Document Reordering for Faster Intersection

Qi Wang
New York University
Computer Science & Engineering
NY, U.S.A
qiwang@nyu.edu

Torsten Suel
New York University
Computer Science & Engineering
NY, U.S.A
torsten.suel@nyu.edu

ABSTRACT

A lot of research has studied how to optimize inverted index structures in search engines through suitable reassignment of document identifiers. This approach was originally proposed to allow for better compression of the index, but subsequent work showed that it can also result in significant speed-ups for conjunctive queries and even certain types of disjunctive top-k algorithms. However, we do not have a good understanding of why this happens, and how we could directly optimize an index for query processing speed. As a result, existing techniques attempt to optimize for size, and treat speed increases as a welcome side-effect.

In this paper, we take an initial but important step towards understanding and modeling speed increases due to document reordering. We define the problem of minimizing the cost of queries given an inverted index and a query distribution, relate it to work on adaptive set intersection, and show that it is fundamentally different from that of minimizing compressed index size. We then propose a heuristic algorithm for finding a document reordering that minimizes query processing costs under suitable cost models. Our experiments show significant increases in the speed of intersections over state-of-the-art reordering techniques.

PVLDB Reference Format:
Qi Wang, Torsten Suel. Document Reordering for Faster Intersection. *PVLDB*, 12(5): 475-487, 2019.
DOI: https://doi.org/10.14778/3303753.3303755

1. INTRODUCTION

Large search engines use significant hardware and energy resources to process billions of user queries per day. This has motivated a large body of research that aims to reduce the cost of processing queries, including work on index compression, caching, optimized top-k query processing, index tiering and clustering, query routing and load-balancing, and cascading and other optimizations for complex rankers.

Search engines typically use inverted index structures to efficiently identify suitable candidate results for a query, which are then further analyzed and reranked using more complex rankers. Given the number of documents indexed by search engines, inverted indexes can reach sizes of many terabytes, distributed over thousands of machines via index sharding, and traversal of the index structures during query time is a significant cost factor for such engines. As a result, a lot of research has focused on techniques for better compression and faster traversal of inverted indexes.

In this paper, we focus on one specific approach that has been studied, *document reordering*, also called *document ID reassignment*. The basic idea, first proposed in [11, 39], is to assign document IDs (docIDs) to indexed documents such that consecutive docIDs are assigned to textually similar documents. The result is that the sequences of docIDs in the inverted lists that make up the index are becoming more bursty or skewed, with clusters of close-by or consecutive docIDs separated by larger docID gaps. In the extreme case, we might get large runs of consecutive documents that all contain a particular term, say pages from the same web site that all contain certain terms common to the site. It is well known that such bursty sequences of docIDs can be more succinctly compressed than in the case where docIDs are assigned to pages at random, and a number of different approaches for assigning docIDs to documents have been proposed.

It was observed in [48] that reordering not only decreases index size, but also substantially increases the speed of conjunctive queries. Subsequent work in [21, 24, 35, 42] extended this observation to important classes of disjunctive top-k algorithms, in particular WAND [13], MaxScore [45], and more recent Block Max-based approaches [16, 24, 42]. Thus, document reordering has the potential to significantly increase the efficiency of the candidate selection phase in search engines, and this benefit arguably outweighs the compression gains in many scenarios. However, we still have very limited understanding of why this happens (beyond some naive speculation, e.g., in [48]), and no techniques that would allow us to model this effect and to optimize the ordering directly for increased speed. Instead, reordering techniques typically focus on minimizing compressed size, and treat any speed gains as a welcome but somewhat mysterious side-effect.

In this paper, we address this challenge, and study the problem of computing a document reordering that minimizes query costs, for the case of conjunctive queries. We formally define the problem of reordering to minimize cost, given an index and a query distribution, and discuss its complexity. We also show that optimizing for cost is quite different from

Proceedings of the VLDB Endowment, Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303755

optimizing for size, and that there are scenarios where significant reductions in size give little or no reductions in cost, and vice versa. Finally, we propose a heuristic algorithm that searches for document reorderings that minimize query processing costs under certain cost models, and show that it achieves significant cost reductions over state-of-the-art reordering approaches.

The remainder of this paper is organized as follows. In the next section, we provide some technical background and discuss related work. Section 3 introduces our approach, defines the basic optimization problem, and discusses its complexity and relationship to the problem of minimizing size. Section 4 describes our algorithm for document reordering, while Sections 5 and 6 present the experimental setup and the results. Finally, Section 8 provides some concluding remarks.

2. BACKGROUND AND RELATED WORK

We now provide some necessary technical background and discuss the most closely related previous work.

2.1 Document Collections and Inverted Indexes

We are given a collection of n documents $C = \{D_0, D_2, \ldots, D_{n-1}\}$, where each document D_i is identified by its document ID (docID) i. An inverted index for C is a data structure that stores for each distinct term t in C the document IDs (docIDs) are all documents containing t. More precisely, an inverted index consists of inverted lists, one for each term t. Each inverted list is a sequence of postings, where each posting contains the docID of a document containing t, plus potentially other information such as the number of times t occurs in the document or the locations of these occurrences. The postings in each lists are stored in order of increasing docIDs. Inverted lists are typically stored in highly compressed form, and many fast and effective compression methods have been proposed in the literature. We refer to [49] for more details on inverted indexes and index compression.

2.2 Basic Query Processing and Intersections

Inverted indexes are used in most current search systems to efficiently process user queries. In a nutshell, we can think of query processing as happening in several stages, typically an initial Boolean filter based on conjunctions or disjunctions, followed by, or in some cases interleaved with, a first ranking of the documents passing the Boolean filter based on a fairly simple and fast ranking function such as BM25. Then several subsequent phases of reranking are performed where increasingly complex and costly ranking functions are applied to fewer and fewer of the top results output by the previous ranker, in order to obtain the final top-k results [46].

In this paper, we focus on the initial stage, and in particular on how to efficiently perform conjunctions, or intersections, between inverted lists. This is basically the problem of intersecting two or more sorted lists of integers, which has been studied extensively over several decades as a basic building block in many applications. It also continues to be an important operations in state-of-the-art search systems. In particular, see [25] for recent work on faster intersection processing, and [36] for background on how intersections are used inside the current Bing search architecture, which transforms an incoming user query into a set of intersections on subsets of the query terms. (Of course, other search systems may make different choices, and some might rely more on disjunctions rather than conjunctions in the initial phase.)

2.3 Intersection Algorithms

We first describe two basic algorithms that are widely used for intersecting k sorted lists of integers L_0 to L_{k-1}, and then discuss additional methods and optimizations. Assume that the lists have been arranged in order of ascending length, such that L_0 is the shortest and L_{k-1} is the longest list. We note that the problem of intersecting two lists of lengths l_0 and l_1 with $l_0 \leq l_1$ can be solved in time $O(l_0 \cdot \log(l_1/l_0))$, and there is a matching worst-case information-theoretic lower bound in the comparison-based model.

The SvS algorithm first intersects the two shortest lists L_0 and L_1, and then intersects the output with L_2, then that output with L_3, and so on. Thus, the algorithm reduces the problem to a sequence of pairwise intersections. The actual intersection is performed via forward seeks from the shorter into the longer list. Here, a forward seek is an operation in a sorted list that moves a pointer from its current location in the list forward to a requested value, or to the next larger value if it does not exist. Forward seeks can be implemented using a variety of techniques such as block-wise skipping, galloping, or even interpolation search in some cases [17, 28].

