SHORT COMMUNICATION

Journal Section

Micro and Macro Optimization of SaaT Search

Andrew Trotman[1] | **Matt Crane**[2]

[1]Department of Computer Science, University of Otago, Dunedin, New Zealand

[2]Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

Correspondence
Andrew Trotman, Department of Computer Science, University of Otago, Dunedin, New Zealand
Email: andrew@cs.otago.ac.nz

Funding information
None

We introduce and test several micro and macro optimizations to the Score-at-a-Time approach to processing impact ordered postings lists in a search engine. Our micro optimizations are at the single assembly instruction level but our macro optimizations are algorithmic. Overall we see an improvement of 37% on our baseline (22% on state of the art). We experiment with parallel search and present evidence that we have hit the memory wall.

KEYWORDS
Search, Efficiency, Inverted Files, Score At A Time

1 | INTRODUCTION

It is often advised that micro-optimization is generally not worth the investment and that macro-optimization is more valuable. Stackoverflow threads asking for advice on micro-optimization for C are often filled with advice to "rely on your compiler to optimise this stuff" and to "concentrate on using appropriate algorithms and writing reliable, readable and maintainable code" [1]. A recent and thorough study by Linares-Vásquez et al. [2] on micro-optimization of Java apps for Android finds that developers rarely implement micro-optimizations, and that the effect of micro-optimizations is negligible in most of the cases. The body of work on micro optimization for C++ (the language used herein) appears to be mostly limited to a set of ad hoc principals on web pages (e.g. Lee [3]) with few exceptions (e.g. Bajwa et al. [4]).

In this investigation we examine micro and macro optimization for the Score-at-a-Time (SaaT) approach to processing postings lists in a search engine, in which we take an interest because of its effectiveness in a time constrained environment [5, 6].

First we microscopically examine the cost of accessing member variables in C++ objects and show that there is essentially no overhead to it, leading to a design decision to object orient the implementation of our new parallel search engine.

We then microscopically examine the cost of accessing an array versus dynamically allocated memory and show that the one additional instruction needed to access dynamically allocated memory can have a substantial effect on the latency of the search engine.

We are aware of only two open source SaaT search engines, ATIRE [7] and JASS [5]. Both use a heap to maintain the top-k results, and both use the same algorithm to determine when a heap update is necessary. We examine that algorithm and show that, in general, more comparisons are performed than necessary. We introduce an optimized algorithm and show that using it results in further efficiency gains.

Finally, we believe we are the first to examine query-per-thread parallel search in SaaT. Our experiments suggest that the changes we present take us to the memory wall. That is, our search engine fully utilizes the memory bandwidth and using additional cores simply results in those cores (mostly) sitting idle waiting for data.

2 | SAAT SEARCH

A document ordered index stores, for each unique term in the collection, a postings list, $\langle d_1, tf_1 \rangle \ldots \langle d_N, tf_N \rangle$, in increasing order of document id, d (where tf is the term frequency of the term in the document). An impact ordered index [8] stores the postings list, $\langle i_1, \langle d_{1,1} \ldots d_{1,n_1} \rangle \rangle \ldots \langle i_I, \langle d_{I,1} \ldots d_{I,n_I} \rangle \rangle$, where i is the impact score of all documents, d, in the segment. The impact score, computed at indexing time, is the influence of the term in the document. That is, for BM25 ranking [9], the BM25 score of that term for that document. These scores are normally quantized into a small integer range [10] and so the number of segments per term is small.

Both JASS and ATIRE participated in the SIGIR 2015 RIGOR workshop [11]. JASS was shown to be more efficient than ATIRE (and all other participants), but they share a processing strategy. They allocate a set of accumulators, $\mathbb{A}$, (one per document) then pull the postings lists for all the query terms, order the segments from highest to lowest impact, then processes them in that order. Each segment is processed by loading the i score from the segment, then, for each d, i is added to $\mathbb{A}_d$. To compute the top-k, a heap of pointers to the top k accumulators is kept up-to-date while processing. SaaT processing is a tight loop and takes advantage of CPU pre-fetch and branch prediction [12]. We use JASS as one of our baselines.

