Information Retrieval Research 2 (2026) 35-58 Submitted 8/25; Revised: 2/26; Published 3/26

Simple Techniques for Efficient Top-k Batch Query Processing

Zhixuan Li ZHIXUAN.LI@UQ.EDU.AU
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, Queensland, Australia

Joel Mackenzie JOEL.MACKENZIE@UQ.EDU.AU
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, Queensland, Australia

Editor: Ismail Sengor Altingovde

Abstract

When faced with a large number of related tasks, like clearing unread emails or tackling a mountain of laundry, it's often most efficient to handle them together as a single *batch*. The simple observation is that grouping similar tasks together can ultimately streamline the entire process. In the context of Information Retrieval, batch processing has been applied to the problem of *conjunctive querying* by re-using partially computed results to avoid redundant list intersections or disk reads, reducing the total computational effort required to answer the given query batch. However, there has yet to be any work on exploiting batch query processing in the context of *disjunctive querying* semantics, which are often implemented by highly optimized top-k dynamic pruning algorithms. In this work, we explore two simple yet efficient approaches to reduce the computational cost of top-k querying in the batch processing regime. Our experimentation on two collections, and two unique query batches, demonstrates end-to-end cost reductions of up to 44% over standard query processing algorithms, and lays the foundation for future work towards more sophisticated top-k batch querying techniques.

Keywords: Batch Processing, Caching, Dynamic Pruning, Experimentation

1 Introduction

Efficiently processing queries is a long-standing problem in the field of Information Retrieval (IR), with work dating back several decades to the 1970s and 1980s (Thiel and Heaps, 1972; Buckley and Lewit, 1985). The motivation is simple; more efficient querying generally means that more queries can be processed within a reasonable amount of time – on a fixed hardware budget – enabling better scalability or reductions in end-to-end electricity costs and carbon emissions. An alternative view is that more efficient querying equates to a better user experience (Schurman and Brutlag, 2009), as more expensive and accurate models can be used under that same fixed resource budget while maintaining a reasonable response latency (Bai et al., 2017).

Although there has been a significant amount of work on improving the efficiency of query processing, most of this work has focused on reducing latency or increasing through-

 WWW: https://irrj.org DOI: 10.54195/irrj.23893

put, two fundamentally important aspects for delivering a high-quality user experience (Tonellotto et al., 2018). However, there has been little focus on efficiency from the perspective of total operational cost, or the associated electrical costs (Kayaaslan et al., 2011; Catena et al., 2016; Blanco et al., 2016) or downstream emissions (Scells et al., 2022; Chowdhury, 2012; Zuccon et al., 2023).

One exception is a thread of work on *batch query processing*, first explored by Ding et al. (2011), focusing on the key observation that there may be an entire class of queries that *do not* require immediate responses. This includes queries issued for refreshing caches, mining data, conducting internal testing, for external clients via API calls, or perhaps now to generate training data. Ding et al. investigated how these non-interactive queries might be processed holistically, as a large batch, to reduce the total computational cost of conjunctive query processing for both disk- and memory-resident indexes.

1.1 Contribution

In this work, we re-investigate the notion of batch query processing in IR. Unlike previous work, which has focused exclusively on *conjunctive querying* (Ding et al., 2011; Mackenzie and Moffat, 2023) – where *all* query terms must be present in order to return a document – we are the first to explore how batch query processing can be used in the context of ranked *disjunctive* querying, one form of top-k retrieval.

We propose two simple methods for improving the efficiency of disjunctive ranked retrieval algorithms in the batch processing regime. Both methods are based on the careful re-use of top-k score thresholds to accelerate future queries; one is a novel *static* method, which tries to find and compute promising thresholds *before* processing the batch; the other is a *dynamic* threshold caching approach which stores thresholds opportunistically for use in later queries (Yafay and Altingovde, 2019).

Our results demonstrate that batch querying costs can be reduced for disjunctive querying, even under highly optimized retrieval algorithms. We also demonstrate that these improvements translate to parallel processing contexts, allowing the time to process the entire batch to be reduced while retaining overall cost savings.

2 Background

Modern commercial search engines serve billions of queries each day over enormous volumes of data, incurring similarly enormous operating costs. Given this extreme scale, they are typically implemented as *multi-stage cascades*, where increasingly sophisticated models are applied to successively smaller subsets of documents to allow fine control over the tension between efficiency and effectiveness (Wang et al., 2010; Gallagher et al., 2019). The earliest phase of this process is known as candidate generation, where the goal is to return a large set of candidate documents for re-ranking, and is typically achieved via simple bag-of-words ranking models. Given a query q containing n unique terms, document d is scored as a sum of weights over those terms:

$$\mathcal{S}_{d,q} = \sum_{i=1}^{n} \mathcal{C}(t_i, d). \tag{1}$$

In this formulation, $\mathcal{C}(t, d)$ represents the contribution (or weight) of term t in document d, and captures any additive scoring function like BM25 (Robertson and Zaragoza, 2009), or modern learned sparse retrieval methods (Yates et al., 2024). It is worth noting that while other scoring formulations exist – perhaps including a static per-document quality prior such as pagerank or spam score (Page et al., 1999; Cormack et al., 2011) – we consider these as orthogonal to our study (Shan et al., 2012; Petri et al., 2013).

Typically, the top-k highest ranking documents are returned for further processing. In some circumstances, this formulation may be used to return documents directly to the end user, and in these cases, a small number of results would be retrieved (such as $k = 10$). For candidate generation, k is more likely to be in the hundreds or thousands. We denote the kth highest similarity score achieved when processing all documents in the collection against q as Θ_k. Note that this formulation is inherently *disjunctive* – there is no guarantee that a document will contain all n query terms, and documents can still rank highly even if they only contain a subset of query terms.

In this work, we assume an operating context that emulates a single processing server responsible for candidate generation in a large-scale search engine infrastructure.

2.1 Query Processing

While many index arrangements are possible (Crane et al., 2017; Fontoura et al., 2011), we use the common *document-ordered* index which facilitates fast *document-at-a-time* retrieval. In particular, we assume an inverted index stores a *postings list* for each term in the corpus vocabulary, and these postings lists contain monotonically increasing document identifiers and corresponding payload information. Those payloads can be assumed to store (quantized) $\mathcal{C}(d, t)$ values, or statistics such as term frequencies that allow $\mathcal{C}$ to be readily computed. Postings may be stored on-disk, in-memory, or in some combination thereof (Fontoura et al., 2011), depending on operational requirements and specifications of the hardware.

Given a query, document-at-a-time processing algorithms iterate across the corresponding postings lists simultaneously, allowing the full score of each document d to be computed before continuing to the next candidate document. *Dynamic pruning* algorithms like MaxScore (Turtle and Flood, 1995), WAND (Broder et al., 2003), or Block-Max WAND (Ding and Suel, 2011) facilitate rapid traversal of inverted indexes, often at a fraction of the cost of exhaustive algorithms (Mallia et al., 2019b; Mackenzie and Moffat, 2020). During processing, dynamic pruning algorithms track the evolving top-k highest scoring documents, using the lowest score of those k documents as a threshold, denoted θ. Any document with a score estimated to be below θ will be bypassed, accelerating retrieval without sacrificing effectiveness — so long as these estimates are *overestimates* of the true document score. In other words, the decision to score a document is made using a fast estimation of that document's true score, so an overestimation will ensure we do not skip scoring any candidate document that may actually form part of the top-k results list.

