Information Processing and Management 60 (2023) 103359

Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/ipm

Many are Better than One: Algorithm Selection for Faster Top-K Retrieval

Gabriel Tolosa [a,*], Antonio Mallia [b,1]

[a] Departamento de Ciencias Básicas, Universidad Nacional de Luján, Buenos Aires, Argentina
[b] New York University, NY, United States

ARTICLE INFO

Keywords:
Query processing
Web search
Dynamic pruning
Efficiency

ABSTRACT

Large-scale search engines have become a fundamental tool to efficiently access information on the Web. Typically, users expect answers in sub-second time frames, which demands highly efficient algorithms to traverse the data structures to return the top-k results. Despite different top-k algorithms that avoid processing all postings for all query terms, finding one algorithm that performs the fastest on any query is not always possible. The fastest average algorithm does not necessarily perform the best on all queries when evaluated on a per-query basis. To overcome this challenge, we propose to combine different state-of-the-art disjunctive top-k query processing algorithms to minimize the execution time by selecting the most promising one for each query. We model the selection step as a classification problem in a machine-learning setup. We conduct extensive experimentation and compare the results against state-of-the-art baselines using standard document collections and query sets. On ClueWeb12, our proposal shows a speed-up of up to 1.20x for non-blocked index organizations and 1.19x for block-based ones. Moreover, tail latencies are reduced showing proportional improvements on average, but a resulting dramatic decrease in latency variance. Given these findings, the proposed approach can be easily applied to existing search infrastructures to speed up query processing and reduce resource consumption, positively impacting providers' operative costs.

1. Introduction

Web search engines enable users to find content available on the web trying to match their information needs. In general, users issue keyword queries, expecting that the search engine responds with a result page in a limited time, namely, in less than half a second. However, the number of available documents (in a broad sense) on the web has skyrocketed during the last decades, so the task has become extremely challenging. Although the hardware is getting less expensive and more powerful every day, the web is an evolving system, and the size of the data and the number of searches is growing at an even faster rate (Bosch, Bogers, & Kunder, 2016).

To answer tens of thousands of user queries per second with sub-second response times large-scale web search engines need the most advances in algorithms and techniques. In practice, answering queries at a proper speed is vital to a commercial web search engine. The giant volume of the web suggests that improvements in query processing are critical to meet demands, so deploying efficient search techniques is mandatory to manage the cost of the hardware infrastructures. Furthermore, even small increases in

* Corresponding author.
E-mail address: tolosoft@unlu.edu.ar (G. Tolosa).
URL: https://tolosoft.gitlab.io/ (G. Tolosa).
[1] Work done prior to joining Amazon.

https://doi.org/10.1016/j.ipm.2023.103359
Received 20 September 2022; Received in revised form 13 March 2023; Accepted 19 March 2023

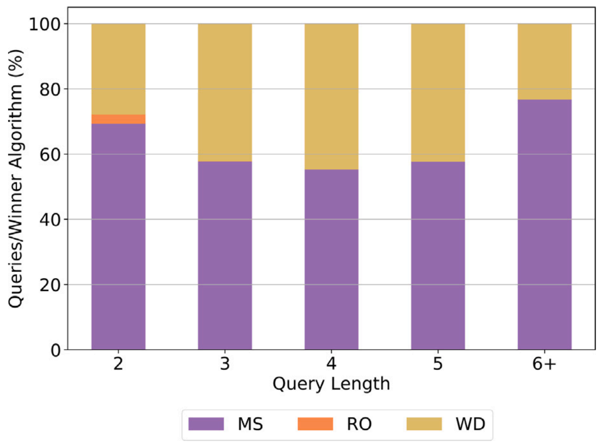


Fig. 1. Winner algorithms by query length (x-axis) using the Clueweb12 documents collection and TREC 2005 Terabyte Track Efficiency Task queries (Clarke, Scholer, & Soboroff, 2005). These results correspond to Ranked_OR (RO), Maxscore (MS), and WAND (WD) algorithms and $k = 1000$.

query processing efficiency can translate into considerable monetary savings in data centers hardware and energy costs (Barroso, Clidaras, & Hölzle, 2018). On the users' side, high response times prevent them from issuing more queries. This situation results in fewer ads posted on result pages by the search provider, reducing revenues (Schurman & Brutlag, 2009), and also decreasing the users' engagement with the search engine in the long term (Arapakis, Bai, & Cambazoglu, 2014).

Query processing in search engines is a fairly complex process. Search engines use a cascading approach to solve a query that starts from simple ranking functions (such as BM25 Robertson & Jones, 1976) that process millions of documents and return a much smaller set of ranked candidates (top-k) that are then evaluated by increasingly more complex functions (Chen, Gallagher, Blanco, & Culpepper, 2017; Wang, Lin, & Metzler, 2011). In general, first-stage ranking algorithms quantify the contribution of each query term to documents while successive functions improve the ranking to form the final result set (i.e. to build a SERP).

Different enhancements in query processing techniques have been developed over several decades of research to improve the first stage of the retrieval pipeline, resulting in many algorithms performing several optimizations to maximize efficiency (one among all is dynamic pruning), such as Maxscore (Turtle & Flood, 1995) and WAND (Broder, Carmel, Herscovici, Soffer, & Zien, 2003) and their variants (Chakrabarti, Chaudhuri, & Ganti, 2011; Dimopoulos, Nepomnyachiy, & Suel, 2013; Ding & Suel, 2011; Jonassen & Bratsberg, 2011; Mallia, Ottaviano, Porciani, Tonellotto, & Venturini, 2017; Mallia & Porciani, 2019; Shan, Ding, He, Yan, & Li, 2012; Strohman, Turtle, & Croft, 2005).

User queries answered by web search engines can be extremely diverse and, regardless of their characteristics, they need to operate under similarly strict latency constraints. In particular, a common requirement is to reduce the high-percentile (e.g. 90th, 95th, or 99th percentile) response time (also known as the tail of the latency distribution) which is often considered as important as the mean response time (Dean & Barroso, 2013). This issue exacerbates with higher cut-off values (also referred to as k) since the running times show a significant increase due to the larger number of retrieved documents (Mallia, Siedlaczek and Suel, 2019).

Many researchers have focused their attention on developing algorithms able to increase query processing efficiency by exploiting specific data patterns and distributions. Finding an algorithm that is the on all queries is not always possible, with some of them being extremely fast on average but showing long-tail latency. Others, on the other hand, show more stable behavior while being slower on average.

1.1. Motivating scenario

The intuition behind this work comes from the observation that there is no single algorithm that performs best for all queries submitted to a search engine, making any fixed selection of the retrieval algorithm to adopt a sub-optimal choice. This is particularly true for real-world commercial search engines which surface a wide variety of queries, making the problem even harder. Different query processing features, such as the number of query terms to process or the number of documents that need to be scored, impact on the final response time. As a motivating example, we executed three well-known disjunctive query processing algorithms (exhaustive processing or Ranked_OR, Maxscore, and WAND) using standard datasets of documents and queries (and a set of fixed k values) and reported what algorithm performs the best, namely the *winner*. Fig. 1 shows the results according to different query lengths.

The percentage of queries that each algorithm processes faster than others changes according to the query lengths. An interesting observation is that Ranked_OR, an exhaustive algorithm that does not make use of any optimizations, is still useful for short queries. An analogous experiment could be made by varying the number of returned results (k) and we would observe similar results, showing no outright winner.

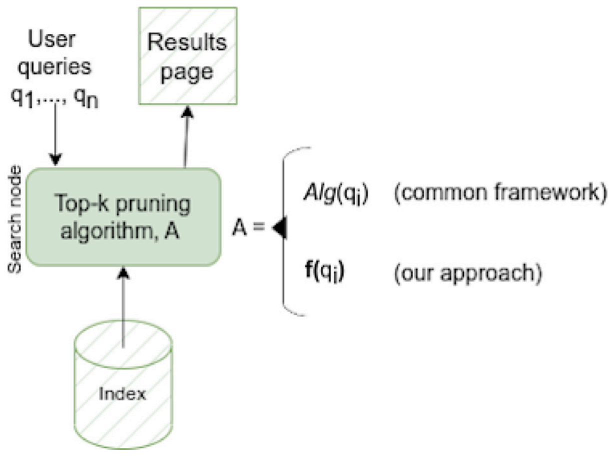


Fig. 2. Overview picture of our proposal. A common approach is using a single top-k algorithm (*Alg*, for example, Maxscore) for all queries. On the other hand, our work proposes to define a function ($f(q_i)$) on a learning framework that selects a particular one on a query basis.

1.2. Objective

The main objective of this work is to speed up top-k selection by combining different disjunctive query processing algorithms in a safe setting, that is, by producing the exact outcome of an exhaustive algorithm. We propose an approach that minimizes the execution time by selecting the most promising method for each submitted query. Fig. 2 presents the general idea. The method relies on a supervised machine-learned framework that infers the algorithm to use given a set of query features.

1.3. Contributions

The contributions of our work can be summarized as follows:

- We model the algorithm selection as a classification problem, carefully engineering and selecting the necessary features in order to estimate the query execution cost. We perform the classification using different tree-based methods of increasing complexity and performance and build a final voting scheme combining them.
- Furthermore, we propose an approach that combines all the methods and configurations we experiment with.
- Finally, we perform an extensive analysis of average and tail running times using standard datasets according to different retrieval methods and index organization.

We also report a detailed experimental evaluation of our proposal comparing it to a range of baseline approaches and show that:

- Our algorithms selection approach shows a speed-up of up to 1.20x for non-blocked index organizations and 1.19x for block-based ones ClueWeb12, with a higher impact on tail queries and logs containing longer queries (TREC 2006 Büttcher, Clarke, & Soboroff, 2006 Track Efficiency Task).
- Our approach also improves tail latencies, in particular when non-block-based algorithms are employed and large k values are set.
- Combining all the methods and variables we work with offers another step of improvement according to the size of the problem. That is, a bigger value of k and long-running queries obtain more significant speedups (1.37x).

In contrast to existing work which attempts to predict long-running queries using features available at indexing time for accelerating tail queries (Hwang, Kim, He, Elnikety, & Choi, 2016; Jeon et al., 2014), we aim at predicting the most suitable query processing algorithm for a target query. Furthermore, additional works exist (Broccolo et al., 2013; Tonellotto, Macdonald, & Ounis, 2013) aiming at selecting or modifying the query processing algorithm depending on the load of the system.

Fang, Marbach, Wang, and Liu (2020) appears to be the most related work to ours. While it presents a similar framework, we consider different algorithmic and data configurations whose details offer deeper insights into the applicability of the methods. In particular, we explore two possible index organizations that have different space requirements and are used by different pruning techniques. This setup may be particularly useful for commercial, web-scale search architectures that may not benefit from the advantages of a block-based inverted index. Besides, we provide a detailed description of the prediction framework using new features and four classification models together with the main relevant parameters that help to explain the results. We train and test the models using different query sets, which facilitate the generalization of the predictions. We also explore a wide range of k values that enable the implementation of two-stage architectures (including a re-ranking one) and test our framework on document collections of different sizes, which has implications for the achieved improvements. Finally, we consider only safe techniques which lead to exact results without reducing effectiveness.

Our results are also helpful for deploying more efficient industry-scale search infrastructures. For example, Lucene ([2]), which has become a widely adopted platform in the industry for building search applications, uses Block-Max WAND and/or Maxscore algorithms (depending on the version) (Grand, Muir, Ferenczi, & Lin, 2020) and may benefit from using our approach.

The remainder of the paper is organized as follows. Section 2 introduces background concepts and summarizes related work. We then define our research problem and introduce the proposed approach (Section 3). In Section 4, we present the experimental evaluation framework while the results are shown and discussed in Section 5. We state some practical implications of this work in Section 6. Finally, we conclude and discuss future work in Section 7.

2. Background and related work

In order to achieve fast response times on user queries, search engines rely on compressed data structures (i.e. inverted index) and efficient query processing techniques that dynamically skip portions of the data to speed up computation. We now briefly review some of the key concepts and techniques and provide essential technical background about the main algorithms and data structures.

2.1. Inverted index

An inverted index I is a fundamental data structure used in most current large-scale text search systems. It is composed of posting lists I_t, one for each distinct term t in a document collection C (Zobel & Moffat, 2006). The set of unique terms forms the vocabulary (V) of C. A posting list is a sequence of document identifiers (ID) where the document (d) contains the corresponding term (t), usually along with respective in-document frequencies ($f_{t,d}$) or other information required for ranking purposes.

Given the considerable size of the collections indexed by current search engines, even a single node of a large search cluster typically contains billions of pairs of integers representing the document ID and the corresponding frequency. Posting lists are usually compressed with algorithms that trade space usage with access time (Pibiri & Venturini, 2020) and can allow index structures to fit in the main memory. Despite the trade-off between effectiveness and efficiency, small index sizes are still highly desirable. Postings in an inverted index are often stored in increasing order of their IDs since it improves compression ratio and allows safe skipping to reduce posting list traversal cost.

An important optimization is represented by document identifiers reassignment, clustering similar documents appearing close together in the document identifier space. While new and promising document reordering techniques are available (Dhulipala et al., 2016; Mackenzie, Mallia, Petri, Culpepper, & Suel, 2019), a common heuristic involves ordering documents by their URL (Silvestri, 2007). Other document-ordering layouts exist, such as impact-ordered indexes, which have received some attention (Lin & Trotman, 2015). In this case, the postings are ordered by decreasing impact score, which indicates the contribution of a term t in a document d. These scores are normally quantized into a small integer range (Anh & Moffat, 2006).

Some approaches propose splitting the inverted index into more than one tier to accelerate query processing (multi-tier). The general idea is to start processing a query on a small portion of the inverted index (highest level tier) that stores documents considered as more relevant ones. The algorithm only accesses and processes the next tier if the result set is unsatisfactory. Important performance improvements emerge when the algorithm answers the query without accessing deeper tiers. In this case, there is no guarantee that the results are the same compared to the exhaustive query processing (non-safe). The works of Daoud et al. (2016) and Ntoulas and Cho (2007) use this approach while facing the safety drawback. In the first case, the authors combine a multi-tier architecture with a mechanism to create a pruned index (first tier) that guarantees that the top results do not suffer any quality degradation for most queries (up to 73% of the queries with a tier of 30% of the full size). In the second case, the author split computation into three phases. The first one runs on a small tier and selects documents that are candidates to be present in the top results. This strategy also enables computing a discarding threshold to use later. In the second phase, the method traverses the second tier to find missing occurrences of documents in the first tier (the two tiers are disjoint) and to compute the complete scores of all the candidate documents. After that, the algorithm checks whether the result provided by the second phase preserves the top-k ranking results, and, when needed, it performs a second pass in the second tier to update just the top-k entries. The authors show that their new algorithm considerably improves the performance over the baseline when computing top-10 results.

2.2. Document scoring

To perform top-k document retrieval an inverted index is traversed collecting relevant documents w.r.t. a user's query. Results are reported to the user ranked from higher to lower relevance score. In simple Boolean retrieval, a query expressed as a set of terms can be processed in conjunctive (AND) or disjunctive (OR) modes, retrieving the documents that contain all the terms or at least one of them, respectively. Although the former is significantly faster to compute, it is also more restrictive and thus can miss some highly relevant documents. On the other hand, disjunctive queries can potentially produce an extremely large list of unranked results, where only a few are highly relevant and hard to find as they do not appear in any specific order.

[2] https://lucene.apache.org/.

For this reason, document scoring is used to produce a list of ranked documents (higher to lower score), where the relevance score is a function of the query-document pair. In disjunctive query processing, only documents that contain at least one query term are retrieved, so documents that do not contain any are assumed to have a score equal to 0.

Typically, the document scoring function must be computationally inexpensive since it is invoked repeatedly for all the disjunctive query reports. Popular scoring functions rely on the *query term independence assumption*, which allows computing term-document scores for the query terms matching in the document and are then linearly combined. This function can be expressed as follows:

$$s(q,d) = \sum_{t \in q \cap d} s_{t,d}$$

where $s_{t,d}$ are scores for each term-document pair usually computed from the frequency and some extra data about the inverted list and document lengths.

