Early Termination Heuristics for Score-at-a-Time Index Traversal

Joel Mackenzie
RMIT University
Melbourne, Australia
joel.mackenzie@rmit.edu.au

Falk Scholer
RMIT University
Melbourne, Australia
falk.scholer@rmit.edu.au

J. Shane Culpepper
RMIT University
Melbourne, Australia
shane.culpepper@rmit.edu.au

ABSTRACT

Score-at-a-Time index traversal is a query processing approach which supports early termination in order to balance efficiency and effectiveness trade-offs. In this work, we explore new techniques which extend a modern Score-at-a-Time traversal algorithm to allow for parallel postings traversal. We show that careful integration of parallel traversal can improve both efficiency and effectiveness when compared with current single threaded early termination approaches. In addition, we explore the various trade-offs for differing early termination heuristics, and propose hybrid systems which parallelize long running queries, while processing short running queries with only a single thread.

1 INTRODUCTION

As collection sizes continue to grow, a major challenge for large scale search engines is the ability to return results reliably and efficiently. Prior work has shown that long response times directly impact the user experience, leading to both user abandonment and a loss of potential revenue [17, 30]. Since large scale systems must perform efficiently for as many queries as possible, the median and mean times are often less important than the 95th or 99th percentile response times, commonly referred to as the *tail latency*.

Although various approaches for reducing tail latency have been explored, the majority of this effort has been focused on document-ordered index layouts [14, 16]. Recently, Lin and Trotman [20] revisited impact-ordered indexes, showing that *Score-at-a-Time* (SaaT) index traversal is both efficient and effective in large-scale collections, and can be used to effectively control tail latency through early termination, at the expense of some effectiveness. Furthermore, a recent comparative study that compared document-ordered and impact-ordered indexes showed promise for impact-ordered indexes, as the efficiency of SaaT traversal with early termination is not substantially impacted by the length of the query, nor the number of required results [6]. This has implications for large-scale search systems, which will often retrieve a large set of candidate documents for further consideration by machine-learned rankers [22]. However, only fixed early termination strategies were explored in prior works, whereby the SaaT system has its execution terminated after processing a predefined and constant number of postings [6, 20].

In this paper, we explore the implications of different early-termination heuristics for SaaT retrieval [1], and explore how *selective parallelization* [14] can be implemented within a SaaT architecture to meet restrictive service level agreements (SLAs). We consider the following research questions:

RQ1 What are the efficiency and effectiveness trade-offs for current state-of-the-art early-termination heuristics?

RQ2 How can parallel processing accelerate Score-at-a-Time index traversal?

RQ3 How can long queries be processed efficiently without sacrificing effectiveness?

These issues contribute further knowledge on the behaviour of SaaT index traversal and early termination, including both the efficiency and effectiveness profiles of these heuristics. Furthermore, we explain the design and implementation of a parallel SaaT algorithm, and show how selective parallelization can be used to accelerate queries which are predicted to run slower than a pre-specified budget, while allowing short running queries to be processed in a single-threaded manner, thus reducing overall resource usage.

2 BACKGROUND & RELATED WORK

2.1 Query Processing

Ranked retrieval systems are designed to return the top-k documents for a given query q by applying a *similarity function* r onto a set of matching documents, and then ranking the matched documents by their respective scores. Each document can be assigned a similarity score S by summing the contribution of the query terms from q that also appear in d:

$$S_{d,q} = \sum_{t \in q \cap d} r(d, t)$$

The similarity function may represent any additive scoring function such as BM25, TF×IDF, and many others. Typically, r will weight the similarity of a document d with respect to a single term t by utilizing both term and document collection statistics such as *term frequency, document length, inverse document frequency*, and possibly other similar statistics.

Statistics used by r to rank documents need to be stored such that ranking can be carried out efficiently. In order to maintain and organize these statistics, an *index* is built across the document collection. The most common indexing approach is the *inverted index*. For each unique term t that is discovered during indexing, the inverted index stores a corresponding *postings list*, $\mathcal{L}_t$. Each document d that contains term t will have a corresponding *posting* in $\mathcal{L}_t$, which stores a unique Document Identifier (DocID) for d, as well as some payload information used for ranking, such as the

ADCS 2017, December 7–8, 2017, Brisbane, QLD, Australia

ACM ISBN 978-1-4503-6391-4/17/12…$15.00
https://doi.org/10.1145/3166072.3166073

number of occurrences of t in d (the term frequency, $f_{d,t}$). At query time, the postings list for each unique query term can be looked up, and then traversed efficiently, to find the top-k documents. The method used to traverse the postings lists is known as the *index traversal strategy*, and different strategies are most amenable to specific index organizations.

Document-ordered Indexes. *Document ordered* indexes ensure that each postings list is sorted by increasing DocID, and are most commonly used for *Document-at-a-Time* (DaaT) traversal, whereby each candidate postings list is traversed simultaneously, and a single document is scored before moving forward to the next document. Such indexes are well studied, and many efficient DaaT traversal algorithms have been explored in the literature, such as the DaaT MaxScore [31, 36], Wand [5], and Bmw [10, 11] algorithms.

Frequency-ordered Indexes. *Frequency-ordered* indexes group DocIDs together into "segments", where each segment contains a term frequency, followed by a corresponding sequence of DocIDs which contain that particular frequency. Then, the segments are ordered such that the *highest* frequency segments appear first. The aim of this organization is to efficiently process the most *important* segments first, as higher term frequencies usually result in larger similarity scores from r. Such processing can be realized using the *Term-at-a-Time* (TaaT) traversal strategy, whereby each postings list associated with the query is considered in isolation [26, 28]. Such processing requires an additional structure to store the partial results from r, known as an *accumulator table*. The accumulator table must efficiently support updating a score for any particular document identifier, and is usually implemented using a hashtable or array-based structure [15].

