, the goal is to produce a pruned index I' of size at most S that maximizes the average quality of queries under distribution Q.

Problem 2 – Trading Off Size and Query Cost: Given a bound S on index size and a bound R on result quality, the goal is to produce a pruned index I' that satisfies these bounds while minimizing average query processing cost.

A related problem we have studied but do not address in this paper is the following: given a bound C on query cost, produce a pruned index I' that maximizes the average quality while achieving an average query processing cost of at most C. The size and cost models introduced in our work can be easily adjusted to solving this problem as well. We leave the experimentation to future work.

The above definitions require some way to model query probabilities, index size, query cost, and result quality. These models will be defined next.

B. Learning to Minimize Index Size

We start with Problem 1, where we optimize result quality given a bound on index size. The idea is simple: during pruning we optimize for a simple model of result quality, the expected number of top-k postings that are kept in the pruned index under a random query. We then use more realistic models of result quality, such as the number of top-k results preserved, and standard information retrieval measures such as P@10 and MAP, to evaluate the pruned index. We note that there is no guarantee that maximizing the number of preserved top-k postings also maximizes these other measures; however, we do not know how to directly optimize for these measures.

We model index size as just the number of postings. It would be difficult to model the exact size increase when including a posting in the pruned index under state-of-the-art compression methods, especially since that increase depends on how many other postings from the same inverted list are kept.

Formally, we say that a posting $p = (d, f)$ is part of a top-k result for a query q if document d is among the top-k results for q. We define $Pr[p \in topk]$ as the probability that p is part of a top-k result for q, and $Pr[p \in q]$ as the probability that the term associated with posting p occurs in q, given a random query q from the query distribution. Note that of course $Pr[p \in topk] = Pr[p \in q] \cdot Pr[p \in topk | p \in q]$. Our main goal is then to get a good estimate of $Pr[p \in topk]$.

Approach

The first step is to select a set of features that are likely to be useful for predicting $Pr[p \in topk]$. Before giving the list of all the features, we give a short description of the more involved and interesting features that we used:

- $Pr[p \in q]$**:** We build a language model on a subset of a few ten thousand queries, which allows us to get reasonable estimates of the probability of a term occurring in a random query. (This subset is of course disjoint from the queries used in the evaluation.) With the exception of [15], no previous work seems to have used this feature for pruning. Using this, we define an additional feature for each document d, called $xdoc$, as $\sum_{p \in d} Pr[p \in q]$. This feature measures how likely it is that a document is at least marginally relevant to any of the query terms.
- **Doc-hit features:** Work in [4] used a set of training queries and then counted how often each document was returned in the top-m results. A greedy pruning heuristic then kept the documents with the highest counts or, as we say, the highest number of *hits*. One problem is that it is not clear how to choose m – it is not clear that m should be the same as the number k used when we evaluate top-k result quality. In fact, a larger m will give us more data and thus coverage of more documents, and we observed that if a document has occurred before in, say, the top-100, this actually increases its chance to occur in the top-10 in the future. Thus we may choose m much larger than k. However, for such large m we need to suitably weigh hit counts, since a past hit at a rank around 1000 is not as predictive as one in the top-10. Our solution is to partition the ranks from 1 to m into a number of ranges, and collect hit counts separately for each range, to be used as features for the learner.
- **Post-hit features:** Work by [1], [2] showed how to use *post-hits*, i.e., past cases where an individual posting was part of a top-m result. The main problem with this approach, however, is sparsity: large indexes have billions of postings, and for each past query, only a few postings are part of the top results. For example, if the average query has 3 terms and we consider top-10 post-hits, then at most 30 postings will see their counts increased. We address this by choosing a large m, keeping separate counts for different rank ranges, and then relying on the machine learner to figure out how to weigh these features versus other features.

A complete table of the features we used for learning $Pr[p \in topk]$ is given in Table I.