Document score estimations are typically obtained by storing the upper-bound value of $\mathcal{C}$ for each postings list in the index, and examining combinations of these scores to determine which document to score next. Figure 1 shows an example of this process; different combinations of term upper-bounds, in conjunction with the heap threshold θ,

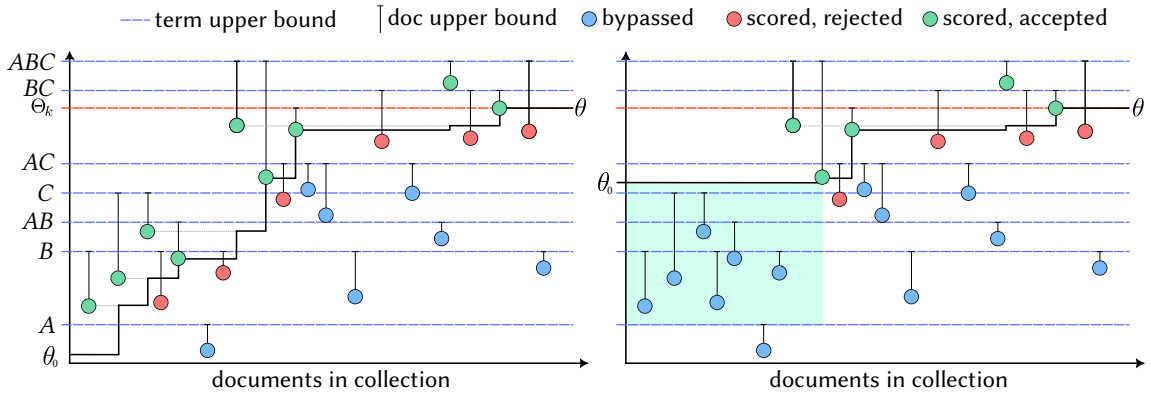

Figure 1: An example of how a dynamic pruning algorithm would find the top $k = 2$ documents on a toy document collection for a three term query (terms A, B, and C). In the left pane, the heap threshold θ grows naturally from zero; in the right pane, the initial threshold θ_0 is *primed* via some non-zero estimate, allowing additional documents to be bypassed (shown with a shaded background). The best possible setting of θ without sacrificing retrieval quality is the terminal value, shown as Θ_k. Figure adapted from Petri et al. (2019).

invoke different levels of skipping; the closer the value of θ to its terminal value, Θ_k, the faster retrieval will be.

This is the key direction of our investigation, and in the following paragraphs, we describe a number of approaches that can use specific properties of the query batch to estimate the *initial* value of θ, denoted θ_0, to accelerate query processing. We refer the interested reader to the work of Petri et al. (2013) and the survey of Tonellotto et al. (2018) for a more comprehensive description of dynamic pruning algorithms.

2.2 Batch Query Processing

Given a batch of queries $\mathcal{B}$, the goal is to return answers to all individual queries $q \in \mathcal{B}$ at a minimal total cost (where this cost is defined according to some metrics of interest such as total data read from disk, or total CPU cycles spent). Ding et al. (2011) were the first to explore batch processing for information retrieval, demonstrating considerable gains for ranked conjunctive queries on both disk and memory-resident inverted indexes (Zobel and Moffat, 2006). These gains were primarily due to the observation that, in batch processing, all queries are known ahead of time. This enables optimizations that would not otherwise be possible over query streams where future queries are not known, including reordering the sequence of queries, optimizing cache behavior, and re-using list intersections to reduce redundant processing. Recently, Mackenzie and Moffat (2023) re-examined this problem in terms of unranked conjunctions, demonstrating further improvements are possible by modeling the cost-vs-benefit of caching specific *pairs* of terms.

Despite the utility of these approaches – shown to achieve up to 2× speedups over standard query-by-query processing – top-k retrieval algorithms have not yet been explored in the batch processing regime. However, since top-k retrieval algorithms typically employ

disjunctive term matching semantics, previous batch optimization techniques based on list intersections are not viable. One exception is the work of Choudhury et al. (2018), which focuses on top-k *spatial-textual* queries for disk-resident indexes. However, their enhancements are tailored towards geospatial indexes, not classic text retrieval problems.

3 Fast Disjunctive Batch Processing

In this section, we outline a series of approaches that can be applied to the problem of disjunctive batch query processing, including simple baselines and novel offline and online processing mechanisms.

3.1 Query-at-a-Time Retrieval

The most obvious baseline, which we denote Naïve, is to simply process each query $q \in \mathcal{B}$ one-by-one with a highly optimized top-k processing algorithm. This has the same characteristics as a typical online retrieval system, and makes no use of the properties of $\mathcal{B}$ to improve efficiency. For each unique query, the initial value of the heap threshold, θ_0, will be initialized as 0, and will grow organically as the query is processed. It is worth noting that, although this algorithm does not make use of $\mathcal{B}$, it is embarrassingly parallel in nature, allowing it to easily scale up with the number of available CPUs if a reduced end-to-end batch-level *latency* is desired.

3.2 Clairvoyant Thresholding

On the other end of the spectrum, we also measure the cost of a Clairvoyant algorithm which "knows" the terminal — and thus optimal — threshold. While this algorithm is not attainable in practice, it represents the best possible speedup of the processing algorithm under threshold estimation; this is a useful yardstick since our primary focus in this work is on threshold-based techniques.

The Clairvoyant algorithm proceeds as follows. Before processing a given query, θ_0 is initialized to the query's corresponding Θ_k value, allowing the processing algorithm to bypass the maximal number of documents (see Figure 1). That is, each Θ_k is pre-computed and is assumed to be freely available for use by the Clairvoyant algorithm.

3.3 Deterministic Term-Based Thresholding

Although the clairvoyant baseline is not attainable in practice, other deterministic estimators are. The most simple of these augments the inverted index with the kth highest impact score for *every* term in the vocabulary for a set of common values of k (such as 10, 100, and 1,000) (Petri et al., 2019; Kane and Tompa, 2018; de Carvalho et al., 2015). Then, θ_0 can be set by taking the *maximum* of these kth highest pre-computed impacts across the terms in the query. This estimator is guaranteed to be safe, as taking the kth highest score for each term assures at least k greater-or-equal scoring documents will be retrieved. This mechanism is referred to as Q_k in our experiments. The key limitation of this approach is that it relies on the value of k being pre-determined, and may not be suitable for situations where k can vary widely.

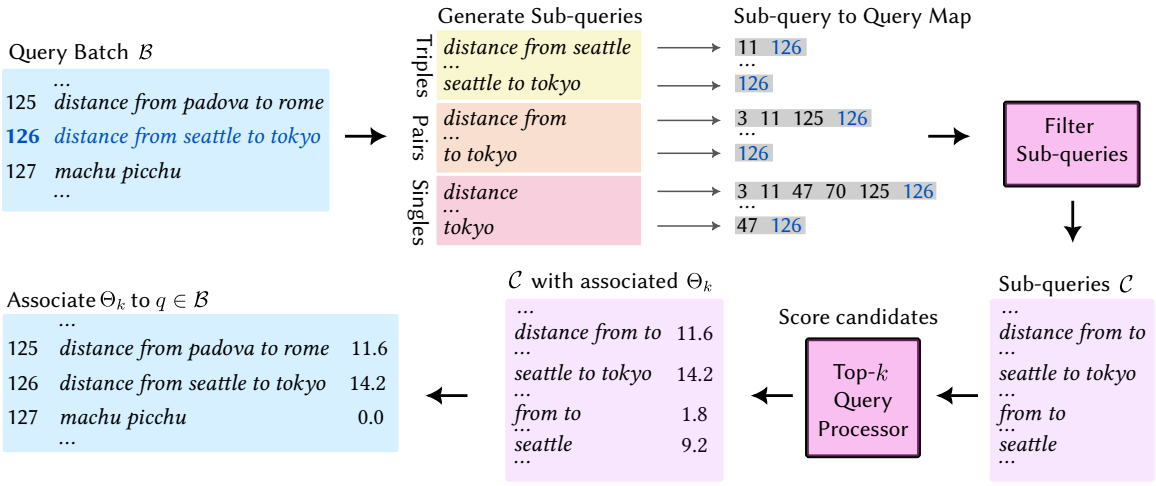

Figure 2: High-level sketch of the *static caching* algorithm with a focus on the query *"distance from seattle to tokyo."* Firstly, all 1-, 2-, and 3-term sub-queries are generated across the entire batch. These are then filtered such that only common sub-queries are considered; these sub-queries are then scored to find their optimal threshold values, and these thresholds are associated back to their parent queries to accelerate processing. In this example, the threshold 14.2 will be used, which was derived from the sub-query *"seattle to tokyo."*