The standard *Document-At-A-Time* (DAAT) algorithm for intersection places a pointer at the beginning of each list. The lists are again sorted by length in ascending order for efficiency (i.e., L_0 is the shortest list). Then it fetches the next (first) element from L_0, and does a forward seek for a matching element in L_1. If this seek is successful, resulting in a matching item, the algorithm then forward seeks into L_2, L_3, and so on, until we either find a match in every list, or one of the seeks fails. When a seek fails, it usually returns the next larger item x from the list into which the seek was performed. We then go back to L_0 and seek the first item that is at least x, and then again try to perform forward seeks into L_1, L_2, etc on this item. The algorithm terminates when a forward seek reaches the end of a list.

While our goal here is to optimize intersections between k lists, our method focuses on optimizing pairwise intersections between lists L_0 and L_1. It is known that for typical queries and collections, most of the accesses in intersections occur in the two shortest lists, and thus one would expect that improving performance on these lists also improves performance on k lists. In fact, we present results in Subsection 6.1 that support this choice, by showing that most of the accesses occur in the two shortest lists, and in fact that DAAT and SvS mostly perform the exact same accesses into these lists. We also note that for the case of two lists, SvS and DAAT reduce to the same approach, also called zig-zag join in databases, where we alternate forward seeks into the two lists.[1]

The problem of intersecting sorted lists has been extensively studied, and is also related to that of merging sorted lists [26, 27]. Prior research has provided many optimization methods for intersection that can be roughly divided

[1]Though there are various ways to implement the forward seeks or lookups at the lower level.

into four categories: adaptive intersection, hierarchical intersection, hash-based intersection, and parallel intersection.

Adaptive algorithms [3, 4, 5, 6, 7, 8, 18, 19] try to adapt to a given instance of the intersection problem. This is in contrast to SvS and DAAT above, which usually drive the intersection from the shortest to the longest list. Adaptive algorithms often assume a comparison-based model, and try to achieve a number of comparison that is close to the minimum required for the particular instance. While such adaptive algorithms can do much better on highly skewed or bursty data, the adaptivity usually introduces other overheads that make the algorithms perform worse than DAAT and SvS on common data sets in information retrieval [19, 22]. Our work here is influenced by adaptive algorithms, and in particular [18], but is also different as our goal is to optimize for the more commonly used SvS and DAAT approaches rather than for adaptive methods.

Hierarchical approaches utilize data structures such as AVL trees [14], treaps [12] and skip-lists [28] to speed up the intersection. A two-level representation of posting lists was proposed in [37]. The lower level splits the range of docIDs into buckets based on their most significant bits, and the upper level provides the information leading from the most significant bits of a docID to the corresponding bucket in the lower level. This enables better compression for lower level data, while the upper level allows binary search over the buckets. In [44], a small partition table is built for the upper level, and the lower-level data is partitioned into small subsets consisting of uniformly distributed values, allowing interpolation search within subsets.

Hash-based representation can speed up intersections significantly when the lists have very skewed sizes. This is achieved by looking up all elements of the shorter lists in the hash tables of the much longer lists. In [9], Bille proposed a hash-based approach which first maps the elements in the original set to smaller values using some hash function. Then it computes the intersection using the hash values. Since the hash values use fewer bits than the original values, this can increase intersection speed. In the end, the original values in the intersection are recomputed, with "false positives" removed. Ding [22] provided a hash-based approach using a two-level representation. The input set is first partitioned into small subsets, and then the values in each subset are mapped to bitmaps that have the word-size of a machine processor. During the intersection, results are calculated using bit-wise AND instructions. This approach works especially well when the result set is much smaller than the input data.

Finally, parallel set intersection algorithms focus on exploiting multi-core architectures [43, 44], Graphical Processing Units (GPUs) [1, 2, 47], and SIMD instruction sets in modern CPUs [31, 38] to obtain further improvements in performance.

Our approach is influenced by the work of Demaine et al. [18, 19] on adaptive set intersection. In particular, [18] observes that different instances of the intersection problem have different complexity, and that adaptive algorithms can adjust to the complexity of the instance and thus perform much better on easy problem instances. This lead us to conjecture that existing reordering methods result in easier instances of the intersection problem, and that with a suitable model we might be able to directly optimize for this. However, our actual model is quite different and in fact simpler than the one in [18], since the goal of our reordering is to obtain instances on which the widely used DAAT and SvS algorithms run faster, rather than instances that are easier for adaptive algorithms.

2.4 Document Reordering Techniques

Recall that documents in the collection are identified by document IDs (docIDs), and that the postings in each inverted list are sorted by docIDs. Thus, by changing the assignment of docIDs to documents, we get different sequences of docIDs in the resulting inverted lists. This lead to the idea, first proposed in [11, 39] and later studied, e.g., in [10, 23, 20, 40, 41], of trying to find an assignment of docIDs to documents that improves the compressibility of the resulting inverted lists.

The basic idea in these *document reordering* or *docID reassignment* techniques is to assign consecutive or close-by docIDs to documents that have a lot of terms in common. This means that the sequence of integer docIDs in many of the inverted lists will be clustered, with some dense areas having many close-by docIDs, and other sparse areas with large gaps between docIDs. It is well known that such clustered sequences can be much better compressed than sequences resulting from random assignment of docIDs to documents, by using suitable compression techniques such as Interpolative Coding [33], OptPFD [48], or Partitioned Elias-Fano [34] that exploit this case.

Most of the proposed document reordering techniques fall into three classes: (1) heuristics that use ideas such as sorting by URL [40] or by size [29, 48] to assign docIDs, (2) bottom-Up approaches that connect similar documents with edges and then assign docIDs via a TSP-like traversal of the resulting graph [10, 23, 39], and (3) top-down techniques that cluster documents into smaller and smaller subsets based on similarity [10, 11, 20]. Of particular interest to us here are the recent Recursive Bisection (BP) algorithm in [20], which is the current state-of-the-art method in terms of compressed size, and the docID assignment via URL sorting proposed in [40], which is extremely simple while also obtaining very good compression on suitable data sets. Our own approach will adapt the BP algorithm to a new objective.

While document reordering was originally proposed with the goal of achieving better compression, it was observed in [48] that is can also significantly increase intersection speed, by reducing the number of forward seeks performed. Subsequently, a number of studies including [24, 32, 35] observed speed improvements not just for intersection, but also for several disjunctive top-k query processing algorithms including WAND [13], MaxScore [45], and BMW [24].

However, while many researchers have obtained speed increases with existing reordering schemes originally designed for better compression, we currently do not have a good model for why reordering increases speed, and we have no methods that can directly optimize the ordering for maximum speedup. The main goal in this paper is to make an initial step in this direction for the case of intersection, by proposing a framework for understanding and modeling speed increases, and by proposing an optimization algorithm that attempts to find a reordering that maximizes speed under a simple model. To the best of our knowledge, no previous work has attempted to directly optimize intersection speed via document reordering.

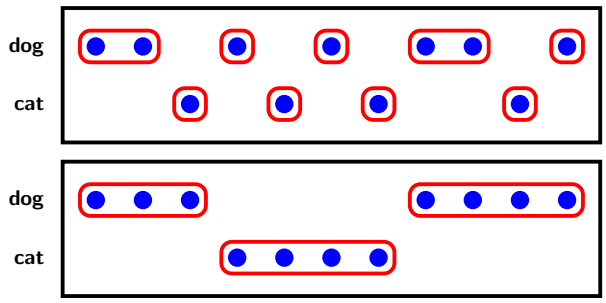

Figure 1: Illustration of an easier (bottom) and harder (top) instance of the intersection problem between two term lists for *dog* and *cat*.

3. REORDERING FOR INTERSECTION

In this section, we describe the basic idea underlying our approach, define the resulting optimization problem, and discuss how it is fundamentally different from that of reordering for improved compression. Finally, we show that the problem is NP-Hard, via a reduction to the Minimum Feedback Arc problem.