3 | EXPERIMENTAL ENVIRONMENT

We used the TREC .GOV2 collection [13] of 25,205,179 documents along with all 20,000 TREC Million Query Track queries pertaining to it (TREC 2007 [14] and 2008 [15]). We build a QMX [16] compressed BM25 ($k1 = 0.9$, $b = 0.4$) impact ordered index quantized into 8-bit impact scores and use that index in all experiments (with our search engine and with JASS). We load the entire index into memory on startup. We used TREC topics 701–850 to verify that the top $k = 10$ results from both our search engine and JASS were identical, and that the precision of both was in line with expectations.

Except where noted, experiments were run on a 4-core iMac with a 3.2GHz Intel Core i5 CPU and 24GB of 1867MHz DDR3 RAM running macOS 10.13.3 with Xcode 9.2 (Apple LLVM version 9.0.0 (clang-900.0.39.2)) and maximum compiler optimizations. Only one core was used to measure latency, but more cores were used to measure throughput.

Timings were measured using the `std::chrono::steady_clock` provided by C++. Each experiment was run 5 times and the reported times are the micro-averages. Search was to completion (no early termination) with $k = 10$. Prior work has shown that k has little effect on efficiency in a SaaT system [6].

LISTING 1 Global and class member accesses.

```
class access
{
public:
   uint32_t m_first, m_second;
   static uint32_t g_first, g_second;

   access() : m_second(2) {}
   uint32_t second_member();
   uint32_t second_global();
};

uint32_t access::g_second = 2;

uint32_t access::second_member()
{
return m_second;
}

uint32_t access::second_global()
{
return g_second;
}
```

4 | GLOBAL VERSUS MEMBER VARIABLES

Intuitively, access to a member variable in a C++ object (or a C `struct`) is more expensive than access to a global. In the former the CPU loads the address of `this`, adds the offset of the member, then accesses the value. In the latter the address is known in advance and does not need to be computed. Consequently, we expect a search engine using global variables for housekeeping (such as JASS) to outperform a search engine that manages the housekeeping in an object (such as ATIRE)—and JASS has been shown to be more efficient than ATIRE [11]. In this section we examine this perceived efficiency gain.

Listing 1 presents simple C++ class illustrating the different access types. In this code, variables starting `g_` are global while those starting `m_` are members. Two of each have been allocated to ensure that offsets are non-zero.

We are interested in the difference between the assembly generated for `second_member()` and `second_global()` so we compiled this code and examined it.[1]

The assembly for access to `m_second`,

```
mov eax, dword ptr [rdi + 4]
```

[1]Interested readers can verify this, and experiment, here: `https://godbolt.org/g/zf5vvE`

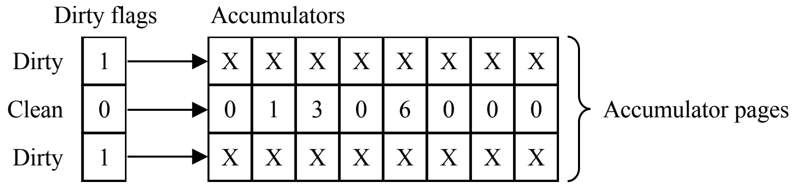

FIGURE 1 The accumulator management of Jia et al. [18] breaks the accumulators into pages and keeps a dirty flag for each page. Dirty flags are initialized on search start but accumulator pages are on demand zeroed (clear-on-write).

uses relative addressing via `rdi`. That is, the `rdi` register holds the address of the object (the `this` pointer), the member is 4 bytes past the start of the object, and the value is loaded into the `eax` register.

Access to `g_second` is similar,

```
mov eax, dword ptr [rip + access::g_second]
```

in this case being relative addressing via the `rip` register (known as rip-relative addressing).

