Using Conjunctions for Faster Disjunctive Top-k Queries

Michał Siedlaczek
michal.siedlaczek@nyu.edu
New York University
New York, US

Antonio Mallia
antonio.mallia@nyu.edu
New York University
New York, US

Torsten Suel
torsten.suel@nyu.edu
New York University
New York, US

ABSTRACT

While current search engines use highly complex ranking functions with hundreds of features, they often perform an initial candidate generation step that uses a very simple ranking function to identify a limited set of promising candidates. A common approach is to use a disjunctive top-k query for this step. There are many methods for disjunctive top-k computation, but they tend to be slow for the required values of k, which are in the hundreds to thousands.

We propose a new approach to safe disjunctive top-k computation that, somewhat counterintuitively, uses precomputed conjunctions of inverted lists to speed up disjunctive queries. The approach is based on a generalization of the well-known MaxScore algorithm, and utilizes recent improvements in threshold estimation techniques as well as new ideas to obtain significant improvements in performance. Our algorithms are implemented as an extension of the PISA framework for search-engine query processing, and available as open-source to support replication and follow-up work.

CCS CONCEPTS

• **Information systems → Information retrieval query processing**.

KEYWORDS

query processing; top-k retrieval; candidate generation

ACM Reference Format:
Michał Siedlaczek, Antonio Mallia, and Torsten Suel. 2022. Using Conjunctions for Faster Disjunctive Top-k Queries. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining (WSDM '22), February 21–25, 2022, Tempe, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3488560.3498489

1 INTRODUCTION

Modern search engines use highly complex ranking functions to return high-quality results. These functions may involve hundreds of features and are often obtained using learning-to-rank techniques [20] or, more recently, neural network technologies such as transformers [19]. While complex rankers significantly improve relevance in the top positions compared to simple ranking schemes, such as BM25 or query likelihood, they are also expensive to evaluate on large numbers of documents. As a result, most systems use a cascading approach to ranking [36], where a very simple ranking function is first used to retrieve a limited number of candidates that are subsequently reranked using more complex and expensive methods. This initial phase is often referred to as *candidate generation*, and is the focus of our work here.

One popular approach to candidate generation involves running a disjunctive top-k query on the query terms to retrieve a few hundred to a few thousand initial candidates. This approach is known to achieve better retrieval quality than a purely conjunctive approach, which requires all query terms to be present in candidate documents. However, it comes at the cost of a significant increase in processing time, since a disjunctive query has to score many more documents than a conjunctive one.

This has motivated a lot of work on more efficient ways to process disjunctive top-k queries, with continuous improvements over the past three decades. We focus on so-called safe methods, which are guaranteed to return all correct top-k results under the simple ranking function, as opposed to unsafe methods, which may miss some results. Well-known algorithms for disjunctive top-k retrieval include MaxScore [35], WAND [3], Block-Max WAND [12] and related methods such as BMM [4, 11] and VBMW [23], and JASS [8]. While these methods perform well for small k (e.g., $k = 10$), many of them slow down significantly when k increases to 1000 or more [8, 25], which is common in candidate generation [5, 6, 9, 22, 29, 40]. It was also observed [25] that for larger k, the fastest method is often the much older and simpler MaxScore [35].

The goal of this work is to improve the performance of disjunctive top-k query processing for larger k by extending MaxScore. In particular, we generalize the concept of essential lists in MaxScore by allowing precomputed intersections (conjunctions) of query terms to be used as essential lists. Since intersections of two lists tend to be much shorter than either list, this can result in much smaller essential index structures, and thus faster processing under a MaxScore-style approach. Our approach requires a good initial estimate of the top-k threshold, and then uses the concept of result classes to select an optimal set of essential structures as access paths for candidate retrieval. Our main contributions are:

(1) We propose a novel and interesting generalization of the Maxscore approach for safe disjunctive top-k queries that allows intersections to be selected as essential lists.
(2) We describe methods for choosing which intersections to precompute at indexing time, which index structures to use as essential structures at query time, and which lookups to perform during index traversal.
(3) We present extensive experiments on carefully optimized implementations of our methods and previous work that show the benefits of our approach. While we require significant additional data structures, the overhead of retrieving these from SSD is shown to be small compared to overall cost.

WSDM '22, February 21–25, 2022, Tempe, AZ, USA

ACM ISBN 978-1-4503-9132-0/22/02…$15.00
https://doi.org/10.1145/3488560.3498489

(4) We provide an implementation of the proposed approach as an extension of the open-source PISA IR framework [24].

2 BACKGROUND AND RELATED WORK

In this section, we define inverted indexes and disjunctive top-k query processing, and discuss disjunctive query processing algorithms. Then we discuss previous methods that exploit intersections for faster query processing, and the role of threshold estimation in accelerating retrieval algorithms.

2.1 Inverted Indexes and Top-k Queries

An *inverted index* is a basic data structure used in almost all search systems. It stores information about where each term occurs in the collection; more precisely, the inverted index contains an *inverted list* l_w for each distinct term w. An inverted list l_w is a sequence of *index postings*, where each index posting is a pair (d, f), where d is the document ID (docID) of a document containing w, and f is the frequency (number of occurrences) of f in the document. We assume that postings in each inverted list are sorted by docID.

A ranking function is defined as a function s that, given a query q, assigns each document d a score $s(d, q)$. We assume a simple ranking function of the form $s(d, q) = \sum_{i=0}^{m-1} f(d, w_i)$, where the w_i are the m terms occurring in q and f is a function that can be efficiently computed from the information in the index postings for w_i plus some limited global statistics such as document sizes and inverted list lengths. This means that scores $f(d, w_i)$ can be precomputed, quantized, and stored in place of frequencies to speed up query processing. We note that well-known families of ranking functions such as Cosine measures, BM25, and some methods based on language modeling satisfy this condition.

Given a ranking function s and query q, a disjunctive top-k query returns the k highest scoring documents among those that contain at least one of the query terms. Conversely, a conjunctive top-k query only selects from documents that contain all terms in q.

Current engines use highly complex ranking functions with hundreds of features. For efficiency, they often perform an initial selection of candidate documents using a top-k query on a simple ranking function, for k in the hundreds or thousands. Candidates are then reranked using the complex ranker. While conjunctive top-k queries are more efficient for the initial step, disjunctive queries often give better results and are thus preferred in many scenarios.

2.2 Disjunctive Top-k Algorithms

Given the importance of disjunctive top-k queries, there has been a lot of work on efficient algorithms. Approaches can be divided into *safe* and *unsafe* methods. The former always return the same top-k results as an exhaustive approach that scores all documents containing at least one query term, while the latter may miss some results but return most of the same, or similar quality, results. We focus on safe methods. We also assume that the inverted index is kept in main memory, or that at least enough data is cached to eliminate disk access as a major bottleneck, which is a realistic assumption for current large-scale search architectures.

Commonly studied safe algorithms include MaxScore [35], WAND [3], Block-Max WAND [12] and related methods such as BMM [4, 11], VBMW [23] and other block-max based approaches [26], JASS [8], the algorithm by Strohman and Croft [32], and methods based on Fagin's TA algorithm [13]. A recent study [25] shows that block-max based approaches are very fast on common data sets when k is small (say, $k \leq 100$). However, for larger k they tend to slow down, while the simpler and older MaxScore often performs best. We focus on the MaxScore approach, and show how to further improve its speed for larger k.

We now briefly describe MaxScore. For simplicity, we assume that we already know the top-k threshold th for the query, and thus the goal is to find all documents with score above th. Also, for each term w in the inverted index, we have precomputed the *maxscore* of w, defined as the maximum score $f(d, w)$ of any document d.

We first sort the query terms in order of ascending maxscore. Let this order be $w_0, w_1, \ldots, w_{m-1}$. We select a maximum prefix of terms $w_0, \ldots w_{m'-1}$ such that $\sum_{i=0}^{m'-1} maxscore(w_i) < th$. We call the inverted lists of these terms *non-essential lists*, and the remaining ones *essential lists*. Clearly any document that scores at or above the threshold must occur in at least one essential list. This leads to a fairly simple algorithm that performs a disjunctive traversal of the essential lists, and then makes lookups into the non-essential lists for any documents found in the essential lists. Since the max scores of inverted lists are usually inversely correlated with their lengths, the non-essential lists typically contain most of the postings for the query terms, assuming a large enough threshold th. Thus, we reduce a disjunctive query to a disjunctive traversal of a much smaller set of lists, plus some lookups into the other lists.