3.4 Static Caching

Since the entire query batch is known ahead of processing, it should be possible to pre-determine a set of thresholds that can be used to expedite the processing of the queries within the batch itself. This approach, denoted *static caching* (SC), aims to associate each query with a best-endeavor threshold based on commonly occurring sub-queries across the entirety of $\mathcal{B}$. The idea is that we can selectively pre-process promising sub-queries that are common to many queries within the batch, with the aim of providing accurate *initial* thresholds to many queries without significant pre-processing costs. The SC algorithm consists of following main conceptual stages, and is outlined in Algorithm 1:

- **Generate**: We first generate all 1-, 2-, and 3-term sub-queries for all $q \in \mathcal{B}$. While computing these sub-queries, we maintain a mapping from sub-queries back to their parent queries (Algorithm 1, lines 1–4). While we could extract longer sub-queries, prior work has found that longer sub-queries are not typically useful as they do not commonly occur (Yafay and Altingovde, 2019).
- **Filter**: Secondly, we ignore any sub-query that occurs in fewer than F queries (Algorithm 1, line 9). The intuition here is that we only want to *pay* for processing a sub-query if it has a good chance of paying itself off later. We denote the remaining candidate query set as $\mathcal{C}$.
- **Score**: Each sub-query in $\mathcal{C}$ is then processed via the top-k retrieval algorithm to find its final threshold value Θ_k (Algorithm 1, line 11). Note that this query runs from a "cold start" – its threshold is initialized to zero.

Algorithm 1 Static Caching Algorithm
Input: An array of all $q \in \mathcal{B}$; a filtering factor F.
Output: The results for each query in $\mathcal{B}$.

```
1:  subqueries ← {}                          ▷ A set of sets mapping each subquery to its associated queries
2:  for q ∈ B do
3:      for all subquery ∈ q where |subquery| = {1, 2, 3} do
4:          append q.id to subqueries[subquery]
5:  thresholds ← {}                                  ▷ Value is initialized to 0 if not found
6:  for subquery in subqueries do          ▷ Iterate the keys of the subqueries structure
7:      qlist ← subqueries[subquery]                 ▷ Get list of associated query identifiers
8:      C ← ∅
9:      if |qlist| ≥ F then
10:         Add subquery to C
11:         Θ_k ← process_query(subquery, 0)                          ▷ Get heap threshold
12:         for qid ∈ qlist do        ▷ Loop over identifiers of all queries with this subquery
13:             thresholds[qid] ← max(Θ_k, thresholds[qid])
14: for q in B \ C do                                  ▷ Process remaining queries
15:     result ← process_query(q, thresholds[q.id])
```

- **Associate**: For each query in the batch that is associated with the current sub-query, we update its threshold if the newly found threshold is larger than the one currently associated (Algorithm 1 lines 12–13).
- **Process**: Finally, we process the entire query batch $\mathcal{B}$ using the thresholds obtained previously to "jump start" query processing (Algorithm 1, lines 14–15).

Figure 2 shows a high-level illustration of the conceptual steps taken by the SC algorithm.

Offline-vs-Online Compute With SC, there is a trade-off between the cost of pre-computing thresholds and the subsequent gains achieved during the final batch processing. If F is too high, the size of the candidate set $\mathcal{C}$ will be small, meaning that pre-processing will be fast but with perhaps little or no gain during batch processing. Conversely, if F is too low, the pre-processing cost of SC may outweigh any benefits arising in the batch processing stage. In our experiments, we deploy SC with three values of F (the number of queries that must contain a sub-query to be retained), namely $F = \{40, 80, 160\}$,[1] meaning that only commonly occurring sub-queries will be utilized; exploring alternative settings is left for future work. It is also worth noting that SC can be thought of as a *pre-processing* step to executing the batch, and may be used in conjunction with further optimizations.

Variant: All Singles We also experimented with a variant of the SC algorithm that pre-computes and caches *all single-term sub-queries* irrespective of their frequency. This is effectively the same as Algorithm 1 except that F is set to zero for single term sub-queries. The intuition is that these singles are likely to be cheap to compute, and appear

1. Selected from preliminary experiments on `MSMARCO-v1` with $F = \{10, 20, 40, 80, 160\}$ – we found that both $F = 10$ and $F = 20$ incurred pre-processing times that heavily outweighed any benefit gained during query processing. We do not experiment with these settings further.

more frequently than term pairs or triples across the batch. However, we found that this all-singles variant was always within about 1-2% of the vanilla SC approach, so we do not report it further.

Parallel Implementation Converting Algorithm 1 to a parallel algorithm is relatively trivial. The most expensive part of the static caching approach is the *scoring* phase, where the thresholds for the selected sub-queries are generated; this can be made parallel by simply rearranging the main `for` loop on line 6 to ensure there are no data races. In particular, we explicitly separate the *score* step from the *association* step on line 12–13 so there are no data races on the *thresholds* hash table, allowing the scoring to be done in a parallel `for` loop. Similarly, the final batch processing operation on line 14 can be converted into a parallel `for` loop.

3.5 Dynamic Caching

One of the key stages in the static caching algorithm involves *scoring* each candidate sub-query to gather its final threshold. Intuitively, processing the candidate sub-queries is no different to processing the query batch $\mathcal{B}$ itself; a query is handed to a top-k retrieval algorithm, and the top-k documents are returned along with their scores. A simple idea follows: Can we re-use the thresholds derived from processing the queries in $\mathcal{B}$ without requiring sub-queries to be computed at all? That is, can we arrange the query batch to maximize the likelihood of re-using the thresholds from previously processed queries?

Previously, Yafay and Altingovde (2019) proposed a family of rank-safe *threshold caching* approaches which store $\langle q', \Theta_k \rangle$ pairs in a hash table during query processing. In this work, we denote this family of methods (DC) to mean *dynamic caching* (as compared to the *static caching* method introduced in Section 3.4). Given a new query q, the hash table is probed for sub-queries of q until a suitable threshold is found, allowing the initial setting of θ to be safely estimated. They proposed three distinct heuristics, outlined as follows:

- DC1: Given a query q, all sub-queries of length 3 are probed; if multiple are found, the one with the maximum Θ_k is used. If no length 3 sub-query is found, we repeat the process for 2-term sub-queries; and then again for 1-term sub-queries. If no sub-query is found, θ_0 will be initialized to 0. Algorithm 2 outlines the operation of the DC1 variant for batch processing.
- DC2: Similar to DC1, all sub-queries of length 3, 2, and 1 are probed. However, instead of backing out once a non-zero threshold is found, this heuristic exhaustively enumerates *all* sub-queries. The intuition here is that, for example, some 1-term sub-queries may still have larger values of Θ_k than their 2-term sub-queries, and so enumerating all possible sub-queries will maximize the value of θ_0; see AB vs C for a concrete example of this in Figure 1. Algorithm 2 can be modified to the DC2 variant by simply omitting lines 8 and 9.
- DC3: Instead of examining 3-, 2-, and 1-term sub-queries, only sub-queries of length $|q|-1$ are probed; if found, these are expected to yield high thresholds, but they are less likely to appear in the map. However, there will also be fewer map look-ups, as the number of candidate sub-queries is linear in $|q|$. Algorithm 2 can be modified to the DC2 variant by modifying line 5 to examine only queries with $l = |q| - 1$.