Impact-ordered Indexes. Taking the frequency-ordered organization one step further are *impact-ordered* indexes. The idea is the same as frequency-ordered indexes, but instead of storing a term frequency for each segment, the output of r is quantized before being stored. Since the quantization occurs at index time, the cost for calculating r is not factored into the live search system, which can result in more efficient query processing, at the cost of additional indexing time [1, 7]. Usually, impact-ordered indexes are processed using a *Score-at-a-Time* (SaaT) processing algorithm [1]. SaaT processing involves scoring segments in order of decreasing impact score, whereby the highest scoring segments are processed first. The main observation with SaaT traversal is that since the segments are processed in descending impact order, the largest score contributions will be added early on in the processing, and the ranking will be gradually refined as the traversal progresses. A recent implementation of an efficient SaaT query processing system is the Jass system, proposed by Lin and Trotman [20], which is the focus of this study.

2.2 Large Scale IR Systems

Since large scale IR systems need to service thousands of queries across millions or billions of documents per second, the system is designed to maximize efficiency. In particular, the document index is divided into many smaller *shards* [3, 18], and each shard is allocated to one or more *Index Server Nodes* (ISNs). When a query enters the search system, a *broker* decides which ISNs to query [18], and then forwards the query to those ISNs. The role of the ISN is simple; return the top-k results as efficiently as possible. After this, the broker may merge and then forward these results to other processing nodes, which may employ more resource-intensive approaches such as machine learned models to re-rank the candidate documents, which can then be returned to the end user.

2.3 Tail Latency

Traditionally, IR practitioners were concerned with improving the mean or median time that a system takes to process queries – by improving the efficiency, more queries could be processed at any given time, allowing greater throughput. Recently, however, more focus is being put on tail latency [9, 25].

Reducing the Tail. Various approaches have been taken to reduce tail latency, such as enforcing early termination [20], predicting and parallelizing long running queries [14, 16], improved scheduling [23, 37], selective pruning [4, 32], and query rewriting [24]. However, the majority of this effort has been conducted with Document-at-a-Time (DaaT) index layouts, which have vastly different performance characteristics to other index layouts and which, in comparison, have not been well explored [6]. The most relevant work to this study is the work from Jeon et al. [14], which uses an efficiency predictor to determine when a query is likely to be long running. If the query is long running, multiple threads are used to accelerate this process, whereas short running queries are processed with just a single thread to improve resource consumption of ISNs. This work was extended by introducing an even more advanced prediction framework, known as Delayed, Dynamic, Selective prediction, which focused on extreme (99.99th percentile) tail latencies. As discussed later, SaaT techniques need not apply sophisticated models, as the processing time is very consistent and predictable, which may not be the case for DaaT dynamic pruning algorithms [23].

Service Level Agreements. A service level agreement (SLA) is an agreed performance budget that the search system adheres to [13, 14, 16, 37]. SLAs can be set such that the entire search process does not degrade the experience for end users, and SLAs allow various parts of the entire search system to have distinct and precise budgets. Typically, a SLA will enforce a high percentile latency, which ensures the entire system does not often go over the pre-specified budget. In this paper, we explore efficient processing techniques which guarantee a particular running time, and minimize effectiveness loss. These guarantees are becoming increasingly important in large scale, commercial, multi-stage search engines where the initial index traversal is largely a "filter" which gathers as many candidates as possible in a fixed amount of time [12, 27].

3 SCORE-AT-A-TIME TRAVERSAL

In this section, we describe the inner workings of the Jass system, including the structures and algorithms used to efficiently process queries using a SaaT index traversal.

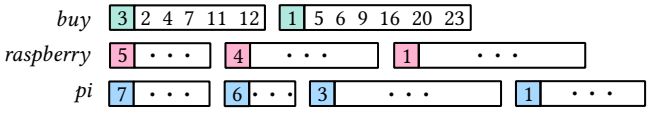

Figure 1: Internal representation of three JASS postings lists for the query '*buy raspberry pi*' . Each segment consists of an impact score, followed by an ascending list of document identifiers. Segments are processed from highest to lowest impact, and ties are broken by processing shorter segments first.

3.1 JASS Processing

System flow control. Given a set of candidate postings lists to process, each consisting of an impact score followed by an ascending list of document identifiers, the first step is to extract out the segment metadata and store it as a vector. This vector contains a $\langle$impact, offset, length$\rangle$ tuple for each segment in the provided set of postings lists, which stores the segment impact value, the byte offset in which the segment begins in the postings, and the length of the segment. For example, consider Figure 1. The metadata tuples for the term '*buy*' would be $\langle 3, b_1, 5\rangle$ and $\langle 1, b_2, 6\rangle$ where b_1 and b_2 represent the byte offset for the respective segments.

The metadata vector is then sorted in descending order of the impact values, with ties broken by length (shorter segments before longer segments). Posting processing is initiated by iterating through the sorted vector, and processing the next segment if and only if the sum of the processed postings plus the length of the candidate segment do not exceed the upper-bound number of postings to process, ρ, explained further below. After each segment is processed, the processed postings counter is updated by adding the length of the segment that was just processed. Once the stopping rule is applied, a top-k heap which is maintained by JASS can be iterated to efficiently return the top-k highest scoring documents.

