Towards an Efficient and Effective Search Engine

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

Xiang-Fei Jia
Department of Computer Science
University of Otago
Dunedin, New Zealand
fei@cs.otago.ac.nz

Matt Crane
Department of Computer Science
University of Otago
Dunedin, New Zealand
mcrane@cs.otago.ac.nz

ABSTRACT

Building an efficient and effective search engine requires both science and engineering. In this paper, we discuss the ATIRE search engine developed in our research lab, and both the engineering decisions and research questions that have motivated building ATIRE.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing – Indexing methods; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – Search process

General Terms

Algorithms, Performance

Keywords

Indexing, Storage, Efficiency, Pruning, Procrastination

1. INTRODUCTION

Information retrieval has been a hot research topic for decades due to the need to quickly and accurately answer users' queries across very large document collections, for example the web. Building such an efficient and effective search engine involves not only science but also engineering. Science provides a range of algorithms for fast searching and better ranking, and engineering is required so that systems can be tuned to their optimal performance.

There are a number of existing search engines, both proprietary and open-source, for example, Google, MG and Apache Lucene. However, we have built a new search engine called ATIRE from the ground up, to ensure we have a fast robust baseline in order to compare new information retrieval technologies; and to conduct state-of-the-art information retrieval research questions. The ATIRE search engine is a cross-platform search engine — running on Windows, Linux and Mac OSX — written in C/C++, with traditionally non-interoperable sections hand-coded to avoid the use of a third-party abstraction layer.

The questions we want to address are:

- **How to build a fast indexer:** It is very challenging to build a fast indexer due to the complexity of how much work is involved. Our indexer is multi-threaded with a unique pipeline methodology. We also implemented a memory management subsystem in the indexer for fast memory allocation. The indexer also supports the merging of multiple indexes into a single index.
- **What is the most efficient structure for the inverted index?** The full structure of the inverted index is rarely discussed in the literature, previous discussions have only discussed the techniques used for index representation [34]. In this paper, we discuss how we engineered our index structure.
- **How to search efficiently without sacrificing effectiveness?** We have been working on the optimisation of the term-at-a-time approach for query evaluation and for future work this will be used as a baseline for comparing various pruning algorithms; and comparing between term-at-a-time and document-at-at-a-time processing.
- **Does term proximity work?** We question whether term proximity and phrase searching are effective under current evaluation methodologies.
- **Other research questions?** There are a number of other research questions we intend to address in future work: generalisation of our fusion of ranking functions such as BM25 and PageRank; an exploration into the juxtaposition between diversity and relevance feedback; and fully distributed indexing and searching.

2. FAST INDEXING

The experiments and results shown were conducted on a collection of standard collections from both INEX and TREC forums, as described in Table 1. The experiments, with the exception of ClueWeb09 collections, were conducted on a dual quad-core Intel Xeon E5410 2.3GHz, DDR2 PC5300 9GB main memory, Seagate 7200RPM 500GB hard drive, and running Linux with kernel version 2.6.30. The ClueWeb-09 collection experiments were performed on an quad cpu AMD Opteron 6276 2.3GHz 16-core, 512GB PC12800 main memory, 6x 600GB 10000 RPM hard drives, and running Linux with kernel version 2.6.32.

In order to produce an index quickly, the indexer in ATIRE uses several optimisations and a unique pipeline procedure based on the producer/consumer model.

The main optimisation that ATIRE uses when indexing is the use of an internal memory management system that

SIGIR 2012 Workshop on Open Source Information Retrieval.
August 16, 2012, Portland, Oregon, USA.

Collection	Size			Documents	Words	
	Collection	Index	%		Unique	Total
Wall Street Journal [14]	517MB	64MB	12.4%	173,252	229,514	84,881,717
WT10g [4]	10GB	837MB	8.4%	1,692,096	5,512,114	1,348,119,626
2009 INEX Wikipedia [29]	50.7GB	1.6GB	3.2%	2,666,190	11,874,077	2,341,271,195
WT100g/VLC2 [16]	100GB	7.1GB	7.1%	20,616,457	25,250,355	12,690,145,498
.gov2 [9]	400GB	12GB	3%	25,205,179	40,641,599	32,573,784,848
ClueWeb09 Category B	1.5TB	32GB	2%	50,220,423	96,298,556	71,319,689,402
ClueWeb09 Category A	12.5TB			503,903,810		
(excl. 70% spam)	3.8TB	76GB	2%	150,954,279	127,651,335	189,731,940,667

Table 1: Summary of Collections Used

Memory Manager	Indexing Time (mm:ss)
System	14:18
ATIRE	10:37

Table 2: Indexing times for INEX 2009 Wikipedia collection across four .tar.gz files with different memory managers

requests large blocks of memory from the system and divides this up as necessary, this overhead can be measured by compiling without this intermediate manager and using the system memory management. This optimisation alone reduces the time taken to index the 2009 INEX Wikipedia collection by one-third, as shown in Table 2, these times are taken from a single run, with disk caches flushed between runs, but are indicative of typical performance.