3.1 The Basic Idea

Recall that the problem of intersecting two sorted, monotonically increasing lists of integers of lengths l_1 and l_2 with $l_1 <= l_2$ has a worst-case complexity of $O(l_1 \cdot \log(l_2/l_1))$. However, as shown in [18], some instances of the intersection problem are easier than others, and there are fairly simple algorithms that can adapt to the complexity of a particular instance [19]. We illustrate this difference in complexity between instances in Figure 1, where we have an easy instance with longer runs of dog and cat postings, and a harder instance where postings from the two lists are very finely interleaved.

This leads to the following natural questions: (1) Do the document reordering techniques that have been proposed for improving the compression of inverted indexes also result in simpler instances of the intersection problem for commonly occurring queries? (2) Can we design new reordering techniques that directly optimize for intersection speed rather than compression, by minimizing the complexity of the resulting intersection instances for common queries?

3.2 A More Formal Approach

In order to address these questions, we need to define a suitable model of the complexity of an intersection instance, simple enough to be used for optimization, but with a reasonable correspondence to the actual running time of commonly used algorithms on the instance. Note that the approach taken in [18] to characterize the complexity of an instance, based on the encoding sizes of *intersection proofs*, does not seem to be useful here, because it seems too complex to optimize for, and because it optimizes for a different objective, namely, the running time of adaptive algorithms. Its complexity is partly due to its use of a comparison-based model, and due to its modeling of k-wise intersection, which is significantly more complex than the pairwise case. Instead, we define a much simpler notion, which we call *run complexity*, that formalizes the above intuition of "runs" for pairwise (two-term) intersection instances. In the following,

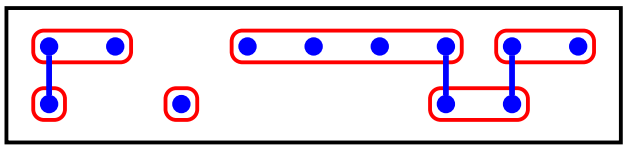

Figure 2: An intersection problem with 6 runs, where there is an intersection of size 3.

we use the term substring of L to refer to a sequence of consecutive elements in a sorted list of integers L, and use $min(s)$ and $max(s)$ to refer to the minimum and maximum element in a substring s.

Definition: Run Complexity. Given a pairwise intersection instance I consisting of two sorted lists of integers L_0 and L_1, we say that two substrings s_0 in L_0 and s_1 in L_1 are non-excluding if either $max(s_1) \leq min(s_2)$ or $max(s_2) \leq min(s_1)$. A legal partitioning P of L_0 and L_1 is a set of substrings such that every element in I is in exactly one substring in P, and any two substrings in P from different lists are non-excluding. We define the *run complexity* of I as the size of the smallest legal partitioning, and refer to its substrings as runs. **End of Definition**

Under this definition, the instances in Figure 1 have run complexities of 9 (top) and 3 (bottom). In fact, for two disjoint lists, the notion of runs and run complexity is quite obvious. However, the situation is somewhat more subtle for instances with non-empty intersection, requiring the somewhat complex definition above. In Figure 2 we show a more complex example with a legal partitioning of size 6, where we have an intersection size of 3 between the lists. In fact, 6 is the size of the smallest legal partitioning for this instances, and thus its run complexity. We also note that this definition is similar to the notion of an alternation of Barbay and Kenyon [5].

We now make two claims about this definition, and briefly sketch their proofs.

Claim 1: Given an intersection instance, we can compute a legal partitioning of minimum size in time linear in the instance size, i.e., the number of items.

For the proof, note that for any pair of items i_1 and i_2 from different lists such that $i_1 < i_2$ and there is no other item j with $i_1 \leq j \leq i_2$, it is obvious that there is a run ending with i_1 and another run in the other list starting with i_2. Thus, after applying these cuts, the remaining question is how to cut around items that are in the intersection. Thus, assume we have a maximal sequence S of $k \geq 1$ consecutive items in both lists that are in the intersection, and let j be the last item before the first item in S. If this item exists, it is unique as it is not in the intersection. We now greedily build runs by adding to the run containing j the next item in the same list, then creating a run from the next two items in the other list, then the next two items from the first list, and so on, until we get to the end of S (if necessary mopping up a singleton with its own run at the end, or if possible extending the last run beyond S). If there is no j preceding S, we can build greedily from the end of S, using the first element following S, if such an element exists. If neither exists, then all items are in the intersection, and we start with a singleton run followed by greedily alternating as before, followed by a final singleton.

Claim 2: Given an intersection instance with run complexity r and an intersection size of c, let N_{fs} be the number of forward seeks made by the DAAT algorithm for intersection on this instance. Then the following holds: $r + c \leq N_{fs} \leq r + c + 1$.

To prove this, assume runs have been created with the process described above. Then the zig-zag algorithm will do forward seeks to the first item in every run, with the only exception being the very first run. This first run may be bypassed if the algorithm starts by first seeking in the other list, provided this first run does not contain any element in the intersection. In addition, if a run contains an item in the intersection as its last element, this item also has to be visited, since all items in the intersection are visited by the algorithm. The greedy construction used for Claim 1 creates exactly c runs where the last item is in the intersection (or $c - 1$ in the case where all items are in the intersection, in which case however the first run is not bypassed). In addition, we assume one extra seek to an end-of-list sentinel at the end of one of the two lists.

The importance of Claim 2 is that it shows that the run complexity can be used as a proxy for minimizing the number of forward seeks needed. This is because the size c of the intersection is obviously not affected by the ordering. It was shown in previous work starting with [48] that the reduction in the number of forward seeks seen after reordering was in turn roughly proportional to the observed reduction in running time. We now formally set up the problem of computing an ordering of the documents in a collection as an optimization problem where we minimize the number of expected runs for a random query.

Definition: Document Reordering for Faster Intersection. Given a collection of documents C and a probability distribution over all 2-term queries P, the goal is to compute an ordering of the documents that minimizes the expected number of runs given a query sampled from P, i.e.,

$$\min_{\pi \in \Pi} \sum_{t_1, t_2 \in L} P(t_1, t_2) \cdot runs(t_1, t_2, \pi)$$

where Π is the set of all permutations of the documents, L is the set of all terms occurring in the collection, and $runs(t_1, t_2, \pi)$ is the run complexity created by permutation π on terms t_1 and t_2. **End of Definition**

Note that in this definition, we assume two-term queries, for two reasons. First, the case of more than two terms is more complex, as seen when looking at the approach in [18]. Second, it has been shown [37] that in practice, the running time of an intersection of more than two lists is dominated by the time to intersect the two shortest lists. We will revisit this claim in Subsection 6.1, where we provide additional support for our approach. The distribution P in the definition assigns to each pair of terms a probability of occurring in a (memoryless) stream of two-term queries. In practice, P will be based on a suitable language model derived from a set of training queries and other data, where training queries are first pruned to keep only the two terms with the shortest lists. Thus, while our optimization only looks at the two shortest lists of a query, the approach will also yield improvements for longer queries under SvS and DAAT approaches, as we will show later on real testing data.

As discussed above, we believe that the number of forward seeks that is the goal of our optimization is in fact a decent proxy for running time. In contrast, [18] considers a comparison-based model that is more complex and we believe less predictive of actual running time. Moreover, it is not difficult to modify our optimization to take into account the length of a forward seek, as discussed later. In this case, longer seeks could be modeled as having a higher cost, say logarithmic to account for galloping, or other terms to account for block-wise decompression effects.