Segment processing. To process a segment, the corresponding metadata tuple is used to access the correct byte offset to the desired segment. Then, the monotone list of document identifiers is decompressed into a local buffer. Next, JASS simply iterates the DocIDs in the buffer, adding the segment impact score to the corresponding accumulator for the given DocID. When an accumulator value is updated, JASS ensures that the top-k score heap is updated to reflect any potential change caused by the accumulator update. The simplicity of the flow control and lack of branching allows extremely efficient processing.

3.2 Aggression Settings

When processing a query with JASS, aggressive early termination is achieved through setting a parameter, ρ, which corresponds to the maximum number of postings which will be evaluated. A candidate segment will only be processed if the sum of previously processed postings plus the length of the candidate segment does not exceed ρ. We now describe alternative ways of setting ρ, and the implications of each.

For a given query q made up of n unique terms $t_1, t_2, \ldots, t_n$ with corresponding postings lists $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$, exhaustive processing requires setting ρ as:

$$\rho_{exhaustive} = \sum_{i=1}^{n} |\mathcal{L}_i|$$

In practice, this setting will cause processing efficiency to be sensitive to both the number of query terms provided, and the length of the corresponding postings lists.

One way to enforce aggressive early termination is to fix the value of ρ to some constant. This heuristic setting for ρ can be expressed as:

$$\rho_{fixed} = c, \text{ where } c > 0$$

Based on empirical observations from a set of web topics, Lin and Trotman [20] found that setting $c = 0.1 \cdot |C|$, where $|C|$ is the number of documents in the collection, yielded an efficient configuration without sacrificing too much effectiveness. This setting was shown to provide efficient and effective results across other collections [6, 19, 20]. With this heuristic, the processing time becomes largely independent of query length or the term statistics of query terms, as the algorithm will simply back out once it has processed ρ postings

Another possible approach is to set ρ based on a percentage of the candidate postings on a per-query basis:

$$\rho_{percentage} = \frac{z}{100} \cdot \sum_{i=1}^{n} |\mathcal{L}_i|, \text{ where } z > 0$$

Using this configuration, JASS will process a variable number of postings depending on both the value of z, the number of query terms, and the length of the corresponding postings lists. Intuitively, as z increases, so too would the effectiveness of the traversal, whereas the efficiency would decrease. This setting has not yet been explored in the literature, and is explored in this work.

3.3 Parallel Postings Traversal

In order to efficiently process long queries, or queries with a large number of candidate postings, we propose a simple extension to the JASS system which enables concurrent postings traversal. Similar extensions have been proposed for DAAT dynamic pruning algorithms [29], but are much more difficult to implement since dynamic pruning effectiveness depends on tracking dynamic thresholds which are then used to avoid processing documents that can not make the top-k results. SAAT traversal is much more amenable to parallelization, as the various worker threads need not communicate with each other during processing. We now explain this extension, partially answering RQ2 in the process.

Multi-threaded flow control. In the multi-threaded extension of JASS, the main processing loop explained above is modified to be *thread-safe*. Firstly, the metadata vector is obtained as in the single-threaded version. Next, we divide the processing load by allowing each thread T_n to process every nth segment in the segment vector, each updating a *local* processed postings counter. To this end, the early termination heuristic is on a per-thread level rather than a global level as in the single-threaded implementation.

Multi-threaded segment processing. Segment processing is similar to the single-threaded version, with a few additional caveats (Algorithm 1). Since multiple segments can be processed in parallel,

Algorithm 1: Multi-threaded Segment Processing

Input : A segment metadata tuple m, the accumulator table, A, and the lock table, L
Output : None

```
1 B ← decompressSegment(m)
2 for d in B do
3     while getLock(L_d) do
4         continue
5     end
6     A_d ← A_d + impact(m)
7     releaseLock(L_d)
8 end
9 return
```

there is no guarantee that an accumulator can be safely updated, as another thread may currently be updating the score. To rectify this issue, we associate an atomic flag with every DocID. When a thread wishes to update the accumulator of a particular document, it must first obtain the flag (lines 3–5), at which point the thread is safe to update the accumulator (line 6), then release the flag (line 7). The atomic flag is implemented in user space to avoid expensive kernel calls, and allows each active thread to remain on its respective processor without an expensive context switch occurring. This synchronization technique can be thought of as a lightweight spinlock. The process of 'locking' and 'unlocking' the atomic flag is indeed an atomic operation, and is therefore thread-safe.

Managing the accumulator table. By default, JASS manages its accumulators using the efficient accumulator initialization method of Jia et al. [15]. The main idea of this approach is to break the global accumulator table into a number of subsets, each subset associated with a single bit. At query time, the accumulators in a given subset are initialized only if one of the accumulators in the subset needs to be set. At this point, the subset of accumulators are initialized to 0, and the associated bit is set. The key problem with this approach is that adding a contribution to an accumulator involves the checking (and potentially setting) of the bit, which is not thread safe. Indeed, synchronization methods could be applied to the reading/writing of the bit, but this will result in a performance hit (as it is much more likely to be a point of contention between multiple worker threads). For this reason, we revert to using a simple, global accumulator table, with the overhead of initializing *all* accumulators before processing. Using `uint16_t` accumulators, and the vectorized STL `std::fill` operation, this took 13 ± 1.5 ms across 100 individual benchmarks.