Algorithm 2 Dynamic Caching Algorithm (Variant DC1)
Input: An array of all $q \in \mathcal{B}$.
Output: The results for each query in $\mathcal{B}$.

```
1:  B ← sort(B)                          ▷ On query length, ascending, and then lexicographically
2:  thresholds ← {}                                  ▷ Maps queries to their final threshold
3:  for q ∈ B do
4:      θ_0 ← 0
5:      for l ∈ {3, 2, 1} do                          ▷ Enumerate sub-queries of length l
6:          for all q' ∈ q where |q'| ≡ l do
7:              θ_0 ← max(θ_0, thresholds[q'])
8:          if θ_0 > 0 then          ▷ If a non-zero threshold was found, stop searching
9:              break
10:     ⟨result, Θ_k⟩ ← process_query(q, θ_0)
11:     thresholds[q] ← Θ_k
12:     output result
```

Adapting this technique to batch processing, we maintain a dynamic threshold cache, where each $\langle q, \Theta_k \rangle$ pair is added to this cache immediately after processing q. Observe, however, that the *order* in which queries are processed will have a large influence on the effectiveness of this approach, as a given query q can only benefit from the threshold cache if a previously executed query is a sub-query of q. As such, an important pre-processing step is required – we can optimize the cache behavior of the DC algorithms by simply sorting $\mathcal{B}$ in *ascending order of query length*, and then lexicographically *within* each length group, thereby maximizing the threshold cache hit rate. This simple modification to the original DC approach allows it to be tailored to the batch processing context, exploiting the fact that the full set of queries is known ahead of time.

Parallel Implementation Converting the DC algorithms into parallel versions is slightly more complex than for their static counterpart. Observe that in the main processing loop, the *thresholds* hash table is read for a set of sub-queries from each query (Algorithm 2, line 7). However, that same structure is also written to once each query has been processed (Algorithm 2, line 11), causing a data race when multiple readers/writers are present. While synchronization could be achieved via locking, this would significantly impact the efficiency of the algorithm.

Instead, we proceed by treating the querying process as a series of "rounds" – each made up of queries sharing the same length – and applying an explicit synchronization step between each round. In short, the parallel algorithm proceeds as follows. All single term queries are processed in parallel, and the resulting thresholds are maintained in a temporary array. Then, before processing all two term queries, these thresholds are propagated into the global *thresholds* cache (hash table). This process repeats for two term queries, and again for three term queries, and so on, treating each set of queries with the same length as a discrete, parallelizable operation. The key observation is that for the thresholds cache to be effective, it only needs to contain the thresholds for queries that are *shorter* than those currently being processed. Synchronization is fast, and simply involves inserting the

Table 1: Index statistics for the two document corpora used in our experiments. Note that both collections are *passage* collections, with relatively short documents.

Collection	Documents	Unique Terms	Postings	Size (GiB)
MSMARCO-v1	8,841,823	2,660,824	266,247,718	0.9
MSMARCO-v2	138,364,198	16,579,071	8,629,430,400	22.8

key/value pairs from the temporary array into the global *thresholds* table for the next round of processing.

4 Experimental Setup

Before discussing our results, we first outline our experimental setup and methodology.

4.1 Hardware

All experiments are conducted entirely in-memory on a Linux server with a 3.6GHz AMD Threadripper Pro 5975WX and 512GiB of RAM. Unless otherwise specified, experiments use a single processing thread on an otherwise idle machine, allowing end-to-end latency to be used as a proxy for total computing effort.

4.2 Document Collections

Following the recent batch processing study from Mackenzie and Moffat (2023), we use the MSMARCO-v1 and MSMARCO-v2 (Bajaj et al., 2018) passage collections, containing 8.8 and 138.4 million passages, respectively (see Table 1).

4.3 Query Batches

Since we expect batch processing to be useful for large query sets, we make use of the ORCAS (Craswell et al., 2020) query log as our primary batch. Following the same methodology of Mackenzie and Moffat (2023), the entire set of 10 million ORCAS queries were normalized (stopped, stemmed, case-folded) with the default English Lucene tokenizer. Then, all within-query duplicate terms were removed, and only queries with no out-of-vocabulary terms on both indexes were retained. This resulted in a batch containing 6,761,892 unique queries with 3.2 terms per query on average. We also make use of the training queries from the MSMARCO Web Search (MSM-WS) collection (Chen et al., 2024); more detail is provided in Section 5.7. Figure 3 plots the distribution of query lengths across each of these logs. While the distributions are similar, the ORCAS batch has a higher proportion of two-term queries, and a lower proportion of four-term queries.

4.4 Software and Settings

Our experiments were implemented within the efficient PISA system (Mallia et al., 2019a) and compiled using gcc 11.4.0 with -O3 optimization. We used the software from Yafay and Altingovde (2019) to re-implement their DC mechanisms. Each collection was indexed

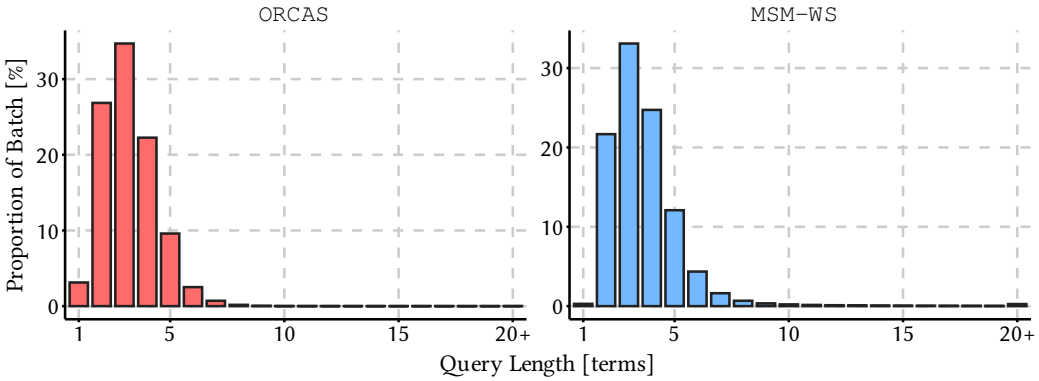

Figure 3: The distribution of query lengths across both the ORCAS (left, 6.76 million queries) and the MSMARCO Web Search (right, 3.83 million queries) query batches.

using Anserini (Yang et al., 2018) and converted to PISA with the CIFF tool (Lin et al., 2020). PISA indexes were reordered with bipartite graph partitioning (Dhulipala et al., 2016; Mackenzie et al., 2022), quantized to 8 bits (Crane et al., 2013) with PISA's BM25 variant (Robertson and Zaragoza, 2009; Kamphuis et al., 2020), and compressed with SIMD-BP128 (Lemire and Boytsov, 2015).

For query processing, we use two representative algorithms: the MaxScore algorithm (Turtle and Flood, 1995); and the highly optimized *Variable* Block-Max WAND (VBMW) algorithm (Mallia et al., 2017), with an average block size of 40 ± 0.5 elements. These settings were based on best practices recommended by Mallia et al. (2019b) and Mackenzie and Moffat (2020).

5 Experiments

This section describes our empirical experimentation, and subsequent analysis, focusing on overall time savings and space overheads, before conducting further analysis on the behavior of the proposed algorithms.

5.1 Time Improvements

Our first experiment applies each batch querying mechanism to process the entire ORCAS batch on both MSMARCO-v1 and MSMARCO-v2. The main metric of interest is the total time, in seconds, to process the batch, with lower values representing a more efficient processing regime. Table 2 and Table 3 present the end-to-end time required to process all 6.76 million queries in the batch for the algorithms we tested, on both small and large settings of k, the number of results to retrieve.