TABLE I: Learning to Prune Feature List

Family	Feature	Description
term	tf	term frequency in the corpus
	tl	term list size
	p_bm25	partial BM25 score
	$Pr[p \in q]$	probability of posting in a random query
document	docSize	# words in document
	docTerms	# unique terms in document
	xdoc	promise of document
dochits	bins 1 - 17	dochit ranges, quantized by rank
	top10	top10 dochits from language model queries
	top1k	top1k dochits from language model queries
posthits	bins 1 - 17	posthit ranges, quantized by rank
	top10	top10 posthits from language model queries
	top1k	top1k posthits from language model queries

Another problem in generating post-hit features is that the Clueweb09 data set only comes with a few hundred thousand queries, not enough to actually *hit* most of postings of the index. Previous work using hits relied on the AOL query trace, but we felt that it was not appropriate to use this set, which also does not really "fit" with the Clueweb09 data anyway. Instead, we used the language model created for queries, as described in connection with the $Pr[p \in q]$ feature, to generate millions of artificial queries, and then ran these queries. (Since our evaluation queries are disjoint from the queries used to train the model, this is acceptable.) Our results show that this actually works pretty well, though one could argue that a larger real query trace might do even better.

We create labeled training data by generating features for a subset of postings by running another set of queries on the whole index, and checking how often the postings are in top-k results. Finally, the trained estimator is run on all postings, and the highest scoring postings kept in the index.

Given this model, it is straightforward to generate a selection algorithm to maximize quality: greedily select postings for the pruned index based on $Pr[p \in topk]$, up to a given bound on size S.

More details, e.g., on the number of training queries and machine-learning tools used, and the method used to generate the hit ranges are provided in Section IV.

C. Exploring the Size-Speed Trade-Off

Next, we address Problem 2, how to trade size versus execution cost in static index pruning. To do so, we first need to discuss a suitable model for query execution cost that can be used for pruning decisions.

Cost Model

One model that has been used in a number of previous studies models the cost of a disjunctive query as the sum of the lengths of the inverted lists of the query terms. The advantage of this model is that it allows us to assign a query execution cost to each individual posting: such a posting has an expected execution cost per query of $Pr[p \in q]$, since it causes one unit of cost whenever its term is part of an incoming query!

In summary, including a posting p in the pruned index increases query execution cost by $Pr[p \in q]$, index size by one, and result quality by an amount proportional to $Pr[p \in topk]$ (where both $Pr[p \in q]$ and $Pr[p \in topk]$ are estimations based on the language model and the machine learning approach in the previous subsection). When evaluating the proposed pruned index, we will of course use not just estimated running times, but also wall clock times, and quality measures such as P@10 and MAP.

Size-Speed Trade-Off

Given an upper bound S on total size and a lower bound R on result quality, we wish to minimize query processing costs. This is in fact a generalized version of a Knapsack problem where each item has a cost and a benefit, and we need to select at most S items with a total benefit at least R such that total cost is minimized.

While a precise solution to this problem is difficult, we can solve a fractional version (where we can choose a fraction of a posting to be in the pruned index) using the following Linear Programming relaxation:

$$min \sum_i x_i \cdot c_i$$

where $\sum_i x_i \cdot b_i \geq R$, $\sum x_i \leq S$, and for all i, $0 \leq x_i \leq 1$. Here, i ranges over all postings, and c_i and b_i are the cost and benefit of the ith posting. The resulting solution (the x_i's) tells us what fraction of each posting is kept in the pruned index. We note that such a fractional solution can be easily transformed into an integer solution (where a posting is either in or out of the pruned index) via randomized rounding [19], by randomly including each posting in the index or not with probability x_i. The expected benefit, cost, and size of this solution would be the same as for the fractional solution, and given the large number of postings the solution would be almost guaranteed to be very close to the expected value.