Managing the heap. Another major issue with concurrent accumulator access is that the top-k heap cannot be efficiently maintained. For example, assume we are using 8 threads to process a query. At any time, there may be up to 8 threads updating unique accumulator values. If we wish to maintain a top-k heap, then the heap updates must be made thread-safe, which results in a large efficiency decay due to contention. To remedy this, we do away with the score heap entirely, and opt for iterating the accumulator table once the postings have been traversed, collecting the top-k documents, and returning them once the process is complete. Prior work has found that increasing k does not greatly impact the performance of (single threaded) JASS, since the only practical difference in processing is the number of heap operations that will occur [6]. By removing the score heap, this is taken one step further with parallel JASS, where the value of k has no notable difference on processing efficiency, since a constant overhead is added (regardless of k). Across three sets of 100 individual benchmarks with $k = \{10, 100, 1000\}$, this operation took 71 ± 3ms, with very little variance between the different values of k.

Query Length	2	3	4	5	6	7+	Total
Count	620	1,744	1,881	891	363	183	5,682

Table 1: The number of queries for each length across the UQV collection. Stopwords and duplicate terms were removed from queries before processing, and 83 single term queries were dropped.

4 EXPERIMENTS

4.1 Experimental Setup

We use the original implementation of JASS[1] to run all single threaded experiments. Our multi-threaded implementation of JASS was derived from this codebase, and is made available[2] for reproducibility. Experiments were conducted on the standard TREC ClueWeb12-B13 collection, which contains 52,343,021 web documents. This collection was indexed using the ATIRE system [34], which JASS then reads and rewrites into its own internal structure. Since JASS requires quantized indexes, ATIRE builds its postings with 9 bit quantization [7] after applying the BM25 similarity model. The JASS indexes were compressed using the QMX [33, 35] compression scheme. Although recent work has shown that *uncompressed* indexes can outperform compressed indexes when using SAAT retrieval strategies, the additional space overhead for such gains can be prohibitive [21]. In any case, the findings presented here are orthogonal to the compression scheme, as the same compression scheme is used for all instances of the search system.

In order to test our hypothesis that a fixed ρ setting reduces system effectiveness as query length increases, we use the UQV100 collection of Bailey et al. [2]. This collection contains 5,764 queries based on a set of 100 "backstories", and contains shallow judgments suitable for measuring early precision metrics. After normalization, which included s-stemming, stopping, and removal of duplicate terms within a query, 5,682 queries of varying lengths remained (Table 1). Note that in our analyses, we group all queries with 7 or more terms together (7+).

Timings are performed in memory on an idle Red Hat Enterprise Linux Server (v7.2) with two Intel Xeon E5-2690 v3 CPUs and 256GB of RAM. Each CPU has 12 physical cores with hyper-threading enabled, resulting in 48 available processing cores. Algorithms were compiled using GCC 6.2.1 with the `-O3` optimization flag. All timings are reported in milliseconds unless otherwise stated, and

[1] http://github.com/lintool/jass
[2] http://github.com/JMMackenzie/Multi-Jass

are the average of 3 runs. We denote mean values in boxplots with a diamond.

4.2 Early Termination Trade-Offs

Our first experiment explores the relationship between the early termination heuristic, and the effectiveness of the processed query. In order to measure the effectiveness loss, we first ran all 5,682 queries exhaustively using $\rho_{exhaustive}$. Next, we ran JASS using the percentage based heuristic $\rho_{percentage}$ for $z = \{5, 10, \ldots, 95\}$. We also ran JASS using the fixed heuristic ρ_{fixed} for various values of c between 250 thousand and 75 million. Then, for each heuristic configuration, we retrieved the top-10 documents for each query, and computed the difference in NDCG@10, with respect to the exhaustive result. Next, we calculated the proportion of wins, ties, and losses that the heuristic traversal had with respect to the exhaustive traversal. Since some improvements or losses may have been small, we consider deltas $\leq 10\%$ of the exhaustive value to be a tie. Figure 2 shows the density of wins, losses, and ties when comparing both fixed and percentage-based heuristics with the exhaustive run across various query lengths. As the length of the query increases, so too does the magnitude of postings that must be processed to achieve a close-to-exhaustive performance. This indicates that setting a fixed value of ρ may result in reduced effectiveness as the query length (and, more importantly, the number of candidate postings) increases. Additionally, the percentage setting appears to give a more predictable effectiveness trade-off than the fixed setting.

Our next experiment explores the behaviour of the different heuristic settings with respect to execution time. From each heuristic, we select four configurations, and each retrieves the top-10 documents. For the fixed setting, we set c as $c = \{5, 10, 15, 25\}$ million postings, and for the percentage setting, we set $z = \{20, 40, 60, 80\}$. We also run an exhaustive instance of JASS. The results are shown in Figure 3. As expected, fixing ρ to be a constant value allows for strict control of the upper-bound processing time, whereas using percentage-based settings may result in more variance.

So, to answer RQ1, the added control on the effectiveness trade-off provided by using percentage-based heuristics results in a loss of control on the tail latency. Conversely, fixing ρ, while reducing effectiveness, allows very strict control of the tail-latency.

4.3 Impact of Threading on Efficiency

Next, we wish to test the efficiency of our proposed parallel extension to the JASS system, and in particular, how the number of threads impacts the efficiency of the processing for varying query lengths, thereby showing the improvements of our proposed approach and answering RQ2. For simplicity, we assume that when processing in parallel, we will exhaustively process all candidate postings lists ($\rho = \rho_{exhaustive}$). We ran the parallel version using between 8 and 40 threads inclusive, increasing by 8 threads between each run. We then calculated the speedup as the average percentage improvement of the threaded run with respect to the exhaustive, single-threaded run, across each query length (Table 2).

Somewhat unsurprisingly, we notice that the speedup for a given number of threads generally increases as the length of the query increases. This is because the processing cost for longer queries is dominated by the postings traversal, whereas the cost for shorter queries is more often a combination of both the index initialization, accumulator traversal, and so on.