Firstly, we observe that the algorithmic Q_k predictor is a strong baseline, leading to latency reductions of between 4% and 33% over the naïve baseline.

The *static caching* algorithms manage to reduce querying latency further – up to 35%, with larger improvements observed with larger values of k. These improvements represent savings of up to 570 *minutes* worth of CPU time in the best case. Interestingly, there is

Table 2: Total time (including all pre-processing operations) and relative latency change over the Naïve baseline when executing all 6.76 million queries on both MSMARCO-v1 and MSMARCO-v2 indexes, using the MaxScore query processing algorithm.

(a) Setting: top-$k = 10$

Strategy	MSMARCO-v1		MSMARCO-v2	
	Total (sec)	Change (%)	Total (sec)	Change (%)
Naïve	3258	—	25,205	—
Q_k	2851	↓ 12.5	23,354	↓ 7.3
SC $F = 40$	2617	↓ 19.7	25,198	0
SC $F = 80$	2693	↓ 17.3	25,319	↑ 0.5
SC $F = 160$	2829	↓ 13.2	25,572	↑ 1.5
DC1	2379	↓ 27.0	21,460	↓ 14.9
DC2	2344	↓ 28.1	21,389	↓ 15.1
DC3	2429	↓ 25.4	22,019	↓ 12.6
Clairvoyant	1792	↓ 45.0	17,140	↓ 32.0

(b) Setting: top-$k = 1{,}000$

Strategy	MSMARCO-v1		MSMARCO-v2	
	Total (sec)	Change (%)	Total (sec)	Change (%)
Naïve	11,958	—	95,881	—
Q_k	8910	↓ 25.5	64,145	↓ 33.1
SC $F = 40$	7857	↓ 34.3	61,738	↓ 35.6
SC $F = 80$	7846	↓ 34.4	66,498	↓ 30.6
SC $F = 160$	7987	↓ 33.2	68,611	↓ 28.4
DC1	8399	↓ 29.8	55,343	↓ 42.3
DC2	7520	↓ 37.1	53,332	↓ 44.4
DC3	7936	↓ 33.6	55,562	↓ 42.1
Clairvoyant	7173	↓ 40.0	48,849	↓ 49.1

a clear balance between the total time taken and the value of F, with $F = 80$ performing best on the smaller collection, and $F = 40$ on the larger collection; we explore this trade-off in more detail shortly.

The three *dynamic caching* mechanisms demonstrate even further time savings – up to 87 minutes on MSMARCO-v1, and 709 minutes on MSMARCO-v2. Both DC1 and DC2 slightly outperform DC3, indicating that the cost of cache access (even across all 1, 2, and 3 term subqueries) is outweighed by the subsequent reduction in total processing cost once a suitable threshold is found; these trends hold irrespective of both the query processing algorithm, and the value of k applied.

Table 3: Total time (including all pre-processing operations) and relative latency change over the Naïve baseline when executing all 6.76 million queries on both `MSMARCO-v1` and `MSMARCO-v2` indexes, using the VBMW algorithm.

(a) Setting: top-$k = 10$

Strategy	MSMARCO-v1		MSMARCO-v2	
	Total (sec)	Change (%)	Total (sec)	Change (%)
Naïve	3492	—	19,597	—
Q_k	3155	↓ 9.7	18,793	↓ 4.1
SC $F = 40$	2790	↓ 20.1	19,248	↓ 1.8
SC $F = 80$	2902	↓ 16.9	19,411	↓ 1.0
SC $F = 160$	3044	↓ 12.8	19,675	↑ 0.4
DC1	2519	↓ 27.9	17,716	↓ 9.6
DC2	2484	↓ 28.9	17,546	↓ 10.5
DC3	2582	↓ 26.1	17,478	↓ 10.8
Clairvoyant	1850	↓ 47.0	13,006	↓ 33.6

(b) Setting: top-$k = 1{,}000$

Strategy	MSMARCO-v1		MSMARCO-v2	
	Total (sec)	Change (%)	Total (sec)	Change (%)
Naïve	14,213	—	65,967	—
Q_k	10,911	↓ 23.2	57,432	↓ 12.9
SC $F = 40$	9435	↓ 33.6	51,296	↓ 22.2
SC $F = 80$	9383	↓ 34.0	53,380	↓ 19.1
SC $F = 160$	9496	↓ 33.2	55,656	↓ 15.6
DC1	9021	↓ 36.5	47,296	↓ 28.3
DC2	9045	↓ 36.4	46,257	↓ 29.9
DC3	9541	↓ 32.9	48,781	↓ 26.1
Clairvoyant	8004	↓ 43.7	36,516	↓ 44.6

Overall, these results are encouraging, especially for larger values of k, with latency reductions generally ranging between 15% to 42% of the baseline latency. One exception is the larger `MSMARCO-v2` collection with $k = 10$, where only modest improvements were observed. We believe this is due to the scaling behavior of dynamic pruning algorithms (Gallagher et al., 2025). In particular, this setting (large collection, small k value) is more favorable to dynamic pruning algorithms than others which, in turn, means that the benefits of batch processing are eroded. Contrasting these findings to the Clairvoyant algorithm shows that there may still be significant room for improvement, although this idealistic baseline does not incur any cost for *finding* the optimal threshold, something that is impossible in practice.

Table 4: Space overhead, measured in MiB, for the given algorithms on the larger ORCAS batch. The Q_k algorithm assumes storage of three k settings for each term in the index; the other algorithms are sensitive to only the batch itself. Note that all variations of the DC algorithm use the same caching scheme, and thus have the same overhead.

Q_k	Static			Dynamic
	$F = 40$	$F = 80$	$F = 160$	
30–190	161	144	130	197

5.2 Space Overheads

Table 4 reports the maximum space consumption of each strategy in MiB. Note that the space usage of the Q_k strategy depends on the size of the vocabulary; we have assumed that the kth highest score is stored for three values of k, using four bytes each. For the Static and Dynamic algorithms, the cache grows in terms of the size of $\mathcal{B}$, rather than the index itself. Given that the indexes are 837 MiB and 22.7 GiB for MSMARCO-v1 and MSMARCO-v2, respectively, the relative overhead of the threshold cache is much higher on the smaller index, approaching 25% of the total index size. On the larger collection, however, this overhead is less than 1%, making it an attractive mechanism for large-scale batch processing tasks. We acknowledge that this space could also be used for alternative enhancements – such as caching postings lists, or pre-computing results for specific sub-queries – and we plan to investigate alternative uses of this space in future work.

5.3 Static Caching Costs

Looking further at the cost of each component of the SC algorithm reveals that the most significant component is, as expected, the *scoring* stage. Generating all sub-queries for the 6.76 million query batch was observed to take around 30 seconds; filtering out all sub-queries occurring less than F times takes about 1 second; and once the thresholds are available, associating them back to their parent queries in $\mathcal{B}$ takes around 1 second. However, the sub-query *scoring* operation was observed to take up to 229 seconds for MSMARCO-v1, and 1165 seconds for MSMARCO-v2, using VBMW under the most expensive setting of $F = 40$ (with $k = 1{,}000$). On the other end of that spectrum, sub-query scoring with $F = 160$ takes a total of 40 seconds on MSMARCO-v1, and 374 seconds on MSMARCO-v2. Yet, the total run-time shown in Table 3 demonstrates that savings arising from pre-processing are not recouped during the final query processing phase, indicating the absence of useful thresholds.

Based on these results, a clear avenue for improvement is to improve the sub-query selection process. For example, many queries contain common term pairs such as *"how do"* or *"who is,"* yet these pairs are unlikely to yield high Θ_k values, and may not accelerate querying at all, making their computation a waste of resources. On the other hand, the sub-query scoring only takes up to 3% of the total processing time, and given the large gap between the aspirational clairvoyant approach and the static caching algorithm, more intelligent (and perhaps resource intensive) sub-query selection could yield much better outcomes.