Note that at the surface the above also resembles the approach by Fontoura et al. [14]. However, they used a heuristic to select an initial set of terms that is processed first to estimate the threshold used in the remaining phase. As discussed later, we can compute a good threshold estimate before starting posting traversal, and thus use a more systematic approach to minimize the number of processed postings. Furthermore, as we show, this approach can be generalized to use intersections of lists for additional speedup.

2.3 Using Intersections for Faster Querying

There is some previous work that uses precomputed, cached, or on-the-fly generated intersections of two or more inverted lists to accelerate query processing. Long and Suel [21] used a set of training queries to select intersections that should be precomputed and added to the index. Then incoming conjunctive top-k queries are rewritten to incorporate available precomputed intersections. Subsequent improvements include [7, 34, 41].

While the above work considers conjunctive queries, there is also previous work that uses intersections to approximate disjunctive queries, and in particular as a faster alternative to full disjunctions in the context of candidate generation. Most relevant to our work are the approaches in [2, 30, 37]. In particular, [30] describes candidate generation in the Bing search engine, where a query is rewritten into a number of conjunctions that are used as efficient access paths for retrieving candidates. A different approach to candidate generation is described in [38], where intersections are precomputed and placed into the first layer of a layered index. Finally, Vigna [2] proposed a method called Model B that first attempts to process a query as a pure intersection query, and then backs off into a more and more disjunctive format if not enough results are produced.

The above approaches are unsafe as they cannot guarantee the same results as an exhaustive query. This motivated the question we address here, whether we can use intersections for safe disjunctive query processing. Our basic idea is to precompute and store certain intersections as part of the index, and to then replace some of the essential lists in MaxScore with intersections. Since intersections tend to be much smaller than their constituent lists, this should decrease the size of the essential lists that need to be traversed.

2.4 Top-k Threshold Estimation

Given a query q, disjunctive top-k threshold estimation is the problem of estimating the score of the k-th highest scoring document, called the *threshold*. The goal is to get a good estimate much faster than executing the query. A strong initial threshold can speed up many common top-k query processing algorithms [12, 28, 39]. However, it is important to avoid overestimates, since that might result in fewer than k results being returned by the algorithm. Several approaches for threshold estimation exist, including Taily [1], random sampling [31, 33], machine learning [28], and quantile-based techniques [10, 16, 27, 39]. We use a hybrid method combining sampling and quantiles from [27], described in Section 3.3.

3 OUR APPROACH

We now describe our approach. We explain the basic idea with examples, outline the necessary steps, and give details on each step.

3.1 Overview

Recall that in MaxScore, the inverted lists for the query terms are divided into essential and non-essential lists based on maxscores. By designating many low-maxscore, and thus usually fairly long, lists as non-essential, we minimize the number of postings in the essential lists. Put another way, we are trying to select the smallest possible set of essential postings such that any document in the top-k for the query must have at least one representative posting in the set. Thus, essential lists are basically used as a compact and efficient access path to retrieve all top-k candidates.

Consider a query q with query terms A, B, and C. We refer to the corresponding inverted lists as $[A]$, $[B]$, and $[C]$. Let their maxscores be 5.0, 6.0, and 8.0, respectively, and let the top-k threshold for q be 12.0. Then the MaxScore approach would select $[C]$ as the only essential lists, since any top-k result must contain C.

Suppose we also have precomputed intersection lists, denoted as $[A, B]$, $[A, C]$, and $[B, C]$. A posting in an intersection list consists of a docID and two scores, one for each list. While any top-k document must contain C, this is not sufficient – any top-k result must either contain C and A, or C and B. This gives us the idea to select $[A, C]$ and $[B, C]$ as essential lists, and to label all three lists $[A]$, $[B]$, and $[C]$ as non-essential. The MaxScore algorithm then proceeds as before, traversing the union of the essential structures, and making lookups as needed into other lists on documents encountered during traversal. We expect the two intersection lists to be much smaller than $[C]$, leading to faster traversal. Moreover, for this example, we do not need any lookups at all: if we find a document in both essential lists, we know its complete score. If we only find it in one list, say in $[A, C]$ but not in $[B, C]$, then we know that it cannot have a posting in $[B]$ (and vice versa for the other case and [A]).

Suppose the threshold for the query is 10.0 instead. Then MaxScore would choose $[B]$ and $[C]$ as essential lists. With intersections, we have several alternative choices. We could choose $[C]$ and $[A, B]$ as essential lists, since any top-k result that does not contain C must contain both A and B. In this case, for any docID found in both essential structures, no lookup is needed. For any docID only found in $[A, B]$, we also need no lookup, since we know that C cannot occur. For any posting found only in $[C]$, we can first perform a lookup into B. If this lookup retrieves a posting, no further lookup on $[A]$ is needed as the document cannot have both A and B. Or we could choose all three intersections as essentials, which would likely result in even fewer essential postings (though the first solution is still useful in cases where not all intersections are available).

Finally, assume an additional term D with maxscore 9.0 and a top-k threshold of 16.0. MaxScore would then choose $[C]$ and $[D]$ as essential lists. Our approach could choose, e.g., $[D]$ and $[B, C]$, or $[C, D]$ and $[A, B]$, or $[B, D]$ and $[C]$, and so on. In summary, we see that there is potential for improvements by using intersections in a MaxScore approach. We also see that this can get complicated, with many possible choices depending on the maxscores, top-k threshold, and available intersections, and thus we need a more formal approach. We propose the following mechanisms, which are described in detail in the following:

- **Precomputing Intersections:** At indexing time, we select a set of intersections that should be created, subject to a space limit. We should choose intersections commonly occurring in queries that are much smaller than each of the two lists.
- **Threshold Estimation:** Given an incoming query, we need a good and fast estimate of the top-k threshold to allow us to select a small set of essential structures.
- **Selecting Essential Structures:** Given a threshold estimate, maxscores for the query terms, and a list of available intersection of query terms, we select essential structures in a way that minimizes query processing costs.
- **Avoiding Unnecessary Lookups:** During traversal of the selected essential structures, we need to decide which lookups into non-essential lists must be performed. The answer depends on the threshold and the already accumulated partial score of a document, as well as on the intersections selected as essential structures, in a complex way, and we need a very fast way to make these decisions.

3.2 Selecting Precomputed Intersections

First, we discuss how to select the intersections that are built at indexing time. Given a space budget B, our goal is to select a set of intersections of total size at most B that maximizes the expected query processing speedup. We estimate three quantities for each considered intersection: (1) how likely the intersection will be useful, F, (2) the savings in query processing cost that occur when it is used, C, and (3) the amount of space taken up by the intersection, S. Our goal is to maximize the expected benefit per space: $F \cdot C/S$

We approximate the first quantity with a language model on a set of training queries, to estimate the likelihood that a random incoming query from the query distribution contains both terms in the intersection. This is a rough approximation since the intersection may not actually be useful even if both terms are in the query. The second quantity is approximated as the difference between the length of the shorter of the two lists and the intersection, where the length of the intersection is scaled to account for the fact that intersections are more expensive to traverse, per posting, than single-term lists. Finally, the space S is simply estimated as the number of postings in the intersection.

We then first identify a large set of candidates by taking intersections with F above a minimum threshold. We estimate C and S for these intersections, and greedily choose intersections based on $F \cdot C/S$, until the space budget for intersections is spent.