Threads	Query Length					
	2	3	4	5	6	7+
8	0.78	1.25	1.03	1.74	1.81	1.92
16	1.03	1.60	1.57	2.44	2.59	2.75
24	1.13	1.50	1.81	2.70	2.22	3.04
32	1.18	1.75	1.98	**2.73**	2.98	3.27
40	**1.20**	**1.76**	**2.08**	2.51	**3.04**	**3.37**

Table 2: Average speedup when adding threads to the processing load. As the length of the query increases, so too does the speedup, as the parallel portion of processing becomes larger for longer queries.

More surprisingly, however, is the very low rate in which speedup increases when adding more threads. Since the workload is divided by allocating segments to threads, there may be cases where all threads are not utilized (for example, when there are 24 worker threads, but only 8 segments). This is due to the *inter-segment* method in which the processing is divided – a single segment will only ever be processed with a single thread. This implies that processing will be at least as slow as the slowest running segment, a potential bottle-neck which may be alleviated with *intra-segment* processing, whereby a segment can be processed by multiple threads. We leave further exploration into this phenomenon as future work.

4.4 Meeting Service Level Agreements

From the lessons learned in the prior experiments, we now attempt to add selective parallelization [14] to the JASS system in order to improve efficiency and effectiveness while meeting two pre-specified SLAs. The proposed SLAs are realistic, and provide strict time budgets on both the 95th (P_{95}) and 99th (P_{99}) percentile response latency:

- $P_{95} \leq 200$ms, and
- $P_{99} \leq 250$ms.

In addition, we wish to minimize effectiveness degradation whenever possible.

Efficiency Modelling. To efficiently predict the approximate running time for a query with ρ postings, we follow Lin and Trotman [20] in building a simple linear model. We sampled 1,000 web queries from a MSN query log [8], and ran them across our ClueWeb12-B13 index using various fixed settings of ρ, collecting both the number of postings processed and the corresponding efficiency in milliseconds. We then derive our model by conducting a linear regression on the time taken and the postings processed for each query. The linear model is a good fit for the sample queries, with a coefficient of determination $R^2 = 0.926$. Our model can be used to predict the largest value of ρ that is acceptable for a given time bound t:

$$m_\rho(t) = \frac{t - 35.541}{2.28 \times 10^{-5}}$$

We note that many more advanced prediction models have been explored in the literature, especially for quantifying the processing

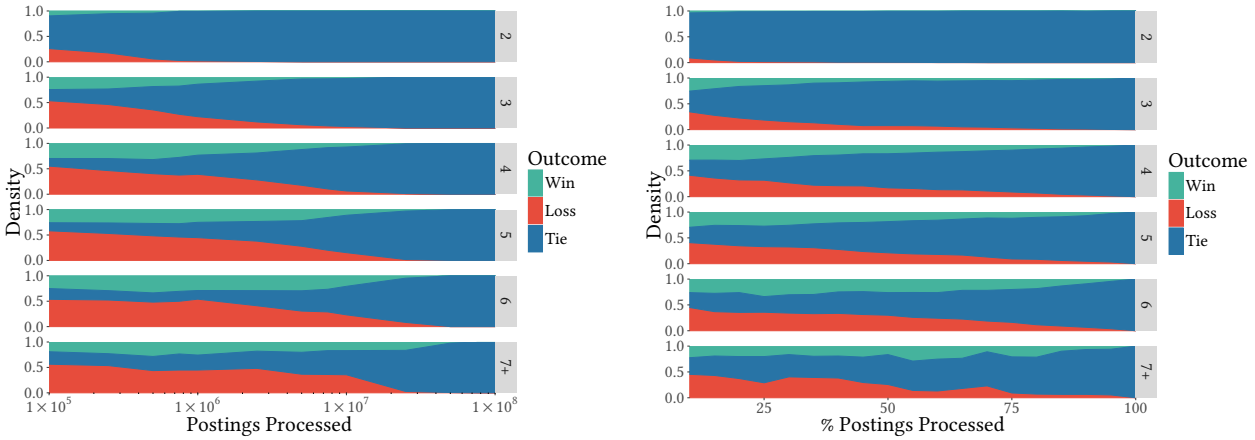

Figure 2: Density of wins, ties and losses, when comparing the heuristic settings with the exhaustive run, across all queries. The leftmost plot uses the fixed ρ heuristic, whereas the rightmost plot bases the value of ρ on a percentage of the postings on a per-query basis. Setting ρ based on the percentage of candidate postings appears to give greater effectiveness control than selecting a system-wide fixed value of ρ.