2.3. Disjunctive top-k query processing

Much research has been devoted to accelerating disjunctive query processing. In disjunctive top-k retrieval, we are interested in retrieving only the k highest-scored documents in the collection instead of returning all the matching documents. In this case, it is possible to perform index pruning to eliminate parts of an inverted index from query processing. While there exist solutions to perform this task statically at index construction, *dynamic pruning* (sometimes also referred to as early termination) does it on the fly at query time. We consider early termination techniques to be *safe* if they produce the exact outcome of an exhaustive algorithm and, so, if the result would be the same as it would be without pruning the index.

There are two primary schemes to traverse posting lists within an index: Term-at-a-Time (TAAT) and Document-at-a-Time (DAAT). In the first case, the results are obtained by sequentially traversing each posting list while maintaining intermediate scores in memory accumulators. On the other hand, in a DAAT strategy, the query term posting lists are processed in parallel, and the score of each document is fully computed before moving to the next document, thus no auxiliary per-document data structures are necessary. In this work, we limit ourselves to safe Document-At-A-Time algorithms for disjunctive top-k queries as they are more amenable to dynamic pruning techniques.

Score-at-a-Time (SAAT) is another disjunctive top-k query processing approach that takes advantage of impact-sorted indexes (Crane, Culpepper, Lin, Mackenzie, & Trotman, 2017). This strategy processes blocks of postings in decreasing impact score order. In a SAAT approach, it is possible to stop processing when the remaining documents cannot make it into the final top-k heap (according to their impact scores) to prevent scoring all the documents that contain at least one query term. This stopping criterion avoids scoring many unuseful documents but does not guarantee that all previous documents are fully scored. For this reason, this technique is considered non-safe. JASS (Lin & Trotman, 2015) is a modern retrieval engine that implements the SAAT evaluation strategy.

Maxscore

The DAAT MaxScore (Turtle & Flood, 1995) is a safe algorithm that relies on the maximum possible contribution of each term (i.e. term upper bounds or maxscores) to perform fewer document scoring operations. During query processing, the score of the k ranked document, θ, becomes large enough to allow pruning candidate documents that only appear in the posting list with small accumulated term upper bounds contribution (safe skipping). In this way, Maxscore computes the full score only on those documents that can make it into the top-k list ($score_d > \theta$). As the query processing proceeds the threshold increases and the pruning becomes more aggressive.

The algorithm starts with the posting lists of each term in the current query and the information of the upper bounds of each one, γ_t, (sorted in n increasing order). Then, it computes a *pivot index* that corresponds to the terms whose accumulated upper bounds exceed the threshold θ. This pivot splits the posting lists into essential and non-essential lists.

MaxScore retrieves candidate documents to score from the essential lists only and uses the non-essential lists to perform a direct lookup for exact scoring computation. That is, each document d_i that appears in the essential lists is fully scored by moving the corresponding cursors of each list to d_i (or greater) and accumulating the score (essential lists first and then the non-essential ones). Once d_i is fully processed, its score is compared against θ to enter into the top-k heap (or not). In the positive case, the algorithm updates θ and computes a new pivot index.

WAND

WAND (*weak*, or *weighted and*) is a safe DAAT strategy for disjunctive query processing proposed by Broder et al. (2003). This approach also relies on term upper bounds and maintains a heap of scores to keep the top-k documents with the largest ones. The score of the kth document in a heap (θ) acts as a discarding threshold.

WAND performs a two-level evaluation approach to prevent computing a similarity score for every appearing document in the considered posting lists. In the first level, WAND reorders the candidate postings by increasing document identifiers (docID). It processes the lists (in that order) to determine the first document for which the sum of the terms' scores exceeds the threshold θ (this is called the *pivot term*).

The algorithm iterates in parallel over all query term postings and identifies candidate documents using a preliminary evaluation using the WAND operator.

It takes as input a list of n boolean variables $(X_0, \ldots, Xn-1)$, a list of n associated weights $(w_0, \ldots, w_{n-1})$ and the threshold θ. By definition, WAND becomes true if and only if $\sum_{i=0}^{(n-1)} w_i x_i \geq \theta$

where x_i is the indicator variable for X_i (x_i is equal to 1 if X_i is true, and 0 otherwise).

The docIDs smaller than the pivots' will never accumulate enough term upper bounds to enter the top-k heap. The documents that meet the condition of the first phase (pivot docIDs) undergo a full evaluation (second-level) that computes the actual scores (i.e. by applying BM25) and might be included in the current top-k heap. Then, the cursors move forward, and the terms are resorted, getting the next pivot term. The algorithm safely finishes when it cannot find a new candidate document that can surpass the current value of θ.

BlockMax WAND

Both Maxscore and WAND use a global per-term upper bound (or maxscore) for the whole list processing. This upper bound could be larger than most of the (average) score contribution of a given term, thus limiting the possibility to avoid scoring documents unnecessarily.

The introduction of block-max indexes a decade ago by Ding and Suel (2011) tackles this problem enabling more opportunities to skip portions of the index. Posting lists are often split into constant-sized blocks (i.e. 64 or 128 docIDs) for compression purposes. The authors propose to augment the index structure by storing the uncompressed maximum docID, the maxscore, and the size of each block in a table. Based on the logic of WAND, the new algorithm uses these block-based, more accurate upper bounds that allow a more aggressive (and still safe) posting skipping, thus avoiding the decompression of unnecessary blocks. This new approach is known as Block-Max WAND (BMW). Extending this work, other researchers (Mallia et al., 2017) introduce a refinement that uses variable-sized blocks, intending to consider the regularities of the scores in the lists. This approach, called Variable BMW, improves the query processing times, and it is considered one of the state-of-the-art safe dynamic pruning techniques.

Khattab, Hammoud, and Elsayed (2020) present an exciting work that analyzes WAND and MaxScore-based strategies under different ranking models. They consider the strengths and weakness of each strategy under different configurations, propose another simple query evaluation strategy called LazyBM and claim that it outperforms all of the tested strategies in both mean and tail query latency. This work builds upon the observation that neither WAND-based nor Maxscore-based techniques perform the best in all cases.

In a recent paper, Shao, Qiao, Ji, and Yang (2021) propose an approach to boost the performance of BlockMax Wand (and two of its variants) on a window-based index navigation scheme. Their strategy modifies the top-k search flow using adaptive probing to tighten the threshold bound earlier. The evaluation of the proposal shows an improved speed for short queries with a limited k value.

On the other hand, Mallia, Siedlaczek, and Suel (2021) focus on disjunctive top-k queries but for larger k (thousands or tens of thousands of documents). Their approach relies on a live-block filtering approach, taking advantage of the SIMD capabilities of modern CPUs. They analyze the space overhead required to store pre-computed (and required) block-max data and offer computational time trade-offs. The proposed method outperforms the standard ones (without the live-block mechanism) on large values of k.

Finally, we mention a very recent paper by Mackenzie, Petri, and Moffat (2022) that deals with retrieving the second page of top-k results in search applications efficiently. This work assumes that only a fraction of submitted queries requires the second page of results. They propose and evaluate methods based on the Document-at-a-Time evaluation strategy and BM25 scoring functions to address the question. Similarly to our work, they consider Wand and Block-Max Wand top-k algorithms.

2.4. Threshold initialization

Another important orthogonal innovation that speeds up query processing is top-k threshold initialization. It proposes to use initial values for the top-k threshold (θ) to allow a more aggressive skipping. A common approach is to get an approximation of the final value of the threshold at the beginning of the evaluation process. An accurate initial estimate of the top-k threshold (minimizing or avoiding overestimations) allows top-k algorithms such as Maxscore or Block-Max WAND to speed up processing, according to their evaluation strategy.

In recent work, Kane and Tompa (2018) examine the use of initial thresholds using scores for individual terms and suggest that they should be used when k values are large. In a different approach, Yafay and Altingovde (2019) build a cache for scores of full queries and portions of them, showing that the approach helps accelerate the list pruning. Furthermore, Petri, Moffat, Mackenzie, Culpepper, and Beck (2019) propose an estimation method based on a learning framework and explore techniques to balance the risk of over-predictions with re-executing the query to get the correct answer. In subsequent work, Mallia, Siedlaczek, Sun, and Suel (2020) offer a detailed analysis and comparison of different techniques and scenarios.

2.5. Tail latency

Tail latency refers to those running times beyond a given percentile of the distribution (typically, 95% or 99%) (Kim, He, Hwang, Elnikety, & Choi, 2015). A query with response time in the *tail* may be considered an "outlier" and poses a challenge to a search engine, at a system level and user satisfaction, as well. The lower the tail latency, the better the global service. This requirement is essential in distributed systems where the slower node dominates the user-perceived response time. Controlling and/or reducing tail latency is challenging because document collections grow constantly, user queries exhibit a highly variable demand, and query processing algorithms behave individually differently for each submitted query.

2.6. Top-k algorithm combination

The idea of combining different algorithms on a query basis is relatively recent, and a few articles are covering this approach. Mackenzie et al. (2018) use a framework to predict which value of k and processing algorithm minimize computing time and effectiveness loss for a given query q. They combine BMW and JASS algorithms using an index mirroring approach (given that both strategies use different index organizations) and devise a hybrid system that minimizes both effectiveness loss and query latency.

Here, we review the closest work related to ours proposed by Fang et al. (2020). The authors combine algorithms from the DAAT and SAAT (Score-at-a-Time) families according to a classification model. For the evaluation, they use the GOV2 collection and TREC05 and TREC06 queries. Using VBMW as a baseline, they find that the proposed approaches reduce the average query time and, mainly, tail latency. However, this work does not provide safe results given that it uses JASS, which sacrifices some effectiveness to achieve a more stable tail behavior.

In this work, we focus our attention on document ID sorted indexes, DAAT query evaluation, and safe methods. We exploit the fact that we can traverse the indexes in a non-blocked and blocked fashion and split the analysis according to this fact. We also consider different cut-off values (k) and three standard document collections, observing essential performance differences among the algorithms. We also consider combining all pruning methods, query lengths, and cut-off values, obtaining extra performance improvements. Besides, recent work (Mallia, Siedlaczek and Suel, 2019) shows that Maxscore is still competitive, mainly for long queries and/or a large k so we take it into account. Our findings with this technique go in the same direction.

2.7. Machine learning classification

Machine learning (ML) is the area of computer science that studies and develop a wide range of algorithms that improve a given performance metric automatically through experience (Mitchell, 1997). A particular case is the area of Supervised Learning, where the algorithms initially build a model based on “training data” to make future predictions on newly seen data. One of the key applications of ML algorithms is data classification, an instance of supervised learning techniques. In brief, a classifier is trained to build a model M that specifies a function $f_M(x)$ that assigns a label to a new input sample (x) according to the model inferred using the training instances.

Data classification problems may be tackled using different approaches such as Decision Trees (DT) (Rokach & Maimon, 2005) and more sophisticated variants. In a tree-like model, each node specifies a splitting criterion on a given variable (model feature), and each branch becomes a possible outcome. The leave nodes in the tree correspond to a final decision, namely the predicted label (or class). DT models are pretty straightforward to interpret, efficient to build, and suitable for many problems, but may exhibit high variance, becoming a weak predictor. As an alternative, ensemble models may be used to overcome these weaknesses based on using many trees to generate more robust predictions. The random Forests (Breiman, 2001) technique belongs to this group. This approach builds multiple decision trees at training time and then combines them to get more accurate and stable predictions. The number of trees is a crucial hyperparameter to tune. It determines the accuracy of the model and the running time (which may be a limitation for real-time settings).

An evolution of these ideas is *boosting* technique, an ensemble tree-based method that builds consecutive small trees, where each one focuses on correcting the error from the previous trees. This strategy allows for improving the predictions by correcting the misclassifications from the initial instances. The final result is the weighted sum of all partial (individual) predictions. One of the most effective extensions of this approach is *gradient boosting trees*, which uses gradient descent for optimization. This more sophisticated technique allows us to handle non-linear relationships and deal with imbalanced training data. The advancement of the tree-based methods improves the accuracy of the predictions. However, they require more processing time in the training and testing phases and more effort to tune the hyperparameters. In particular, we use eXtreme Gradient Boosting (Chen & Guestrin, 2016), a widely used technique to build tree-boosting approaches that scales beyond billions of examples while using computing resources rationally.

3. Algorithm selection method

3.1. Problem definition

First, we define the problem we aim to solve. Given a query q, a series of top-k disjunctive query processing algorithms A, how can we select the processing algorithm $a_i \in A$ which minimizes the execution time of the query without requiring to execute all the query processing algorithms in A?

3.2. Proposed approach

As we mention in the Motivation paragraph (Section 1.1), the main idea is to combine more than one top-k algorithm in a search engine to solve queries faster than using a single strategy. Some authors have proposed to use different algorithms according to the number of terms in each query. For example, Mallia, Siedlaczek and Suel (2019) found that Maxscore performs better than other methods (even against Variable-BlockMax Wand) with long queries and suggest a hybrid retrieval method that switches between algorithms. In previous work, Dimopoulos et al. (2013) propose a combination of Block-Max WAND and Block-Max Maxscore by selecting the best method according to the number of query terms.

However, we propose deciding the retrieval method on a query basis, considering not only the number of terms but using other properties of individual queries. For this, we build a framework that enables us to select the presumably *best* algorithm in each case without performance degradation (i.e. with minimum overhead). We propose to model the algorithm-selection issue as a classification problem and solve it using Machine Learning techniques. The feature space corresponds to several *signals* extracted from queries, and the target value (or *class*) is the algorithm to use for solving a particular query. In other words, at running time, the framework aims to predict the algorithm that performs the best (i.e. the fastest) for each query. The whole process encompasses two steps:

1. **Classifier training:** We start with a set of unique queries Q and a series of top-k retrieval algorithms A (i.e. Maxscore and WAND). Then,

   (a) We execute each query $q \in Q$ using each algorithm $a \in A$ and record the retrieval time, $rt_{q,a}$. The winning method is the one that $\arg\min_a rt_{q,a}$.
   (b) We extract some properties of the queries themselves (i.e. number of terms) and some from the individual terms (such as the length of their posting lists in a given inverted index). This information comprises the features associated with the target class (the winner algorithm from the previous step).
   (c) With this information, we train different classification models to predict the winning algorithm.

   It is worth mentioning that we use query features computed over the posting lists of each query's terms. Each time the algorithm locates a posting list inside the inverted index, all the necessary information to train the model becomes available.
2. **Using predictions:** At running time, we decide which top-k algorithm to use per query basis. We compute the feature space for each query q_i and feed them into the classification model. The resulting class becomes the algorithm we use to solve a particular query. Analogous to the training process, the features become available when the algorithm locates all the posting lists required to solve the query.

We separate retrieval methods that do not require a block-based inverted index organization (Non-blocked indexes) from those that do (Blocked indexes). We are aware that some methods (i.e. standard Maxscore) may run on the top of a blocked index organization (without the use of the information of the block), but the reverse option is not possible. However, although some algorithms that exploit blocking information become state-of-the-art techniques (i.e. Variable-Blocks Block-Max WAND), they do not perform the best for all possible configurations (Mallia, Siedlaczek and Suel, 2019). For this reason, we explore the performance of different algorithms built on the top of the two indexes organizations but then, we make a complete comparison by combining the best-performing algorithm in each case. In the first case, we consider three retrieval methods:

- Ranked_OR (RO)
- Maxscore (MS)
- WAND (WD)

In the second case, we consider two variants that use the Block-Max index organization, with variable block sizes, such as:

- Variable Block-Max Maxscore (BM)
- Variable Block-Max WAND (BW)

3.3. Prediction framework

The approach to building the prediction framework relies on models that can run quickly and effectively using a reduced number of features, with the primary goal of not incurring significant processing overhead. We build our prediction framework on the tree-based classifier family (DT), using three algorithms of increasing complexity and, in general, also better performance. We select this family of models because it offers different choices on the complexity/efficiency tradeoff. Also, tree-based models are easy to interpret (for example, to determine the importance of the features), efficient to build, and have low overhead at prediction time. In addition, there are efficient approaches for big data classification in the case of the more complex choices (Weinberg & Last, 2019).