In order to support runtime object code relocation (i.e. DLLs) compilers such as `clang` output assembly that accesses global data relative to the instruction pointer. In short, the absolute address of a global cannot be known until a DLL is loaded, so a fixed address cannot be allocated at compile or link time. The global address is relative to the DLL load address, but the x86_64 instruction pointer, `rip`, is used for convenience. Rip-relative addressing also results in smaller (and faster) instructions than absolute addressing as rip-relative offsets are 32-bits but absolute addresses are 64-bits.

We ran an experiment and verified that the overhead of placing variables in objects is nil. We do not report on this further. We conclude that accessing members is no more expensive than accessing globals as both use relative addressing (ATIRE is not slower than JASS because of this). We accept that calling into an object has a cost because `rdi` must be loaded, and calling a `virtual` function has a further cost. But once `rdi` is loaded there is no overhead in accessing members. Hereon in we adopt an object oriented design.

5 | ACCUMULATORS

Several accumulator management schemes for SaaT search have been proposed [17, 18, 19]. The simplest appears to be that of Jia et al. [18]. They allocate an array of accumulators, one per document, and sum the impact scores into this array. Their structure is small (16-bit accumulators, or 19MB for 10 million documents). They observe that the most expensive operation in their search engine is zeroing this array at the start of each search. To address this they break the array into a series of pages and keep a dirty-flag for each page.

This is illustrated in Figure 1. All dirty-flags are set to 1 on search initialization. During query processing, when writing to an accumulator, if the flag is set then it is cleared, the accumulator page is zeroed, and the write proceeds. Otherwise, the page is already clean and the write goes ahead unhindered.

The usual implementation is to use a page size of $2^{\lfloor \log_2 \sqrt{D} \rfloor}$, where D is the number of documents in the collection (Jia et al. [18] provide a proof of optimal page size). This has two efficiency advantages. First, converting from a document identifier into a dirty-flag index is performed with a single instruction (bit-shift). Second, all pages are aligned

LISTING 2 Access to an array versus dynamic memory.

```
class difference
{
public:
   uint8_t *dynamic_data;
   uint8_t array_data[1024];

   difference()
     {
     dynamic_data = new uint8_t[1024];
     }
   char dynamic_access(size_t element);
   char array_access(size_t element);
};

char difference::dynamic_access(size_t element)
{
return dynamic_data[element];
}

char difference::array_access(size_t element)
{
return array_data[element];
}
```

with CPU cache lines (except in tiny collections) as the page size is a whole power of 2. The usual implementation is to use an array of bytes to store the flags which is space inefficient, but avoids bit-level operations.

Both ATIRE and JASS use dynamic allocation to allocate the accumulator array and dirty-flags, but this is inefficient for memory accesses. In detail, to access a dirty-flag the CPU must first load the dynamic address of the array, then compute the offset, and then accesses the flag. As this is happening for each and every posting in all postings lists for each term of each query, any improvement in this could result in a substantial throughput gain.

We observe that by allocating as arrays rather than through dynamic allocation, one instruction per access can be avoided. If the address of the dirty flags is known in advance then a load of the address can be avoided.

The two approaches are illustrated in Listing 2. Both `_access()` methods access a contiguous segment of memory as if it were an array, but, `dynamic_data`, is dynamically allocated while `array_data` is allocated as an array. In the case of `dynamic_access()`, the assembly to access the data is

```
mov rax, qword ptr [rdi]
movsx eax, byte ptr [rax + rsi]
```

that is, load the address from the object, compute the offset of the data, then load the value into `eax`. In the case of `array_access()`, the assembly is

Allocation	Algorithm	ms/Query	Reduction	c.f. JASS
Dynamic	1	177	0%	-24%
Array	1	166	6%	-16%
Dynamic	2	164	8%	-14%
Array	2	152	14%	-6%
JASS	**1, inline**	**143**	**19%**	**0%**
Array	2, inline	112	37%	22%

TABLE 1 Execution time for 20,000 TREC queries on .GOV2 and the different approaches in this paper. Savings sum to an overall reduction in execution time of 37% on our baseline and 22% on state of the art. We believe the times reported here for JASS are for the same code-base used in Crane et al. [6], but the times per query differ because the query set is different (TREC Million Query Track queries here verses TREC Topics 701-850 there) and due to hardware is differences.

```
movsx eax, byte ptr [rdi + rsi + 8]
```

that is, compute the address of the data, and load the value into `eax`. It takes one instruction for an array access but two if dynamic allocation is used.