There is however one major problem with this approach, the size of the LP, as we have one variable for each posting in the index. State-of-the-art LP solvers can deal with millions of variables, but not with the billions required here. Luckily, there is a fairly easy solution. We simply quantize the benefit and cost values into a smaller value range. Thus, we quantize benefit and cost each into only $m = 1000$ distinct values using a simple form of logarithmic quantization. As a result, each posting belongs to one of m^2 classes based on benefit and cost. The resulting LP now has only m^2 variables x_i, where $0 \leq x_i \leq n_i$ with n_i is the number of postings in class i.

In summary, we estimate costs and benefits of each posting as before using an ML approach, quantize these into a smaller range of values, solve the resulting LP relaxation using a state-of-the-art LP solver, and then create a feasible solution by rounding the x_i to integer values. We note that there are some possible pitfalls with approaches that run an optimization method on top of machine-learned estimates. First, the actual quality achieved depends on the quality of these estimates, and moreover we would prefer these estimates to be unbiased over

the whole range of values. Second, when picking postings for the pruned index based on an estimate, we run into a form of selection bias where we are more likely to pick items whose benefit we overestimate; this leads to a pruned index whose overall quality is lower than what one would expect from naively adding up the estimated contributions of the selected postings. In fact, as we will see later, these problems show up in some of our data points.

IV. Experimental Setup

A. Corpus, Parsing and Indexing

We run all our experiments on the ClueWeb-09 Category B English text collection. We use a customized version of MG4J [7] as our search engine, using defaults parameters, no stemming, no stop word removal and no positional information. The relevant statistics for our initial baseline index are shown on table II.

TABLE II: ClueWeb 09 Cat B Text - Full Index

documents	5002579
terms	90382443
postings	16748354659
size on disk	26.2Gb

B. Query Log

We collected 165k queries from the TREC[1] 01-09 and 2002-2009 Adhoc, Terabyte and Web tracks for our work. The queries were then randomly split into 4 disjoint sets. We also have a set of 10M queries derived from the language model, which we explain in the next section. The final collection of query sets is shown in table III.

TABLE III: Query Log - Set partitioning

$100k$	for language modeling
$60k$	for hits generation
$5k$	for evaluation queries
TREC-2009 Web-track topics	for relevance measurements
$10M$	for feature generation

C. Language Model

We used the OpenGrm NGram[2] toolkit to build a 5-gram language model from our 100K query set. We modify the model by adding 5k randomly sampled pages from the CW09B corpus, at an interpolation factor of 85% (query set) + 15% (clueweb corpus).

We use the language model in two ways:

- We use the 5-gram model to generate a set of 10 million queries. We use these to create document and postings hits to use as a features in our machine learning.
- We use the model's unigram probability as the probability $Pr[p \in q]$, which forms the basis of our cost model.

[1]https://trec.nist.gov

[2]http://www.openfst.org/twiki/bin/view/GRM/NGramLibrary

D. Feature Engineering

Our features were introduced in section III-B. In this section, we elaborate on how we generate the hits features.

Hit Features

We ran 2M randomly sampled queries, from the 10M query set created by our language model, through the search engine, and captured the first 1000 results for both document and postings. For each document and posting, we classify the hits into one of 17 bins as shown in table IV. The document and posting bins become features in our machine learning.

TABLE IV: Hits - binning/quantizing

result rank	bin # (hits)
1...10	1...10 (one-to-one mapping)
11...20	11
21...40	12
41...80	13
81...160	14
161...320	15
321...640	16
641...1000	17

We evaluated the hits model quality, using MAP and P@10, at several sizes of between 400k and 2M queries. Each incremental size in query set showed an increase in model quality on those two metrics. We chose 2M queries as a trade-off between feature generation efficiency and quality. As noted elsewhere, access to a large corpus of real-world queries may also produce a better model.

Training Set

We ran our 60k query set on the full index and collected all postings that are in the first 1000 results of each query. This gives us a collection of several million postings as our training set.

During querying, we also collect the number of times each posting is in the top10 result of a query. This top10 count becomes the label for our supervised machine learning.