The simplest method we use is a Decision Tree, which offers a first approximation to a solution and an intuition about the interaction between features. The second one is Random Forest, an ensemble method that builds multiple tree models using different subsets of features to remove correlations. These two methods have fewer parameters to tune and maybe train quickly. Finally, we use the Extreme Gradient Boosting (XGBoost) method. Gradient Boosting approaches work by fitting subsequent models to the residuals from prior outputs, while the particular case of XGBoost adds regularization to reduce overfitting. These methods require a proper tuning of numerous parameters, which leads to a more considerable training time.

3.3.1. Query features

We use a set of features corresponding to statistics computed from the posting list of the terms that form each query. Some of them can be computed offline during indexing and stored with a negligible space overhead inside the vocabulary of the inverted index. Other features must be computed online because they summarize properties of all query terms (usually a few) and do not represent an extra running time.

We focus on a reduced set of features that may explain the query running time and devise which algorithm should perform the best. Previous works considered some of these features in effectiveness (Tonellotto, Macdonald, & Ounis, 2011) and cut-off prediction (Culpepper, Clarke, & Lin, 2016) proposals.

Given a query q, composed by n terms t_j (with $j = \{1, 2, \dots, n\}$) we compute the following groups of features:

List Length-based

We compute these features because longer lists mean (a priori) a higher running time. Even though the number of scored documents depends on the dynamic pruning strategy of the top-k algorithm, this set of three features captures the size of the problem and some indications of whether more lookups of a document identifier of one list inside the other one may occur.

1. Lengths of the posting lists of each term in q (pl_len): This corresponds to the number of documents containing a term t_j. This information is usually inside the vocabulary as the *DF* (Document Frequency) field. As we build different models according to query lengths (from 2 up to 6+), each one contains a diverse number of them. For each query q, we sorted these features from the longest to the shortest one. To limit the number of models needed to answer all query lengths, we only leverage the longest posting lists for queries with more than 6 terms. We base this decision on the observation that there are much fewer longer queries (i.e. than those up to 5 terms) which make it challenging to have enough items for training purposes. We also observe that selecting the n longest posting lists does not degrade the classification performance for this particular set. As a practical consideration in a production scenario, a practitioner can carefully decide which query length to group the queries into one single bucket based on the query length distribution.
2. Average length of the posting lists (avg_plen): This value is computed online with the information of each individual term in q. This value tries to capture a single length value for the whole query.
3. Relative posting lists length size (rel_len): This corresponds to the relative size of the longest posting list regarding the second one. This feature captures the length balance between the two most costly posting lists.

Frequency-based

The idea behind these features is to capture signals regarding the chance of a list containing documents that may appear in the top-k final list.

4. Relative frequency value (rel_val): This value corresponds to the relative value of the greatest frequency (of all posting lists) regarding the second one. This feature complements the previous one. The intuition behind this feature is to capture the possibility that a single list contains many documents that make it inside the final top-k answer. This feature is computed online.
5. Average value of doc's frequencies (avg_val): For each term t, we compute all documents' average frequency value in the posting list of that term.
6. Value of highest frequency of t_i (ub_val): This value and the posting list length are the main components of the upper bound score associated with each term.
7. Position of the highest frequency of t_i (ub_pos): This feature corresponds to the location (expressed as a percentage) where the highest frequency appears inside the list of t_j. The idea is that if the highest frequency of a term is located early inside the list, the remaining documents may have less chance of entering the top-k answer.

Identifier-based

Complementarily, we model the density of the docID space of each posting list. A denser list could indicate more lookups of a document identifier of one list inside the other one.

8. Average value of term's docIDs (id_avg): This features corresponds to $\frac{max\{docid_j\} - min\{docid_j\}}{|I_j|}$, for a given term j and its associated posting list, I_j. This value must be computed offline and stored inside the vocabulary.

4. Experimental evaluation

In this section, we analyze the performance of our proposed method with an extensive experimental evaluation in a realistic and reproducible setting, using state-of-the-art baselines, standard text collections, and three large query logs.

4.1. Datasets

Document Collections: We perform experiments using three standard document collections, such as:

- **GOV2**: This dataset is the TREC 2004 Terabyte Track (Clarke, Craswell, & Soboroff, 2004) test collection consisting of around 25 million documents crawled in 2004 from the *.gov* domain (documents are truncated to 256 kB).
- **CW09**: This dataset corresponds to the Category B part of the ClueWeb09 collection (Callan, Hoy, Yoo, & Zhao, 2009) that contains around 50 million English web pages crawled between January and February 2009.

Table 1
Basic properties of the document collections.

	Documents	Terms	Postings
GOV2	25,205,178	32,407,061	5,264,576,636
CW09	50,220,110	87,896,391	14,996,638,171
CW12	52,343,021	133,248,235	14,173,094,439

Table 2
Percentage of queries that differ in the winner algorithm between the original and the stopped one. (We use 10 000 queries from the EFF06 set and the CW12 documents collection.)

Changed winners (%)		
k	**nb**	**b**
10	0.12	0.18
100	0.15	0.18
1 000	0.22	0.21
10 000	0.23	0.26

- **CW12**: This is the official Category B13 subset of the ClueWeb12 collection,[3] which contains around 52 million English web pages crawled between February and May 2012.

Table 1 summarizes the main properties of the document collections.

The three datasets are processed using the Gumbo parser.[4] We assign document identifiers (docID) in the lexicographic order of their URLs (Silvestri, 2007) and apply standard processing techniques such as lowercasing and stemming (using the Porter2 algorithm). We decide to apply stemming in our processing pipeline, considering our proposal becomes the first phase of a two-stage retrieval process, followed by a re-ranking approach. In this scenario, stemming increases recall allowing the re-ranking process to move up most relevant documents (mainly for large k values). From an effectiveness perspective, this decision is worth analyzing but is out of the scope of this work. From an efficiency perspective, stemming generates longer posting lists, which are harder to traverse to score documents. Our method becomes even more helpful in this challenging scenario, allowing a decrease in processing time.

We do not remove stopwords either from the collection or the queries. Applying stopping is inherently lossy, and the results may differ between the original and the pruned query (Mackenzie & Moffat, 2020). However, we analyze the number of queries for which the fastest algorithm differs between the original and the stopped query. Table 2 reports the results showing that around 20% of the queries (on average) have a different *winner* algorithm when considering both versions of the query (for example, a query solved faster with MS and then, became WAND faster once we remove the stopwords). This situation holds for the different values of k and the two index structures.

This suggests that different settings require training-specific classification models. For the global testing, we stick to the experimental settings used in other works on efficiency and keep the stopwords, which is a realistic scenario for modern search engines (Crane et al., 2017; Mallia et al., 2017).

After parsing and indexing each collection, we compress the final inverted indexes using the SIMD-BP128 algorithm (Lemire & Boytsov, 2015) and store them on disk.

Query Logs: To train the classification models and evaluate the performance of the proposed approach, we select three different query logs commonly used in efficiency experiments. For model training, we use Million Query Track (MQT) from TREC 2007[5] (Allan et al., 2007), TREC 2008 (Allan, Aslam, Carterette, Pavlu, & Kanoulas, 2008) and TREC 2009 (Carterette, Pavlu, Fang, & Kanoulas, 2009). In this case, we randomly select 15,000 queries (3000 of each considered query length to train with balanced data samples).

For the retrieval experiments, we select queries from the TREC 2005[6] (Clarke et al., 2005) (∼50k queries) and TREC 2006[7] (Büttcher et al., 2006) (∼90k queries) Terabyte Track Efficiency Task (we call them EFF05 and EFF06, respectively). Then, we randomly select 10,000 queries from each query log in the same range of lengths as with MQT. The three query sets correspond to the TREC community's efforts to create standard data collection for search systems evaluation, mainly for comparing the efficiency and scalability issues and determining query processing times under different methods. The query logs do not contain repeated instances themselves, and the train (MQT) and test (EFF05 and EFF06) overlap by less than 1%, which does not represent an issue when evaluating efficiency (as it would be when evaluating caching methods) and training the chosen classification methods. It is important to note that the training and testing sets come from different query logs. This decision represents a more challenging scenario that prevents the learners' overfitting and allows a more robust generalization capability of the resulting models.

3 http://www.lemurproject.org/clueweb12/.
4 https://github.com/google/gumbo-parser.
5 https://trec.nist.gov/data/million.query07.html.
6 https://trec.nist.gov/data/terabyte05.html.
7 https://trec.nist.gov/data/terabyte06.html.

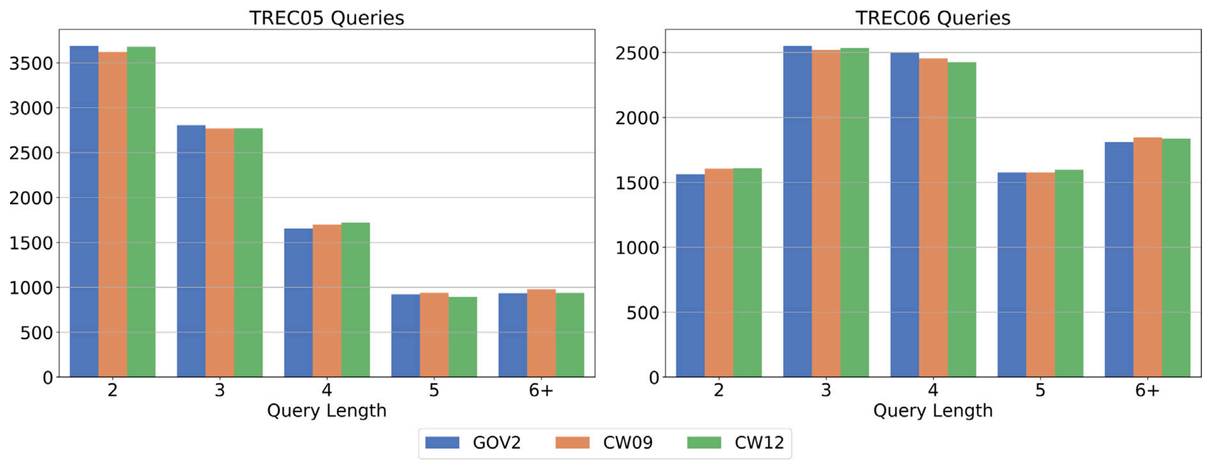


Fig. 3. Query length distributions.

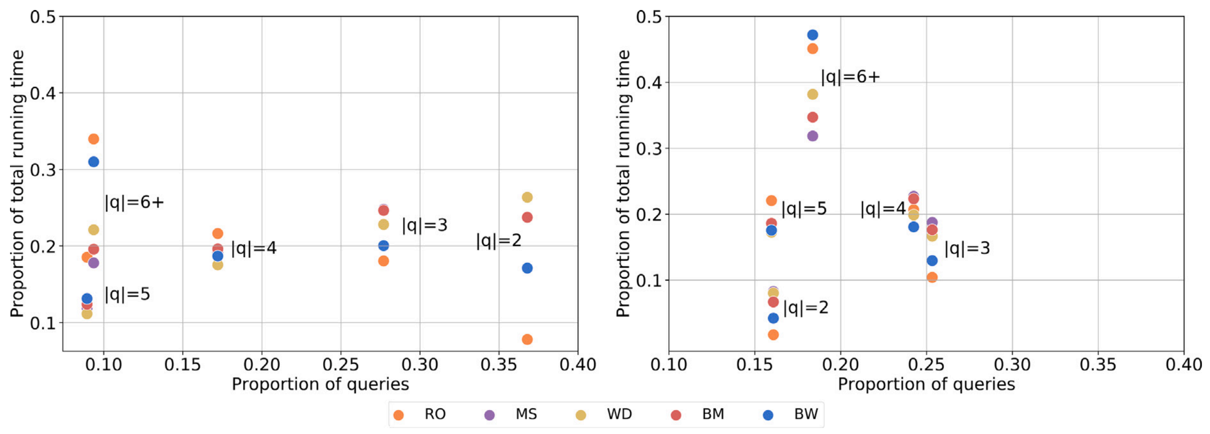


Fig. 4. Fraction of total running time according to the proportion of queries in EFF05 (left) and EFF06 (right), CW12 document collection.

Fig. 3 shows the query length distributions of the query sets by document collection. We plot queries whose length is in $[2-6]$. Longer queries ($|q| > 6$) are not discarded but considered as with six terms (6+). In all cases, we only keep queries whose terms appear in each collection dictionary.

Considering the different top-k retrieval methods, we compute the fraction of the total running time according to the proportion of queries of each length. Fig. 4 shows the results for the two query logs on the CW12 collection (the methods perform similarly on the remaining datasets). It is worth noting here the impact of long queries on the fraction of the total time.

4.2. Hardware and software environment

We run the retrieval experiments using the PISA search engine (Mallia, Siedlaczek, MacKenzie and Suel, 2019), compiled with GCC++ 7.5.0 (on Linux 4.15.0) using the highest optimization settings. The hardware platform is a two Intel(R) Xeon(R) E5-2667 v2 CPU (32 cores), clocked at 3.30 GHz, with 256 GiB RAM.

4.3. Testing details

Before timing experiments, the indexes are memory-mapped to avoid the cost of loading the posting lists and additional data structures. We use BM25 (Robertson & Jones, 1976) as the scoring function (computed during retrieval). Petri, Culpepper, and Moffat (2013) have measured the performance improvements of dynamic pruning approaches in the context of the alternative Language Models' similarity score computation and found that the dramatic efficiency gains reported in previous studies are no longer achievable. Most optimized query processing algorithms considered the de-facto standard in retrieval systems exploit BM25 score distributions. Recently, Mackenzie, Trotman and Lin (2021) have shown that also learned sparse models reduce the opportunities for

Table 3
Performance of the classification methods weighted by the class sizes.

k	Method	Non blocked			Blocked		
		P	R	F1	P	R	F1
10	Decision tree	0.806	0.806	0.806	0.874	0.886	0.876
	Random forest	0.840	0.842	0.840	0.888	0.896	0.886
	XGBoost	0.840	0.840	0.840	0.886	0.894	0.886
100	Decision tree	0.820	0.822	0.816	0.878	0.884	0.876
	Random forest	0.850	0.850	0.850	0.884	0.888	0.878
	XGBoost	0.858	0.860	0.860	0.890	0.894	0.888
1000	Decision tree	0.848	0.846	0.844	0.866	0.870	0.868
	Random forest	0.864	0.862	0.862	0.884	0.888	0.884
	XGBoost	0.862	0.862	0.862	0.882	0.884	0.884
10 000	Decision tree	0.880	0.882	0.880	0.886	0.888	0.886
	Random forest	0.894	0.896	0.892	0.892	0.894	0.892
	XGBoost	0.890	0.890	0.890	0.888	0.890	0.888

skipping and early exiting optimizations. Optimizing for other relevance scoring functions is out of the scope of this paper. Although these models show reduced benefits, our approach is still orthogonal to future adaptations to each of the algorithms used.

For each combination of collection/query log, we recover the top-k documents and report the minimum value of five independent runs. We decide to test a large range of k values, from k = 10 up to 10 000, considering that some recent work (Crane et al., 2017) shows that the performance of some top-k algorithms degrade as k becomes larger. Moreover, many applications such as commercial search engines use a second-stage reranking phase, using a complex machine-learned ranker. This kind of process requires a greater number of results than just the top-10 or top-100 results. In our experiments, we find that the best-performing top-k algorithms also depend on the value of k, for some particular queries.

4.4. Classification models

We build different classification models according to the length of the queries. We process the inverted index for each collection to obtain terms' information related to their posting lists, document identifiers, and upper bounds. We use Scikit-Learn[8] to build, evaluate and test the models. We also use the *Randomized Search* technique (Bergstra & Bengio, 2012) to tune the main hyper-parameters of each method. This technique is more efficient than grid search experiments when using various learning algorithms on several datasets (our case). Moreover, not all hyper-parameters are equally important to adjust.