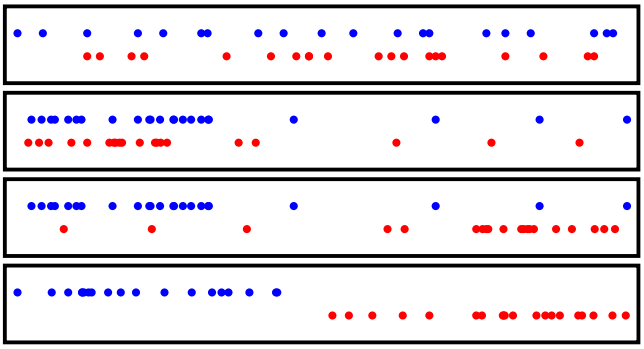

Figure 3: Two inverted lists, in blue and red, under a random order, and under three different document reorderings.

3.3 Optimizing For Size Versus Speed

As mentioned, previous work focused on reordering to decrease compressed size, and treated speed improvements as a welcome side effect. However, given that previous techniques obtain significant improvements in both size and speed, one could argue that maybe the two objectives are closely related. In this subsection, we show that they are in fact fundamentally different problems. More formally, we show that we can construct cases where reordering greatly improves both size and speed, other cases where we get great improvements in size but no improvements in speed, and cases where we get great improvements in speed but very little in size.

Our constructions are illustrated in Figure 3, which shows two inverted lists, in red and in blue, under four different orderings: a random document ordering, an ordering where both lists are heavily clustered on the left with few postings on the right, an ordering where one list is clustered on the left and the other one on the right, and an ordering where one list is randomly spread out in the left half and the other in the right half (from top to bottom). For simplicity, we show only two lists, and assume that while the collection may have many other terms, the query distribution P is dominated by this pair, which has an empty intersection. More complex scenarios could be built with non-empty intersections and more diverse query distributions.

Note that compared to the random case on the top, the second case results in significant size reductions for both lists, as postings are heavily clustered in docID space. However, the number of runs, and thus forward seeks, is not decreased at all. The third case shows reductions in size, due to both lists being clustered in docID space, and also in speed as the number of runs is significantly reduced, since both lists are clustered in different areas. Finally, the case at the bottom shows only two runs, but size decrease is lim-

ited to about one bit per posting over the random case, as average gaps are decreased by only a factor of two.

These cases show that optimizing for size and speed are fundamentally different problems. To optimize for size, we need to cluster each inverted list in docID space, but it does not matter how other lists are clustered as each list is compressed by itself. To optimize for speed, on the other hand, we have to look at pairs of common query terms, and the goal is to separate the red and blue postings into largely disjoint clusters. Nonetheless, previous results indicate that existing techniques seem to be fairly successful at achieving both goals, and thus might be closer to the third case from the top. For example, under URL sorting, we could imagine that there are some web domains that are about the red term and do not contain many occurrences of the blue term, or vice versa, while some domains are about both terms. In this case, we would see fewer runs and thus increased speeds while traversing the former, more common but less useful, domains, but not while in the latter domains. Of course, our goal here is do even better in terms of speed by designing techniques specifically geared towards this goal.

3.4 Problem Complexity

We now show that our problem of finding an document ordering that minimizes the expected intersection cost, given a document collection and query distribution, is NP-hard. Given the known hardness results for the problem of reordering documents to minimize compressed size shown in [20], this is not surprising. However, while those results perform a reduction from Minimum Linear Arrangement and related problems, we use a reduction from Minimum Feedback Arc Set, reflecting the fact that our problem, as discussed above, is fundamentally based on pairs of term, rather than optimizing a measure for each term and inverted list. Formally, we have:

Theorem: Given a collection of documents C and a distribution over all 2-term queries P, it is NP-Hard to compute an ordering of the documents that minimizes the expected number of runs for a query sampled from P.

Proof: Given a directed graph G, the Minimum Feedback Arc Problem is the problem of ordering the vertices in G to minimize the number of back edges (feedback arcs), i.e., edges going from a later to and earlier vertex. Given a directed graph $G = (V, E)$, we now show how to construct a document collection C and a query distribution P, such that solving our document ordering problem also provides an optimum solution to the Minimum Feedback Arc Problem on G.

For each vertex v in V, we create a corresponding document D_v in C. Moreover, for each edge $e_i = (u, v)$ in E, we generate two new terms, $from_i$ and to_i, and add $from_i$ to D_u and to_i to D_v. Moreover, we create two additional documents S and T, and four additional terms s_1, s_2, t_1, and t_2. We then add s_1 and all of the $from_i$ to S, and t_2 and all of the to_i to T. In addition, we add s_2 and t_1 to all nodes D_v where $v \in V$.

Given this document collection, we define our query distribution. We have $|E|$ queries $(to_i, from_i)$ with frequency $0.2/|E|$ each, and queries (s_1, s_2) and (t_1, t_2) with frequency 0.4 each.

Now consider a document ordering where we start with S and end with T, and where the documents D_v in between S and T are ordered according to an optimal solution of the Minimum Feedback Arc problem on the graph G. Let B be the total number of backward edges in the optimal solution for G, which is at most $|E|/2$ as otherwise we could improve the ordering by reversing it. We will use a series of observations to show that all optimal solutions are of this form, or are the reverse of this form.

First, we observe that the expected number of runs for a random query drawn from P under this ordering is $2 * P(s_1, s_2) + 2 * P(t_1, t_2) + (2 + 2 * B/|E|) \sum_i P(from_i, to_i)$, which is $2 + 0.4 * B/|E| <= 2.2$. This is because any query (s_1, s_2) or (t_1, t_2) results in 2 runs, while a query $(from_i, to_i)$ results in 2 runs if edge e_i is a forward edge, and 4 runs otherwise.

Second, we can assume that any optimal solution starts with S and ends with T. This is because any solution where S is arranged in between the documents D_v would result in 3 instead of 2 runs for any occurrence of query (s_1, s_2), resulting in at least 2.4 expected runs per query overall; the case of T is symmetric. Thus, S and T must be at the beginning or end. Moreover, if at least half of the edges e_i are forward edges in the document ordering, then S should be at the beginning and T at the end; if less than half, we can reverse the ordering without changing the number of runs, making this also true.

Thus, we can assume an optimal ordering starts with S, and ends with T, with some arrangement of the documents D_v in between. Moreover, the precise ordering of the D_v between S and T does not impact the number of runs created by queries of the form (s_1, s_2) and (t_1, t_2), while the number of runs due to queries of the form $(from_i, to_i)$ is directly proportional to the number of backward edges e_i in the ordering of the D_v, and thus the best ordering of the D_v would directly provide an optimal solution to the Minimum Feedback Arc problem on G. **End of Proof**

We note that the Minimum Feedback Arc problem is in fact APX-Hard, and hard to approximate to any factor smaller than 1.3606 (or 2 if the Unique Game Conjecture is true) [30]. However, our reduction here does not preserve any approximability results. We conjecture that a more careful construction might be able to establish APX-Hardness of our ordering problem.

4. ALGORITHMIC APPROACH

In this section, we describe a heuristic algorithm for solving the optimization problem defined in the previous section. The algorithm is based on an adaptation of the recent algorithm for reordering based on recursive bisection proposed in [20], which was shown to outperform previous methods for both document reordering and graph reordering in terms of compressed size. We will show how to change the partitioning criterium to focus on decreasing the number of runs.

4.1 The Recursive Bisection Approach (BP)

We now briefly outline the approach in [20], called *BP*, for the case of document reordering. The reader may want to consult [20] for more details. In a nutshell, the algorithm starts from any initial ordering of the document collection, and then recursively bisects the collection into smaller and smaller subsets, as follows:

1. If the subset has less than C documents, sort them by URL and return.
2. Split the subset into a left half and a right half.