3.3 Threshold Estimation

Our threshold estimation uses a hybrid between sampling and quantile-based methods from [27]. In particular, we implemented a quantile-based method called Q_k^4 that stores the k-th highest score for each term in the collection, and for each pair, triple, and 4-tuple of terms encountered in queries of a large training query trace. We combined this with the sampling-based method from [27] as follows: Given a query, we execute the quantile-based method. If the full query is contained in one of the pairs, triples, or 4-tuples that was stored, then the quantile-based method guarantees an exact threshold. Otherwise, we execute the sampling-based method (with an initial threshold equal to the one estimated by Q_k^4), and take the maximum of the two estimates. The quantile-based method never returns an overestimate. The sampling-based method can overestimate, but the probability of doing so can be easily bounded as discussed in [27], and this bound also applies to the hybrid.

Note that our overall query processing method is safe, i.e., always returns the correct top-k results, even in the presence of overestimates. The reason is that an overestimate is easily detected once the query has been executed, as we end up with fewer than k results above the threshold. In this rare case, we simply reexecute the query using a baseline method, and all our results include this cost.

3.4 Selecting Essential Structures

Next, we discuss how to select essential index structures given a top-k threshold estimate and a set of available intersections. This task is more complex and requires some new definitions.

Result Classes and Score Bounds. Given an m-term query $q = (w_0, w_1, \ldots, w_{m-1})$, a result class is identified by a bitvector c of length m, and consists of all documents that contain all w_i where $c[i]$ is set to one, and none of the w_i where $c[i]$ is set to zero. For example, if $q = (\text{dog}, \text{cat}, \text{mouse})$; then result class $(1, 0, 1)$ consists of all documents that contain "dog" and "mouse" but not "cat". For simplicity, we use a binary vector c as a synonym for its result class.

Thus, there are 2^m result classes for a query of length m, where the class defined by a vector of all zeros consists of documents containing none of the query terms, and the class for a vector of all ones consists of documents containing all the terms. The 2^m classes form a lattice with a partial order $\ll$, where $c_1 \ll c_2$ iff the 1-bit positions in c_2 properly contain those in c_1.

Given a query q and a result class c, let $u(q, c)$ be an upper bound on the score of any document in c. The easiest way to get such a bound is to add up the relevant maxscores, i.e., $u(q, c) = \sum_{0 \leq i < m \ \& \ c[i]=1} maxscore(w_i)$. Tighter upper bounds are possible, for example by storing additional maxscore information for some intersections, though this appears to give only minor improvements in the bounds. It is reasonable to assume that $c_1 \ll c_2$ implies that $u(q, c_1) \leq u(q, c_2)$, for the types of bounds we can efficiently compute – though the actual top score in c_1 might sometimes be higher than that in c_2. We say that a result class c is *critical* for q if $u(q, c) \geq th$, where th is the (estimated) top-k threshold of q.

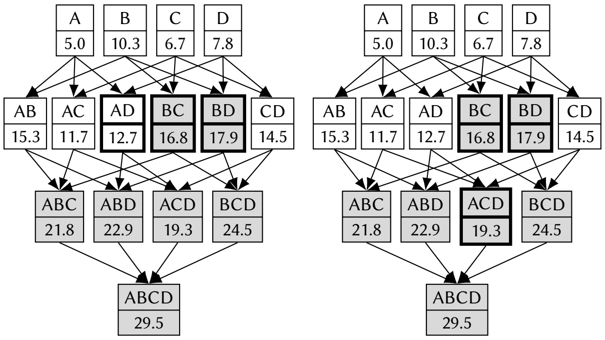

Figure 1: Result class lattice for query with terms A, B, C, D, assuming a threshold of 16.0.

Index Structures and Retrieval of Result Classes. Given an incoming query q, we have a set of available index structures for the query terms, including single term lists, two-term intersections, and in principle even intersections of more than two terms. An inverted list for query term w_i contains a posting for every document containing w_i. Translating this to result classes, this means that the list contains a posting for every document that is in a result class c with $c[i] = 1$. Formally, we say that the index structure *retrieves* all these classes c. Furthermore, an intersection list for terms w_i and w_j retrieves all result classes c with $c[i] = c[j] = 1$, and correspondingly for intersections of more than two terms.

Given these definitions, we can restate the task of selecting essential index structures. We say that a set of essential index structures is *safe* for a query q and threshold th if every critical result class is retrieved by at least one structure. Note that this implies that we can find all top-k results by traversing the essential structures and then performing lookups into non-essential lists as needed. Thus, our goal can be stated as follows:

Problem Definition: *Given a query q, a top-k threshold th, and a set of available index structures on the query terms, the goal is to select a safe subset of the structures that minimizes the cost.*

We assume here that the cost is simply the total number of postings in the selected essential structures, with a weight factor to account for the fact that postings from multi-term structures take more time to process. The actual running time also includes the cost of lookups into non-essential lists. However, the lookup cost is hard to estimate, as the number of necessary lookups per posting can differ significantly between lists. Thus, our goal is to select essential structures of minimal total size, in the expectation that smaller essential structures typically lead to faster overall times.

Example. Figure 1 shows an example of a query with terms A, B, C, and D. Critical result classes are shown in grey. An index structure retrieves the result class with the corresponding label and any class reachable from it. The left side shows a safe set of three intersections, $[A, D]$, $[B, C]$, $[B, D]$, assuming these are available in the index. The right side shows another safe set where an intersection of three terms, $[A, C, D]$, is selected together with $[B, C]$ and $[B, D]$.

Our approach is to treat this as a *weighted set cover* problem. Here, the elements of the set cover problem are any critical classes that are not reachable from other critical classes, while the sets are defined by the available index structures. A set associated with an index structure contains a critical result class iff the index structure retrieves that class. In Figure 1, the set associated with $[B]$ is $S_B = \{(BC), (BD)\}$. We could also include the rest of the critical (shaded) classes containing B in S_B, but it is unnecessary as they are reachable from those in S_B. The goal is to cover everything at least once while minimizing the total cost of the selected posting lists (we expand on the chosen cost model in Section 4).

The general Set Cover problem is NP Complete [15], though our version has additional structure that might make it easier. However, the number of critical result classes that are part of the input to our problem can be exponential in the number of query terms, while the number of available index structures can be quadratic for pairwise intersections, or higher for intersections of more than two terms. Moreover, the complexity of the problem may depend on the upper bounds for the result classes, e.g., if they are assumed to be sums of term-wise maxscores, or if they could be arbitrary values. A complexity analysis is deferred to future work.

We implemented two methods for selecting essential structures: an exhaustive and a greedy one. Both start by first finding critical classes not reachable from other critical classes, and then identifying the subsets that correspond to the available single-term or pairwise index structures. The exhaustive method considers all possible solutions to find the one minimizing cost. This is quite fast for short queries and cases where only few intersections are available, but slow in other cases. The greedy method uses the standard greedy approximate algorithm for Set Cover. We found this to be highly accurate for the case where only single terms and pairs of terms are considered. We discuss the results in the experimental section.

3.5 Avoiding Unnecessary Lookups

Next, we describe our policies for performing lookups into the non-essential lists. We start by describing lookup policies for the standard MaxScore algorithm, with only single-term index structures. Suppose you have a 4-term query with terms A, B, C, and D, and maxscores 4.0, 5.0, 6.0, and 7.0, respectively. If the threshold is, say, 13.0, then MaxScore will select $[C]$ and $[D]$ as essential lists. When a document is found in the essential lists, we consider whether to look up its scores in the non-essential lists, usually in order from highest to lowest maxscore. If the partial score accumulated from the essential lists is less than 4.0, then no lookups are needed as the threshold cannot be reached. If the score is higher, then at least one more lookup is needed. In general, before each lookup, we check the partial score accumulated so far, and the maximum possible score from further lookups (the sum of the maxscores of the remaining lists). If the sum of these two is less than the threshold, we can discard the document without further lookups.