We split the training data into train and test sets (80/20%) to evaluate performance using a 10-fold cross-validation procedure. We use queries from the MQT set to train the models and the remaining queries from EFF05 and EFF06 to evaluate the proposal (query time performance). We present the performance of the three classification methods applied to CW12 collection in Table 3 using standard metrics (averaged over all query lengths), namely: (a) Precision (P) is the number of true positive cases over the total positive predictions ($P = \frac{TP}{TP+FP}$). In other words, precision computes the fraction of predicted positives that are actually positive. (b) On the other hand, Recall (R) measures the model's ability to predict the positives ($R = \frac{TP}{TP+FN}$). (c) Finally, the F1 score is the harmonic mean of precision and recall ($F1 = 2 \times \frac{P \times R}{P+R}$). Given that the target class is not balanced, we compute weighted versions of the metrics according to the size of each one.

The sizes of the generated models are small compared to the other data structures. For example, the average size of XGB models is 0.021 and 0.012 GB for the non-blocked and blocked index organization over the CW12 document collection (averaged over all query lengths and the four values of k). We do not consider the overhead incurred in making predictions at a running time because this cost is an order of magnitude less than query times. To support this decision, we run a benchmark to measure the impact of the prediction process. We run 5000 predictions for both non-blocked and blocked indexes and for all k values. The overhead is around 20 and 11 μs (on average) for non-blocked and blocked indexes, respectively, which becomes negligible compared to the processing time. Previous works (Fang et al., 2020; Mackenzie et al., 2018) report similar approaches under similar conditions.

We also compute feature importance, as a metric that indicates which features are most relevant in explaining the target variable and the model behavior. Given that we use tree-based classification methods, we use an impurity-based approach where feature importance corresponds to the decrease in node impurity weighted by the probability of reaching that node. This analysis offers some insights into each feature's usefulness and where to focus on improving the model. Fig. 5 shows an example of the XGB method and the different values of k (averaged over all query lengths), according to the two index organizations.

Not surprisingly, the results show that the lengths of the posting lists of each individual term (pl_len) are the most informative features together with the average length (avg_len). Here, pl_len corresponds to the highest importance offered by any posting list of a term in a query. This behavior is similar in the remaining document collections.

Complementing Fig. 1, we compare the proportion of queries solved by each algorithm in the case of *true* winners and those selected by our methods. These *true* winners are our *ground truth* to compare the models with. Fig. 6 shows an example for both types of index organization considering the sel_XGB approach (CW12 document collection, EFF05 query log, and k = 1000).

[8] https://scikit-learn.org/.

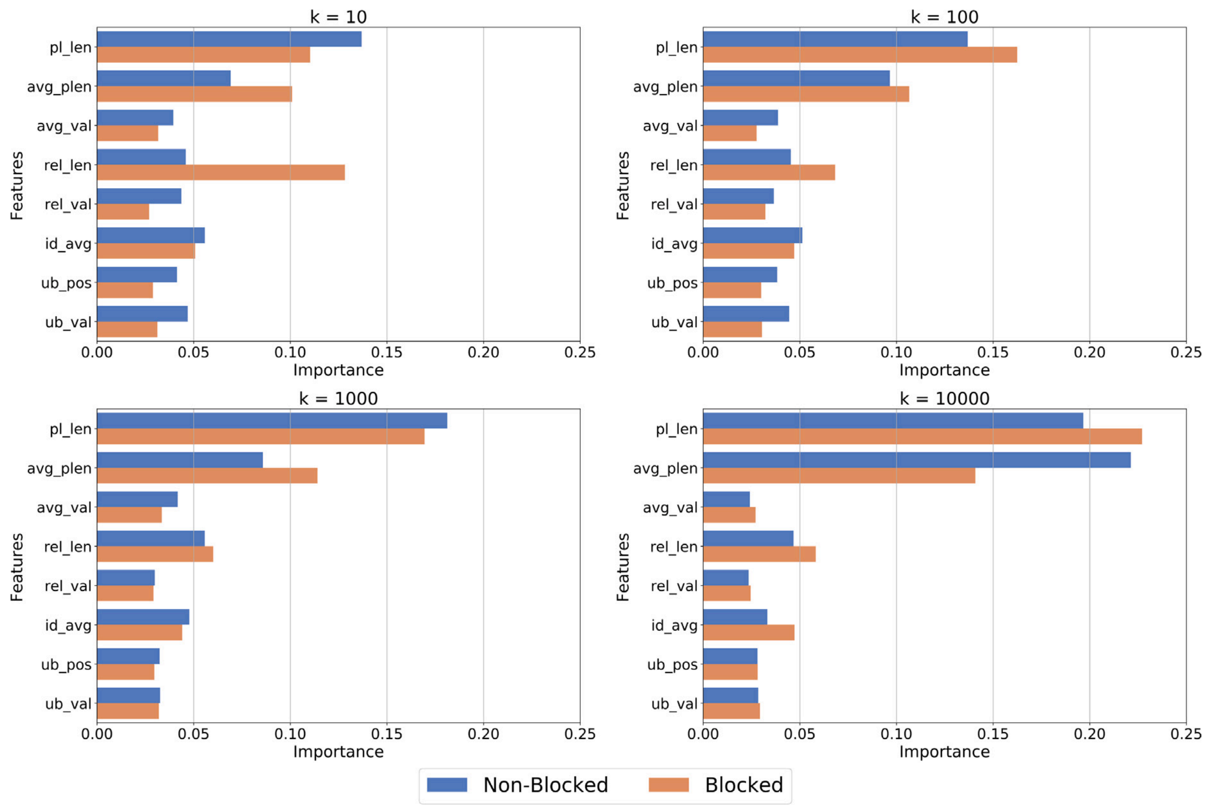


Fig. 5. Features importances computed for the XGB method on the (CW12 collection) for different values of k and index organizations.

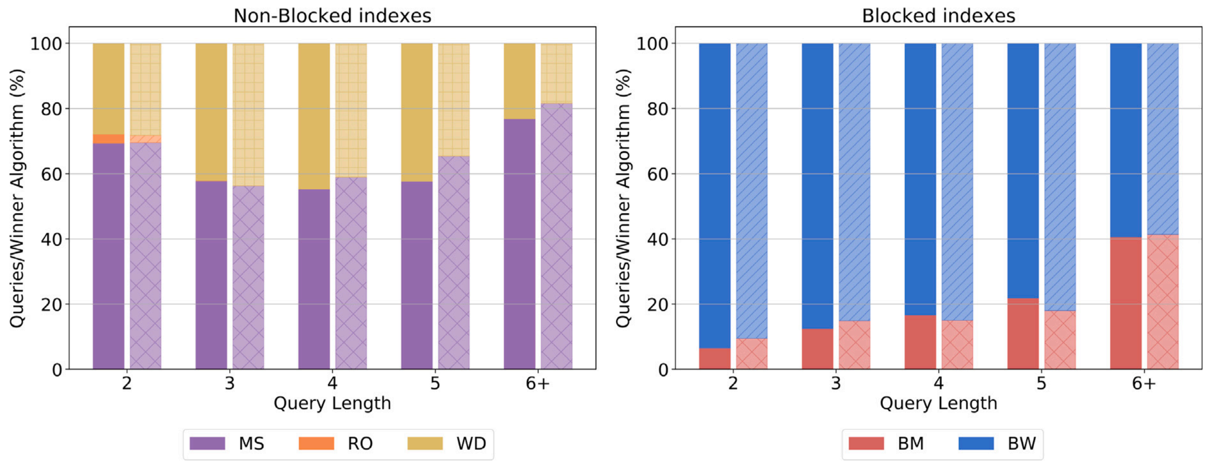


Fig. 6. The proportion of queries solved by different algorithms (separated by query length): true winners (solid bars, left side) and selected by the sel_XGB approach (striped bars, right side). Both figures correspond to the CW12 document collection, EFF05 query log, and k = 1000.

In the case of Non-blocked index organizations, the sel_XGB approach performs close to the *true* winners. For example, considering the MS method, differences are about 0.001%, −0.027%, 0.061%, 0.11%, and 0.059% for query lengths of 2, 3, 4, 5, and 6, respectively. For Non-blocked index organizations differences are still smaller (around −0.033%, −0.027%, 0.019%, 0.048%, and −0.013%, for the same query lengths).

This result offers an overall idea of the classification process's performance. However, it does not define the final timing efficiency due to the overlapping between the two sets is not perfect. We explore the execution performance in the next section.

Table 4
Query processing times averages (in milliseconds) for GOV2 collection and EFF05 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	4.80	4.52 (×1.06)	4.44 (×1.08)	**4.43** (×1.08)	4.44 (×1.08)	4.31 (×1.11)	4.10 (×1.17)
100	7.26	6.95 (×1.04)	6.83 (×1.06)	6.85 (×1.06)	**6.83** (×1.06)	6.68 (×1.09)	6.46 (×1.12)
1 000	13.45	12.96 (×1.04)	12.89 (×1.04)	12.93 (×1.04)	**12.88** (×1.04)	12.75 (×1.05)	12.47 (×1.08)
10 000	34.34	34.12 (×1.01)	**33.92** (×1.01)	33.98 (×1.01)	33.94 (×1.01)	33.74 (×1.02)	33.43 (×1.03)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	3.95	**3.70** (×1.07)	3.81 (×1.04)	3.87 (×1.02)	3.82 (×1.03)	3.62 (×1.09)	3.52 (×1.12)
100	7.46	7.08 (×1.05)	6.89 (×1.08)	6.94 (×1.08)	**6.90** (×1.08)	6.73 (×1.11)	6.55 (×1.14)
1 000	16.50	15.31 (×1.08)	15.00 (×1.10)	15.06 (×1.10)	**15.02** (×1.10)	14.75 (×1.12)	14.44 (×1.14)
10 000	47.27	43.39 (×1.09)	42.89 (×1.10)	42.88 (×1.10)	**42.82** (×1.10)	42.34 (×1.12)	41.59 (×1.14)

5. Retrieval results

This section analyzes the performance of each proposed method against the two state-of-the-art baselines that achieve the best overall performance. We first execute all the algorithms described in Section 3.2 with Non-Blocked and Blocked index organizations using the three document collections, three query logs, and the four considered values of k. These initial results show that, overall, Maxscore and Variable Block-Max WAND become the fastest ones according to each index type. These results are consistent with very recent literature that considers these algorithms as strong baselines (Bortnikov, Carmel, & Golan-Gueta, 2017; Fang et al., 2020; Khattab et al., 2020; Mackenzie & Moffat, 2020; Mackenzie, Petri and Moffat, 2021; Mallia et al., 2020; Siedlaczek, Mallia, & Suel, 2022). While Variable Block-Max WAND is the fastest query processing method for a small value of k, MaxScore performs better for a large k and, also, for long queries (Mallia, Siedlaczek and Suel, 2019). Considering these facts, we use both algorithms as references for each type of index organization in our work.

We build the classification models using the information from the experiments with the MQT query log and then compute the execution time using different top-k methods according to the output of each classifier. We also add two reference methods to obtain a comprehensive view regarding the maximum possible speedup that we may achieve using different top-k algorithms and, in another view, considering only the information of the three classification methods. We name the methods:

- **sel_Tree**: Time achieved when selecting the top-k algorithm using a Decision Tree classifier.
- **sel_Forest**: Time achieved when selecting the top-k algorithm using a Random Forest classifier.
- **sel_XGB**: Time achieved when selecting the top-k algorithm using an eXtreme Gradient Boosting classifier (XGB).
- **sel_Vote**: We build a fourth method using the previous ones, combining them in a *majority voting* ensemble scheme (Valentini & Masulli, 2002). Then, *sel_Vote* corresponds to the time achieved when using the voting ensemble of classifiers.
- **bestCLF**: Time achieved when selecting the best performing algorithm among Decision Trees, Random Forest, and XGB classifiers ($bestCLF = min(sel\_Tree, sel\_Forest, sel\_XGB)$). The intention behind *bestCLF* is to offer an insight into the best possible performance gains that the proposed framework may achieve using the classification methods (and settings) mentioned above.
- **Oracle**: Time achieved for a perfect algorithm selector who *knows* which one performs the best for each query. The Oracle method shows the upper limit in time optimization when combining top-k methods.

5.1. Average query times

We show the performance improvements for each method against the baselines. Tables 4, 5, 6, 7, 8 and 9 show the overall results for the three collections and the two query logs used for evaluation (EFF05 and EFF06). The first interesting overall observation appears when comparing Non-blocked indexes (NI) versus Blocked indexes (BI). The *Oracle* method shows a decrease in the possible best performance in NI when k increases. Nevertheless, this trend reverses in the BI case, where a greater value of k benefits more from the combination of the algorithms. This situation correlates with the observation that some Non-blocked methods (precisely Maxscore) achieve better performance than Blocked ones (such as BW) when increasing k (Mallia, Siedlaczek and Suel, 2019). A more in-depth inspection shows that many complex (long) queries run faster with Block-Max Maxscore instead of Block-Max WAND and our proposal enables the possibility of combining them.

Considering the GOV2 collection (the smallest one) and the EFF05 query log, the best-proposed strategy (*sel_XGB*) achieves up to x1.08 ($k = 10$) for NI and 1.10x ($k = \{1000, 10\,000\}$) for BI of speedup concerning the corresponding baselines, while perfect algorithm selector (*Oracle*) improves x1.17 and x1.14, respectively. These results remain consistent when using the EFF06 query log. In the case of the CW09 collection, the *sel_XGB* method approaches more to the Oracle method (x1.12 vs x1.16 for NI and x1.16 vs x1.18 for BI using EFF05 query log). Finally, when considering the biggest document collection (CW12) the method achieves an x1.20 speedup (vs 1.26, Oracle) and x1.16 (vs 1.19, Oracle) using the EFF05 query log. This overall behavior becomes similar using the EFF06 queries.