3. Repeat the following 20 times or until converged:

   a. For each document, estimate how moving it to the other half of the subset would impact compressed size. That is, if we were to move this document over, in the process increasing the frequencies of the document's terms in the other half and decreasing them in the current half, would this decrease or increase the expected overall compressed index size for both halves, assuming a random ordering of documents in each half and a log-based cost model for the expected resulting docID gaps.

   b. In each half, sort the documents in decreasing order of moving benefit.

   c. Now swap the highest scoring document in the left half with the highest scoring document in the right half, then the second highest, etc., until the benefit of a swap becomes negative.

4. Recurse on the left half and on the right half.

The repetition in Step 3 is needed because when we estimate the benefit of moving a document in 3a, this under the assumption that only this document is moved. However, in 3c, we then move a large number of documents, which impacts the estimated benefits in the next operation, so that we may now want to move additional documents, or may want to move some documents back. As reported in [20], after about 20 iterations this process is close to converging in that very few documents still want to move. Also, once a subset size of about 20 or less is reached, there is little benefit in recursing further, and we may just sort the subset by URL (or use some other heuristic) and return that ordering.

In addition to achieving very small compressed sizes, the BP algorithm is also very fast, with tens of millions of documents being processed in less than one hour under a highly optimized implementation. One reason for this is that Step 3a can be performed highly efficiently by first computing for each term the benefit of moving one posting of the term to the other side, and then for each document summing up these precomputed values for all its terms.

4.2 Our Adaptation: BP-RUN

In order to use the Recursive Bisection approach, we mainly need to modify Step 3a, by using a different definition of the benefit of moving a document to the other half. As we saw in Subsection 3.3, when optimizing for size we look at each inverted list by itself and try to improve its compressibility. In the context of Step 3a, this means trying to make each inverted list as imbalanced as possible between the two halves, by putting most postings into one half. However, when optimizing for speed, our goal is to separate out common pairs of terms, by putting most postings of one term into one half, and most postings of the other term into the other half.

Now consider two terms t_1 and t_2, where there are f_1 postings of t_1 and f_2 postings of t_2 in a particular subset. We can now estimate the expected number of runs between t_1 and t_2, assuming as before that documents are ordered at random in that subset, and that the intersection is empty (or at least much smaller than the two lists). A posting of t_1 is the start of a run if it is not the first among the f_1+f_2 items, and if the next smaller item is term t_2 rather than t_1. Under a random ordering, this probability is $f_2/(f_1+f_2)$, and thus the expected number of runs between t_1 and t_2 is estimated as $ER(f_1, f_2) = (f_1 \cdot f_2)/(f_1 + f_2) + (f_2 \cdot f_1)/(f_2 + f_1) = 2 \cdot f_1 \cdot f_2/(f_2+f_1)$. Note that this estimate ignores "boundary effects" between different subsets and halves, and assumes that runs can only start inside a subset. Thus, it is a rough estimate for the number of runs generated under a random ordering.

Algorithm 1 BP-RUN
Input: document set D, term lexicon T_1 and T_2
Output: reordered document set D

```
 1: procedure PARTITION(D, T1, T2)
 2:     if size of D < threshold C then
 3:         FinalizeAndOutput(D)
 4:         return
 5:     equally split D into D1 and D2
 6:     repeat
 7:         reset T1 and T2
 8:         for d in D1 do
 9:             for t in d do
10:                 T1 ← IncreaseCount(t, T1)
11:         for d in D2 do
12:             for t in d do
13:                 T2 ← IncreaseCount(t, T2)
14:         for d in D do
15:             gains[d] ← GetMoveGain(d, T1, T2)
16:         D1 ← sorted D1 in descending order of gains
17:         D2 ← sorted D2 in descending order of gains
18:         for d1 in D1, d2 in D2 do
19:             if gains[d1] + gains[d2] > 0 then
20:                 swap d1 and d2
21:             else
22:                 break
23:     until converged or more than 20 iterations
24:     Partition(D1, T1, T2)
25:     Partition(D2, T1, T2)

26: procedure GETMOVEGAIN(d, T1, T2)
27:     gain ← 0
28:     for t1 in d do
29:         for t2 not in d and prob(t1, t2) > threshold do
30:             l1 ← GetCount(t1, T1)
31:             l2 ← GetCount(t2, T1)
32:             r1 ← GetCount(t1, T2)
33:             r2 ← GetCount(t2, T2)
34:             gain += prob(t1, t2) · B(l1, l2, r1, r2)
        return gain
```

Given the formula above, we can now estimate the benefit of moving a document D from one half to the other, as follows: First, for any pair of terms t_1 and t_2 that do not occur in D at all, moving D has no impact on the number of runs on t_1 and t_2. However, even if only t_1 occurs in D, this may have an impact on the number of runs. Thus, if we start out with l_1 and l_2 postings of t_1 and t_2 in the left half, and r_1 and r_2 postings of t_1 and t_2 in the right half, then moving a document containing t_1 but not t_2 from left to right half has a benefit of $B(l_1, l_2, r_1, r_2) = ER(l_1, l_2) + ER(r_1, r_2) - ER(l_1-1, t_2) - ER(r_1+1, r_2)$. For the case that a document containing both t_1 and t_2, no matter which side this document is in, there will be an unavoidable comparison

between this query pair. Besides, this case is much rarer compared to the the case when the document contains only t_1 and in practice it has negligible impact on the final result. For efficiency reason, we treat this case the same as the document containing only t_1.

As in the case of [20], we can precompute the benefit of moving one posting of t_1 from, say, the left half to the right half, as follows:

$$Ben(l_1, r_1) = \sum_{t_2 \neq t_1} P(t_1, t_2) \cdot B(l_1, l_2, r_1, r_2),$$

where l_2 and r_2 are the number of postings of t_2 in the left and right half. The case of moving from right to left is symmetric. As before, the benefit of moving a document can be computed as the sum of benefits of moving all its terms. There is however a performance problem with the above formula, as we cannot afford to loop over all $t_2 \neq t_1$ in the computation of $Ben(l_1, r_1)$, as we have millions of distinct terms. Instead, we loop only over a limited number of terms t_2 such that $P(t_1, t_2)$ is above a certain threshold under the distribution P derived from a training query set. In Algorithm 1, we present pseudo-code for the BP-RUN algorithm.

4.3 Further Enhancements

We now describe a few more enhancements to the above algorithm that we explored, as follows:

Splashback: We found that when directly using the formula for $B(l_1, l_2, r_1, r_2)$ above, in many scenarios the algorithm never got close to converging in 20 or even more iterations, as many documents would be repeatedly moved back and forth between halves. The problem seemed to be that the formula assumes that we move one posting of t_1 to the other half. However, in Step 3c we actually swap our document with a document going the other direction, and there is a chance that this document also contains a t_1. In the extreme case when all documents in the other half already contain t_1, this results in an infinite loop where we try to add more occurrences of t_1 to that side, but each time another document containing t_1 is returned. We decided to adjust the formula by assuming that the document moving in the other direction is chosen at random, and thus has a r_1/n_r chance of containing t_1, where n_r is the number of documents in the right half. Thus, we change the definition of $B(l_1, l_2, r_1, r_2)$ to $ER(l_1, l_2) + ER(r_1, r_2) - ER(t_1 - x, t_2) - ER(r_1 + x, r_2)$ where $x = 1 - r_1/n_r$. This idea of moving a fractional posting seemed odd at first, but resulted in all instances getting close to convergence in 20 iterations or less, while also improving the resulting number of runs. Thus, we used this approach in all subsequent experiments. The same idea is also potentially applicable in the original scenario in [20], but in tests we did not see any improvements for it.