This becomes more complicated if we allow intersections as essential lists. There are two challenges: First, while traversing the essential lists, there is additional overhead as a document may be encountered in two overlapping intersections, say in $[B, D]$ and in $[C, D]$. This needs to be detected so that we do not add up the score from $[D]$ twice, creating some overhead. Second, rules for avoiding lookups are more complex. As in standard MaxScore, we do not need to do any lookups into single-term lists that are essential structures. However, suppose $[B, D]$ is an essential lists, but the document being evaluated was not found in it. Then we know that the document does not contain both B and D, but it could contain either one. If we know that the document contains B, say because either $[B]$ or $[B, C]$ is another essential list containing the document, or we have done a lookup into $[B]$, then no lookup in $[D]$ is needed. Thus, intersections result in more complex rules about what lookups are necessary, and more complex bounds for the maximum score from further lookups – if $[C]$ and $[B, D]$ are essential, then a document discovered in $[C]$ but not in $[B, D]$ has a maximum score from further lookups of $maxscore(A) + max(maxscore(B), maxscore(D))$, as it cannot contain both B and D.

We designed two methods for dealing with these challenges. The first one, *eager*, focuses on the first challenge, at the cost of additional lookups. It treats all selected single-term essential lists as in standard MaxScore, aggregating their essential scores and then performing lookups into the other lists, while ignoring the existence of any essential intersections. If the same document also occurs in one of the essential intersections, then it will be processed again separately, with additional lookups as needed. However, intersections are usually shorter than single-term lists, so that most documents are found only in single-term lists or in at most one essential intersection. Thus, the amount of extra work is limited. After aggregating results for single-term essential lists, we aggregate the results from each intersection, one by one, with lookups into non-essential single-term lists not in the intersection. Finally, the different result sets are merged, and any duplicates removed.

The second approach, *state*, focuses on the other challenge, avoiding lookups as much as possible, while speeding up the logic for deciding what lookups are needed. It maintains a *lookup state* for any document found in the essential structures, which is an integer in $[(n+1) \cdot 2^n]$ for an n-term query. Terms are ordered from highest to lowest maxscore, and lookups occur in this order. The state encodes in its higher bits which terms have already been considered for lookups (a number from 0 to n as lookups occur in a fixed order), and in the lower n bits which lookups retrieved a posting.

At the start of a query, after essential structures have been selected, we precompute two small arrays indexed by lookup state, one called *next* for telling us which lookup to consider next, and one called *mps* that stores the maximum possible additional score from further lookups, given the current state. All the logic for deciding what lookups to perform is precomputed into these tables, which are consulted as we evaluate a document. In particular, we check *mps* to see if the maximum possible scores in the current state could lead to a top-k result. If not, we terminate lookups for this document; otherwise, we check *next* to decide on the next lookup, update the lookup state, and continue.

Experiments showed *eager* to be faster for up to three terms, while *state* is faster otherwise. This is because the logic of *state* is more complex, which causes some overhead, including computing *next* and *mps* tables, but the potential savings due to avoiding unnecessary lookups increase with the number of query terms. Thus, we used a hybrid, with the eager version for short queries, and the state version otherwise. We include the pseudocode for both algorithms in Appendix A. We also encourage interested readers to study the implementation details on Github.[1]

[1] https://github.com/elshize/using-conjunctions-docker

Table 1: Accuracy of greedy selection for different k.

Collection	Queries	10	100	1000	10 000
ClueWeb09B	TREC 2005	0.94	0.94	0.94	0.95
ClueWeb09B	TREC 2006	0.92	0.91	0.89	0.89
ClueWeb12B	TREC 2005	0.94	0.93	0.93	0.95
ClueWeb12B	TREC 2006	0.90	0.88	0.87	0.86

4 EXPERIMENTS

We now analyze the performance of the proposed methods with an extensive experimental evaluation in a realistic and reproducible setting, using state-of-the-art baselines and standard data sets.

Setup. All methods were implemented based on the PISA framework [24], written in C++ and compiled with GCC 8.3 with highest optimization settings. Tests are performed on a machine with 8 Intel Core i7-4770 3.50GHz Haswell cores, with 32GiB RAM, running Linux 4.15. Only a single core is used in each run. Inverted indexes were saved to disk after construction, and memory-mapped for querying; thus, there are no hidden costs due to loading of additional data structures in memory. Before timing the queries, we ensure that all required single-term posting lists are in memory.

We precomputed and stored different sets of pairwise structures (intersections) according to the algorithm in Section 3.2. Due to their larger size, these structures were not loaded into memory, but instead kept on an SSD drive and fetched during query processing. We used a Samsung 860 EVO 2.5 Inch SATA III Internal SSD drive with sequential read speed up to 550 MB/s and up to 98K random read IOPS. This is a low-cost drive, at about $200 for the 2TB version, and not as fast as many available NVMe drives. Nonetheless, we found the overhead SSD accesses to be small. Complete source code, data, and a docker image will be made available at publication time.

Datasets. We used two standard text collections, ClueWeb09B and ClueWeb12B, with 50.2M and 52.3M documents. Collections were parsed using PISA, with terms stemmed using the Porter2 stemmer. Data was then exported to Common Index File Format [18] files, which are the input to our scripts. This makes it easier to reproduce results, or to use indexes parsed by other software such as Apache Lucene. For each collection, we built one inverted index with frequencies (*non-quantized*), and one with quantized scores (*quantized*) using 8-bit linear quantization. Posting docIDs and frequencies were encoded using SIMD-BP128 [17]. To evaluate query processing speed, we use TREC 2005 and TREC 2006 Terabyte Track Efficiency Task data. From each, we selected 1000 random queries with 2 to 16 terms. We also used the AOL query log of 20M queries as training data for threshold estimation and to select intersections.

Query Cost Model. Recall that traversing intersections is more expensive than traversing single-term posting lists, due to having two frequencies or quantized scores per posting. We found empirically that multiplying the number of postings by 1.25 for intersections gives the best results, and thus we use this factor to select intersections at indexing time and essential structures at query time.

4.1 Preliminary Experiments

We start with an initial set of experiments to explore the impact of threshold estimation and greedy selection on performance. Due to space constraints, we cannot give results for all data sets.

Table 2: Average query times (in ms) for the Max-Inter-All algorithm with $k = 1000$, with selections performed by the greedy and exhaustive approaches. We show query times without selection cost (Time) and with selection cost (+Sel).

		Greedy		Exhaustive	
Collection	Queries	Time	+Sel	Time	+Sel
ClueWeb09B	TREC 2005	9.22	9.23	9.16	9.62
ClueWeb09B	TREC 2006	11.16	11.17	10.77	11.77
ClueWeb12B	TREC 2005	8.19	8.21	7.99	8.44
ClueWeb12B	TREC 2006	9.9	9.91	9.39	10.37

Table 3: Average number of postings in the essential lists P_E, and lookups into the non-essential lists L_{NE} for $k = 1000$ on ClueWeb09B.

	TREC05		TREC06	
	P_E	L_{NE}	P_E	L_{NE}
MaxScore	1 232 970	146 952	1 508 983	269 374
MaxScore-T	1 063 127	101 341	1 272 617	206 692
Max-Inter-x2	470 384	105 473	791 108	182 588
Max-Inter-x5	420 660	85 707	669 347	160 966
Max-Inter-x10	392 747	74 285	582 454	140 337
Max-Inter-x15	385 677	65 346	512 719	123 677
Max-Inter-x20	375 209	60 222	482 628	111 823
Max-Inter-All	230 483	19 056	125 560	23 022

Threshold Estimation. As discussed in Section 3.3, we used a hybrid of random sampling and quantile methods, building on work in [27] and [39]. Our quantile method, Q^4_k*-log*, stores top-k quantile information for all terms in the index, and for all pairs, triples, and 4-tuples encountered in a large query log. Following [27], we used the AOL query log, resulting in about 13 million unique term pairs, 60 million unique triples, and 340 million 4-tuples. This results in space overhead of about 2GB, though this could be significantly reduced at little loss in precision by selecting fewer triples and 4-tuples. For random sampling, we chose a sample size of 0.5% and limited the expected overestimation rate to 1% (see [27] for more details). Overestimates are detected when fewer than k results above the threshold are returned. In this case, the query is rerun with MaxScore, similar to [27, 28], and the cost of doing so is included in the reported running times.