We ran an experiment to measure the size of this saving (20,000 queries, .GOV2, etc.). Table 1 presents the results, of which the first two rows are for this experiment. They show that avoiding dynamic allocation results in a 6% reduction on our baseline – the tight loop in SaaT search appears to be sufficiently tight that 6% of the time is spent loading and performing these extra instructions. The performance of JASS is shown in bold. Our baseline is not as competitive, but our improvements eventually sum to a 22% reduction over JASS.

The saving from this micro-optimization is substantial, but the improvement comes at a loss of functionality. The maximum size of the two arrays must be known at compile time. That is, the largest document collection the search engine will see, not the size of the actual collection (which can be substantially smaller). Our implementation allocates accumulators for 50,000,000 documents (95MB), and the appropriate number of dirty-flags (48MB) for the 25,205,179 documents. We rely on operating system delayed allocation strategies to ensure physical memory is not allocated to those parts of the array that are not used – which is much of it in the experiments herein.

6 | HEAP MANAGEMENT

The search engine must compute the top-k documents to return to the user. The standard solution for SaaT is to use a heap to keep track of the top-k while the query is progressing [7]. The standard algorithm, seen in both JASS and ATIRE, is presented as Algorithm 1.

The first clause checks to see whether the heap has fewer than `topk` elements, and if so then it puts the accumulator into the heap, building the heap once it is full. The second clause checks to see if the accumulator is already in the heap and if so then it rebuilds the heap. The third clause checks to see if the accumulator should be added to the heap, and adds it if it should.

We observe that the usual case is that the heap is full, that the accumulator is not in the heap, and does not

Algorithm 1 Heap Update

```
1:  if $needed\_for\_topk > 0$ then
2:      $old\_value \leftarrow acc[index]$
3:      $acc[index] \leftarrow acc[index] + score$
4:      if $old\_value = 0$ then
5:          $dec(needed\_for\_topk)$
6:          $heapk[needed\_for\_topk] \leftarrow \&acc[index]$
7:      end if
8:      if $needed\_for\_topk = 0$ then
9:          build the heap on $heapk$
10:     end if
11: else if $acc[index] >= *heapk[0]$ then
12:     $acc[index] \leftarrow acc[index] + score$
13:     $heap\_update()$
14: else
15:     $acc[index] \leftarrow acc[index] + score$
16:     if $acc[index] > *heapk[0]$ then
17:         $heap\_insert(\&acc[index])$
18:     end if
19: end if
```

Location	Times
First clause (line 2 is executed)	199,965
Second clause (line 12 is executed)	143,406
Third clause, in `if` (line 17 is executed)	73,733,775
Third clause, `else` of `if` (`else` of line 16)	378,923,996,448

TABLE 2 Clause execution counts for Algorithm 1.

need to enter the heap, that is, all comparisons are false. To confirm this we annotated the source code to count the number of times each clause was executed (20,000 queries, .GOV2, etc.). The results are presented in Table 2 which shows that in 378,923,996,448 cases, three comparisons are performed in order to determine that the heap need not change.

If the heap is primed with a single entry whose value is lower than the lowest possible accumulator value (i.e. 0), then a single comparison of a new accumulator value against the top of the heap is sufficient to determine that, in most cases, no change to the heap is needed. Reducing the number of comparisons not only reduces the number of instructions executed, but also decreases the probability of a CPU pipeline stall, that is, there is a double efficiency gain. The new algorithm is presented as Algorithm 2.

We sanity checked the new algorithm against the original, and indeed they produce identical results. We then annotated the new source code and re-ran the experiment. The results are presented in Table 3, where it can be seen that in the vast majority of cases (over 99.9%) only one `if` is evaluated and it is normally false.