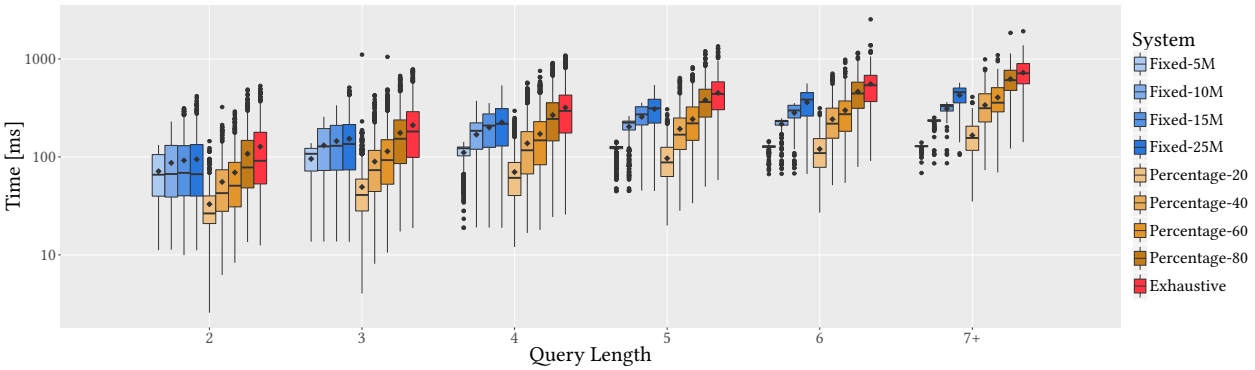

Figure 3: This figure shows the efficiency of various heuristic settings for ρ across all queries. Diamonds denote the mean efficiency. Fixed settings of ρ tend to exhibit very little variance as query length increases, whereas the percentage-based settings exhibit more variance.

time of DaaT or TaaT traversal [14, 16, 23, 24]. These models are almost certainly overly complicated for SaaT traversal prediction, as SaaT traversal is sensitive only to the number of postings to be processed [6].

Baseline Systems. The most obvious baseline is the exhaustive, single-threaded system. This system exhaustively processes all queries in a single-threaded manner, and provides the desired effectiveness that other systems should aim to achieve. Given the various time/effectiveness profiles that can be achieved using the different aggressiveness heuristics, we employ several instances of each as baseline. For the ρ_{fixed} baseline, we set $c = 5$ million as suggested by Lin and Trotman, as well as $c = 7$ million (which corresponds to the calculated upper-bound ρ to meet the P_{95} SLA). These systems are denoted 'Fixed-c', where c corresponds to the fixed constant used. For $\rho_{percentage}$, we let $z = \{20, 40, 60, 80\}$, and denote these systems as 'Percentage-z' (or 'Perc-z').

Selective Parallelization. Next, we propose a number of hybrid approaches which use selective parallelism to accelerate queries which are predicted to run slowly. The key idea with selective parallelization is to only parallelize queries which are predicted to run slower than the given budget, as parallelizing short queries is often a waste of resources [14]. Based on our model, we know that queries with more than 7 million postings are likely to exceed the 200ms time budget, so we parallelize any query with over 7 million candidate postings. We exhaustively process *all* queries (and thus, lose no effectiveness), and call this system *Selective Parallelization (SP)*. Another hybrid approach is to use *aggressive* processing on top of the selective parallel system, where the parallelized queries will be used in conjunction with the fixed heuristic. We use 3 large values of c, namely $c = \{20, 50, 100\}$ million, as this value is divided equally among the processing threads (that is, each of the n threads will have a local ρ of $\frac{\rho}{n}$). This approach is called *Aggressive Selective Parallelization (ASP-c)*.

We run all aforementioned systems across the entire 5,862 queries, each system retrieving the top-10 documents. Table 3 shows both

the efficiency and effectiveness results for this experiment for different numbers of worker threads.

Effectiveness Analysis. In general, the SP/ASP systems outperform the fixed and percentage heuristic systems with respect to effectiveness, as they often process more postings than their counterparts. For example, the SP system exhaustively processes all queries, and thus loses no effectiveness. In addition, the Fixed-7M system outperforms the Fixed-5M system and all of the percentage based systems except for Percentage-80. The aggressive SP systems were generally at least, if not more, effective than the Fixed-7M system, except for ASP-20M with 32 and 40 threads. Note also that the ASP systems tend to become less effective as more threads are added, because the stopping rule is met more readily.

Efficiency Analysis. First, we examine the baselines presented in Table 3. The Fixed-5M configuration, based on the recommended value of c from Lin and Trotman, was able to meet the efficiency SLAs quite easily. In addition, the linear prediction model was quite accurate in predicting that processing 7 million postings is possible within the given budget, as the Fixed-7M model was also able to meet both SLAs. Conversely, the percentage based systems all violate both SLAs with the exception of Percentage-20. Next, we examine the efficiency of both the selective parallelization and the aggressive selective parallelization systems for a number of different worker threads. As expected, selective parallelization is able to accelerate the exhaustive processing. For example, the safe-to-k SP systems are between 1.5× and 2.3× faster than exhaustive processing when considering the mean latency, and 1.5× to 3.5× faster when considering the 99th percentile latency, with no effectiveness loss. However, all of the SP variants violated the service level agreements, as they could not effectively control the tail latency of the postings traversal. Fortunately, the aggressive SP methods were generally able to meet the SLAs, while achieving a higher effectiveness than the fixed systems. For convenience, Figure 4 summarizes the timings for the various systems, including the mean, median, P_{95} and P_{99} latencies. Based on our findings, the best approach for efficiently processing longer queries without a large reduction in effectiveness is to use parallel processing to accelerate them, while still processing a high percentage of the candidate postings to ensure effective results are returned, thus answering RQ3.

Scalability for Large Candidate Sets. As a final experiment, we run our multi-threaded implementation of JASS exhaustively across all queries, retrieving the top-10,000 documents for each query. As in earlier experiments, we use 8, 16, 24, 32 and 40 threads. The aim of this experiment is to understand the scalability of the multi-threaded implementation when retrieving large sets of candidate documents, as is necessary in modern, multi-stage retrieval systems [12, 27]. In comparing the runs which retrieve the top-10,000 documents to those which retrieve the top-10 documents, we find that the large increase in k only results in increases of mean time of between 5.5 and 10.3 ms. This makes sense given that the same amount of work was conducted in processing the postings lists (which dominates the processing time) regardless of k.