Table 7 in Appendix B shows the performance of our estimator using the mean under-prediction fraction (MUF) measure proposed in [28]. The method performs very well on our data, with mean estimates mostly above 90% of the real threshold, and even better numbers for larger k. Table 8 in Appendix B shows the cost of threshold estimation in microseconds for different configurations. As expected, costs increase with k and query length. Results on other data sets were very similar.

Selecting Essential Structures. Next, we evaluate our greedy essential list selection algorithm and compare it against the exhaustive approach. Due to exponential complexity of the latter, we could only perform this comparison for queries with up to five terms. As shown in Table 1, the greedy algorithm made exactly the same selections of essential structures between 86% to 95% of the time, depending on collection, query log, and k. Table 2 furthermore

Table 4: Average query times (in ms) for top-1000 on non-quantized indexes, using estimated thresholds.

	Number of query terms					Avg
	2	3	4	5	6+	
ClueWeb09B TREC 2005						
VBMW-T	9.23	15.91	22.58	38.82	79.63	22.16
MaxScore-T	19.46	20.58	22.97	30.02	36.6	22.91
Max-Inter-x2	9.32	16.41	19.36	25.98	33.46	16.34
Max-Inter-x5	8.98	14.88	17.99	25.39	33.02	15.52
Max-Inter-x10	8.9	14.31	16.64	23.48	31.67	14.84
Max-Inter-x15	8.63	14.05	**16.14**	23.25	30.55	14.46
Max-Inter-x20	**8.5**	**13.94**	16.18	**22.1**	**30.51**	**14.28**
Max-Inter-All	7.68	9.63	10.86	13.68	22.93	10.6
ClueWeb09B TREC 2006						
VBMW-T	8.15	16.75	29.1	46.03	115.49	41.35
MaxScore-T	16.19	20.59	27.63	37.2	56.59	30.88
Max-Inter-x2	7.92	17.3	24.78	33.58	55.33	27.2
Max-Inter-x5	7.63	15.79	23.36	30.67	52.59	25.46
Max-Inter-x10	6.39	14.9	21.48	28.98	50.63	23.95
Max-Inter-x15	6.24	14.52	20.82	27.86	49.55	23.3
Max-Inter-x20	**6.16**	**14.2**	**20.06**	**26.95**	**49.09**	**22.8**
Max-Inter-All	5.0	8.6	13.68	18.2	37.69	16.16
ClueWeb12B TREC 2005						
VBMW-T	10.71	16.76	21.96	35.15	89.09	23.43
MaxScore-T	22.15	23.38	25.17	30.79	44.8	25.83
Max-Inter-x2	7.77	14.12	16.57	23.53	32.07	14.34
Max-Inter-x5	7.79	12.8	15.05	22.2	30.29	13.5
Max-Inter-x10	7.77	12.17	13.91	21.51	29.99	13.07
Max-Inter-x15	7.69	11.88	**13.57**	20.07	29.37	12.73
Max-Inter-x20	**7.62**	**11.74**	13.6	**19.5**	**28.68**	**12.55**
Max-Inter-All	7.17	8.06	9.09	12.11	21.18	9.41
ClueWeb12B TREC 2006						
VBMW-T	5.35	11.85	22.17	34.72	88.34	31.15
MaxScore-T	14.39	17.1	25.28	32.65	50.17	27.24
Max-Inter-x2	5.3	14.1	23.24	29.04	48.22	23.55
Max-Inter-x5	4.99	13.6	22.13	27.19	46.43	22.48
Max-Inter-x10	4.87	12.4	19.84	25.52	44.99	21.09
Max-Inter-x15	**4.78**	11.78	18.78	**24.29**	**43.68**	20.22
Max-Inter-x20	**4.78**	**11.28**	**18.48**	24.5	44.32	**20.18**
Max-Inter-All	3.94	6.73	13.34	16.48	33.42	14.36

Table 5: Average query times (in ms) for top-1000 on quantized indexes, using estimated thresholds.

	Number of query terms					Avg
	2	3	4	5	6+	
ClueWeb09B TREC 2005						
VBMW-T	6.5	13.8	20.41	35.49	75.02	19.44
MaxScore-T	7.63	8.96	11.65	15.98	21.24	10.62
Max-Inter-x2	5.47	6.86	9.22	13.15	19.94	8.42
Max-Inter-x5	4.72	6.3	8.42	12.52	18.96	7.7
Max-Inter-x10	4.36	6.03	7.59	11.48	18.76	7.25
Max-Inter-x15	4.0	5.93	7.36	11.39	18.17	6.98
Max-Inter-x20	**3.82**	**5.86**	**7.29**	**10.56**	**17.21**	**6.71**
Max-Inter-All	2.85	3.97	4.32	6.3	13.32	4.64
ClueWeb09B TREC 2006						
VBMW-T	6.64	14.4	26.29	42.17	108.85	37.99
MaxScore-T	6.55	9.51	14.38	20.29	35.59	16.75
Max-Inter-x2	6.34	7.77	12.52	18.38	35.54	15.53
Max-Inter-x5	5.97	7.12	11.49	16.05	33.63	14.33
Max-Inter-x10	4.59	6.74	10.48	15.35	32.46	13.44
Max-Inter-x15	4.27	6.54	9.85	13.95	31.92	12.87
Max-Inter-x20	**3.78**	**6.36**	**9.52**	**13.49**	**30.96**	**12.41**
Max-Inter-All	2.47	3.78	5.53	8.22	23.3	8.33
ClueWeb12B TREC 2005						
VBMW-T	5.12	10.46	15.17	26.01	61.52	15.12
MaxScore-T	7.34	7.95	9.72	13.64	19.79	9.6
Max-Inter-x2	4.29	5.88	7.8	12.06	18.81	7.26
Max-Inter-x5	3.93	5.23	6.81	11.01	17.48	6.58
Max-Inter-x10	3.71	4.95	6.24	10.67	17.9	6.34
Max-Inter-x15	3.49	4.84	6.12	9.98	17.11	6.07
Max-Inter-x20	**3.34**	**4.77**	**6.04**	**9.59**	**16.53**	**5.89**
Max-Inter-All	2.72	3.32	3.67	5.86	12.29	4.19
ClueWeb12B TREC 2006						
VBMW-T	3.99	10.28	19.42	31.27	82.54	28.25
MaxScore-T	5.01	8.14	12.78	17.12	30.65	14.35
Max-Inter-x2	3.72	6.65	11.59	14.71	29.77	12.93
Max-Inter-x5	3.36	6.33	10.96	13.38	28.54	12.2
Max-Inter-x10	3.09	5.87	9.55	12.57	27.79	11.44
Max-Inter-x15	**2.91**	5.61	8.89	11.78	26.96	10.91
Max-Inter-x20	2.99	**5.3**	**8.51**	**11.51**	**26.84**	**10.69**
Max-Inter-All	2.0	3.22	5.88	7.34	20.87	7.6

shows that the exhaustive method results in only minor improvements in running time that are more than erased by extra selection cost. Thus, potential benefits of using a smarter algorithm for selection are very limited. Note that greedy selection costs are included in all subsequent results. Also, selection is only performed once in our approach, right after threshold estimation. While it might be beneficial to rerun selection once the actual top-k threshold grows beyond the initial estimate during query processing, this also creates additional overheads that limit the possible gains.

4.2 Comparison of all Algorithms

We now compare the performance of our approach to several baselines. We implemented the following MaxScore-based methods:

(1) **MaxScore** is an implementation of the MaxScore algorithm, with initial threshold 0 and thus no threshold estimation.

(2) **MaxScore-T** is a version of Maxscore where threshold estimation is used to select an initial set of essential lists. When the threshold grows beyond the initial estimate, the algorithm recomputes the selection of essential lists.