Algorithm 2 Fast Heap Update

```
1: $acc[index] \leftarrow acc[index] + score$
2: if $acc[index] >= *heapk[0]$ then
3:     if $needed\_for\_topk > 0$ then
4:         if $acc[index] = score$ then
5:             $dec(needed\_for\_topk)$
6:             $heapk[needed\_for\_topk] \leftarrow \&acc[index]$
7:             if $needed\_for\_topk = 0$ then
8:                 build the heap on $heapk$
9:             end if
10:        end if
11:    else
12:        if $acc[index] - score < *heapk[0]$ then
13:            $heap\_insert(\&acc[index])$
14:        else
15:            $heap\_update()$
16:        end if
17:    end if
18: end if
```

Location	Times
Line 3 is executed	74,077,146
Line 3 is not executed	378,923,996,448

TABLE 3 Clause execution counts for Algorithm 2.

To measure this efficiency gain we measured the mean time per query for each algorithm both with and without the changes in Section 5. The result are presented in Table 1 where it can be seen that an 8% reduction is due to this algorithm, and an overall reduction of 14% is seen if arrays are also used.

During our experiments we noticed that despite using C++ templates the compiler did not `inline` our changes. We altered our code to force inlining and re-ran the experiment. The result (marked inline in Table 1) shows that all three improvements result in a substantial reduction in execution time of 37%, a 22% reduction on JASS (which inlines Algorithm 1). Inlining and careful optimization results in a large efficiency improvement.

7 | PARALLEL SEARCH

One reason to object orient our design is parallel search. Mackenzie et al. [20] discuss the use of multiple threads to decrease latency in a SaaT search engine. We believe we are the first to examine parallel search with SaaT to increase throughput.

We build a lock-free queue of all the queries. In parallel, each thread looks for the next unassigned query, marks it as assigned, then executes it. This way the total workload is evenly distributed across the threads and there is no

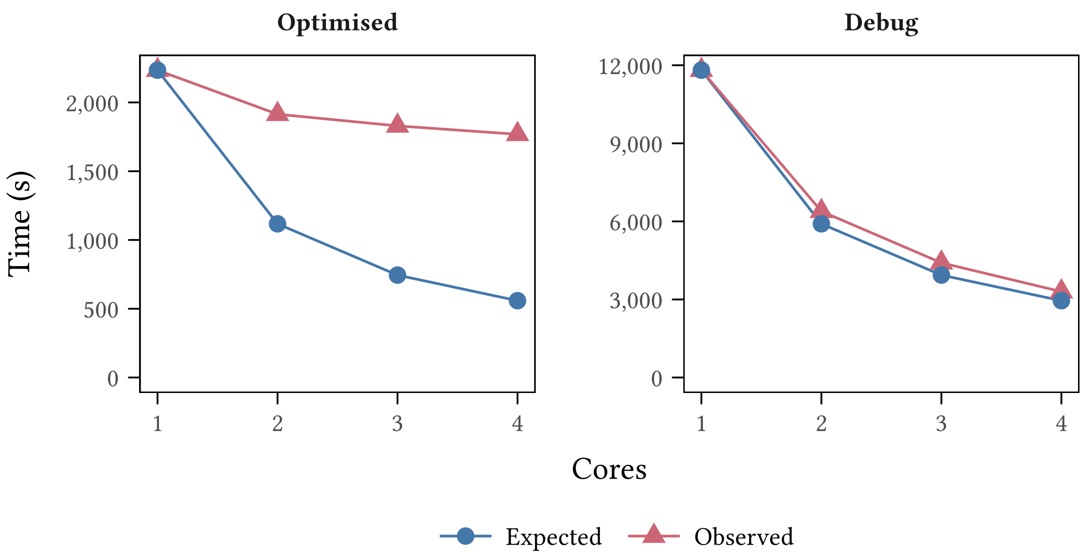

FIGURE 2 Total execution time for 20,000 queries as the core count increases (left: all optimizations, right: no optimization) suggest that the memory wall has been hit.

job stealing at the end.