Considering Boundaries: Recall that our formula $ER(f_1, f_2)$ for the expected number of runs did not try to model the case of a new run starting with the first item of a half, as this would require information about the items preceding this one. However, we can extend $ER()$ with two extra cases: (a) when looking at the first item in the left half, we have already fixed the ordering to the left, and thus know whether the preceding posting was t_1 or t_2; (b) for the first item in the right half, we can look at the frequencies of t_1 and t_2 in the left half to estimate how likely it is that the last item in the left half was t_1 or t_2. Thus, a more complex formula can estimate the expected number of runs including runs that start at boundaries. We refer to this version of the algorithm as BP-RUN-CB. As we show later, this version led to very slight overall improvements.

Neighbor Swapping: We added another optimization after running the bisection algorithm, where we perform several rounds of neighbor swaps, where in odd (even) rounds we consider swapping documents in positions $2i$ and $2i+1$ ($2i-1$ and $2i$), respectively. That is, we swap documents if this would reduce the expected number of runs. This version of the algorithm, named BP-RUN-CB-NS, resulted in additional consistent but fairly limited improvements, as shown later.

Weighted Cost Models: Finally, instead of treating each run, and thus each corresponding forward seek, as having unit cost, we can apply a cost function that depends on the distance traveled by a seek. For example, if galloping is used during forward seeks, then the cost can be modeled as logarithmic in the distance. Or if index decompression is done in blocks of B postings, then for seeks of distance $d < B$ the likelihood of having to decompress a new block as part of the seek could be estimated as d/B. The existing BP-RUN approach can fairly easily accommodate such cost models, by estimating in Step 3a not just the expected number of runs, but also their expected lengths.

5. EXPERIMENTAL SETUP

In this section, we discuss details about the experimental setup.

Document Collections: Our experiments were conducted on two widely used benchmark data sets:

- **Gov2** is the TREC 2004 Terabyte Track test collection, which consists of about 25 million pages from the `gov` top level domain crawled in early 2004.
- **ClueWeb09B** is the ClueWeb 2009 TREC Category B test collection, which consists of about 50 million English web pages crawled between January and February 2009.

For each document in the collection, body text was extracted using Indri[2], and terms were lower-cased and stemmed using the Porter2 stemmer; no stopwords were removed. The basic statistics for the two collections are reported in Table 1.

Query Sets: We use the TREC 2006 Terabyte Query Track queries (Trec06) for experiments on Gov2, and the TREC 2009 Million Query Track queries (Million09) on Clueweb09B.

- **Trec06** consists of 100k queries. This query set was collected in 2006 specifically for use with Gov2.
- **Million09** consists of 40k queries. This query set was collected in 2009 for use on ClueWeb09B.

We randomly sampled two sets of 20k queries each from Trec06 and Million09, drawing only queries with at least two terms where all terms are present in the collection dictionary. We randomly selected 1k (5%) of the queries for testing and the rest (95%) for training. To obtain the language model P for two-term queries, we use standard Language

[2]http://lemurproject.org/indri.php

Table 1: Basic statistics of the two test collections.

	Gov2	ClueWeb09B
Documents	25,205,179	50,220,423
Terms	39,177,863	87,136,498
Postings	5,673,089,220	15,769,928,283

Modeling tools, in particular the MIT Language Modeling (MITLM) toolkit[3], based on Kneser-Ney smoothing. Before training the model, we kept for each training query only the two terms with shortest list length. We then thresholded the query distribution by only using pairs with probability at least 10^{-6} according to the language model during Step 3a of the reordering.

Algorithms: We implemented several existing ordering methods as baselines, in particular random ordering (Random), ordering based on URL sorting (URL) [40], ordering based on recursive bisection (BP) [20], and a version of BP that weighs terms using a uni-gram query language model (BP-LM), to show that simply adding term weighting to the existing approach does not give significant improvements.

For BP and BP-LM, we performed the recursive bisection by considering all terms in the collection. We stopped the recursion when the number of documents in the subset is no more than 12, since this gave us the best performance. For BP-LM, we weighted the estimated benefits in compression during Step 3a using the uni-gram weights, thus giving more weight to terms frequently used in queries. For the two BP algorithms, we started with URL ordering and also sort each subset according to URLs at the lowest level of the recursion.

To report numbers for compression, we used several methods, and include numbers for Partitioned Elias-Fano (PEF) [34], OptPFD [48], and Binary Interpolative Coding [33] (BIC). For evaluating query processing speed, we report results for algorithms for DAAT-type intersection, top-10 WAND, and top-10 Maxscore, using OptPFD with block size 128. Relative behaviors on PEF were very similar. We use BM25 as the ranking function in WAND and Maxscore.

Testing details: The algorithms were implemented using C++11 and compiled using gcc with -O3 optimization. We did not use any SIMD instructions or special processor features. The experiments were conducted on a single core of a 2.5Ghz Intel Xeon Platinum CPU, with 64GiB RAM, running Ubuntu 16.04. All data structures and indexes are memory-resident.

6. EXPERIMENTAL EVALUATION

We now present and discuss our experimental results, where we compare the various methods based on the number of forward seeks, query processing speed, and compressed index size. We start out with the running times of the new method. Our BP-RUN-CB algorithm took about 8 hours to run on Gov2, and 17 hours on ClueWeb09B, using a single thread. The running time could be significantly improved with multiple threads and distributed implementations, as reported in [20].

6.1 Two-term and Multi-term Intersections

We start with a set of preliminary experiments on SvS and DAAT to support our claim in Section 3.2 that multi-term intersection times are dominated by the running times on the two shortest lists, and that optimizing for the shortest lists can benefit multi-term intersection under both SvS and DAAT intersection algorithms.

In these experiments, we ran experiments on both Gov2 and Clueweb09B under URL ordering, using for each query length 1000 queries selected from Trec06 and Million09, respectively. First, in Tables 2 and 3, we show the average ratio of forward seeks that occur in the two shortest lists, for SvS and DAAT.

Table 2: Fraction of forward seeks in the two shortest lists under SvS, for different query lengths.

Query length	Avg.	2	3	4	5	≥6
Gov2	0.8937	1.0000	0.9050	0.8718	0.8494	0.8421
Clueweb09B	0.9187	1.0000	0.9194	0.8964	0.8893	0.8792

Table 3: Fraction of forward seeks in the two shortest lists under DAAT, for different query lengths.

Query length	Avg.	2	3	4	5	≥6
Gov2	0.8750	1.0000	0.8909	0.8519	0.8201	0.8119
Clueweb09B	0.9064	1.0000	0.9124	0.8881	0.8726	0.8591

For two-term queries, the ratio is obviously 1. However, for longer queries we still find that most of the work is done in the shortest lists; even for queries with 6 and more terms, more than 80% of the accesses are into the two shortest lists. The average overall ratio for queries of all lengths on both Gov2 and Clueweb09B is more than 0.87 and 0.90, respectively. The ratio on Gov2 is lower than on Clueweb09B; we conjecture that this is because Gov2 is known to be more clustered under URL ordering than Clueweb09B and in fact contains many page duplicates and near-duplicates, resulting in more forward seeks performed outside the two shortest lists. We can see from the results that the majority of the forward seeks happen within the two shortest lists for multi-term queries. This is well known for SvS [37] but also not surprising for DAAT, which also prioritizes accesses into shorter lists.

In fact, we next show something much stronger for DAAT: not only are most of its accesses into the two shortest lists, but in fact most of these accesses are exactly the same accesses as performed in SvS on those lists! That is, they are forward seeks to the same document ID, starting from the same current document ID, as in the case of SvS.