Table 5
Query processing times averages (in milliseconds) for GOV2 collection and EFF06 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	7.33	6.78 (×1.08)	**6.68** (×1.10)	6.70 (×1.09)	**6.68** (×1.10)	6.46 (×1.14)	6.12 (×1.20)
100	10.81	10.13 (×1.07)	10.01 (×1.08)	**9.97** (×1.08)	10.00 (×1.08)	9.73 (×1.11)	9.32 (×1.16)
1 000	19.33	18.43 (×1.05)	18.38 (×1.05)	**18.36** (×1.05)	18.37 (×1.05)	18.06 (×1.07)	17.61 (×1.10)
10 000	48.36	47.59 (×1.02)	**47.41** (×1.02)	47.46 (×1.02)	47.42 (×1.02)	47.14 (×1.03)	46.63 (×1.04)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	7.09	6.87 (×1.03)	6.87 (×1.03)	6.86 (×1.03)	**6.85** (×1.03)	6.67 (×1.06)	6.48 (×1.09)
100	12.79	12.22 (×1.05)	**12.07** (×1.06)	**12.07** (×1.06)	12.09 (×1.06)	11.78 (×1.09)	11.38 (×1.12)
1 000	26.49	24.73 (×1.07)	**24.39** (×1.09)	24.44 (×1.08)	**24.39** (×1.09)	24.00 (×1.10)	23.40 (×1.13)
10 000	72.26	65.93 (×1.10)	65.59 (×1.10)	65.66 (×1.10)	**65.58** (×1.10)	64.86 (×1.11)	63.84 (×1.13)

Table 6
Query processing times averages (in milliseconds) for CW09 collection and EFF05 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	17.53	15.89 (×1.10)	15.69 (×1.12)	**15.67** (×1.12)	15.68 (×1.12)	15.47 (×1.13)	15.08 (×1.16)
100	22.97	20.79 (×1.10)	20.71 (×1.11)	20.76 (×1.11)	**20.70** (×1.11)	20.49 (×1.12)	20.05 (×1.15)
1 000	36.37	33.65 (×1.08)	**33.43** (×1.09)	33.52 (×1.09)	**33.43** (×1.09)	33.20 (×1.10)	32.72 (×1.11)
10 000	82.89	79.17 (×1.05)	79.11 (×1.05)	**78.94** (×1.05)	79.06 (×1.05)	78.62 (×1.05)	76.58 (×1.08)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	16.51	15.01 (×1.10)	14.80 (×1.12)	14.95 (×1.10)	**14.79** (×1.12)	14.46 (×1.14)	14.07 (×1.17)
100	28.08	23.93 (×1.17)	**23.75** (×1.18)	23.84 (×1.18)	**23.75** (×1.18)	23.47 (×1.20)	22.94 (×1.22)
1 000	51.03	43.70 (×1.17)	43.65 (×1.17)	43.56 (×1.17)	**43.53** (×1.17)	43.08 (×1.18)	42.41 (×1.20)
10 000	115.25	99.87 (×1.15)	99.80 (×1.15)	99.76 (×1.16)	**99.66** (×1.16)	99.02 (×1.16)	97.70 (×1.18)

Table 7
Query processing times averages (in milliseconds) for CW09 collection and EFF06 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	27.99	23.70 (×1.18)	23.50 (×1.19)	**23.45** (×1.19)	**23.45** (×1.19)	22.94 (×1.22)	22.05 (×1.27)
100	37.35	32.52 (×1.15)	32.54 (×1.15)	32.49 (×1.15)	**32.44** (×1.15)	31.88 (×1.17)	30.91 (×1.21)
1 000	57.29	52.63 (×1.09)	52.50 (×1.09)	52.49 (×1.09)	**52.42** (×1.09)	51.96 (×1.10)	51.02 (×1.12)
10 000	114.17	111.87 (×1.02)	**111.78** (×1.02)	111.81 (×1.02)	111.80 (×1.02)	111.43 (×1.02)	110.55 (×1.03)

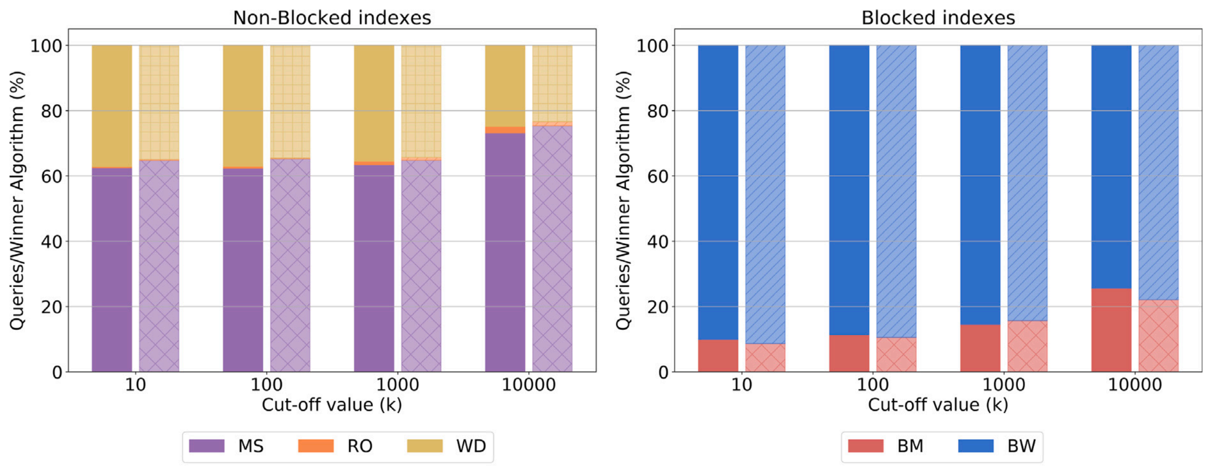
(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	28.86	25.65 (×1.12)	**25.39** (×1.14)	25.48 (×1.13)	**25.39** (×1.14)	25.00 (×1.15)	24.21 (×1.19)
100	46.67	40.47 (×1.15)	40.20 (×1.16)	40.35 (×1.16)	**40.19** (×1.16)	39.85 (×1.17)	38.96 (×1.20)
1 000	85.82	69.83 (×1.23)	69.55 (×1.23)	69.63 (×1.23)	**69.54** (×1.23)	68.85 (×1.25)	67.86 (×1.26)
10 000	191.48	159.42 (×1.20)	159.22 (×1.20)	159.17 (×1.20)	**159.12** (×1.20)	158.16 (×1.21)	156.65 (×1.22)

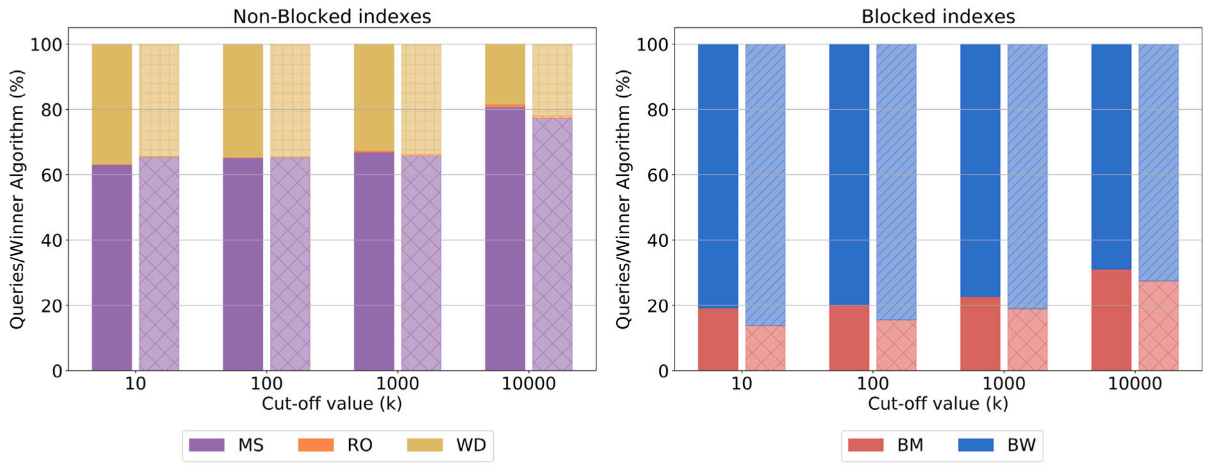
Another observation is that bigger collections (CW09 and CW12) benefit more from this approach (up to 1.2x speedup). In the same way, more costly queries (EFF06[9]) obtain, in general, better improvements, no matter the document collection considered. The ensemble-based method (*sel_Vote*) does not perform better than *sel_XGB*, except for a few exceptions that we may not consider significant. This is a rather expected behavior given that the voting decision comes from three methods based on similar approaches (Decision Trees). We investigate its usefulness later when analyzing tail latencies.

All methods perform close to *bestCLF*, which models the best possible reachable performance using the proposed classification-based approach. Moreover, *bestCLF* is also close (in general) to the perfect selector (Oracle), which means that there is some space for improving the model on a classification basis. However, it is crucial to use low-overhead strategies (as proposed) that do not impact query processing time.

[9] EFF06 query set contains longer queries (see Fig. 3) and terms with longer posting lists (i.e. 35% longer on average for CW12 collection).



(a) EFF05 query log.



(b) EFF06 query log.

Fig. 7. The proportion of queries solved by algorithms combinations according to k. The solid (left side) and striped (right side) bars correspond to the *Oracle* and *sel_XGB* methods, respectively.

Then, we observe the proportion of queries solved by each algorithm, the *Oracle* method, and our best one (*sel_XGB*) while increasing k. In this case, we compare EFF05 and EFF06 query logs (Fig. 7). The proportion of queries where MS becomes the best method is rather stable for $k = \{10, 100, 1000\}$ but this value increases around 10% and 14% for $k = 10\,000$. The *sel_XGB* method follows this behavior with a slight difference of around 2%. However, in the case of Blocked indexes, BW steadily decreases according to k increases. BM becomes more efficient when the query response set becomes larger (and the queries more costly as in the EFF06 case). Our method follows this trend and slightly *prefers* BM over BW (around 4%). This result shows that while the required response list size increases, the combination of different top-k algorithms (BM and BW in this example) attains greater importance than using a single one.

We also analyze the whole distribution of running times. Figs. 8–10 show the results. The overall observation is a reduction in the boxes' upper limit (q75) in all the proposed methods approaching the optimal. More precisely, we observe a reduction of the Interquartile Range (IRQ) up to 40% in Non-Blocked methods for *sel_XGB* (while the Oracle method is close to 43% lower). In Blocked methods, we found a slight reduction in the IQR (up to 5%). Median values also exhibit a decrease that reaches around 20% for Non-Blocked methods (and *sel_XGB*). This result indicates that the proposed approach obtains a better distribution of values around the median that correlates with the running time improvements.

5.2. Tail latencies

We now turn our attention to high percentile tail latencies (P_{95}) for different methods and configurations. Some previous work (Crane et al., 2017) shows that both WD and BW are significantly impacted by outlier values, thus exhibiting a great variance in tail latency, at any k cut-off. In the case of Non-blocked indexes, we did not expect to improve much this metric over MS (the

Table 8
Query processing times averages (in milliseconds) for CW12 collection and EFF05 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	18.30	15.61 (×1.17)	15.32 (×1.19)	**15.25** (×1.20)	15.32 (×1.19)	15.04 (×1.22)	14.57 (×1.26)
100	23.64	20.25 (×1.17)	20.06 (×1.18)	**20.05** (×1.18)	20.06 (×1.18)	19.76 (×1.20)	19.25 (×1.23)
1 000	37.47	33.05 (×1.13)	32.87 (×1.14)	32.98 (×1.14)	**32.88** (×1.14)	32.57 (×1.15)	32.00 (×1.17)
10 000	75.51	72.56 (×1.04)	**72.36** (×1.04)	72.40 (×1.04)	**72.36** (×1.04)	72.05 (×1.05)	71.27 (×1.06)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	14.35	13.17 (×1.09)	**12.99** (×1.11)	13.10 (×1.10)	**12.99** (×1.11)	12.76 (×1.12)	12.39 (×1.16)
100	23.92	21.39 (×1.12)	21.09 (×1.13)	21.10 (×1.13)	**21.04** (×1.14)	20.80 (×1.15)	20.36 (×1.17)
1 000	44.50	39.11 (×1.14)	**38.91** (×1.14)	39.10 (×1.14)	38.93 (×1.14)	38.48 (×1.16)	37.79 (×1.18)
10 000	105.68	91.09 (×1.16)	**90.86** (×1.16)	91.09 (×1.16)	**90.86** (×1.16)	90.25 (×1.17)	89.00 (×1.19)

Table 9
Query processing times averages (in milliseconds) for CW12 collection and EFF06 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	20.79	18.05 (×1.15)	17.74 (×1.17)	**17.70** (×1.17)	17.71 (×1.17)	17.41 (×1.19)	16.81 (×1.24)
100	27.34	24.57 (×1.11)	**24.38** (×1.12)	24.39 (×1.12)	24.39 (×1.12)	23.98 (×1.14)	23.42 (×1.17)
1 000	44.18	41.11 (×1.07)	40.91 (×1.08)	40.99 (×1.08)	**40.90** (×1.08)	40.62 (×1.09)	40.05 (×1.10)
10 000	87.58	86.40 (×1.01)	**86.13** (×1.02)	86.23 (×1.02)	86.16 (×1.02)	85.87 (×1.02)	85.17 (×1.03)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	23.27	20.74 (×1.12)	20.73 (×1.12)	20.71 (×1.12)	**20.67** (×1.13)	20.19 (×1.15)	19.68 (×1.18)
100	37.01	32.28 (×1.15)	31.99 (×1.16)	32.00 (×1.16)	**31.94** (×1.16)	31.49 (×1.18)	30.85 (×1.20)
1 000	66.89	57.07 (×1.17)	57.02 (×1.17)	57.02 (×1.17)	**56.92** (×1.18)	56.31 (×1.19)	55.52 (×1.20)
10 000	149.42	126.17 (×1.18)	125.98 (×1.19)	125.97 (×1.19)	**125.86** (×1.19)	125.20 (×1.19)	124.00 (×1.21)

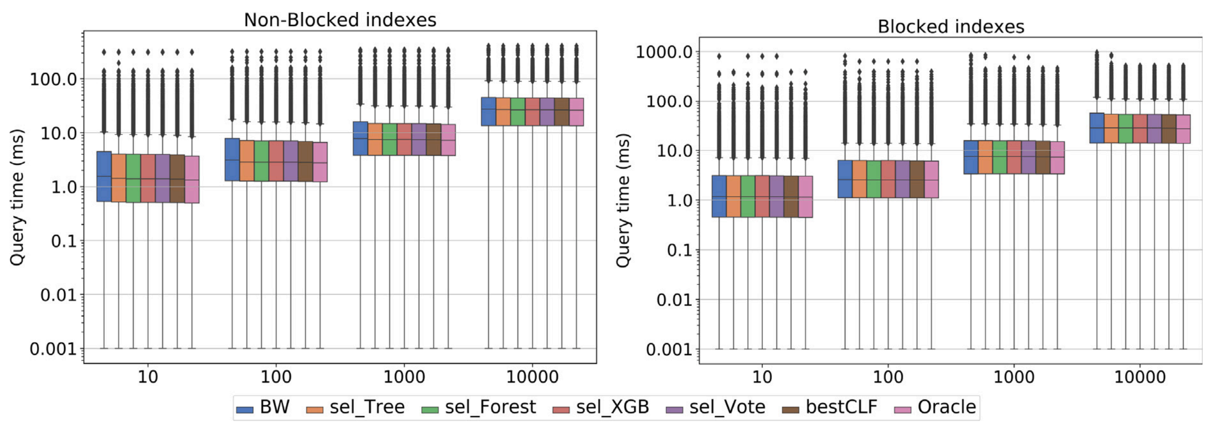


Fig. 8. Running times distributions for GOV2 collection and EFF05 query log.

baseline) given that the combination of algorithms includes many queries solved using WAND. However, on the other hand, we expect better performance improvements by combining BW and BM. In both cases, any reduction of tail latencies becomes important in production systems (Kim et al., 2015) that use many servers (such as commercial search engines) because the response time of the slowest server that processes a query dominates its overall performance.

Our results (Tables 10, 11, 12, 13, 14, 15) show an overall behavior similar to average query times. The methods show a performance decrease when *k* increases and this trend reverses for Blocked indexes. As expected, the best method is *sel_XGB* in all cases, and the ensemble of classifiers (*sel_Vote*) offers a slightly better performance in most of the instances. Another global observation is that all methods benefit more from the biggest document collection (CW12) and the most costly set of queries (EFF06). For example, considering CW12 and EFF06, *sel_Forest* achieves a speedup of x1.08 (vs the Oracle method that reaches x1.13) for the Non-Blocked organization. In the case of Blocked indexes, the speedup reaches x1.28, while the Oracle method performs x1.30 better than the baseline.

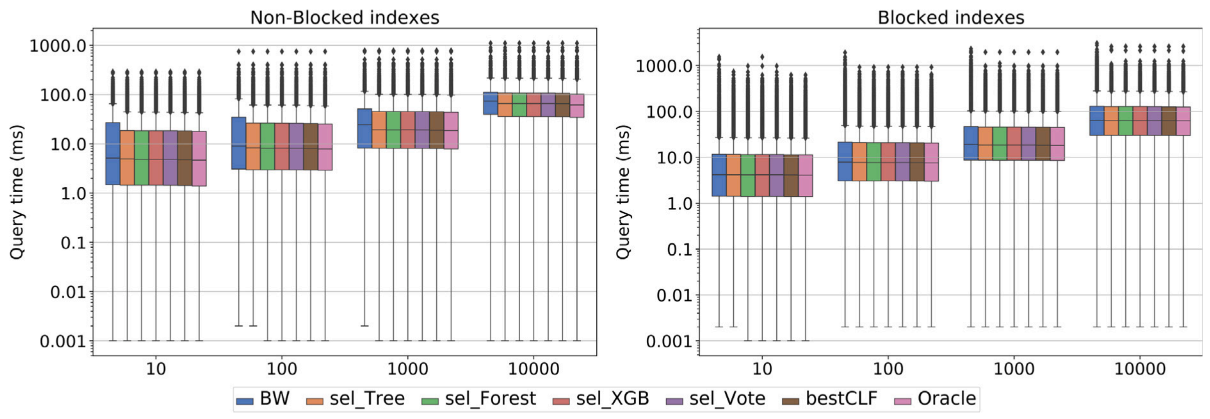


Fig. 9. Running times distributions for CW09 collection and EFF05 query log.