We ran an experiment using 1 to 4 threads (CPU cores) and all our optimizations. The left of Figure 2 presents the observed time needed to execute all 20,000 queries along with the expected time ($\frac{single\_core\_time}{core\_count}$). A 21% reduction in time is seen between 1 and 4 cores, where a 75% reduction is expected. Extensive further investigation on several machines suggests this is because parallel SaaT fully saturates the memory to CPU channels (hits the memory wall). An improvement of only 3% is seen when going from 3 to 4 cores because the memory cannot keep up with CPU demand.

To illustrate our hypothesis, we turned off all improvements herein, compiled without optimization, and ran on a 2.6GHz Intel Core i7 with 16GB 2133MHz LPDDR3 RAM (more CPU intensive, slower CPU, faster RAM) which, in effect, moves the memory wall. The results are presented on the right of Figure 2 which shows that if the task were more CPU intensive a near perfect improvement would be seen (72% vs. 75%).

SaaT search as we have measured it is a purely CPU and memory task (the I/O time to disk or screen is not being measured) one of which must be the bottleneck. Figure 2 illustrates this point. On the left the process is memory access bound, on the right it is CPU bound. Optimization is usually a process of identifying a bottleneck and identifying a way to remove it. As it is unlikely that our search engine has the optimal memory access pattern, we will be carefully examining this pattern for areas of further improvement. One obvious place to look is decompression. Integer compression codecs such as QMX are normally distributed as source code that decodes a large block of integers for later processing (and the version we use does exactly this). Interleaving the decoding and the processing of postings is likely to reduce memory accesses as decoded integers do not need to be written to, and later read from, memory. Unfortunately, a change of this nature comes and the expense of not being able to plug-in a third party pre-optimized codec without adaptation – and it will increase coupling which is not normally considered to be good practice.

8 | DISCUSSION AND CONCLUSIONS

In this investigation we examined micro and macro optimizations to SaaT search and showed that the removal of a single assembly instruction from the tight loop can result in a substantial decrease in latency. We optimized and inlined the heap update algorithms which showed further improvements. Finally we examined parallel search which suggests that we have hit the memory wall. This work covers SaaT search on Intel x86-64 architecture using Intel CPUs and the TREC .GOV2 document collection.

We expect that the optimizations will be effective on other x86-64 CPUs (such as AMD), but have not conducted experiments to verify this. Section 7 discusses the effect of changing the CPU to memory access time ratio – faster RAM shifts the bottleneck back to the CPU. We have not experimented with different cache sizes and leave this for further work. RISC architectures such as Power or ARM may not have the same complex instructions as x86-64 and as such the micro-optimizations we outline may not be effective on those architectures. We have not conducted experiments to verify this.

Other postings list processing strategies for search include Document-at-a-Time (DaaT) and Term-at-a-Time (TaaT). We believe that (on x86-64) all the optimizations we outline will decrease latency will work for TaaT as it is similar to SaaT. DaaT processing is radically different from SaaT and TaaT, and does not use an array of accumulators and does not use the same heap management strategy [21], however our observations on arrays, dynamic allocation, and class member variables remains valid in for that work – but likely at a smaller improvement in performance. We leave for future work the micro optimization of DaaT.

We have experimented with just one document collection (.GOV2) as it is commonly used in search engine efficiency experiments. The size of the improvements seen herein is a function of the execution time spent in the main tight loop, and the overhead of starting and stopping that main loop (the pre-amble and post-amble to search). With a larger collection more time is spent processing postings and so the saving will be larger. Conversely, with a smaller collection, relatively less time is spent in the tight loop and so the improvement is likely to be smaller. We leave for future work the measurement of the performance gain as a function of document collection size.

Our memory accesses are not optimal because of a design decision to use pre-existing code for decompression, and without increasing coupling. That code decompresses complete segments of postings lists and stores the results in memory. In future work we will examine the interlacing of the decompression and the processing of the results. Although bad from a design perspective we expect to see a reduction in memory accesses and an improvement in throughout – perhaps even removing the memory access boundedness.