System	Time				NDCG@10	
	Mean	Median	P_{95}	P_{99}	Mean	% W/T/L
1 Thread						
Exhaustive	309.9	270.4	736.5	973.6	0.1588	-/-/-
Fixed-5M†	103.8	120.7	129.7	132.1	0.1534	11/74/15
Fixed-7M†	135.2	156.3	184.6	189.6	0.1541	8/81/11
Perc-20†	69.3	56.5	171.2	231.0	0.1499	21/52/27
Perc-40	134.2	106.5	350.4	486.4	0.1538	15/67/17
Perc-60	167.6	132.3	423.2	593.9	0.1569	12/77/11
Perc-80	260.3	222.8	625.0	842.6	0.1590	7/87/6
8 Threads						
SP	207.9	186.1	414.6	630.4	0.1588	-/-/-
ASP-100M	187.3	190.5	354.1	423.1	0.1587	2/96/2
ASP-50M	179.6	182.5	328.1	379.1	0.1580	4/92/4
ASP-20M	153.5	167.8	241.4	255.3	0.1563	8/84/8
16 Threads						
SP	153.8	150.8	267.9	354.3	0.1588	-/-/-
ASP-100M	149.8	151.6	252.6	284.4	0.1587	2/96/2
ASP-50M	143.0	145.8	243.3	270.0	0.1577	3/93/4
ASP-20M†	128.4	139.9	189.9	200.0	0.1559	7/84/9
24 Threads						
SP	146.8	151.1	251.7	316.0	0.1588	-/-/-
ASP-100M	138.5	147.1	227.7	257.4	0.1583	2/95/3
ASP-50M	136.8	147.7	214.9	243.7	0.1572	4/91/5
ASP-20M†	120.4	135.6	164.5	172.2	0.1556	9/81/10
32 Threads						
SP	135.4	138.9	219.5	277.9	0.1588	-/-/-
ASP-100M	131.8	139.9	207.7	241.6	0.1580	2/95/3
ASP-50M†	127.8	136.4	193.5	222.5	0.1571	4/90/6
ASP-20M†	114.4	125.2	159.2	169.8	0.1538	8/80/12
40 Threads						
SP	134.9	139.3	216.3	273.3	0.1588	-/-/-
ASP-100M†	128.8	137.5	199.3	229.6	0.1577	3/94/3
ASP-50M†	124.9	135.5	183.6	207.3	0.1567	5/89/6
ASP-20M†	109.7	120.3	150.6	169.4	0.1528	9/78/13

Table 3: Time and effectiveness trade-offs for the various approaches. † denotes that a system did not violate the SLAs. W/T/L corresponds to the percentage of Wins/Ties/Losses with respect to the exhaustive system.

5 DISCUSSION AND FUTURE WORK

A major assumption of this work is that longer queries generally have more candidate postings. Although this is often true, it is not always the case. Furthermore, different methods of splitting the workload for multi-threaded processing may result in greater speedup and improved resource allocation and load distribution. For example, we did not consider threading policies which allow *work stealing*, where idle threads will help further divide the workload from busy threads. This idea could be implemented by allowing *intra-segment* processing, where many threads are able to divide a single segment into many smaller blocks of work. Finally, modern IR systems would typically use JASS for early-stage processing, known

as candidate-generation, which is a recall-oriented task. Given the limitations of current collections, query logs, and judgments, we are unable to effectively explore the merits of each heuristic in a recall-oriented scenario. We leave these ideas for future exploration.

6 CONCLUSION

In this work, we explored several early termination heuristics for score-at-a-time retrieval. We found that percentage-based heuristics are more stable with respect to their effectiveness behaviour, but suffer from increased tail latency with respect to the fixed heuristics. Conversely, the fixed heuristics are able to manage the upper-bound efficiency, but are not able to guarantee effective results, especially for longer queries. Furthermore, we showed that SAAT systems can benefit from using multiple worker threads to accelerate longer queries, allowing for improved effectiveness without loss in efficiency. In particular, using selective parallelization with per-thread aggressiveness provided the best trade-off, outperforming fixed-parameter systems for both efficiency and effectiveness when the number of worker threads was sufficiently high.

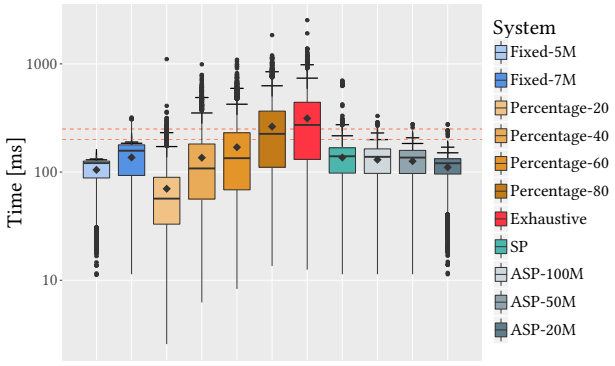

Figure 4: Efficiency characteristics of the baseline and the parallel approaches for all 5,682 queries. The dashed horizontal lines denote the P_{99} and P_{95} SLA bounds. For each system, the values of P_{99} and P_{95} are denoted by a short and long horizontal bar, respectively. For threaded systems, the 40 thread instances are plotted.

ACKNOWLEDGMENTS

This work was supported by the Australian Research Council's *Discovery Projects* Scheme (DP170102231), an Australian Government Research Training Program Scholarship and a grant from the Mozilla Foundation.

REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. 2001. Vector-space ranking with effective early termination. In *Proc. SIGIR*. 35–42.