E. Learning to Prune : $Pr[p \in topk]$

We used a random forest tree regression algorithm to learn $Pr[p \in topk]$. We set the parameters of our learner to gradient boosting, 1500 trees, 1000-bin quantization, learning rate of 0.03, feature fraction of 0.8, and L2 minimization as the objective. We left all other parameters at their default values. Our ML tool is the Microsoft LighGBM library [16].

We run the ML predictor generated by our model for each posting in the full index, and append the predicted top-k value, normalized as a probability, back into the index. Note we could also use the predicted top-k count for pruning decisions, depending on the problem we are solving. We use the predicted probability as our pruning selector during in our solution to Problem 1.

We note that generating features for each posting in the index is computationally expensive, but easily parallelized, and in fact we perform this step in a Hadoop cluster.

F. Optimizing for Cost (MLP-CO)

We quantize all posting's benefit b, and cost c, modeled by $Pr[p \in topk]$ and $Pr[p \in q]$ respectively, into a $k \times k$ table using a simple logarithmic quantization. We use a cell's coordinate as the posting's class, so that each posting maps to exactly one of k^2 cells.

The cell's value is a tuple (K, B, C), where K is the # of postings in cell, B is the cell's benefit, and C the cell's cost, computed as the average value of the benefits and costs of the postings in the cell.

We formulate our linear programming equations as:

$$Objective : \min \sum_{i=1}^{k} (x_i \cdot C_i)$$

$$Constraints : \sum_{i=1}^{k} (x_i \cdot B_i) \geq TotalBenefit$$

$$\sum_{i=1}^{k} x_i \geq PrunedSize$$

$$\sum_{i=1}^{k} x_i \leq MaxSize$$

$$0 \geq x_i \leq K_i$$

where B_i is the cell's benefit, C_i the cell's cost, and K_i is the number of postings that mapped to the cell. In our work, we set $k = 1000$.

We can select a desired pruned size, say 5%, and incrementally relax the $MaxSize$ constraint. The LP solver will solve for the set of x_i variables, and our pruner uses the given solution to select $ceiling(x_i)$ number of postings of class i.

We call this algorithm the *Machine Learned Pruning - Cost Optimized* model (MLP-CO).

G. Computing Resources, Software and Datasets

The hardware environment used for this work consists of two Linux machines, each with 64Gb of RAM and 8Tb of hard disk, and a Hadoop cluster with a computational capacity of 1088 cores, 3TB of RAM, and 128Tb of HDFS storage.

Source code, datasets, query logs and global ordering data are available from the authors upon request.

V. Experimental Results

We used the following metrics as our measures of quality: $P@10$, $P@100$, $P@1K$, MAP, $PK@10$, and $RK@10$. We define $PK@10$, and $RK@10$ as the percentage of postings and results, respectively, retained in the pruned index that were also part of the top-k results against the full index, when issuing the 5K held out queries. All relevance metrics are computed using the TREC 2009 Web track relevance topics.

A. Machine Learned Static Pruning, MLP

As noted earlier, MLP is designed to maximize quality metrics, with the implied assumption that a smaller index leads to lower processing costs.

Table V lists the results of MLP at the very high pruning ratios that are the subject of this study. Our results show that an ML approach can be effective at these ratios. We are not aware of other published work targeting such high pruning ratios. Relevance

TABLE V: Machine Learned Static Pruning (MLP) Metrics

Size	RK@10	PK@10	P@10	P@100	P@1000	MAP
100%	**1**	**1**	**0.2944**	**0.1880**	**0.0397**	**0.1319**
1%	0.6850	0.6850	0.2306	0.1335	0.0293	0.0868
2%	0.7975	0.7980	0.2572	0.1517	0.0324	0.1015
3%	0.8505	0.8510	0.2701	0.1592	0.0342	0.1078
4%	0.8815	0.8720	0.2758	0.1674	0.0353	0.1130
5%	0.8965	0.9329	0.2747	0.1707	0.0360	0.1152
10%	0.9380	0.9525	0.2840	0.1774	0.0377	0.1199
15%	0.9595	0.9774	0.2862	0.1810	0.0384	0.1236
20%	0.9665	0.9875	0.2888	0.1812	0.0389	0.1259
25%	0.9770	0.9876	0.2909	0.1812	0.0389	0.1267
30%	0.9815	0.9926	0.2919	0.1831	0.0392	0.1282