This work directly challenges the assumption that micro optimizations are not generally worthwhile by presenting an application in which the main tight loop is responsible for a sufficiently substantial proportion of the execution time that such optimizations are worthwhile. The optimizations presented sum to a total decrease in latency of 22% on state of the art, and this short contribution offers several areas for further investigation in efficiency.

references

[1] R M, Micro-optimizations in C, Which Ones Are There? Is There Anyone Really Useful?; 2010. `https://stackoverflow.com/questions/2109262/micro-optimizations-in-c-which-ones-are-there-is-there-anyone-really-useful`.

[2] Linares-Vásquez M, Vendome C, Tufano M, Poshyvanyk D. How Developers Micro-optimize Android Apps. Journal of Systems and Software 2017;130:1–23.

[3] Lee ME. Optimization of Computer Programs in C; 1997. http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html.

[4] Bajwa MS, Agarwal AP, Gupta N. Code Optimization as a Tool for Testing Software. In: 3rd International Conference on Computing for Sustainable Global Development (INDIACom); 2016. p. 961–967.

[5] Lin J, Trotman A. Anytime Ranking for Impact-Ordered Indexes. In: ICTIR 2015; 2015. p. 301–304.

[6] Crane M, Culpepper JS, Lin J, Mackenzie J, Trotman A. A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation. In: WSDM 2017; 2017. p. 201–210.

[7] Trotman A, Jia XF, Crane M. Towards an Efficient and Effective Search Engine. In: SIGIR 2012 Workshop on Open Source Information Retrieval; 2012. p. 40–47.

[8] Anh VN, de Kretser O, Moffat A. Vector-space Ranking with Effective Early Termination. In: SIGIR 2001; 2001. p. 35–42.

[9] Robertson SE, Walker S, Jones S, Hancock-Beaulieu MM, Gatford M. Okapi at TREC-3. In: TREC-3; 1996. p. 109–126.

[10] Crane M, Trotman A, O'Keefe RA. Maintaining Discriminatory Power in Quantized Indexes. In: CIKM 2013; 2013. p. 1221–1224.

[11] Lin J, Crane M, Trotman A, Callan J, Chattopadhyaya I, Foley J, et al. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In: ECIR 2016; 2016. p. 408–420.

[12] Lin J, Trotman A. The Role of Index Compression in Score-at-a-time Query Evaluation. IRJ 2017;20(3):199–220.

[13] Clarke CLA, Craswell N, Soboroff I. Overview of the TREC 2004 Terabyte Track. In: TREC 2004; 2004. p. 1–9.

[14] Allan J, Carterette B, Dachev B, Aslam JA, Pavlu V, Kanoulas E. Million Query Track 2007 Overview. In: TREC 2007; 2007. p. 1–20.

[15] Allan J, Aslam JA, Pavlu V, Kanoulas E, Carterette B. Million Query Track 2008 Overview. In: TREC 2008; 2008. p. 1–22.

[16] Trotman A. Compression, SIMD, and Postings Lists. In: ADCS 2014; 2014. p. 50–57.

[17] Anh VN, Moffat A. Pruned Query Evaluation Using Pre-computed Impacts. In: SIGIR 2006; 2006. p. 372–379.

[18] Jia XF, Trotman A, O'Keefe RA. Efficient Accumulator Initialisation. In: ADCS 2010; 2010. p. 44–51.

[19] Anh VN. Impact-Based Document Retrieval. PhD thesis, University of Melbourne; 2004.

[20] Mackenzie J, Scholer F, Culpepper JS. Early Termination Heuristics for Score-at-a-Time Index Traversal. In: ADCS 2017; 2017. p. 8:1–8:8.

[21] Broder AZ, Carmel D, Herscovici M, Soffer A, Zien J. Efficient Query Evaluation Using a Two-level Retrieval Process. In: CIKM 2003; 2003. p. 426–434.