(3) **Max-Inter-xN** uses the greedy algorithm to select both single-term lists and intersections as essential lists, based

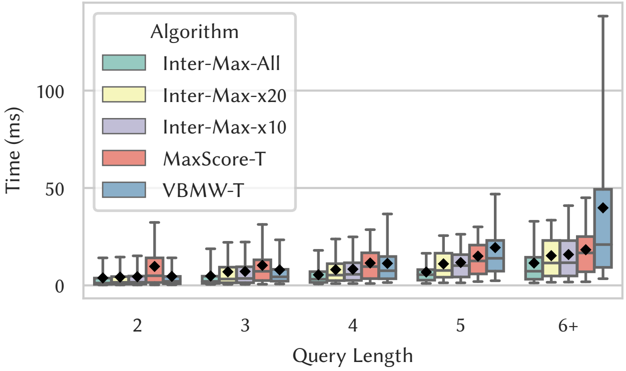

Figure 2: Query times for selected algorithms run on the non-quantized ClueWeb09B index with TREC 2005 queries. The middle bar indicates the median value, while the boxes extend to the first and third quartiles. Whiskers extend to the 5-th and the 95-th percentiles. Means are marked by diamonds. Outliers are removed for better readability.

Table 6: Average query times (in ms) for Max-Inter-x20 (k = 1000) with pairs stored on SSD and in main memory.

		Non-quantized		Quantized	
Collection	Queries	SSD	Memory	SSD	Memory
ClueWeb09B	TREC 2005	14.28	13.98	6.71	6.44
ClueWeb09B	TREC 2006	22.8	22.43	12.41	12.06
ClueWeb12B	TREC 2005	12.55	12.4	5.89	5.64
ClueWeb12B	TREC 2006	20.18	19.47	10.69	10.32

on the initial threshold estimate. The selected structures are then used during the entire query. Intersections are chosen from a large set of precomputed pairwise intersections, as explained in Section 3.2, with a space budget of $N \times |I|$ where $|I|$ is the size of the inverted index. We report results for $N \in \{2, 5, 10, 15, 20\}$. If the selection includes only single-term lists, we use VBMW-T (see below) for two-term queries and MaxScore-T otherwise, as Max-Inter provides no improvements without intersections. Intersections are fetched from SSD, and we include this cost in all reported results.

(4) **Max-Inter-All** is the case of Max-Iter-xN where all pairwise intersections are available, i.e., unlimited space budget.

Essential List Sizes and Lookups. In Table 3, we show two statistics that drive the cost of our approach: the number of essential postings traversed (P_E) and random lookups into non-essential lists (L_{NE}). We see that MaxScore-T significantly reduces both P_E and L_{NE} compared to MaxScore by using an initial threshold estimate. Recall that essential structures in Max-Inter methods are selected to minimize the number of essential postings. Thus, the more precomputed intersections are available, the fewer essential postings are traversed. Moreover, we see that L_{NE} also significantly decreases as P_E gets smaller. While Max-Inter-x20 reduces both statistics significantly over the baselines, Max-Inter-All shows additional reductions of up to 50% for P_E and up to 80% for L_{NE}.

Performance of all Methods. We also implemented two block max-based methods, VBMW [23] and a version of VBMW with initial estimated threshold called VBMW-T. Our experiments showed that MaxScore-T and VBMW-T always significantly outperformed MaxScore and VBMW without initial threshold, and thus in the following we only report numbers for MaxScore-T and VBMW-T.

Tables 4 and 5 show average query times across all methods on non-quantized and quantized indexes, for different query lengths. We see that VBMW-T is fast for very short queries. Also, MaxScore-based methods benefit more from using a quantized index, and for that case significantly outperform VBMW-T. Our new Max-Inter methods achieve promising improvements over the baselines for larger N (such as N = 20), of about 25 to 35%. Improvements are even larger for the idealized case where all intersections are available, indicating that a better algorithm for selecting which intersections to precompute might give further benefits.

In Figure 2 we show the distribution of query processing times for different algorithms as a box-and-whiskers plot. As we see, our new methods do not just outperform the baselines in terms of average times, but also improve mean and 75- and 95-percentile tail latencies.

Further extensive experiments (see Appendix C) show results obtained by using idealized clairvoyant threshold estimates, i.e., estimates equal to the exact threshold obtained at zero cost. This increases the performance advantage of our new methods over baselines to about 30 to 45%, suggesting that our results would benefit from further progress on threshold estimation techniques. Other results how the cost of threshold estimation and essential list selection for the different data sets. Overall, these costs are low (on the order of a few percent) compared to overall query execution costs. Finally, results for different values of k show how our methods do particularly well for larger K, while still obtaining more moderate benefits for small k.

Finally, Table 6 shows the overhead of fetching intersections from SSD. As we see, running times would only be reduced by a small amount if we could hold all intersections in main memory. This justifies our decision to store large sets of precomputed intersections on a low-cost SSD drive.

To summarize, our new results show significant improvements in running time over state-of-the-art baselines, with the potential for additional improvements through better selection of precomputed intersections or improved threshold estimation.

5 DISCUSSION AND CONCLUDING REMARKS

In this paper, we have described a new approach for optimizing safe disjunctive top-k query processing. Our approach is a novel and interesting generalization of the MaxScore algorithm that allows precomputed pairwise intersections to be used as essential lists. The experimental results showed the potential for significant improvements in performance over the standard MaxScore approach. While the improvements required a significant amount of precomputed data structures, we showed that this can be efficiently addressed by storing the structures on a low-cost SSD drive.

ACKNOWLEDGMENTS

This work was partially supported by NSF Grant IIS-1718680 and a grant from Amazon.

REFERENCES

[1] Robin Aly, Djoerd Hiemstra, and Thomas Demeester. 2013. Taily: Shard Selection Using the Tail of Score Distributions. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 673–682.

[2] Paolo Boldi and Sebastiano Vigna. 2006. MG4J at TREC 2006. In *TREC*.

[3] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*. ACM, 426–434.

[4] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. 2011. Interval-based pruning for top-k processing over compressed lists. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 709–720.

[5] Olivier Chapelle and Yi Chang. 2011. Yahoo! learning to rank challenge overview. In *Proceedings of the learning to rank challenge*. 1–24.

[6] Olivier Chapelle, Yi Chang, and Tie-Yan Liu. 2011. Future directions in learning to rank. In *Proceedings of the Learning to Rank Challenge*. 91–100.

[7] Surajit Chaudhuri, Kenneth Church, Arnd Christian König, and Liying Sui. 2007. Heavy-Tailed Distributions and Multi-Keyword Queries. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 663–670.

[8] Matt Crane, J. Shane Culpepper, Jimmy Lin, Joel Mackenzie, and Andrew Trotman. 2017. A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. 201–210.

[9] Nick Craswell, Dennis Fetterly, Marc Najork, Stephen Robertson, and Emine Yilmaz. 2009. Microsoft Research at TREC 2009: Web and relevance feedback tracks. (2009).

[10] Lídia Lizziane Serejo de Carvalho, Edleno Silva de Moura, Caio Moura Daoud, and Altigran Soares da Silva. 2015. Heuristics to improve the BMW method and its variants. *Journal of Information and Data Management* 6, 3 (2015), 178–178.

[11] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. Optimizing Top-k Document Retrieval Strategies for Block-Max Indexes. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. 113–122.

[12] Shuai Ding and Torsten Suel. 2011. 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*. ACM, 993–1002.

[13] Ronald Fagin. 2002. Combining fuzzy information: an overview. *ACM SIGMOD Record* 31, 2 (2002), 109–118.

[14] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Zien. 2011. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment* 4, 12 (2011), 1213–1224.

[15] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*.

[16] Andrew Kane and Frank Wm Tompa. 2018. Split-lists and initial thresholds for WAND-based search. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 877–880.

[17] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[18] Jimmy Lin, Joel Mackenzie, Chris Kamphuis, Craig Macdonald, Antonio Mallia, Michał Siedlaczek, Andrew Trotman, and Arjen de Vries. 2020. Supporting Interoperability Between Open-Source Search Engines with the Common Index File Format. *arXiv preprint arXiv:2003.08276* (2020).

[19] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2020. Pretrained Transformers for Text Ranking: BERT and Beyond. *arXiv preprint arXiv:2010.06467* (2020).

[20] Tie-Yan Liu et al. 2009. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval* 3, 3 (2009), 225–331.

[21] Xiaohui Long and Torsten Suel. 2005. Three-Level Caching for Efficient Query Processing in Large Web Search Engines. In *Proceedings of the 14th International Conference on World Wide Web*. 257–266.