Figure 1 compares MLP against previous work: TCP and UP [14], and UPP [15]. Due to the computational expense of computing previous work, we limit the comparison to a minimum of 90% pruning ratio (our experiment was designed for parallel computation, and allows for more experimentation).

Our results show that MLP can outperform previous methods, while significantly increasing the number of relevant postings and documents retained in a pruned index. This is as expected since we are learning to maximize the number of top-k items retained the pruned index.

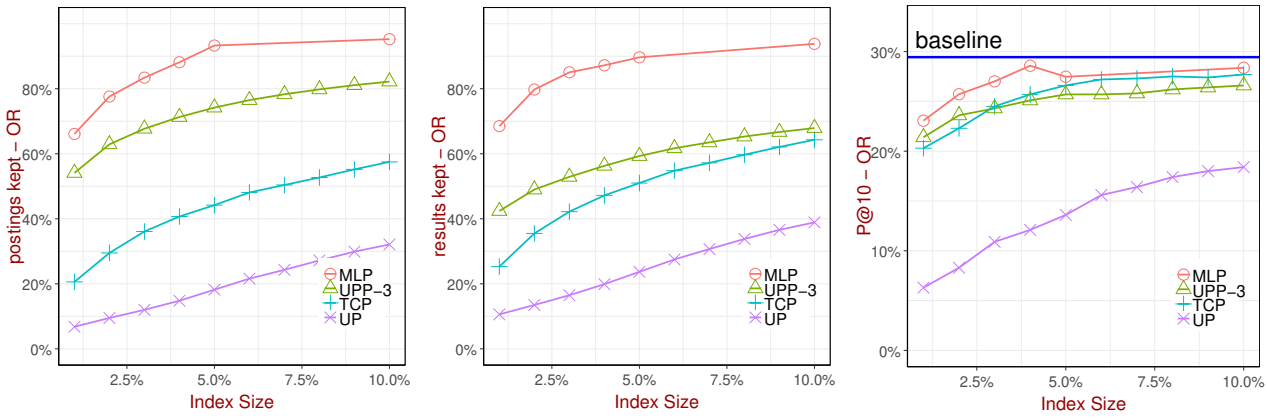
B. Size/Speed Trade-Offs

We selected 5%, 15% and 25% pruned index sizes as the starting point for our speed optimizations. The results of our experiments are shown in Figure 2 and Table VI. Because MLP is a greedy algorithm, it allows for the possibility that other posting combinations exist that can provide the same or better benefit but at a lower cost than one that has been selected by MLP, and in fact MLP-CO is able to optimize for speed at the same original size, with no degradation in quality; e.g. MLP 5% and MLP-CO 5%. This effect is more pronounced at lower prune ratios (higher index sizes).

We also note that quality starts to break down after several step increases, e.g. the 5.7% size level for the 5% pruned size. We believe there is a form of *selection bias* effect from our algorithm, as previously noted: that as we grow the index by adding less costly but also less beneficial postings, our ML method might be overestimating the real benefit of the postings at the lower end of the benefit ranges - that our model is not a fully unbiased estimate of $Pr[p \in topk]$, and that it exaggerates the usefulness of relatively poor postings. We leave the exploration of this effect to future work.