[2] P. Bailey, A. Moffat, F. Scholer, and P. Thomas. 2016. UQV100: A Test Collection with Query Variability. In *Proc. SIGIR*. 725–728.

[3] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23, 2 (2003), 22–28.

[4] D. Broccolo, C. Macdonald, O. Salvatore, I. Ounis, R. Perego, F. Silvestri, and N. Tonellotto. 2013. Load-sensitive Selective Pruning for Distributed Search. In *Proc. CIKM*. 379–388.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*. 426–434.

[6] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman. 2017. A comparison of Document-at-a-Time and Score-at-a-Time query evaluation. In *Proc. WSDM*. 201–210.

[7] M. Crane, A. Trotman, and R. O'Keefe. 2013. Maintaining discriminatory power in quantized indexes. In *Proc. CIKM*. 1221–1224.

[8] N. Craswell, R. Jones, G. Dupret, and E. Viegas (Eds.). 2009. Proceedings of the Web Search Click Data Workshop. In *Proc. WSDM*.

[9] J. Dean and L. A. Barroso. 2013. The Tail at Scale. *Comm. ACM* 56, 2 (2013), 74–80.

[10] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. 2013. Optimizing Top-k Document Retrieval Strategies for Block-Max Indexes. In *Proc. WSDM*. 113–122.

[11] S. Ding and T. Suel. 2016. Faster Top-k Document Retrieval Using Block-Max Indexes. In *Proc. SIGIR*. 993–1002.

[12] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei S. Elnikety, and Y. He. 2017. BitFunnel: Revisiting Signatures for Search. In *Proc. SIGIR*. 605–614.

[13] S-W. Hwang, K. Saehoon, Y. He, S. Elnikety, and S. Choi. 2016. Prediction and Predictability for Search Query Acceleration. *ACM Trans. Web* 10, 3 (Aug. 2016), 19:1–19:28.

[14] M. Jeon, S. Kim, S-W. Hwang, Y. He, S. Elnikety, A.L. Cox, and S. Rixner. 2014. Predictive Parallelization: Taming Tail Latencies in Web Search. In *Proc. SIGIR*. 253–262.

[15] X-F. Jia, A. Trotman, and R. O'Keefe. 2010. Efficient accumulator initialization. In *Proc. ADCS*. 44–51.

[16] S. Kim, Y. He, S-W. Hwang, S. Elnikety, and S. Choi. 2015. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *Proc. WSDM*. 7–16.

[17] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. 2013. Online controlled experiments at large scale. In *Proc. KDD*. 1168–1176.

[18] A. Kulkarni and J. Callan. 2015. Selective Search: Efficient and Effective Search of Large Textual Collections. *ACM Trans. Information Systems* 33, 4 (2015), 17–1–17–33.

[19] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. 2016. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *Proc. ECIR*.

[20] J. Lin and A. Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *Proc. ICTIR*. 301–304.

[21] J. Lin and A. Trotman. 2017. The Role of Index Compression in Score-at-a-time Query Evaluation. *Inf. Retr.* 20, 3 (2017), 199–220.

[22] C. Macdonald, R. L. T. Santos, and I. Ounis. 2013. The whens and hows of learning to rank for web search. *Inf. Retr.* 16, 5 (2013), 584–628.

[23] C. Macdonald, N. Tonellotto, and I. Ounis. 2012. Learning to Predict Response Times for Online Query Scheduling. In *Proc. SIGIR*. 621–630.

[24] C. Macdonald, N. Tonellotto, and I. Ounis. 2017. Efficient & Effective Selective Query Rewriting with Efficiency Predictions. In *Proc. SIGIR*. 495–504.

[25] J. Mackenzie. 2017. Managing Tail Latencies in Large Scale IR Systems. In *Proc. SIGIR*. 1369–1369.

[26] A. Moffat and J. Zobel. 1996. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Trans. Information Systems* 14, 4 (1996), 349–379.

[27] J. Pedersen. 2010. Query understanding at Bing. *Invited talk, SIGIR* (2010).

[28] M. Persin, J. Zobel, and R. Sacks-Davis. 1996. Filtered document retrieval with frequency sorted indexes. *JASIST* 47, 10 (1996), 749–764.

[29] O. Rojas, V. Gil-Costa, and M. Marin. 2013. Efficient Parallel Block-max WAND Algorithm. In *Proc. EuroPar*. 394–405.

[30] E Schurman and J Brutlag. 2009. Performance related changes and their user impact. Velocity. (2009).

[31] T. Strohman, H. Turtle, and W. B. Croft. 2005. Optimization strategies for complex queries. In *Proc. SIGIR*. 219–225.

[32] N. Tonellotto, C. Macdonald, and I. Ounis. 2013. Efficient and effective retrieval using selective pruning. In *Proc. WSDM*. 63–72.

[33] A. Trotman. 2014. Compression, SIMD, and Postings Lists. In *Proc. ADCS*. 50:50–50:57.

[34] A. Trotman, X-F. Jia, and M. Crane. 2012. Towards an efficient and effective search engine. In *Wkshp. Open Source IR*. 40–47.

[35] A. Trotman and J. Lin. 2016. In Vacuo and In Situ Evaluation of SIMD Codecs. In *Proc. ADCS*. 1–8.

[36] H. Turtle and J. Flood. 1995. Query evaluation: strategies and optimizations. *Inf. Proc. & Man.* 31, 6 (1995), 831–850.

[37] J-M. Yun, Y. He, S. Elnikety, and S. Ren. 2015. Optimal Aggregation Policy for Reducing Tail Latency of Web Search. In *Proc. SIGIR*. 63–72.