5.4 Dynamic Caching Hit Analysis

Next, we examine the cache behavior of the DC algorithms in more detail to provide a better understanding of their performance.

Since $\mathcal{B}$ contains only unique queries, it is impossible for the DC algorithms to yield a cache hit for any single term query; as such, the first 211,273 (single term) queries are cache misses (see Figure 3).

Both DC1 and DC2 use the same sub-query lookup sequence, differing only once a cache hit occurs; across the tested batch, only 1,693 queries (ignoring the aforementioned singles) were not associated with a threshold value for these heuristics. In other words, more than 97% of the roughly 6.7 million queries started with a non-zero value of θ under either DC1 or DC2, explaining their strong performance. On the other hand, DC3, which searches for only $|q| - 1$ length sub-queries, had 607,224 misses (about 9%), again ignoring the singles. This high cache hit ratio can be explained by multiple factors. Firstly, recall that the batch is sorted by query length, and then lexicographically within length groups. English language is known to be Zipfian in nature, meaning that common terms occur very frequently. Thus, a common term appearing as a single term query in the batch could result in a large number of cache hits in subsequent queries containing that term, as could a common pair, and so on. Secondly, the ORCAS query log is biased towards popular queries, as only queries that were submitted by multiple unique users *and* led to clickthroughs were retained in the log.

In contrast to the DC algorithms, the SC algorithm does not require any cache look-ups during query processing, with all sub-query association done during pre-processing. However, there is a clear benefit in caching the results of queries during processing; since all queries in $\mathcal{B}$ need to be processed, storing the resulting thresholds and consulting a cache – even considering the cost of look-ups themselves – seems to be a worthwhile optimization.

5.5 Batch Size Effects

In the previous experiment, query batches were made up of all available data. Our next experiment aims to quantify the influence of different batch sizes on the overall performance of the algorithms presented. In order to measure this, we generated random subsets of the ORCAS batch of various sizes, without replacement, and ran the best in-class algorithms from Section 5.1 on these subsets. Table 5 reports the outcomes. As expected, the proposed batch processing algorithms tend to perform similarly to the baseline approach for smaller batches, as there are fewer overlapping query terms to exploit. Interestingly, however, there are still clear benefits to applying batch processing algorithms on inputs that are as small as $|\mathcal{B}| = 10^4$ (with savings observed of up to 50%). From an operational perspective, however, batch processing is unlikely to be a worthwhile optimization for batches of this size, as the total computation required to process them is somewhat negligible; even processing a batch of 10^4 queries on the larger MSMARCO-v2 collection takes just over one and a half minutes with the naïve approach. On the other hand, even when batch processing does not largely help, it does not significantly *hurt* either, making it a reasonably safe optimization to incorporate into a large-scale querying system.

We remark that the notion of drawing *random* subsets of queries is rather strict for the batch processing algorithms as their performance depends on occurrences of overlapping sub-queries. Thus, it is possible that batch processing can potentially be a helpful

Table 5: Total time (including all pre-processing operations), in seconds, to process batches of varying sizes over both collections and settings of k. This experiment uses VBMW processing.

(a) Setting: $k = 10$

Strategy	MSMARCO-v1					MSMARCO-v2				
	10^3	10^4	10^5	10^6	All	10^3	10^4	10^5	10^6	All
Naïve	0.5	5	51	511	3492	5.8	33	284	2869	19,597
SC $F = 40$	1.0	10	106	482	2790	2.9	29	297	2925	19,248
DC2	0.5	5	52	472	2484	2.9	30	298	2892	17,546

(b) Setting: $k = 1{,}000$

Strategy	MSMARCO-v1					MSMARCO-v2				
	10^3	10^4	10^5	10^6	All	10^3	10^4	10^5	10^6	All
Naïve	2.0	21	211	2004	14,213	9.6	96	972	9748	65,967
SC $F = 40$	1.9	19	162	1455	9435	9.7	96	941	8615	51,296
DC2	2.0	21	207	1767	9045	9.8	98	977	8838	46,257

optimization on smaller batches, especially when those batches are likely to contain related queries, such as for processing a set of query variations (Benham et al., 2019) or topically clustered batches. We look forward to exploring this problem in future work.

5.6 Parallel Processing

In our prior commentary, we have assumed that the batch processing engine has no latency requirements, and that processing can proceed on a single CPU. However, we note that there may be situations where the total end-to-end time to process the batch may still need to be as fast as possible. To this end, we also explore how each top-k batch processing method scales in the multi-threaded processing scenario, with the aim of completing the entire query batch in the minimal end-to-end time.

Figure 4 shows the total time taken to process the batch over both indexes, using VBMW with $k = 1{,}000$. It is evident that the parallel processing optimizations do not lead to any adverse behavior, with almost perfect parallelism across all tested algorithms. In turn, this means the computational savings achieved from our enhanced batch query processing methods are not eroded under parallel processing, making them a suitable choice when the overall batch latency must be optimized while still saving on overall computing cost, making these optimizations quite practical for real-world systems.

5.7 Generalizability

Our final experiment aims to verify the generalizability of our batch processing approaches to other query batches. To build a new batch, we used the 9.2 million training queries from

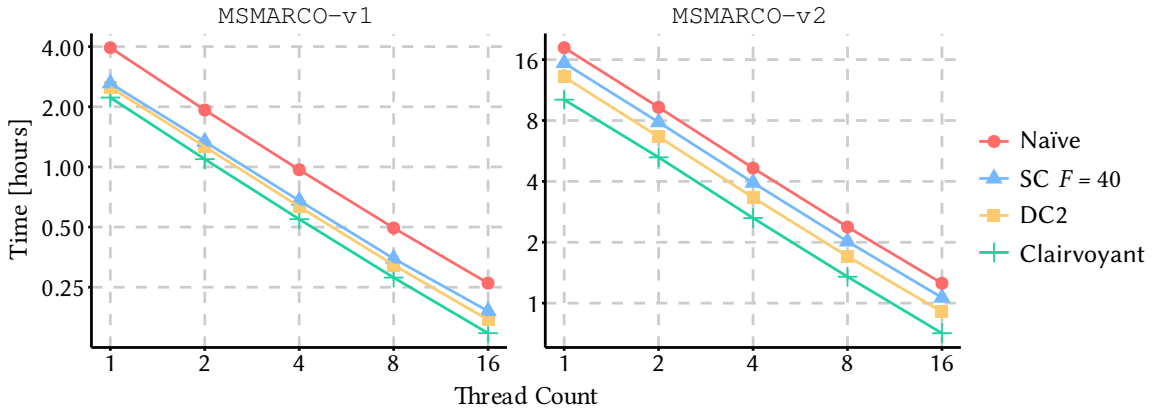

Figure 4: The total latency taken to process the entire batch, in hours, as the amount of processing threads increases. Both experiments are for k = 1,000 with VBMW. All approaches scale almost perfectly as threads are added – note the logarithmic axes.

Table 6: Total time (including all pre-processing operations), in seconds, to process two different batches over MSMARCO-v2 with VBMW and $k = 1{,}000$.