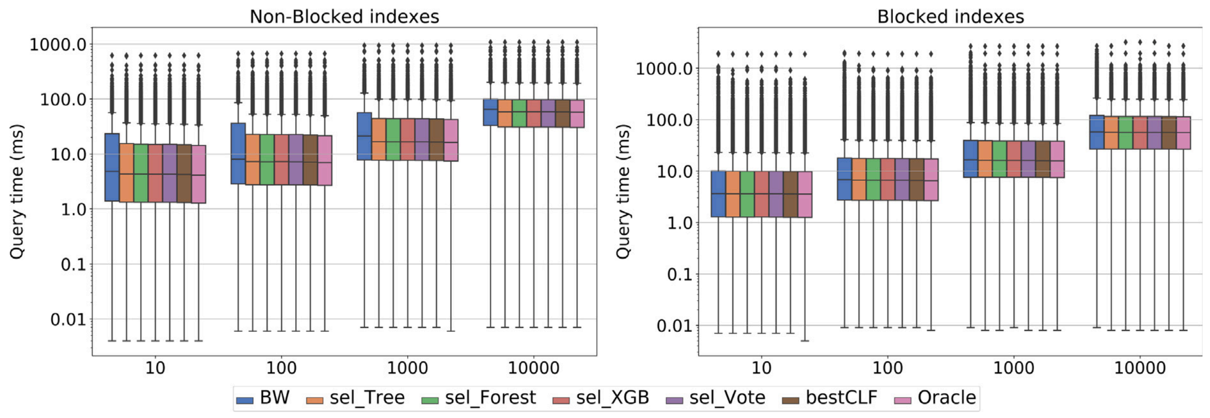


Fig. 10. Running times distributions for CW12 collection and EFF05 query log.

Table 10
Query processing tail latencies P_{95} (in milliseconds) for GOV2 collection and EFF05 query log. Improvements (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	21.27	19.66 (×1.08)	**18.98** (×1.12)	18.99 (×1.12)	19.03 (×1.12)	18.45 (×1.15)	17.08 (×1.25)
100	27.19	26.18 (×1.04)	25.67 (×1.06)	**25.57** (×1.06)	25.67 (×1.06)	24.93 (×1.09)	24.18 (×1.12)
1 000	41.02	40.19 (×1.02)	39.74 (×1.03)	39.74 (×1.03)	**39.69** (×1.03)	39.20 (×1.05)	38.44 (×1.07)
10 000	87.85	87.92 (×1.00)	87.92 (×1.00)	88.02 (×1.00)	**87.82** (×1.00)	87.06 (×1.01)	85.56 (×1.03)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	13.47	13.38 (×1.01)	13.20 (×1.02)	13.38 (×1.01)	**13.19** (×1.02)	13.09 (×1.03)	12.93 (×1.04)
100	25.51	25.87 (×0.99)	**24.57** (×1.04)	25.07 (×1.02)	24.85 (×1.03)	24.04 (×1.06)	23.48 (×1.09)
1 000	53.06	52.35 (×1.01)	51.53 (×1.03)	51.71 (×1.03)	**51.19** (×1.04)	49.79 (×1.07)	48.81 (×1.09)
10 000	139.49	130.09 (×1.07)	128.88 (×1.08)	**128.56** (×1.08)	128.66 (×1.08)	126.86 (×1.10)	124.21 (×1.12)

5.3. Average times by query lengths

In this section, we analyze the impact of the proposed algorithm's combination according to the query lengths. Tables 16 and 17 show the results for Non-blocked and Blocked indexes, respectively (CW12 collection and EFF05 query log). The behavior is different for the two index organizations. In the first case, the *sel_XGB* method exhibits the best performance improvements for queries of length 3 and 4 (no matter the value of k), up to x1.27 over MS ($k = 10$). These time savings are close with regards to the maximum possible offered by the *Oracle* method (x1.33). The benefits decrease as k increases but the relationship between *sel_XGB* and *Oracle* methods holds.

Table 11
Query processing tail latencies P_{95} (in milliseconds) for GOV2 collection and EFF06 query log. Improvements (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	29.50	27.80 (×1.06)	27.81 (×1.06)	27.81 (×1.06)	**27.70** (×1.06)	26.72 (×1.10)	24.91 (×1.18)
100	37.87	36.41 (×1.04)	36.08 (×1.05)	36.10 (×1.05)	**35.94** (×1.05)	35.12 (×1.08)	33.39 (×1.13)
1 000	55.37	54.00 (×1.03)	53.87 (×1.03)	53.94 (×1.03)	**53.93** (×1.03)	53.03 (×1.04)	51.58 (×1.07)
10 000	108.28	108.73 (×1.00)	**108.08** (×1.00)	108.14 (×1.00)	**108.08** (×1.00)	107.52 (×1.01)	106.14 (×1.02)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	27.20	26.65 (×1.02)	26.71 (×1.02)	26.79 (×1.02)	**26.65** (×1.02)	26.07 (×1.04)	25.47 (×1.07)
100	46.48	46.90 (×0.99)	45.62 (×1.02)	**45.61** (×1.02)	45.96 (×1.01)	44.03 (×1.06)	42.18 (×1.10)
1 000	90.90	84.51 (×1.08)	**82.90** (×1.10)	83.24 (×1.09)	83.03 (×1.09)	81.95 (×1.11)	80.24 (×1.13)
10 000	208.49	184.41 (×1.13)	182.66 (×1.14)	**181.82** (×1.15)	183.22 (×1.14)	178.83 (×1.17)	176.03 (×1.18)

Table 12
Query processing tail latencies P_{95} (in milliseconds) for CW09 collection and EFF05 query log. Improvements (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	68.36	70.05 (×0.98)	**68.34** (×1.00)	68.73 (×0.99)	68.38 (×1.00)	67.41 (×1.01)	65.19 (×1.05)
100	81.10	80.69 (×1.01)	80.42 (×1.01)	80.68 (×1.01)	**80.58** (×1.01)	79.88 (×1.02)	77.97 (×1.04)
1 000	106.20	105.73 (×1.00)	105.18 (×1.01)	105.36 (×1.01)	**105.11** (×1.01)	104.88 (×1.01)	103.59 (×1.03)
10 000	188.15	188.15 (×1.00)	**187.98** (×1.00)	188.00 (×1.00)	188.10 (×1.00)	187.37 (×1.00)	185.49 (×1.01)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	60.88	63.32 (×0.96)	61.59 (×0.99)	62.22 (×0.98)	61.76 (×0.99)	60.63 (×1.00)	58.69 (×1.04)
100	113.72	108.64 (×1.05)	**106.96** (×1.06)	107.08 (×1.06)	107.17 (×1.06)	106.50 (×1.07)	102.71 (×1.11)
1 000	191.75	173.23 (×1.11)	172.51 (×1.11)	172.59 (×1.11)	**172.40** (×1.11)	170.61 (×1.12)	166.01 (×1.16)
10 000	368.28	308.21 (×1.19)	307.88 (×1.20)	**306.56** (×1.20)	307.38 (×1.20)	303.68 (×1.21)	298.43 (×1.23)

Table 13
Query processing tail latencies P_{95} (in milliseconds) for CW09 collection and EFF06 query log. Improvements (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	92.47	91.15 (×1.01)	**89.89** (×1.03)	90.28 (×1.02)	90.15 (×1.03)	88.82 (×1.04)	86.74 (×1.07)
100	109.45	108.73 (×1.01)	108.10 (×1.01)	**107.98** (×1.01)	108.12 (×1.01)	107.09 (×1.02)	104.67 (×1.05)
1 000	140.47	140.43 (×1.00)	**139.80** (×1.00)	**139.80** (×1.00)	**139.80** (×1.00)	139.70 (×1.01)	138.46 (×1.01)
10 000	235.93	236.58 (×1.00)	235.54 (×1.00)	235.24 (×1.00)	**235.54** (×1.00)	234.97 (×1.00)	234.79 (×1.00)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	118.94	116.69 (×1.02)	**116.18** (×1.02)	117.07 (×1.02)	116.28 (×1.02)	114.43 (×1.04)	110.77 (×1.07)
100	187.12	170.35 (×1.10)	**168.48** (×1.11)	168.92 (×1.11)	**168.48** (×1.11)	166.95 (×1.12)	160.78 (×1.16)
1 000	306.76	239.26 (×1.28)	**237.02** (×1.29)	237.38 (×1.29)	237.14 (×1.29)	234.57 (×1.31)	227.56 (×1.35)
10 000	571.05	428.00 (×1.33)	425.58 (×1.34)	**422.24** (×1.35)	423.58 (×1.35)	420.66 (×1.36)	417.49 (×1.37)

The methods over Blocked indexes behave quite differently. These exhibit moderate performance improvements for queries up to 5 terms but remarkable gains for longer queries. In the same direction, the performance becomes better according to k increases, up to x1.37% of speedup against the baseline (BW). These values are close to the *Oracle* method (x1.39), as in the previous case. These results show that BM performs better than BW on longer queries and the combination of both algorithms offers more significant time savings. A similar behavior appears when running the EFF06 queries. As an example, Fig. 11 shows the case for $k = 10$.

5.4. Threshold initialization

In this experiment, we explore the impact of threshold initialization methods on the performance of the proposed approach. Instead of estimating the threshold value, we select a clairvoyant method that *knows* in advance the final value of the k_{th} document in the results list. This configuration represents the best-case scenario and offers insights into the impact of this technique on our proposal. Table 18 shows the results using the CW12 document collection and EF6 query set.

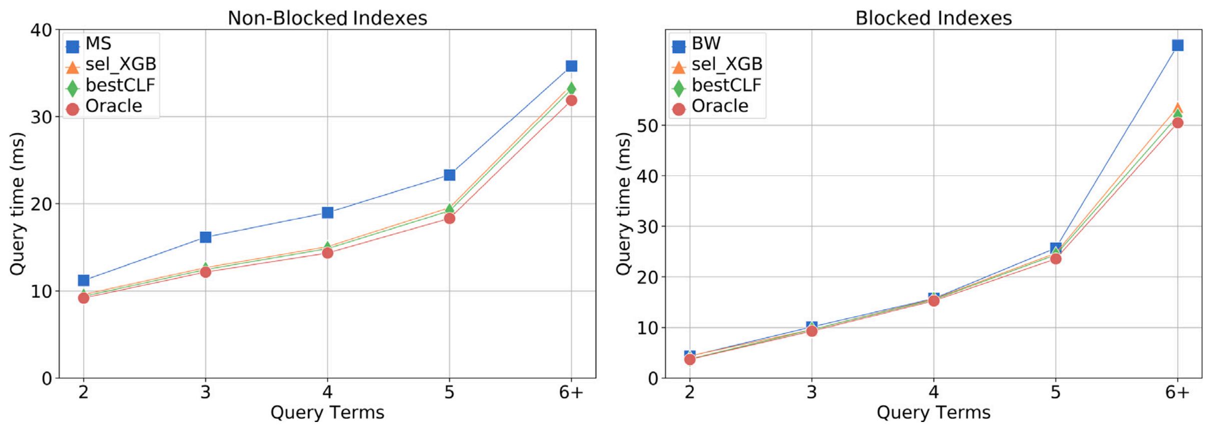


Fig. 11. Time averages by query length CW12 collection and EFF06 query log ($k = 10$).

Table 14
Query processing tail latencies P_{95} (in milliseconds) for CW12 collection and EFF05 query log. Improvements (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	76.31	75.36 (×1.01)	74.64 (×1.02)	**74.44** (×1.03)	74.64 (×1.02)	73.23 (×1.04)	71.44 (×1.07)
100	87.45	85.56 (×1.02)	**84.60** (×1.03)	84.67 (×1.03)	84.77 (×1.03)	83.52 (×1.05)	81.64 (×1.07)
1 000	112.36	111.42 (×1.01)	**110.83** (×1.01)	111.27 (×1.01)	111.25 (×1.01)	110.30 (×1.02)	108.44 (×1.04)
10 000	174.98	175.01 (×1.00)	**174.20** (×1.00)	175.20 (×1.00)	**174.20** (×1.00)	173.97 (×1.01)	172.56 (×1.01)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	51.46	52.21 (×0.99)	50.93 (×1.01)	50.98 (×1.01)	**50.65** (×1.02)	50.10 (×1.03)	49.32 (×1.04)
100	95.40	95.29 (×1.00)	93.33 (×1.02)	92.43 (×1.03)	**92.43** (×1.03)	91.86 (×1.04)	90.35 (×1.06)
1 000	166.70	154.80 (×1.08)	**153.45** (×1.09)	154.38 (×1.08)	**153.45** (×1.09)	150.72 (×1.11)	146.38 (×1.14)
10 000	336.13	278.09 (×1.21)	**277.92** (×1.21)	280.21 (×1.20)	278.46 (×1.21)	275.00 (×1.22)	269.33 (×1.25)

Table 15
Query processing tail latencies P_{95} (in milliseconds) for CW12 collection and EFF06 query log. Improvements (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	76.04	72.18 (×1.05)	**70.68** (×1.08)	70.87 (×1.07)	70.80 (×1.07)	69.79 (×1.09)	67.03 (×1.13)
100	87.45	86.59 (×1.01)	**85.39** (×1.02)	85.71 (×1.02)	85.44 (×1.02)	84.28 (×1.04)	82.05 (×1.07)
1 000	118.21	117.34 (×1.01)	116.74 (×1.01)	117.10 (×1.01)	**116.72** (×1.01)	115.83 (×1.02)	113.64 (×1.04)
10 000	189.86	191.00 (×0.99)	190.77 (×1.00)	189.91 (×1.00)	190.06 (×1.00)	189.68 (×1.00)	189.27 (×1.00)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	92.90	89.09 (×1.04)	**87.74** (×1.06)	**87.74** (×1.06)	**87.74** (×1.06)	85.80 (×1.08)	83.50 (×1.11)
100	142.97	132.38 (×1.08)	130.77 (×1.09)	130.57 (×1.09)	**130.03** (×1.10)	129.16 (×1.11)	124.37 (×1.15)
1 000	234.77	201.65 (×1.16)	202.09 (×1.16)	201.87 (×1.16)	**201.68** (×1.16)	198.26 (×1.18)	192.35 (×1.22)
10 000	448.72	353.84 (×1.27)	**351.87** (×1.28)	353.36 (×1.27)	**351.87** (×1.28)	348.99 (×1.29)	345.08 (×1.30)

The results compare against Table 9 and show that the threshold initialization offers more savings when using non-blocked indexes (up to x1.30 with $k = 10$), and this impact decreases for block-based indexes. However, when comparing a classifier-based result (i.e. sel_XGB) against the Oracle approach, we see that the methods are close to the best possible performance than the same algorithms without threshold initialization. These results suggest that both techniques positively complement each other, reducing the gap with the best possible (theoretical) improvements.

5.5. Combining methods, query lengths, and cut-off values

In our final experiment, we combine all the methods and variables we work with. As we mentioned, one possibility is to select the algorithm according to the length of a given query and the required number of results (k). We build a *new* combined baseline

Table 16
Query processing times averages (in milliseconds) according to different query lengths for non-blocked indexes, CW12 collection, and EFF05 query log. Speedups (in brackets) are computed with respect to Maxscore (MS).