[22] Craig Macdonald, Rodrygo LT Santos, and Iadh Ounis. 2013. The whens and hows of learning to rank for web search. *Information Retrieval* 16, 5 (2013), 584–628.

[23] Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. 2017. Faster BlockMax WAND with variable-sized blocks. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 625–634.

[24] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia.. In *OSIRRC@ SIGIR*. 50–56.

[25] Antonio Mallia, Michał Siedlaczek, and Torsten Suel. 2019. An experimental study of index compression and DAAT query processing methods. In *European Conference on Information Retrieval*. Springer, 353–368.

[26] Antonio Mallia, Michał Siedlaczek, and Torsten Suel. 2021. Fast Disjunctive Candidate Generation Using Live Block Filtering. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*. 671–679.

[27] Antonio Mallia, Michal Siedlaczek, Mengyang Sun, and Torsten Suel. 2020. A comparison of top-k threshold estimation techniques for disjunctive query processing. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2141–2144.

[28] Matthias Petri, Alistair Moffat, Joel Mackenzie, J. Shane Culpepper, and Daniel Beck. 2019. Accelerated Query Processing Via Similarity Score Prediction. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 485–494.

[29] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. 2010. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval* 13, 4 (2010), 346–374.

[30] Corby Rosset, Damien Jose, Gargi Ghosh, Bhaskar Mitra, and Saurabh Tiwary. 2018. Optimizing Query Evaluations Using Reinforcement Learning for Web Search. In *The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1193–1196.

[31] Milad Shokouhi and Justin Zobel. 2009. Robust result merging using sample-based score estimates. *ACM Transactions on Information Systems (TOIS)* 27, 3 (2009), 1–29.

[32] Trevor Strohman and W. Bruce Croft. 2007. Efficient Document Retrieval in Main Memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 175–182.

[33] Paul Thomas and Milad Shokouhi. 2009. SUSHI: Scoring Scaled Samples for Server Selection. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 419–426.

[34] Gabriel Tolosa, Esteban Feuerstein, Luca Becchetti, and Alberto Marchetti-Spaccamela. 2017. Performance improvements for search systems using an integrated cache of lists+ intersections. *Information Retrieval Journal* 20, 3 (2017), 172–198.

[35] Howard Turtle and James Flood. 1995. Query evaluation: strategies and optimizations. *Information Processing & Management* 31, 6 (1995), 831–850.

[36] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A Cascade Ranking Model for Efficient Ranked Retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 105–114.

[37] Qi Wang, Constantinos Dimopoulos, and Torsten Suel. 2016. Fast First-Phase Candidate Generation for Cascading Rankers. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 295–304.

[38] Qi Wang and Torsten Suel. 2019. Document reordering for faster intersection. *Proceedings of the VLDB Endowment* 12, 5 (2019), 475–487.

[39] Erman Yafay and Ismail Sengor Altingovde. 2019. Caching Scores for Faster Query Processing with Dynamic Pruning in Search Engines. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. ACM, 2457–2460.

[40] Min Zhang, Da Kuang, Guichun Hua, Yiqun Liu, and Shaoping Ma. 2009. Is learning to rank effective for Web search?. In *SIGIR 2009 workshop: Learning to Rank for Information Retrieval*. 641–647.

[41] Wanwan Zhou, Ruixuan Li, Xinhua Dong, Zhiyong Xu, and Weijun Xiao. 2015. An Intersection Cache Based on Frequent Itemset Mining in Large Scale Search Engines. In *Proceedings of the 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. 19–24.

A ALGORITHMIC DETAILS

Algorithm 1 shows the pseudocode of the "eager" MaxInter algorithm. The UnionLookup algorithm used in lines 9 and 11 is a slightly modified version of MaxScoreT, where the essential and non-essential lists stay the same throughout the entire run of the algorithm. This modification generalizes it to work with precomputed intersections of of an arbitrary number of terms.

Algorithm 2 shows the pseudocode of the "state" MaxInter algorithm. Algorithm 3 illustrate the process of generating the two auxiliary tables used in the "state" algorithm. These tables are precomputed once per query.

Algorithm 1 Eager Inter algorithm

```
 1: $E \leftarrow$ essential query terms
 2: $N \leftarrow$ non-essential terms sorted by decreasing maxscore
 3: $I \leftarrow$ set of selected pair intersections
 4: $U \leftarrow$ set of selected single-term lists
 5: $\theta \leftarrow$ estimated score threshold
 6: function EagerInter($E$, $N$, $I$, $U$, $k$, $\theta$)
 7:     if $I = \emptyset$ then                    ▷ no pair intersections were selected
 8:         return MaxScore($E$, $N$, $\theta$)            ▷ fall back to MaxScore
 9:     $R \leftarrow$ UnionLookup($U$, $N - U$, $\theta$)
10:     for $L \in I$ do
11:         $R \leftarrow R$ + UnionLookup($L$, $N - L$, $\theta$)
12:     $R \leftarrow$ RemoveDuplicateDocs(R)        ▷ always keep higher score
13:     $R \leftarrow$ SortByScoreDescending(R)
14:     return $\{r_1, r_2, \ldots, r_k\}$
```

Algorithm 2 State Inter algorithm

```
$l_e \leftarrow$ DAAT union of the selected essential posting lists
$L_n \leftarrow$ list of the selected non-essential posting lists
$M \leftarrow$ maximum possible score table
$N \leftarrow$ next lookup table
$|q| \leftarrow$ number of terms in the query
function StateInter($l_e$, $L_n$, $M$, $N$, $|q|$)
    $R \leftarrow$ size-bounded min-heap for accumulating top results
    $M \leftarrow (1 \ll |q|) - 1$
    $S_0 \leftarrow |E| \ll |q|$
    while NonEmpty($U$) do
        $d \leftarrow$ DocId($l_e$)
        $s \leftarrow$ Score($l_e$)
        $S \leftarrow$ State($l_e$) $| \; (|E| \ll |q|)$
        $n \leftarrow N[S]$
        while $n \geq 0$ and WouldEnter($R$, $s + M[S]$) do
            $l \leftarrow L[n - |E|]$
            NextGEQ($l$)
            if DocID($l$) = $d$ then
                $s \leftarrow s+$ Score($l$)
                $S \leftarrow S \; | \; (1 \ll n)$
            $S \leftarrow (s \;\&\; ((1 \ll |q|) - 1)) + ((n+1) \ll |q|)$
            $n \leftarrow N[S]$
function State($l_e$)
    $s \leftarrow 0$
    for $i = 0, 1, \ldots, |q|$ do
        if term $i$ is part of union $l_e$ then
            $s \leftarrow s \; | \; (1 \ll i)$
    return $s$
```

Algorithm 3 Computation of lookup tables

```
$t_0, t_1, \ldots, t_{k-1} \leftarrow$ essential query terms
$t_k, t_{k+1}, \ldots, t_{n-1} \leftarrow$ non-ess. terms sorted by decreasing maxscore
$I \leftarrow$ set of selected pair intersections
function Unnecessary(p, s)
    if $((1 \ll p) \;\&\; s) > 0$ then return true
    for $k \leftarrow \{k | (p, k) \in I\}$ do
        if $((1 \ll k) \;\&\; s) > 0$ then return true
    return false
function PrecomputeNextLookup
    for $t = k, k+1, \ldots, n-1$ do
        for $s = 0, 1, \ldots, 2^n - 1$ do
            $p \leftarrow t$
            while $p < n \wedge$ Unnecessary($p$, $s$) do
                $p \leftarrow p + 1$
            if $p = t$ then $p \leftarrow -1$
            $N_{(t \ll n)+s} \leftarrow p$
    return $N$
function MaximumPossibleScore
    for $i = n, n-1, \ldots, 0$ do
        for $j = 2^n - 1, 2^n - 2, \ldots, 0$ do
            $s \leftarrow (i \ll n) + j$
            $n_t \leftarrow N_s$
            if $n_t = -1$ then $M_s \leftarrow 0$
            else
                $i_a \leftarrow (n_t + 1) \ll n) + (j \; | \; (1 \ll n_t))$
                $a \leftarrow$ MaxScore(i) + $M_{i_a}$
                $i_b \leftarrow (i + 1) \ll n) + j$
                $b \leftarrow M_{i_b}$
                $M_s \leftarrow \max(a, b)$
    return $M$
```

B THRESHOLD ESTIMATION

In Table 7, we can see mean under-prediction fraction (MUF) results for several k values, and for different query lengths when thresholds are estimated on ClueWeb09B and for TREC 2005. Average threshold estimation cost in microseconds when thresholds are estimated on ClueWeb09B and for TREC 2005 are shown in Table 8.