Strategy	ORCAS		MSM-WS	
	Total (sec)	Change (%)	Total (sec)	Change (%)
Naïve	65,967	—	63,677	—
Q_k	57,432	↓ 12.9	59,767	↓ 6.1
SC $F = 40$	51,296	↓ 22.2	59,054	↓ 7.3
SC $F = 80$	53,380	↓ 19.1	59,627	↓ 6.4
SC $F = 160$	55,656	↓ 15.6	60,459	↓ 5.1
DC1	47,296	↓ 28.3	58,662	↓ 7.9
DC2	46,257	↓ 29.9	58,494	↓ 8.1
DC3	48,781	↓ 26.1	61,753	↓ 3.0
Clairvoyant	36,516	↓ 44.6	44,096	↓ 30.8

the MSMARCO Web Search collection (Chen et al., 2024) as a starting point.[2] We then filtered out any non-English queries, normalized all queries, and removed duplicates or queries with out-of-vocabulary terms, following the original experimental setup (Mackenzie and Moffat, 2023). This resulted in a new batch containing 3,832,510 unique queries. While this batch is smaller than the ORCAS batch used in prior experiments, it has a longer average query length (3.7 vs 3.2 terms, respectively, see Figure 3). Repeating our experiments on this new batch showed similar trends; the Clairvoyant algorithm had a 30% to 44% improvement over the Naïve baseline, with both the DC and SC algorithms following behind with a modest 3% to 8% improvement. This result indicates that while our batch processing improvements

2. Note that, despite the name, the MSM-WS query set is not derived from (nor is aligned with) the MSMARCO corpora used in our experimentation – rather, it is based on the ClueWeb22 corpus.

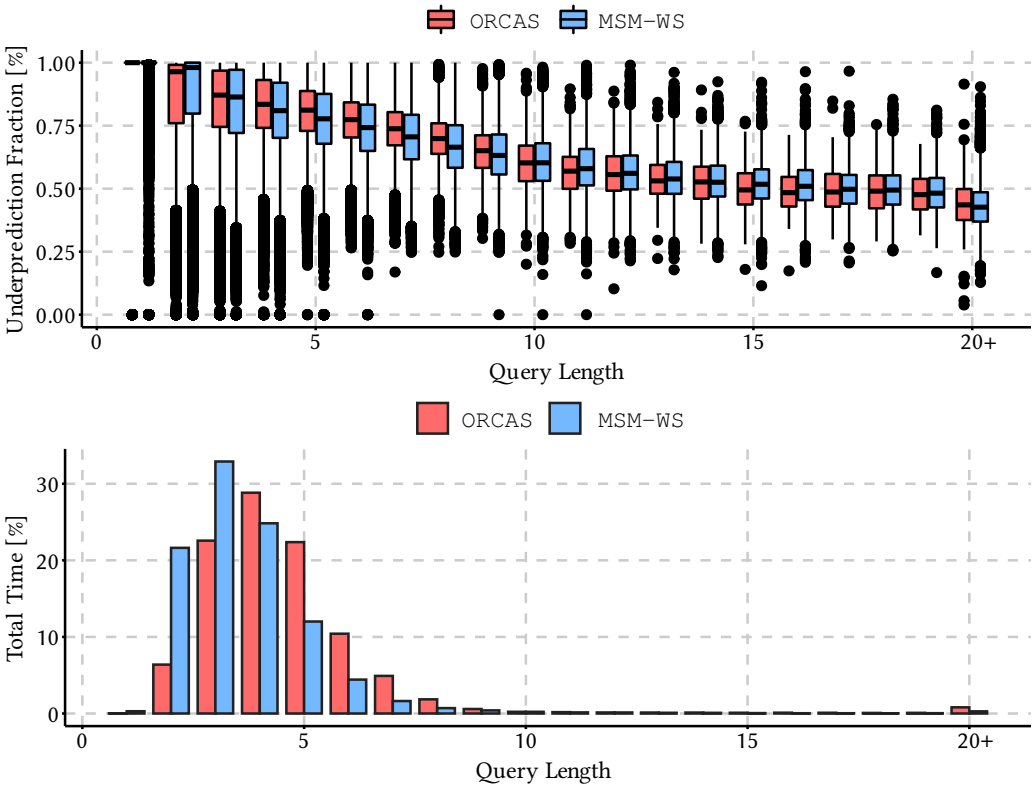

Figure 5: Top: The per-query threshold underprediction distribution across both logs, bucketed by query length. Bottom: The proportion of time spent processing queries across both logs, bucketed by query length. These figures use the same settings as in Table 6: VBMW processing with $k = 1{,}000$ on `MSMARCO-v2`.

do generalize to other query loads, the observed improvements almost certainly depend on the size and properties of the batch, and the corpus, itself.

To further illustrate this, Figure 5 plots the *underprediction fraction* (Petri et al., 2019) of the SC algorithm using $F = 40$. The underprediction fraction represents the difference, as a percentage, of the threshold that was applied to each query against the optimal threshold, meaning that a value of 1.0 represents a perfect initial threshold. We also plotted the distribution of time spent on processing queries of different lengths across the batch. We can observe that, as the length of the query increases, the estimations become worse. This is intuitive, because longer queries result in higher Θ_k scores, but we only use up to three-term sub-queries for initial threshold estimation. More importantly, we see that the underprediction fractions on the `ORCAS` log tend to be higher than on the `MSM-WS` log, especially for queries containing 3 to 9 terms which make up over 70% of the total batch processing time. The higher the underprediction fraction, the more savings accrue.

6 Discussion and Future Work

Our experiments demonstrated that the offline method is generally outperformed by its dynamic, online counterparts. Furthermore, comparing the savings of both approaches to the Clairvoyant yardstick shows that there is still room for improvement.

The key benefit of pre-processing term associations offline is that there is no need to maintain a cache on-the-fly; each query is already associated with the best possible threshold prior to querying, so no enumeration (or hashing) of sub-queries is required when processing the batch. However, as discussed in Section 5.3, there is also a delicate balance between the cost of pre-computing the thresholds for candidate sub-queries, and the savings realized from these thresholds during processing. In this work, we have only applied simple frequency-based cut-offs to determine whether a sub-query is included or not. More sophisticated regimes, including dynamic regimes based on features such as the length of a query, or the estimated utility of a given sub-query, could lead to better trade-offs than those presented here. We leave this investigation to future work.

One promising direction is to adapt both approaches into a *hybrid* algorithm, using both offline and online computation to accelerate the batch. It is unclear, however, whether these mechanisms are complementary (and hence, additive in their effects (Mackenzie and Moffat, 2020; Armstrong et al., 2009)), or if they are simply optimizing the same underlying inefficiencies.

Another possibility for closing this gap is to apply threshold *prediction* algorithms (Petri et al., 2019; Mallia et al., 2020), which use other features to predict the terminal value of the heap prior to processing each query. However, predictive algorithms come at the risk of *overestimation* which can harm effectiveness, something we did not consider here.

Another assumption made in this work is that the value of k – the number of documents to retrieve – is fixed across the entire query batch. However, there could be situations where each individual query is parameterized by a different value of k. In turn, this means that only thresholds computed for sub-queries to depth $k' > k$ can be utilized for rank-safe retrieval, making the batch processing task much more difficult. We leave this interesting idea for future investigation.

Finally, it would be worthwhile to revisit this research in terms of alternative scoring mechanisms, such as *learned sparse retrieval* (Yates et al., 2024), alternative indexing frameworks, like *approximate nearest neighbor* search over dense indexes (Bruch, 2024), or different query processing contexts, like *query-by-document*, which is used for building *corpus graphs* (Dunn et al., 2025).

7 Conclusion

We explored the problem of batch query processing for disjunctive top-k retrieval algorithms. We proposed two baseline methods, as well as two competitive approaches, all based on threshold estimation. These techniques reduce the overall cost of processing a large batch of queries. Our experiments demonstrated that a *dynamic caching* approach, based on the work of Yafay and Altingovde (2019), provided the most benefit, reducing the total cost of processing by as much as 44% – almost halving the processing time – with a small associated space overhead. We also found that these improvements translate to parallel

processing, allowing batches to be processed faster while still retaining cost reductions over naïve query-by-query processing. With many open problems yet to be answered, we hope that our work can motivate further studies into the problem of batch processing, especially for top-k retrieval.