	QL	MS	sel_Tree	sel_Forest	sel_XGB	bestCLF	Oracle
Top-10	2	15.78	13.57 (×1.16)	**13.35** (×1.18)	13.38 (×1.18)	13.27 (×1.19)	12.98 (×1.22)
	3	16.74	13.72 (×1.22)	13.47 (×1.24)	**13.40** (×1.25)	13.20 (×1.27)	12.90 (×1.30)
	4	18.61	15.11 (×1.23)	14.67 (×1.27)	**14.64** (×1.27)	14.34 (×1.30)	13.95 (×1.33)
	5	20.64	18.00 (×1.15)	17.48 (×1.18)	**17.24** (×1.20)	16.87 (×1.22)	15.75 (×1.31)
	6+	30.01	27.88 (×1.08)	27.70 (×1.08)	**27.34** (×1.10)	26.97 (×1.11)	25.82 (×1.16)
Top-100	2	18.92	16.35 (×1.16)	16.30 (×1.16)	**16.28** (×1.16)	16.11 (×1.17)	15.89 (×1.19)
	3	20.45	16.55 (×1.24)	**16.38** (×1.25)	**16.38** (×1.25)	16.19 (×1.26)	15.74 (×1.30)
	4	24.85	20.34 (×1.22)	20.06 (×1.24)	**19.94** (×1.25)	19.63 (×1.27)	18.98 (×1.31)
	5	29.70	25.64 (×1.16)	25.16 (×1.18)	**25.09** (×1.18)	24.35 (×1.22)	23.34 (×1.27)
	6+	43.58	41.27 (×1.06)	**40.86** (×1.07)	41.08 (×1.06)	40.51 (×1.08)	39.44 (×1.10)
Top-1000	2	26.75	23.42 (×1.14)	**23.29** (×1.15)	23.41 (×1.14)	23.14 (×1.16)	22.91 (×1.17)
	3	32.47	26.97 (×1.20)	**26.82** (×1.21)	26.90 (×1.21)	26.53 (×1.22)	26.01 (×1.25)
	4	41.99	36.13 (×1.16)	**35.82** (×1.17)	36.17 (×1.16)	35.38 (×1.19)	34.43 (×1.22)
	5	50.59	45.65 (×1.11)	45.48 (×1.11)	**45.30** (×1.12)	44.79 (×1.13)	43.47 (×1.16)
	6+	73.61	71.16 (×1.03)	70.99 (×1.04)	**70.96** (×1.04)	70.72 (×1.04)	70.03 (×1.05)
Top-10000	2	49.52	47.62 (×1.04)	**47.54** (×1.04)	47.55 (×1.04)	47.37 (×1.05)	46.73 (×1.06)
	3	69.05	64.66 (×1.07)	**64.32** (×1.07)	64.46 (×1.07)	63.92 (×1.08)	63.03 (×1.10)
	4	86.95	82.61 (×1.05)	**82.42** (×1.05)	82.52 (×1.05)	81.98 (×1.06)	81.08 (×1.07)
	5	105.32	102.94 (×1.02)	102.51 (×1.03)	**102.44** (×1.03)	102.20 (×1.03)	100.98 (×1.04)
	6+	147.33	146.46 (×1.01)	**146.26** (×1.01)	146.30 (×1.01)	146.12 (×1.01)	145.72 (×1.01)

Table 17
Query processing times averages (in milliseconds) according to different query lengths for blocked indexes, CW12 collection, and EFF05 query log. Speedups (in brackets) are computed with respect to Variable Block-Max WAND (BW).

	QL	BW	sel_Tree	sel_Forest	sel_XGB	bestCLF	Oracle
Top-10	2	5.87	5.60 (×1.05)	**5.43** (×1.08)	5.58 (×1.05)	5.43 (×1.08)	5.20 (×1.13)
	3	9.12	8.82 (×1.03)	**9.11** (×1.00)	**9.11** (×1.00)	8.81 (×1.04)	8.68 (×1.05)
	4	15.77	15.83 (×1.00)	**15.58** (×1.01)	15.66 (×1.01)	15.35 (×1.03)	14.82 (×1.06)
	5	21.19	20.56 (×1.03)	20.21 (×1.05)	**20.12** (×1.05)	19.88 (×1.07)	19.68 (×1.08)
	6+	54.02	43.87 (×1.23)	**42.47** (×1.27)	43.08 (×1.25)	41.69 (×1.30)	40.19 (×1.34)
Top-100	2	10.52	9.37 (×1.12)	**9.24** (×1.14)	9.25 (×1.14)	9.19 (×1.14)	8.92 (×1.18)
	3	15.71	15.46 (×1.02)	15.46 (×1.02)	**15.44** (×1.02)	15.16 (×1.04)	15.01 (×1.05)
	4	26.17	25.88 (×1.01)	25.02 (×1.05)	**24.95** (×1.05)	24.51 (×1.07)	23.95 (×1.09)
	5	34.76	32.17 (×1.08)	**31.88** (×1.09)	**31.88** (×1.09)	31.40 (×1.11)	31.10 (×1.12)
	6+	86.38	67.67 (×1.28)	**66.79** (×1.29)	67.07 (×1.29)	66.17 (×1.31)	64.35 (×1.34)
Top-1000	2	20.65	18.34 (×1.13)	**18.47** (×1.12)	18.55 (×1.11)	18.23 (×1.13)	17.85 (×1.16)
	3	31.18	29.62 (×1.05)	**29.74** (×1.05)	30.06 (×1.04)	29.28 (×1.07)	28.95 (×1.08)
	4	47.83	44.94 (×1.06)	**44.63** (×1.07)	44.78 (×1.07)	44.14 (×1.08)	42.97 (×1.11)
	5	63.86	59.45 (×1.07)	**58.45** (×1.09)	58.73 (×1.09)	58.10 (×1.10)	57.49 (×1.11)
	6+	153.03	118.62 (×1.29)	**117.21** (×1.31)	117.50 (×1.30)	116.20 (×1.32)	113.99 (×1.34)
Top-10000	2	50.67	45.81 (×1.11)	**45.47** (×1.11)	45.50 (×1.11)	45.36 (×1.12)	44.41 (×1.14)
	3	80.43	74.89 (×1.07)	**74.70** (×1.08)	74.97 (×1.07)	74.13 (×1.09)	73.12 (×1.10)
	4	114.92	106.10 (×1.08)	**106.04** (×1.08)	106.54 (×1.08)	105.15 (×1.09)	103.74 (×1.11)
	5	157.11	141.41 (×1.11)	**141.31** (×1.11)	141.84 (×1.11)	140.10 (×1.12)	138.37 (×1.14)
	6+	330.57	241.31 (×1.37)	**240.99** (×1.37)	241.15 (×1.37)	239.39 (×1.38)	237.01 (×1.39)

($comb\_BSL$) considering the best-performing algorithm between MS ad BW (our best methods for Non-Blocked and Blocked indexes, respectively) for each possible combination of k and query length. As an example, in the case we request the top-10 results, we use BW for $2 \leq |q| \leq 4$ while MS becomes the best when $|q| \geq 5$. We also propose another method ($comb\_XGB$) based on combining the strategies we introduce in this work, which considers using different top-k processing algorithms. In this case, we use our best method which relies on the XGBoost classifier. Specifically, for each length of q and k value, we use the $comb\_XGB$ method on the Blocked or Non-Blocked index organization according to the best-performing case. For example, when $k = 10$, $comb\_XGB$ uses the block-based algorithms for $2 \leq |q| \leq 3$ and non-block-based ones when $|q| \geq 4$. Fig. 12 shows the results for CW12 and the two testing query logs.

The first observation is that both combined methods improve the performance of a single algorithm (MS and BW). The *comb_XGB* method becomes the best in both cases (EFF05 and EFF06 query logs) and all values of k. The exact average values of MS and BW are those in Table 8 and 9. In the case of the EFF05 query log, *comb_XGB* reaches a speedup of x1.63 in the best case ($k = 10$) while the improvements for the second query set reach x1.29 (for the same k).

We also report the results of high percentile (P_{95}) tail latencies for these combined methods. Table 19 shows the results for the CW12 collection and the two query logs. For $k = \{10, 100, 1000\}$ the $comb\_XGB$ method helps to reduce tail latency, mainly for

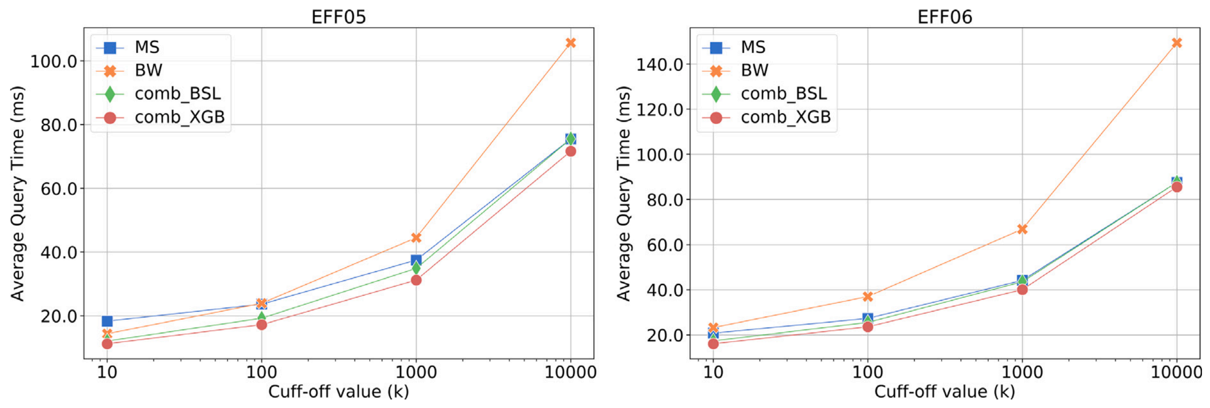


Fig. 12. Time averages for combined methods according to k for the CW12 collection using the two considered query logs.

Table 18
Query processing times with threshold initialization (using an oracle approach) averages (in milliseconds). The results correspond to the CW12 collection and EFF06 query log. Speedups (in brackets) are computed with respect to the baseline algorithm for each index type.

(a) Non-blocked indexes. Baseline: Maxscore (MS)

k	MS	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	17.37	13.73 (×1.26)	13.44 (×1.29)	13.35 (×1.30)	**13.41** (×1.30)	13.09 (×1.33)	13.09 (×1.33)
100	25.39	20.34 (×1.25)	20.20 (×1.26)	20.09 (×1.26)	**20.18** (×1.26)	19.65 (×1.29)	19.65 (×1.29)
1 000	40.23	34.52 (×1.17)	34.34 (×1.17)	34.25 (×1.17)	**34.31** (×1.17)	33.86 (×1.19)	33.86 (×1.19)
10 000	61.67	59.74 (×1.03)	59.52 (×1.04)	59.61 (×1.03)	**59.55** (×1.04)	59.20 (×1.04)	59.20 (×1.04)

(b) Blocked indexes. Baseline: Variable Blockmax Wand (BW)

k	BW	sel_Tree	sel_Forest	sel_XGB	sel_Vote	bestCLF	Oracle
10	16.12	15.58 (×1.03)	15.47 (×1.04)	15.54 (×1.04)	**15.51** (×1.04)	14.87 (×1.08)	14.87 (×1.08)
100	27.12	24.46 (×1.11)	24.36 (×1.11)	24.42 (×1.11)	**24.32** (×1.12)	23.99 (×1.13)	23.99 (×1.13)
1 000	45.47	42.32 (×1.07)	42.02 (×1.08)	42.00 (×1.08)	**41.96** (×1.08)	41.56 (×1.09)	41.56 (×1.09)
10 000	90.66	84.46 (×1.07)	84.35 (×1.07)	84.37 (×1.07)	**84.21** (×1.08)	83.48 (×1.09)	83.48 (×1.09)

Table 19
Query processing tail latencies P_{95} (in milliseconds) for CW12 collection. Improvements (in brackets) are computed with respect to the MS algorithm.

(a) EFF05 query log

k	MS	BW	comb_BSL	comb_XGB
10	76.31	51.46	55.33 (×1.38)	**52.21** (×1.46)
100	87.45	95.40	75.96 (×1.15)	**71.33** (×1.23)
1 000	112.36	166.70	110.68 (×1.02)	**106.81** (×1.05)
10 000	174.98	336.13	174.98 (×1.00)	174.98 (×1.00)

(b) EFF06 query log

k	MS	BW	comb_BSL	comb_XGB
10	76.04	92.90	68.53 (×1.11)	**65.26** (×1.17)
100	87.45	142.97	85.27 (×1.03)	**83.94** (×1.04)
1 000	118.21	234.77	119.51 (×0.99)	**115.95** (×1.02)
10 000	189.86	448.72	189.86 (×1.00)	189.86 (×1.00)

$k = 10$ (x1.46 of speedup with regards to the most competitive baseline (MS)). However, when considering $k = 10\,000$, the combined methods do not outperform MS but they improve considerably on BW.

6. Discussion and practical implications

The results show that our framework helps speeding-up query processing by selecting the (predicted) *best* top-k algorithm. We rely on safe techniques to avoid harming the final result list on a DAAT framework. Combining DAAT and SAAT techniques requires two separate underlying indexes, which have storage implications, mainly allowing indexes to reside in the main memory.

Besides, we evaluate our framework under two different index organizations. Our analysis is particularly useful on existing (and running) search architectures that usually rely on a single index organization. Given that both index organizations are compatible, it is possible to implement the blocked index using the same posting lists organization as a non-blocked one (just by adding block information in a separate data structure). In this way, it is feasible to use, for instance, Maxscore o Block-Max WAND when appropriate.

The results also show that for some specific queries, Block-Max Maxscore (which is not usually taken into account) offers non-negligible processing savings (i.e. longer queries) and may positively complement the state-of-the-art BW algorithm. From the point of view of a search provider, the shown reduction of average processing times (may translate into operational cost savings, while a lower tail latency helps to manage the load balancing of the search distributed system.

Furthermore, it is possible to apply the proposed approach on top of multi-tier index architectures. However, it may reduce the impact of traversing the biggest index layer, which requires a careful analysis we leave for future work. An alternative is to consider specific retrieval algorithms designed to work on multi-tier indexes on the decision framework, such as the case of the Waves method (Daoud et al., 2017).

7. Conclusions and future work

In this work, we propose and evaluate a combined strategy that considers many state-of-the-art disjunctive query processing algorithms and decides which one to use on a query basis. This proposal relies on the observation that no single algorithm performs the best for all queries. Their different characteristics, such as the number of terms or the number of documents to score, impact the final response time. We model and solve the problem on a machine-learned framework using different query features.

We also evaluate the proposal using three different document collections and three query sets, which enable us to train and test the machine learning model using separate data, thus preventing overfitting. These results offer some key baselines for practitioners to consider in a production setting. We also extend the range of k values considered, which may help to consider a setup for a single top-k search engine (i.e. top-10) or a more sophisticated one that relies on a cascading approach that requires more results for the upcoming phases (i.e. top-10000 to feed a Learning-to-Rank approach).

Our experimental analysis using standard datasets shows that the proposed approach outperforms strong baselines, thus improving efficiency in running times, mainly for more demanding setups (big document collections and large cut-off values). On ClueWeb12, our proposal shows a speed-up of up to 1.20x for non-blocked index organizations and 1.19x for block-based ones. Moreover, tail latencies show similar improvements on average, but with higher gains in particular configurations.

As a closing proposal, we combine different methods, query lengths, and cut-off values to build a decision framework that achieves up to 1.37 speedup improvements, even in this highly optimized environment.

We consider different directions to conduct further investigation. First, we plan to extend our work to consider other index organizations (i.e. impact ordered) and processing strategies (i.e. SAAT). This direction also includes the analysis of our proposal on top of multi-tier index architectures where the impact of our approach still needs to be clarified. Complementary, including additional parameters to the problem, such as time budgets or running limits, may be worth investigating. From the modeling perspective, one possible improvement is to evaluate more sophisticated classification methods, such as neural networks and a more extensive set of query features. In this case, it is essential to assess the trade-off of using more complex and costly approaches against the potential benefits. In particular, some models may be prohibitively slow in this time-constrained environment.

CRediT authorship contribution statement

Gabriel Tolosa: Conception and design of study, Acquisition of data, Analysis and/or interpretation of data, Writing – original draft, Writing – review & editing. **Antonio Mallia:** Conception and design of study, Acquisition of data, Analysis and/or interpretation of data, Writing – original draft, Writing – review & editing.

Data availability

Data will be made available on request.

Acknowledgment

Work partially funded by the Universidad Nacional de Luján (Argentina) Disp.CDCB: 284/21, and by the National Science Foundation Grant IIS-1718680 (USA).