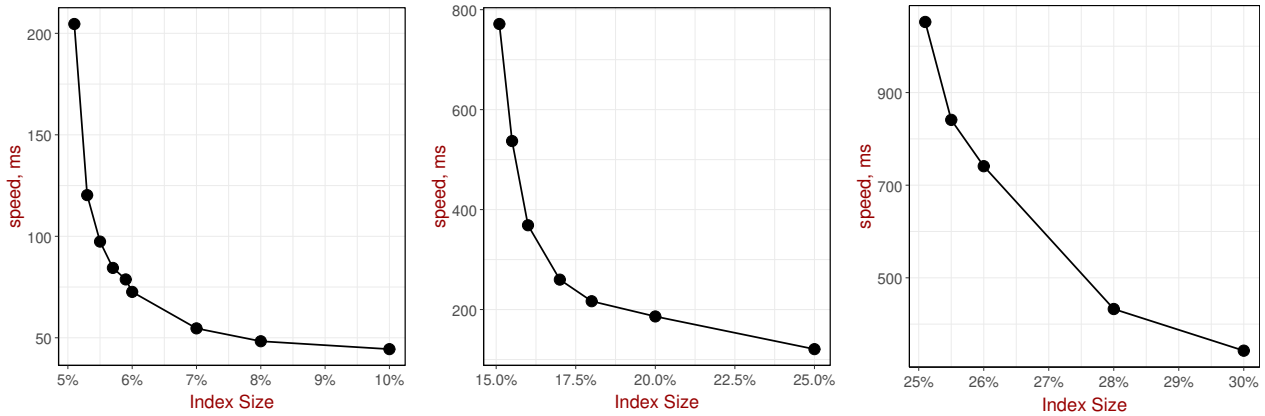
VI. Concluding Remarks

In this paper, we proposed a machine-learning approach for static index pruning based on multiple features that is shown to outperform previous methods. Using this approach, we then



(a) PK@10 (b) RK@10 (c) P@10

Fig. 1: Quality metrics for MLP and previous-work static pruning models. Queried using disjunctive queries.



(a) 5% (b) 15% (c) 25%

Fig. 2: MLP-CO Speed/Size trade-off at several base pruned sizes. Run as disjunctive queries, on a single threaded, single core computer.

explored the trade-off between pruned index size and query processing cost using an LP formulation.

Our results also show how a pruned index that is optimized for size can be made to run substantially faster by allowing a marginally larger index that achieves the same result quality with significantly smaller query costs, sometimes by a factor of 2 to 5.

There are several unresolved questions and new open problems that arise from our work, as follows:

- **A More Robust Trade-Off:** The quality-speed trade-off eventually breaks down. We are currently working on changes in our ML setup that we hope will address this issue.
- **Complex Rankers:** Simple rankers such as BM25 are used to generate an initial set of candidate results that are then re-ranked by a more complex ranker. An interesting open problem is how to build pruned indexes that perform well for the task of candidate generation for the re-ranking phase.

Acknowledgement

This research was partially supported by NSF Grant IIS-1718680; "Index Sharding and Query Routing in Distributed Search Engines", and by a grant from Amazon.

References

[1] I. S. Altingovde, R. Ozcan, and O. Ulusoy, "Static Index Pruning in Web Search Engines: Combining Term and Document Popularities with Query Views," *ACM Trans. Information Systems*, vol. 30, no. 1, pp. 2:1–2:28, Mar. 2012.

[2] I. S. Altingovde, R. Ozcan, and Ö. Ulusoy, "A Practitioner's Guide for Static Index Pruning," in *Advances in Information Retrieval*, 2009, pp. 675–679.

[3] A. Anagnostopoulos, L. Becchetti, I. Mele, S. Leonardi, and P. Sankowski, "Stochastic Query Covering," in *Proc. 4th ACM Int. Conf. on Web Search and Data Mining*, 2011.

[4] A. Anagnostopoulos, L. Becchetti, I. Bordino, S. Leonardi, I. Mele, and P. Sankowski, "Stochastic Query Covering for Fast Approximate Document Retrieval," *ACM Trans. Information Systems*, vol. 33, no. 3, pp. 11:1–11:35, Feb. 2015.