Acknowledgments and Disclosure of Funding

The authors thank Alistair Moffat, as well as the anonymous referees, for useful conversations and suggestions. The second author was supported by a Google Research Scholar grant. In the interest of reproducibility, all experimental code is available at the following URL: `https://github.com/JMMackenzie/disbatch`.

References

T. G. Armstrong, A. Moffat, W. Webber, and J. Zobel. Improvements that don't add up: Ad-Hoc retrieval results since 1998. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 601–610, 2009.

X. Bai, I. Arapakis, B. B. Cambazoglu, and A. Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Trans. on Information Systems*, 36(2):1–42, 2017.

P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang. MS MARCO: A human generated MAchine Reading COmprehension dataset. *arXiv:1611.09268v3*, 2018.

R. Benham, J. Mackenzie, A. Moffat, and J. S. Culpepper. Boosting search performance using query variations. *ACM Trans. on Information Systems*, 37(4):41.1–41.25, 2019.

R. Blanco, M. Catena, and N. Tonellotto. Exploiting green energy to reduce the operational costs of multi-center web search engines. In *Proc. Conf. on the World Wide Web (WWW)*, pages 1237–1247, 2016.

A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 426–434, 2003.

S. Bruch. *Foundations of Vector Retrieval*. Springer, 2024. ISBN 9783031551826.

C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 97–110, 1985.

M. Catena, C. Macdonald, and N. Tonellotto. Load-sensitive CPU power management for web search engines. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 751–754, 2016.

Q. Chen, X. Geng, C. Rosset, C. Buractaon, J. Lu, T. Shen, K. Zhou, C. Xiong, Y. Gong, P. Bennett, N. Craswell, X. Xie, F. Yang, B. Tower, N. Rao, A. Dong, W. Jiang, Z. Liu, M. Li, C. Liu, Z. Li, R. Majumder, J. Neville, A. Oakley, K. M. Risvik, H. V. Simhadri, M. Varma, Y. Wang, L. Yang, M. Yang, and C. Zhang. MS MARCO Web Search: A large-scale information-rich web dataset with millions of real click labels. In *Proc. Conf. on the World Wide Web (WWW)*, pages 292–301, 2024.

F. M. Choudhury, J. S. Culpepper, Z. Bao, and T. Sellis. Batch processing of top-k spatial-textual queries. *ACM Trans. on Spatial Algorithms and Systems*, 3(4):13.1–13.40, 2018.

G. Chowdhury. An agenda for green information retrieval research. *Information Processing & Management*, 48(6):1067–1077, 2012.

G. V. Cormack, M. D. Smucker, and C. L. A. Clarke. Efficient and effective spam filtering and re-ranking for large web datasets. *Information Retrieval*, 14(5):441–465, October 2011.

M. Crane, A. Trotman, and R. O'Keefe. Maintaining discriminatory power in quantized indexes. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1221–1224, 2013.

M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman. A comparison of Document-at-a-Time and Score-at-a-Time query evaluation. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 201–210, 2017.

N. Craswell, D. Campos, B. Mitra, E. Yilmaz, and B. Billerbeck. ORCAS: 20 million clicked query-document pairs for analyzing search. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 2983–2989, 2020.

L. L. S. de Carvalho, E. S. de Moura, C. M. Daoud, and A. S. da Silva. Heuristics to improve the BMW method and its variants. *Journ. Information and Data Management*, 6(3):178–191, 2015.

L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1535–1544, 2016.

S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 993–1002, 2011.

S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel. Batch query processing for web search engines. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 137–146, 2011.

L. Dunn, L. Gallagher, and J. Mackenzie. Approximate bag-of-words top-k corpus graphs. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 174–182, 2025.

M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proc. Conf. on Very Large Databases (VLDB)*, 4(12):1213–1224, 2011.

L. Gallagher, R-C. Chen, R. Blanco, and J. S. Culpepper. Joint optimization of cascade ranking models. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 15–23, 2019.

L. Gallagher, J. Mackenzie, and A. Moffat. Empirical asymptotic growth of dynamic pruning mechanisms. In *Proc. Int. ACM SIGIR Conference on Information Retrieval in the Asia Pacific (SIGIR-AP)*, pages 325–330, 2025.

C. Kamphuis, A. P. de Vries, L. Boytsov, and J. Lin. Which BM25 do you mean? A large-scale reproducibility study of scoring variants. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 28–34, 2020.

A. Kane and F. W. Tompa. Split-lists and initial thresholds for WAND-based search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 877–880, 2018.

E. Kayaaslan, B. B. Cambazoglu, R. Blanco, F. P. Junqueira, and C. Aykanat. Energy-price-driven query processing in multi-center web search engines. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 983–992, 2011.

D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software Practice & Experience*, 41(1):1–29, 2015.

J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman, and A. de Vries. Supporting interoperability between open-source search engines with the common index file format. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 2149–2152, 2020.

J. Mackenzie and A. Moffat. Examining the additivity of top-k query processing innovations. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1085–1094, 2020.

J. Mackenzie and A. Moffat. Index-based batch query processing revisited. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 86–100, 2023.

J. Mackenzie, M. Petri, and A. Moffat. Tradeoff options for bipartite graph partitioning. *IEEE Trans. on Knowledge and Data Engineering*, 35(8):8644–8657, 2022.

A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 625–634, 2017.

A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: Performant indexes and search for academia. In *Proc. OSIRRC at SIGIR 2019*, pages 50–56, 2019a.

A. Mallia, M. Siedlaczek, and T. Suel. An experimental study of index compression and DAAT query processing methods. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 353–368, 2019b.

A. Mallia, M. Siedlaczek, M. Sun, and T. Suel. A comparison of top-k threshold estimation techniques for disjunctive query processing. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 2141–2144, 2020.

L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999. URL `http://ilpubs.stanford.edu:8090/422/`. Technical Report.

M. Petri, J. S. Culpepper, and A. Moffat. Exploring the magic of WAND. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 58–65, 2013.

M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck. Accelerated query processing via similarity score prediction. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 485–494, 2019.

S. E. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations & Trends in Information Retrieval*, 3:333–389, 2009.

H. Scells, S. Zhuang, and G. Zuccon. Reduce, reuse, recycle: Green information retrieval research. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 2825–2837, 2022.

E. Schurman and J. Brutlag. Performance related changes and their user impact. Velocity, 2009.

D. Shan, S. Ding, J. He, H. Yan, and X. Li. Optimized top-k processing with global page scores on block-max indexes. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 423–432, 2012.

L. H. Thiel and H. S. Heaps. Program design for retrospective searches on large data bases. *Information Storage and Retrieval*, 8(1):1–20, 1972.

N. Tonellotto, C. Macdonald, and I. Ounis. Efficient query processing for scalable web search. *Foundations & Trends in Information Retrieval*, 12(4-5):319–500, 2018.

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

L. Wang, J. Lin, and D. Metzler. Learning to efficiently rank. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 138–145, 2010.

E. Yafay and I. S. Altingovde. Caching scores for faster query processing with dynamic pruning in search engines. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 2457–2460, 2019.

P. Yang, H. Fang, and J. Lin. Anserini: Reproducible ranking baselines using Lucene. *Journal of Information Quality*, 10(4):1–20, 2018.

A. Yates, C. Lassance, S. MacAvaney, T. Nguyen, and Y. Lei. Neural lexical search with learned sparse retrieval. In *Proc. Int. ACM SIGIR Conference on Information Retrieval in the Asia Pacific (SIGIR-AP)*, pages 303–306, 2024.

J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38 (2):6.1–6.56, 2006.

G. Zuccon, H. Scells, and S. Zhuang. Beyond CO2 emissions: The overlooked impact of water consumption of information retrieval models. In *Proc. Int. Conf. on Theory of Information Retrieval (ICTIR)*, pages 283–289, 2023.