References

Allan, J., Aslam, J., Carterette, B., Pavlu, V., & Kanoulas, E. (2008). Million query track 2008 overview. In *Proceedings of the seventeenth text retrieval conference (TREC 2007)*. Maryland, USA: National Institute of Standards and Technology (NIST).

Allan, J., Carterette, B., Aslam, J., Pavlu, V., Dachev, B., & Kanoulas, E. (2007). Million query track 2007 overview. In *Proceedings of the sixteenth text retrieval conference (TREC 2007)*. Maryland, USA: National Institute of Standards and Technology (NIST).

Anh, V. N., & Moffat, A. (2006). Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on research and development in information retrieval* SIGIR '06, (pp. 372–379). New York, NY, USA: Association for Computing Machinery.

Arapakis, I., Bai, X., & Cambazoglu, B. B. (2014). Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on research and development in information retrieval* SIGIR '14, (pp. 103–112). New York, NY, USA: Association for Computing Machinery.

Barroso, L. A., Clidaras, J., & Hölzle, U. (2018). *The datacenter as a computer: An introduction to the design of warehouse-scale machines, third edition*. California (USA): Morgan Claypool Publishers.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*(10), 281–305.

Bortnikov, E., Carmel, D., & Golan-Gueta, G. (2017). Top-k query processing with conditional skips. In *WWW '17 companion*, *Proceedings of the 26th international conference on world wide web companion* (pp. 653–661). Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee.

Bosch, A., Bogers, T., & Kunder, M. (2016). Estimating search engine index size variability: a 9-year longitudinal study. *Scientometrics*, *107*(2), 839–856.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

Broccolo, D., Macdonald, C., Orlando, S., Ounis, I., Perego, R., Silvestri, F., et al. (2013). Load-sensitive selective pruning for distributed search. In *Proceedings of the 22nd ACM international conference on information and knowledge management* CIKM '13, (pp. 379–388). New York, NY, USA: Association for Computing Machinery.

Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on information and knowledge management* CIKM '03, (pp. 426–434). New York, NY, USA: Association for Computing Machinery.

Büttcher, S., Clarke, C. L., & Soboroff, I. (2006). The TREC 2006 terabyte track. In *TREC, Vol. 6* (p. 39). Maryland, USA: National Institute of Standards and Technology (NIST).

Callan, J., Hoy, M., Yoo, C., & Zhao, L. (2009). Clueweb09 data set. URL: http://lemurproject.org/clueweb09/.

Carterette, B., Pavlu, V., Fang, H., & Kanoulas, E. (2009). Million query track 2009 overview. In *Proceedings of the eighteenth text retrieval conference (TREC 2009)*. Maryland, USA: National Institute of Standards and Technology (NIST).

Chakrabarti, K., Chaudhuri, S., & Ganti, V. (2011). Interval-based pruning for top-k processing over compressed lists. In S. Abiteboul, K. Böhm, C. Koch, & K. Tan (Eds.), *Proceedings of the 27th international conference on data engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany* (pp. 709–720). IEEE Computer Society.

Chen, R.-C., Gallagher, L., Blanco, R., & Culpepper, J. S. (2017). Efficient cost-aware cascade ranking in multi-stage retrieval. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval* SIGIR '17, (pp. 445–454). New York, NY, USA: Association for Computing Machinery.

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* KDD '16, (pp. 785–794). New York, NY, USA: Association for Computing Machinery.

Clarke, C. L. A., Craswell, N., & Soboroff, I. (2004). Overview of the TREC 2004 terabyte track. In E. M. Voorhees, & L. P. Buckland (Eds.), *NIST special publication*, *Proceedings of the thirteenth text retrieval conference, TREC 2004, Gaithersburg, Maryland, USA, November 16-19, 2004*. Maryland, USA: National Institute of Standards and Technology (NIST).

Clarke, C. L., Scholer, F., & Soboroff, I. (2005). The TREC 2005 terabyte track. In *TREC*. Maryland, USA: National Institute of Standards and Technology (NIST).

Crane, M., Culpepper, J. S., Lin, J., Mackenzie, J., & Trotman, A. (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* WSDM '17, (pp. 201–210). New York, NY, USA: Association for Computing Machinery.

Culpepper, J. S., Clarke, C. L. A., & Lin, J. (2016). Dynamic cutoff prediction in multi-stage retrieval systems. In *Proceedings of the 21st Australasian document computing symposium* ADCS '16, (pp. 17–24). Caulfield, VIC, Australia: Association for Computing Machinery.

Daoud, C. M., Silva de Moura, E., Carvalho, A., Soares da Silva, A., Fernandes, D., & Rossi, C. (2016). Fast top-k preserving query processing using two-tier indexes. *Information Processing & Management*, *52*(5), 855–872.

Daoud, C. M., Silva de Moura, E., de Oliveira, D. F., Soares da Silva, A., Rossi, C., & da Costa Carvalho, A. L. (2017). Waves: a fast multi-tier top-k query processing algorithm. *Information Retrieval Journal*, *20*, 292–316.

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, *56*(2), 74–80.

Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., & Shalita, A. (2016). Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* KDD '16, (pp. 1535–1544). New York, NY, USA: Association for Computing Machinery.

Dimopoulos, C., Nepomnyachiy, S., & Suel, T. (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* WSDM '13, (pp. 113–122). New York, NY, USA: Association for Computing Machinery.

Ding, S., & Suel, T. (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* SIGIR '11, (pp. 993–1002). New York, NY, USA: Association for Computing Machinery.

Fang, W., Marbach, T. G., Wang, G., & Liu, X. (2020). Hybrid dynamic pruning for efficient and effective query processing. In *Proceedings of the 29th ACM international conference on information and knowledge management* CIKM '20, (pp. 2013–2016). New York, NY, USA: Association for Computing Machinery.

Grand, A., Muir, R., Ferenczi, J., & Lin, J. (2020). From MAXSCORE to block-max wand: The story of how lucene significantly improved query evaluation performance. In J. M. Jose, E. Yilmaz, J. Magalhães, P. Castells, N. Ferro, M. J. Silva, & F. Martins (Eds.), *Advances in information retrieval* (pp. 20–27). Springer International Publishing.

Hwang, S.-W., Kim, S., He, Y., Elnikety, S., & Choi, S. (2016). Prediction and predictability for search query acceleration. *ACM Transactions on the Web (TWEB)*, *10*(3), 1–28.

Jeon, M., Kim, S., Hwang, S.-w., He, Y., Elnikety, S., Cox, A. L., et al. (2014). Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th international ACM SIGIR conference on research and development in information retrieval* SIGIR '14, (pp. 253–262). New York, NY, USA: Association for Computing Machinery.

Jonassen, S., & Bratsberg, S. E. (2011). Efficient compressed inverted index skipping for disjunctive text-queries. In P. D. Clough, C. Foley, C. Gurrin, G. J. F. Jones, W. Kraaij, H. Lee, & V. Murdock (Eds.), *Lecture notes in computer science*: *vol. 6611*, *Advances in information retrieval - 33rd European conference on IR Research, ECIR 2011, Dublin, Ireland, April 18-21, 2011* (pp. 530–542). Springer.

Kane, A., & Tompa, F. W. (2018). Split-lists and initial thresholds for WAND-based search. In *The 41st international ACM SIGIR conference on research and development in information retrieval* SIGIR '18, (pp. 877–880). Association for Computing Machinery.

Khattab, O., Hammoud, M., & Elsayed, T. (2020). Finding the best of both worlds: Faster and more robust top-k document retrieval. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval* SIGIR '20, (pp. 1031–1040). New York, NY, USA: Association for Computing Machinery.

Kim, S., He, Y., Hwang, S.-w., Elnikety, S., & Choi, S. (2015). Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search. In *Proceedings of the eighth ACM international conference on web search and data mining* WSDM '15, (pp. 7–16). New York, NY, USA: Association for Computing Machinery.

Lemire, D., & Boytsov, L. (2015). Decoding billions of integers per second through vectorization. *Software - Practice and Experience*, *45*(1), 1–29.

Lin, J., & Trotman, A. (2015). Anytime ranking for impact-ordered indexes. In *Proceedings of the 2015 international conference on the theory of information retrieval* ICTIR '15, (pp. 301–304). New York, NY, USA: Association for Computing Machinery.

Mackenzie, J., Culpepper, J. S., Blanco, R., Crane, M., Clarke, C. L. A., & Lin, J. (2018). Query driven algorithm selection in early stage retrieval. In *Proceedings of the eleventh ACM international conference on web search and data mining* WSDM '18, (pp. 396–404). New York, NY, USA: Association for Computing Machinery.

Mackenzie, J., Mallia, A., Petri, M., Culpepper, J. S., & Suel, T. (2019). Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *Advances in information retrieval: 41st European conference on IR research, ECIR 2019, Cologne, Germany, April 14–18, 2019, Proceedings, Part I 41* (pp. 339–352). Springer.

Mackenzie, J., & Moffat, A. (2020). Examining the additivity of top-k query processing innovations. In *Proceedings of the 29th ACM international conference on information and knowledge management* CIKM '20, (pp. 1085–1094). New York, NY, USA: Association for Computing Machinery.

Mackenzie, J., Petri, M., & Moffat, A. (2021). Anytime ranking on document-ordered indexes. *ACM Transactions on Information Systems*, *40*(1).

Mackenzie, J., Petri, M., & Moffat, A. (2022). Efficient query processing techniques for next-page retrieval. *Information Retrieval*, *25*(1), 27–43.

Mackenzie, J., Trotman, A., & Lin, J. (2021). Wacky weights in learned sparse representations and the revenge of score-at-a-time query evaluation. arXiv preprint arXiv:2110.11540.

Mallia, A., Ottaviano, G., Porciani, E., Tonellotto, N., & Venturini, R. (2017). Faster BlockMax WAND with variable-sized blocks. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval* SIGIR '17, (pp. 625–634). New York, NY, USA: Association for Computing Machinery.

Mallia, A., & Porciani, E. (2019). Faster BlockMax WAND with longer skipping. In *Advances in information retrieval: 41st European conference on IR research, ECIR 2019, Cologne, Germany, April 14–18, 2019, Proceedings, Part I 41* (pp. 771–778). Springer.

Mallia, A., Siedlaczek, M., MacKenzie, J., & Suel, T. (2019). PISA: Performant indexes and search for academia. In *CEUR workshop proceedings, Vol. 2409* (pp. 50–56).

Mallia, A., Siedlaczek, M., & Suel, T. (2019). An experimental study of index compression and DAAT query processing methods. In *European conference on information retrieval* (pp. 353–368).

Mallia, A., Siedlaczek, M., & Suel, T. (2021). Fast disjunctive candidate generation using live block filtering. In *Proceedings of the 14th ACM international conference on web search and data mining* WSDM '21, (pp. 671–679). New York, NY, USA: Association for Computing Machinery.

Mallia, A., Siedlaczek, M., Sun, M., & Suel, T. (2020). *A comparison of top-k threshold estimation techniques for disjunctive query processing* CIKM '20, (pp. 2141–2144). New York, NY, USA: Association for Computing Machinery.

Mitchell, T. M. (1997). *Machine learning*. New York: McGraw-Hill.

Ntoulas, A., & Cho, J. (2007). Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* SIGIR '07, (pp. 191–198). Association for Computing Machinery.

Petri, M., Culpepper, J. S., & Moffat, A. (2013). Exploring the magic of WAND. In *Proceedings of the 18th Australasian document computing symposium* ADCS '13, (pp. 58–65). Association for Computing Machinery.

Petri, M., Moffat, A., Mackenzie, J., Culpepper, J. S., & Beck, D. (2019). Accelerated query processing via similarity score prediction. In *Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval* SIGIR '19, (pp. 485–494). New York, NY, USA: Association for Computing Machinery.

Pibiri, G. E., & Venturini, R. (2020). Techniques for inverted index compression. *ACM Computing Surveys*, *53*(6).

Robertson, S., & Jones, K. S. (1976). Relevance weighting of search terms. *Journal of the American Society for Information Science*, *27*, 129–146.

Rokach, L., & Maimon, O. (2005). Decision trees. In *Data mining and knowledge discovery handbook* (pp. 165–192). Boston, MA: Springer US.

Schurman, E., & Brutlag, J. (2009). Performance related changes and their user impact. In *Presented at velocity web performance and operations conference, June 2009*. O'Reilly.

Shan, D., Ding, S., He, J., Yan, H., & Li, X. (2012). Optimized top-k processing with global page scores on block-max indexes. In E. Adar, J. Teevan, E. Agichtein, & Y. Maarek (Eds.), *Proceedings of the fifth international conference on web search and web data mining, WSDM 2012, Seattle, WA, USA, February 8-12* (pp. 423–432). ACM.

Shao, J., Qiao, Y., Ji, S., & Yang, T. (2021). Window navigation with adaptive probing for executing BlockMax WAND. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval* SIGIR '21, (pp. 2323–2327). New York, NY, USA: Association for Computing Machinery.

Siedlaczek, M., Mallia, A., & Suel, T. (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, (pp. 917–927). New York, NY, USA: Association for Computing Machinery.

Silvestri, F. (2007). Sorting out the document identifier assignment problem. In G. Amati, C. Carpineto, & G. Romano (Eds.), *Advances in information retrieval* (pp. 101–112). Berlin, Heidelberg: Springer Berlin Heidelberg.

Strohman, T., Turtle, H. R., & Croft, W. B. (2005). Optimization strategies for complex queries. In R. A. Baeza-Yates, N. Ziviani, G. Marchionini, A. Moffat, & J. Tait (Eds.), *SIGIR 2005: Proceedings of the 28th annual international ACM SIGIR conference on research and development in information retrieval, Salvador, Brazil, August 15-19, 2005* (pp. 219–225). ACM.

Tonellotto, N., Macdonald, C., & Ounis, I. (2011). Query efficiency prediction for dynamic pruning. In *LSDS-IR '11, Proceedings of the 9th workshop on large-scale and distributed informational retrieval* (pp. 3–8). New York, NY, USA: Association for Computing Machinery.

Tonellotto, N., Macdonald, C., & Ounis, I. (2013). Efficient and effective retrieval using selective pruning. In *Proceedings of the sixth ACM international conference on web search and data mining* WSDM '13, (pp. 63–72). New York, NY, USA: Association for Computing Machinery.

Turtle, H., & Flood, J. (1995). Query evaluation: Strategies and optimizations. *Information Processing and Management*, *31*(6), 831–850.

Valentini, G., & Masulli, F. (2002). Ensembles of learning machines. In M. Marinaro, & R. Tagliaferri (Eds.), *Neural nets* (pp. 3–20). Berlin, Heidelberg: Springer Berlin Heidelberg.

Wang, L., Lin, J., & Metzler, D. (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* SIGIR '11, (pp. 105–114). New York, NY, USA: Association for Computing Machinery.

Weinberg, A. I., & Last, M. (2019). Selecting a representative decision tree from an ensemble of decision-tree models for fast big data classification. *Journal of Big Data*, *6*(23).

Yafay, E., & Altingovde, I. S. (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* CIKM '19, (pp. 2457–2460). New York, NY, USA: Association for Computing Machinery.

Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, *38*(2), 6–es.

Gabriel Tolosa (https://tolosoft.gitlab.io/) is a professor/researcher of Computer Science at Universidad Nacional de Luján in Argentina. He holds a master's degree in Data Networks from the Universidad Nacional de La Plata and a Dr. degree in Computer Science from the Universidad de Buenos Aires. His research focuses mainly on information retrieval, efficient algorithms, and search technologies. The problems he works on are motivated by dealing with large-scale data.

Antonio Mallia (http://antoniomallia.it) completed his B.Sc. in Electronic Engineering in 2012 and his M.Sc. in Computer Science in 2015 both at the University of Pisa. He also holds a M.Sc. and a Ph.D. in Computer Science from New York University completed in 2020 and 2022 respectively. Antonio Mallia is currently working as an Applied Scientist at Amazon working in the Alexa AI Search team. His main research interests involve query processing, inverted index compression, GPU programming, and Deep Learning.