[5] R. Blanco and A. Barreiro, "Static Pruning of Terms in Inverted Files," in *Proc. of the 29th European Conf. on IR Research*, 2007, pp. 64–75.

[6] R. Blanco and A. Barreiro, "Probabilistic Static Pruning of Inverted Files," *ACM Transactions on Information Systems*, vol. 28, 2010.

TABLE VI: MLP and MLP-CO static pruning results. The area where quality starts to decline is shown on red (lighter color on b/w). The best quality results are in bold. Speed measured on a single-threaded, single processor environment

min size %	max size %	model	speed ms/query	postings scored/query	RK@10	PK@10	P@10	P@100	P@1000	MAP
100	**100**	baseline	**1405.80**	**15,552,973**	**1.0000**	**1.0000**	**0.2944**	**0.1880**	**0.0397**	**0.1319**
5	5.0	MLP	430.60	3,708,027	0.8965	0.9329	0.2747	0.1707	0.0360	0.1152
5	5.0	MPL-CO	417.16	3,640,072	0.8980	**0.9329**	0.2753	0.1708	0.0360	0.1152
5	5.1	MLP-CO	204.61	1,382,320	0.9035	0.9228	0.2768	**0.1731**	0.0312	0.1166
5	5.3	MLP-CO	120.29	661,134	**0.9145**	0.9105	0.2790	0.1727	0.0368	0.1164
5	5.5	MLP-CO	97.39	456,832	0.9110	0.9001	0.2781	0.1719	**0.0368**	**0.1164**
5	5.7	MLP-CO	84.42	346,555	0.9025	0.8896	**0.2821**	0.1713	0.0350	0.1158
5	5.9	MLP-CO	78.75	281,108	0.8940	0.8833	0.2796	0.1698	0.0362	0.1145
5	6.0	MLP-CO	72.61	256,335	0.8840	0.8786	0.2750	0.1687	0.0359	0.1134
5	7.0	MLP-CO	54.64	139,973	0.7750	0.8654	0.2530	0.1520	0.0332	0.1024
5	8.0	MLP-CO	48.30	103,752	0.6800	0.7616	0.2247	0.1375	0.0320	0.0910
5	10.0	MLP-CO	44.41	78,956	0.6080	0.7072	0.1995	0.1326	0.0312	0.0871
15	15.0	MLP	939.98	8,243,194	0.9595	0.9774	0.2862	0.1810	0.0384	0.1236
15	15.0	MPL-CO	868.11	7,992,232	0.9595	0.9774	0.2862	0.1810	0.0384	0.1237
15	15.1	MLP-CO	771.39	6,463,987	0.9590	**0.9774**	0.2857	0.1812	0.0385	0.1240
15	15.5	MLP-CO	537.15	3,969,644	0.9595	0.9726	0.2852	0.1812	0.0387	0.1245
15	16.0	MLP-CO	368.68	2,409,762	0.9625	0.9577	0.2888	0.1816	**0.0389**	0.1255
15	17.0	MLP-CO	259.86	1,669,210	**0.9725**	0.9516	**0.2904**	**0.1816**	0.0386	0.1258
15	18.0	MLP-CO	216.68	1,322,280	0.9675	0.9304	0.2899	0.1794	0.0386	**0.1259**
15	20.0	MLP-CO	186.21	974,019	0.9520	0.9195	0.2899	0.1772	0.0386	0.1249
15	25.0	MLP-CO	120.80	699,919	0.9175	0.9251	0.2828	0.1730	0.0376	0.1198
25	25.0	MLP	1023.92	10,998,628	0.9770	0.9876	0.2909	0.1812	0.0389	0.1267
25	25.0	MPL-CO	971.61	10,511,112	0.9770	0.9876	0.2914	0.1810	0.0389	0.1267
25	25.1	MLP-CO	920.53	9,687,611	0.9770	0.9875	0.2914	0.1813	0.0390	0.1270
25	25.5	MLP-CO	762.58	7,939,274	0.9805	**0.9880**	0.2914	0.1819	0.0391	0.1276
25	26.0	MLP-CO	677.71	6,472,689	0.9805	0.9811	0.2914	0.1823	0.0391	0.1278
25	28.0	MLP-CO	401.99	3,516,511	0.9830	0.9690	0.2929	0.1831	0.0392	0.1286
25	30.0	MLP-CO	332.96	2,749,935	**0.9860**	0.9616	**0.2924**	**0.1842**	**0.0393**	**0.1294**