Table 7: MUF of threshold estimates for different query lengths and k on ClueWeb09B, for TREC 2005 queries.

k	2	3	4	5	6+	avg
10	0.87	0.87	0.87	0.86	0.84	0.86
100	0.92	0.91	0.90	0.90	0.89	0.90
1000	0.96	0.95	0.94	0.95	0.95	0.95
10 000	0.96	0.97	0.96	0.96	0.96	0.96

Table 8: Average cost (in µs) of threshold estimates on ClueWeb09B, for TREC 2005 queries.

k	2	3	4	5	6+	avg
10	110	110	105	148	184	119
100	119	121	124	174	226	135
1000	150	151	176	227	317	177
10 000	203	242	303	399	558	279

C CLAIRVOYANT THRESHOLD ESTIMATES

Tables 9 and 10 show average query times across all methods on non-quantized and quantized indexes, for different query lengths when clairvoyant threshold estimates are employed. All methods benefits from a better threshold estimate. When compared to Tables 4 and 5, we can see that our proposed methods, Max-Inter, achieve even better improvements over the baselines, sometimes requiring a smaller space budget to obtain a similar speedup.

Table 9: Average query times (in ms) for $k = 1000$ on non-quantized ClueWeb09B and ClueWeb12B indexes, using clairvoyant (exact) thresholds.

	Number of query terms					Avg
	2	3	4	5	6+	
ClueWeb09B TREC 2005						
VBMW-T	8.85	14.9	20.47	35.03	69.14	20.1
MaxScore-T	19.04	19.54	21.84	28.24	32.84	21.77
Max-Inter-x2	9.12	15.37	17.96	24.24	29.76	15.27
Max-Inter-x5	8.65	13.72	16.26	22.69	28.9	14.19
Max-Inter-x10	8.43	12.89	14.99	20.62	27.18	13.36
Max-Inter-x15	**8.28**	**12.67**	14.68	20.51	26.4	**13.11**
Max-Inter-x20	8.62	12.95	**14.56**	**18.99**	**26.13**	13.14
Max-Inter-All	7.31	8.26	9.62	12.04	19.73	9.47
ClueWeb09B TREC 2006						
VBMW-T	7.57	15.23	26.62	40.76	99.33	36.42
MaxScore-T	15.57	19.1	25.44	33.72	50.87	28.27
Max-Inter-x2	7.43	15.82	22.36	29.78	48.42	24.28
Max-Inter-x5	7.14	14.5	20.62	25.85	44.96	22.21
Max-Inter-x10	5.91	13.53	19.2	24.63	43.46	20.96
Max-Inter-x15	5.73	13.11	19.08	23.87	44.08	20.79
Max-Inter-x20	**5.66**	**12.73**	**17.85**	**22.71**	**41.74**	**19.77**
Max-Inter-All	4.48	7.27	11.36	14.8	31.34	13.47
ClueWeb12B TREC 2005						
VBMW-T	7.41	11.34	15.98	26.24	57.7	16.07
MaxScore-T	18.24	19.51	19.88	24.21	30.3	20.5
Max-Inter-x2	7.35	13.22	15.56	21.41	27.3	13.15
Max-Inter-x5	7.38	11.84	14.05	20.11	25.9	12.34
Max-Inter-x10	7.44	11.32	12.84	19.44	24.51	11.86
Max-Inter-x15	7.3	10.85	13.72	19.34	27.48	12.1
Max-Inter-x20	**7.21**	**10.66**	**12.46**	**17.42**	**23.76**	**11.3**
Max-Inter-All	6.77	7.06	8.2	10.91	16.81	8.35
ClueWeb12B TREC 2006						
VBMW-T	5.03	10.93	19.78	30.21	76.21	27.32
MaxScore-T	13.5	16.27	23.28	29.76	44.73	24.93
Max-Inter-x2	4.89	13.28	20.44	26.08	42.0	20.98
Max-Inter-x5	4.66	12.57	19.58	23.88	39.95	19.83
Max-Inter-x10	4.52	11.45	17.71	22.55	37.69	18.45
Max-Inter-x15	4.47	10.72	16.9	21.24	37.15	17.75
Max-Inter-x20	**4.44**	**10.2**	**16.07**	**20.79**	**36.84**	**17.29**
Max-Inter-All	3.58	5.75	11.36	13.89	26.96	11.97

Table 10: Average query times (in ms) for top-1000 on quantized ClueWeb09B and ClueWeb12B indexes, using clairvoyant (exact) thresholds.

	Number of query terms					Avg
	2	3	4	5	6+	
ClueWeb09B TREC 2005						
VBMW-T	6.1	12.71	18.18	31.41	63.19	17.19
MaxScore-T	7.35	8.31	10.43	14.33	18.05	9.71
Max-Inter-x2	5.19	6.9	8.2	11.7	16.33	7.68
Max-Inter-x5	4.47	6.22	7.35	10.72	15.41	6.92
Max-Inter-x10	4.14	5.89	6.58	9.5	14.44	6.38
Max-Inter-x15	3.76	5.81	6.48	9.54	13.95	6.15
Max-Inter-x20	**3.64**	**5.78**	**6.28**	**8.47**	**13.68**	**5.93**
Max-Inter-All	2.79	3.86	3.4	5.12	10.4	4.06
ClueWeb09B TREC 2006						
VBMW-T	5.98	13.31	23.5	36.51	91.92	32.88
MaxScore-T	8.4	8.8	12.75	17.56	30.66	15.14
Max-Inter-x2	5.76	7.11	10.8	15.23	29.54	13.24
Max-Inter-x5	5.43	6.53	9.73	12.4	27.04	11.87
Max-Inter-x10	4.07	6.11	8.9	11.69	25.87	11.01
Max-Inter-x15	3.81	5.89	8.4	10.53	25.41	10.52
Max-Inter-x20	**3.5**	**5.71**	**8.07**	**10.43**	**24.24**	**10.11**
Max-Inter-All	2.16	3.25	4.46	5.97	17.53	6.46
ClueWeb12B TREC 2005						
VBMW-T	4.73	9.76	13.86	23.49	53.0	13.56
MaxScore-T	7.01	7.42	8.88	12.17	16.45	8.76
Max-Inter-x2	3.98	5.89	7.1	10.34	14.83	6.5
Max-Inter-x5	3.78	5.21	6.17	9.34	13.87	5.93
Max-Inter-x10	3.58	4.95	5.53	8.9	12.99	5.56
Max-Inter-x15	3.39	4.8	5.44	8.24	12.89	5.36
Max-Inter-x20	**3.26**	**4.72**	**5.39**	**7.92**	**12.54**	**5.22**
Max-Inter-All	2.65	3.21	3.05	4.79	8.74	3.6
ClueWeb12B TREC 2006						
VBMW-T	3.6	9.35	17.42	27.17	70.21	24.51
MaxScore-T	4.75	7.47	11.35	15.09	26.09	12.63
Max-Inter-x2	3.46	5.93	9.78	12.7	24.55	10.99
Max-Inter-x5	3.07	5.67	9.13	11.14	22.85	10.13
Max-Inter-x10	2.81	5.14	8.09	10.54	21.33	9.34
Max-Inter-x15	**2.63**	4.8	7.74	9.85	20.98	8.96
Max-Inter-x20	2.75	**4.56**	**7.2**	**9.59**	**20.91**	**8.74**
Max-Inter-All	1.75	2.65	4.5	5.84	14.82	5.72