Table 4: Fraction of total forward seeks under DAAT that also occur when intersecting only the two shortest lists, for different query lengths.

Query length	Avg.	2	3	4	5	≥6
Gov2	0.8305	1.0000	0.8482	0.7879	0.7639	0.7523
Clueweb09B	0.8640	1.0000	0.8714	0.8332	0.8133	0.8019

As we see in Table 4,the numbers are almost as high as in Table 3. Thus, SvS and DAAT in fact perform many of the exact same accesses, and mainly differ in the order in which they are performed, with SvS doing a list-wise traversal. This provides strong support for our claim that optimizing for the two shortest lists should also result in improvements for multi-term intersections with DAAT.

[3]https://github.com/mitlm/mitlm

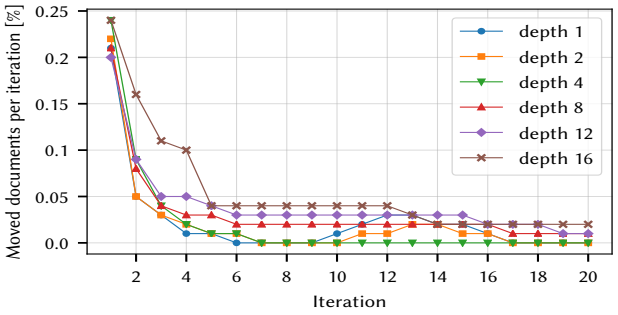

Figure 4: Convergence behavior of BP-RUN-CB: we show the average percentage of documents that are swapped between halves in each iteration, for different depths of the recursion, on Gov2.

6.2 Convergence Behavior

Figure 4 shows that the average percentage of swapped documents decreases quickly in the first several iterations of Step 3 in our method BP-RUN-CB. The smaller depths, on larger subsets, converge faster than the larger depths, on smaller subsets. Note that the method often does not fully converge, and in fact there are cases where documents end up getting swapped forever. However, there are no improvements beyond 20 iterations, so we stop at that point. We also refer to Figure 5 in [20] for similar observations on BP.

6.3 Forward Seeks

In Table 5, we show how BP-RUN and BP-RUN-CB perform in terms of the number of forward seeks, on two-term queries obtained from the test queries by keeping only the two terms with the shortest inverted lists. For both Gov2 and ClueWeb09B, we observe that the best setting is to recurse up to a minimum subset size of 12. We also see that BP-RUN-CB usually outperforms BP-RUN, but only by a very small amount.

Table 5: Average forward seeks per query on two-term evaluation queries, for Trec06 on Gov2 and Million09 on Clueweb09B, and for various thresholds on where we stop the recursion.

	Gov2		ClueWeb09B	
	BP-RUN	BP-RUN-CB	BP-RUN	BP-RUN-CB
192	182,886	182,557	1,006,848	1,002,666
96	182,161	181,573	998,336	994,725
48	181,259	180,888	991,938	987,274
24	180,650	180,429	987,131	984,215
12	180,295	**180,279**	984,783	**983,722**
6	180,484	180,724	987,444	986,912

Table 6: Average forward seeks per query on two-term queries, with minimum subset sizes 12 and 24, and using between 1 and 16 rounds of neighbor swapping at the end, on Gov2.

iterations	1	2	4	8	16
BP-RUN-CB-NS 24	180,068	179,706	179,596	179,401	**179,330**
BP-RUN-CB-NS 12	179,899	179,520	179,204	179,032	**179,001**

In Table 6, we see that the neighbor swapping technique mentioned in Section 4.3 gives only very limited improvements, about a 0.5% reduction in forward seeks after 16 rounds. Given that neighbor swapping slows down reordering significantly, we decided to not use it in subsequent experiments.

Next, in Table 7, we show the number of forward seeks on two-term queries for all different reordering methods. We see that BP-RUN-CB clearly outperforms URL, BP, and the term-weighted method BP-LM. In particular, it outperforms BP, the best previous method, by 19.8% on Gov2 and 14.3% on ClueWeb09B. BP-LM is only slightly better than BP, showing that merely adding a language model to the existing approach is not sufficient.

Table 7: Average forward seeks per query on two-term queries, for Gov2 and ClueWeb09B, under various document orderings.

	Random	URL	BP	BP-LM	BP-RUN-CB
Gov2	590,519	236,421	224,684	220,190	**180,279**
CW09B	2,089,627	1,180,164	1,148,409	1,117,379	**983,722**

6.4 Query Processing Speed

Table 8: Times in ms for two-term queries on Gov2 and ClueWeb09B, for DAAT-intersection, and for WAND and Maxscore with $k = 10$.

	Gov2			ClueWeb09B		
	Inter	WAND	Maxscore	Inter	WAND	Maxscore
Random	15.44	23.09	17.53	54.61	65.47	55.42
URL	6.04	8.61	10.22	35.01	50.62	40.43
BP	5.59	9.54	10.50	31.82	42.93	35.71
BP-LM	5.42	**8.43**	**10.01**	30.68	40.79	**34.72**
BP-RUN-CB	**4.55**	9.26	11.10	**26.17**	**37.65**	35.70

We now look at actual query processing speeds, rather than just counting forward seeks. Table 8 and Table 9 report query processing times in milliseconds on Gov2 and ClueWeb09B under various reorderings. Table 8 shows running times for just the two terms in each query with the shortest list lengths, while Table 9 shows the running times for the full queries. We note that the running times for multi-term intersection are obtained using a state-of-the-art DAAT implementation that maintains pointers into all lists.

Table 9: Times in ms for full queries on Gov2 and ClueWeb09B, for DAAT-intersection, and for WAND and Maxscore with $k = 10$.

	Gov2			ClueWeb09B		
	Inter	WAND	Maxscore	Inter	WAND	Maxscore
Random	21.19	47.70	26.85	60.55	75.43	61.98
URL	8.27	19.03	18.46	37.93	63.03	46.98
BP	7.30	19.86	19.35	33.99	50.99	40.39
BP-LM	7.13	**18.59**	**18.24**	32.91	50.61	**40.35**
BP-RUN-CB	**6.05**	20.11	20.89	**28.13**	**48.76**	43.32

We see that for DAAT intersection (Inter, described in Section 2.3), BP-RUN-CB again performs significantly better than the other orderings, and outperforms BP by 18.6% on Gov2 and 17.8% on ClueWeb09B for two terms, and by 17.1% on Gov2 and 17.2% on ClueWeb09B for the original full term queries. This supports our basic assumption

that we can focus our model and optimization on only the two terms with the shortest list, and still get similar improvements for the full queries even for DAAT intersection algorithms, as the cost is dominated by accesses in the two shortest lists.

However, we also see that BP-RUN-CB does not consistently perform better than the other orderings on the disjunctive top-k algorithms WAND and Maxscore, in some cases slightly outperforming the other orderings, and in some cases underperforming. This is not surprising, as our model does not really apply to these methods. More precisely, there is no obvious connection between our definition of run complexity and the number of forward seeks in either WAND or Maxscore.

6.5 Compression Ratio

Finally, in Table 10, we show the compression in bits per docID under various orderings and compression methods, on Gov2 and ClueWeb09B. BP consistently achieves the smallest index size under all compression methods. BP-LM is slightly worse as it focuses on lists corresponding to common query terms. BP-RUN-CB is worse than BP and BP-LM, and performs similarly to URL. This is to be expected since BP-RUN-CB does not attempt to minimize compressed size but instead focuses on reducing expected intersection query processing costs.

Table 10: Bits per docID for full indexes on Gov2 and ClueWeb09B, for various orderings and compression methods.