[7] P. Boldi and S. Vigna, "MG4J at TREC 2005," in *The Fourteenth Text REtrieval Conf. Proceedings*, 2005.

[8] S. Büttcher and C. Clarke, "A Document-Centric Approach to Static Index Pruning in Text Retrieval Systems," in *Proc. of the 15th ACM CIKM*, 2006.

[9] S. Büttcher, C. L. A. Clarke, and P. C. K. Yeung, "Index Pruning and Result Reranking: Effects on Ad-Hoc Retrieval and Named Page Finding," in *Proc. of the Fifteenth Text REtrieval Conference*, 2006.

[10] R. Chen and C. Lee, "An Information-Theoretic Account of Static Index Pruning," in *Proc. of the 36th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2013.

[11] R.-C. Chen, C.-J. Lee, and W. B. Croft, "On Divergence Measures and Static Index Pruning," in *Proc. of the 2015 Int. Conf. on The Theory of Information Retrieval*. ACM, 2015, pp. 151–160.

[12] J. Cho and A. Ntoulas, "Pruning Policies for Two-Tiered Inverted Index with Correctness Guarantee," in *Proc. of the 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2007.

[13] E. S. de Moura, C. F. dos Santos, D. R. Fernandes, A. S. Silva, P. Calado, and M. A. Nascimento, "Improving Web Search Efficiency via a Locality Based Static Pruning Method," in *Proc. of the 14th Int. Conf. on World Wide Web*, 2005.

[14] R. Fagin, D. Carmel, D. Cohen, E. Farchi, M. Herscovici, Y. Maarek, and A. Soffer, "Static Index Pruning for Information Retrieval Systems," in *Proc. of the 24th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2001.

[15] W. Jiang, J. Rodriguez, and T. Suel, "Improved Methods for Static Index Pruning." in *IEEE Int. Conference on Big Data*, 2016, pp. 686–695.

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 3146–3154.

[17] H. Lam, R. Perego, and F. Silvestri, "On Using Query Logs for Static Index Pruning," in *Web Intelligence*, 2010.

[18] L. Nguyen, "Static Index Pruning for IR Systems: A Posting-Based Approach," in *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009, pp. 25–32.

[19] P. Raghavan and C. D. Tompson, "Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, Dec 1987.

[20] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates, "ResIn: A Combination of Results Caching and Index Pruning for High-Performance Web Search Engines," in *SIGIR conference on Research and development in information retrieval*, 2008.

[21] S. L. Thota and B. Carterette, "Within-Document Term-Based Index Pruning with Statistical Hypothesis Testing," in *Advances in Information Retrieval*, 2011.

[22] S. K. Vishwakarma, K. I. Lakhtaria, D. Bhatnagar, and A. K. Sharma, "An Efficient Approach for Inverted Index Pruning Based on Document Relevance," in *2014 Fourth Int. Conf. on Communication Systems and Network Technologies*, 2014, pp. 487–490.

[23] H. Wu and H. Fang, "Analytical Performance Modeling for Top-K Query Processing," in *Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management*, 2014, pp. 1619–1628.

[24] L. Zheng and I. J. Cox, "Entropy-Based Static Index Pruning," in *Advances in Information Retrieval*, Jan. 2009, no. 5478, pp. 713–718.