	Gov2			ClueWeb09B		
	BIC	OptPFD	PEF	BIC	OptPFD	PEF
Random	6.36	6.89	6.53	5.59	6.26	6.11
URL	2.80	4.01	3.84	3.78	5.39	4.82
BP	**2.18**	**3.41**	**3.12**	**3.20**	**4.63**	**4.30**
BP-LM	2.34	3.54	3.22	3.59	4.98	4.72
BP-RUN-CB	3.04	4.19	3.67	3.88	5.29	4.83

To summarize, our experimental evaluation showed encouraging improvements in both the number of forward seeks and the actual running times of intersection queries for our new approach, with speedups of around 17% over the best previous approach. Not surprisingly, this came at the cost of a larger compressed size when compared to BP. Also, somewhat disappointingly, we did not see consistent improvements in speed for the disjunctive top-k algorithms WAND and Maxscore, and we expect that another approach is needed to optimize for these algorithms.

7. DISCUSSION OF RESULTS

We now discuss our results and point out some limitations.

Size vs. Speed. As shown, our new method improves over BP in terms of speed, but does worse in terms of compressed size. We believe that in many search engine environments, speed is more important, but there will certainly be other cases. One way to look at this is by starting from URL ordering, which achieved significant improvements in both size and speed over a random ordering. The BP algorithm in [20] used an objective function that aims for reduced size, resulting in significant improvements in size but very limited improvements in intersection speed over URL, as shown in our results. Our algorithm BP-RUN chose to go into a different direction, optimizing for speed, and achieved much better speed than URL and BP, but compressed size was about the same as URL and worse than BP. Getting improvements in both measures would be nice, but under a BP-type approach these are very different objectives.

Maintenance: One concern about reordering approaches, including our own, is that they are hard to maintain under document changes. This is partially true, and reordering is probably not a good choice under very quickly evolving document collections such as [15]. However, for collections evolving at a slower rate, the standard solutions for index updates which rebuild and merge index structures in the background (e.g,, the log-merge in Lucene) could be extended to periodically reorder the collection or parts of it. Our as yet unoptimized running times for reordering are only moderately slower than index building times, and could be improved by, e.g., reducing the threshold in the language model that we used.

Direct Support for Multiple Terms: As we showed, by looking at only the two shortest lists, we can in fact optimize the performance of multi-term queries under the DAAT and SvS algorithms. Nonetheless, additional improvements might be possible by extending the language model to triplets (3-grams) of query terms, or even quadruplets, and changing BP-RUN accordingly. However, we expect the improvements to be very limited, since as we saw most accesses are to the shorter lists, and successfully using triplets might also require larger query traces in order to get a strong enough language model. Finally, use of large numbers of triplets would also increase the running time of BP-RUN.

8. CONCLUSIONS AND OPEN PROBLEMS

In this paper, we studied how to reorder document collections in order to increase the speed of intersections on the resulting index. We proposed a model for the complexity of an instance of the pairwise intersection problem, defined the problem of reordering to minimize the resulting complexity, and showed that the problem is fundamentally different from that of reordering for compression. We also proposed and evaluated a heuristic method based on an adaptation of the recent algorithm for recursive bisection in [20] that showed improvement in speed of about 17 percent over the best previous schemes on two standard data sets.

There are a number of open questions left for future research. We plan to run experiments on weighted cost models, discussed in Section 4.3, in future work. Minor additional speedups could also be obtained by improvements in the language model for two-term queries. We further plan to study other possible algorithms for finding better orderings. In fact, we initially attempted to reorder the collection via a bottom-up TSP-based method, but failed to get any improvements over URL sorting and BP. However, there might be other approaches that can outperform BP-RUN.

One major limitation of our approach is that we did not see consistent improvements over BP and URL sorting for disjunctive top-k algorithms such as WAND and Maxscore. We suspect that these algorithms require a different and more involved model for instance complexity. This is also left for future work.

9. REFERENCES

[1] R. R. Amossen and R. Pagh. A new data layout for set intersection on gpus. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 698–708, 2011.

[2] N. Ao, F. Zhang, D. Wu, D. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[3] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, pages 400–408, 2004.

[4] R. A. Baeza-Yates. Experimental analysis of a fast intersection algorithm for sorted sequences. In *String Processing and Information Retrieval*, pages 13–24, 2005.

[5] J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Algorithms*, 4(1):4:1–4:18, Mar. 2008.

[6] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Conference on Experimental Algorithms*, pages 146–157, 2006.

[7] J. Barbay, A. López-Ortiz, and T. Lu. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14:7:3.7–7:3.24, June 2010.

[8] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1976.

[9] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *International Symposium on Algorithms and Computation*, pages 739–750, 2007.

[10] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proceedings of the 29th European Conference on IR Research*, pages 375–387, 2005.

[11] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proceedings of the Data Compression Conference*, page 342, 2002.

[12] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 16–26, 1998.

[13] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, pages 426–434, 2003.

[14] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26:211–226, 1979.

[15] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1360–1369, 2012.

[16] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. In *2011 IEEE 27th International Conference on Data Engineering*, pages 709–720, 2011.

[17] J. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1):1:1–1:25, 2010.

[18] E. D. Demaine, A. López-Ortiz, and J. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752, 2000.

[19] E. D. Demaine, A. López-Ortiz, and J. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, pages 91–104, 2001.

[20] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544, 2016.

[21] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, pages 113–122, 2013.

[22] B. Ding and A. C. Konig. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[23] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proceedings of the 19th International Conference on World Wide Web*, pages 311–320, 2010.

[24] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 993–1002, 2011.

[25] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He. Bitfunnel: Revisiting signatures for search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 605–614, 2017.

[26] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Inf.*, 1(2):145–158, June 1971.

[27] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput*, 1(1):31–39, June 1972.

[28] A. Kane and F. Tompa. Skewed partial bitvectors for list intersection. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 263–272, 2014.

[29] A. Kane and F. W. Tompa. Distribution by document size. In *Workshop on Large-scale and Distributed Systems for Information Retrieval*, 2014.

[30] S. Khot and O. Regev. Vertex cover might be hard to approximate to within 2-ϵ. *J. Comput. Syst. Sci.*, 74(3):335–349, May 2008.

[31] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. *Softw. Pract. Exper.*, 46(6):723–749, June 2016.

[32] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster blockmax wand with variable-sized blocks. In *Proceedings of the 40th International ACM SIGIR Conference on Research*

and Development in Information Retrieval, pages 625–634, 2017.

[33] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, Jul 2000.

[34] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 273–282, 2014.

[35] V. Ramaswamy, R. Konow, A. Trotman, J. Degenhardt, and N. Whyte. Document reordering is good, especially for e-commerce. In *Proceedings of the SIGIR 2017 Workshop On eCommerce*, 2017.

[36] C. Rosset, D. Jose, G. Ghosh, B. Mitra, and S. Tiwary. Optimizing query evaluations using reinforcement learning for web search. In *The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1193–1196, 2018.

[37] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proceedings of the Meeting on Algorithm Engineering and Experiments*, pages 71–83, 2007.

[38] B. Schlegel, T. Dresden, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *In International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage at VLDB*, 2016.

[39] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Inf. Process. Management*, 39(1):117–131, Jan. 2003.

[40] F. Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on IR Research*, pages 101–112, 2007.

[41] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 305–312, 2004.

[42] F. Silvestri and R. Venturini. Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 1219–1228, 2010.

[43] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 738–739, 2009.

[44] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *PVLDB*, 2(1):838–849, 2009.

[45] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Management*, 31(6):831–850, Nov. 1995.

[46] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 105–114, 2011.

[47] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang. A batched gpu algorithm for set intersection. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 752–756, 2009.

[48] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, pages 401–410, 2009.

[49] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.