The indexing pipeline that ATIRE uses internally is unique among open-source search engines. The pipeline consists of a group of parallel producer/consumer inspired objects that either deal with streams of data, or file-like objects that are created from these streams. Each step in the pipeline is focused on only performing one operation on the passed-through data, minimising the amount of inspection performed at each step.

These different stages in the pipeline can be combined quickly and efficiently to allow new types of content to be indexed. For instance, an existing object that un-tars, and an object that un-gzips can be combined to allow the indexing of .tar.gz files.

Objects in the pipeline are allowed to perform secondary functions, for instance, compressing the original document and including it within the index (for post-processing such as focused retrieval and snippet generation). The input pipeline allows the indexer to filter out documents, such as those identified as spam, and a best-effort attempt to clean incoming data to negate any pre-processing of document collections that might otherwise be necessary. This is motivated by our underlying philosophy that the indexer should be able to index any standard test collection out of the box without any pre-processing.

At the end of the pipeline each document is indexed separately and folded into the overall index. This, combined with the indexing pipeline, allow documents to be indexed completely in parallel on a single computer.

The effect that parallel indexing has on the indexing time is shown in Table 3. The times shown are from a single run, with disk caches flushed between experiments, but are typical times experienced. This table shows that as the number

Number Input Files (.tar.gz format)	Indexing Time (mm:ss)
1	19:35
2	11:47
4	10:37

Table 3: Indexing times for INEX 2009 Wikipedia collection with varying number of input files

Input Format	Indexing Time (mm:ss)
.tar	10:35
.tar.gz	10:37
.tar.lzo	10:09
.tar.bz2	19:10
File count	5:40
Extracted line count	6:54
Individual files	64:30

Table 4: Indexing times for INEX 2009 Wikipedia collection under various compression schemes across four files

of files increases, and our ability to index these files in parallel increases with it, that the total time to index is decreased.

This leads us to believe that our indexer is approaching the point where indexing is bound by decompression time. Indexing times for the INEX 2009 Wikipedia collection when split across four files are shown in Table 4, with the time to index the individual extracted files shown for comparison, as a clearly input bound operation (probably by the ability to open and close files). We are pleased to notice that we have already crossed the point at which we take less than twice the time as simply counting the number of files within the tar file. We also show the total time taken to count the number of lines in extracted files as a target to aim for. We have not yet tuned the number of threads our indexer uses against the number of cores in the machine.

The ATIRE search engine defines a word to be a sequence of characters or numbers, where a character or number is determined by the unicode specification. We currently assume that input is in UTF-8 format, and can process encoding errors that may be encountered such as missing continuation bytes. The input is decomposed, normalised and lower-cased following the unicode specifications. ATIRE supports CJK, and includes chinese segmentation, and has been used in experiments at NTCIR. Currently ATIRE does not support entities such as `&aacute;` and ignores any processing directives contained within the document, except for comments.

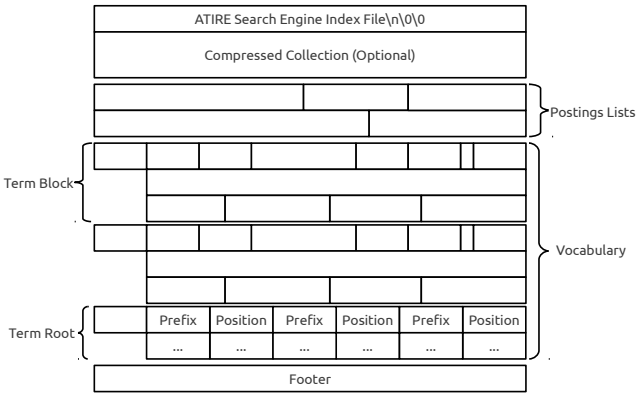

Figure 1: The overall structure of the index file

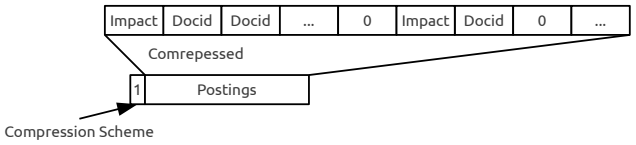

Figure 2: The structure of a postings list

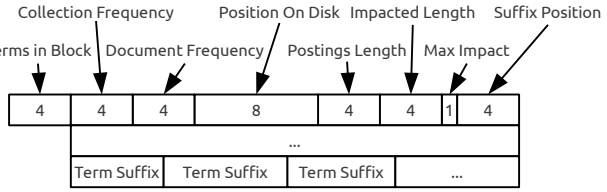

Figure 3: The structure of a vocabulary leaf

ATIRE performs all indexing in memory on a single machine, although distributed indexing is being investigated. To work around this limitation, ATIRE also includes a tool to combine previously generated indexes with minimal memory overhead. By processing each indexed term separately the merge tool only requires enough memory to contain the merged postings list, and to maintain the first level of the dictionary structure described below. This allows the indexing of collections that otherwise could not be indexed on commodity hardware, for example the ClueWeb09 Category A collection.

3. INDEX STRUCTURE

The ATIRE search engine generates a single index file that consists of multiple distinct sections. By restricting the index to a single file we minimise the likelihood of a user not having all parts of the index at search time.

The first few bytes of the index file contain the string `ATIRE Search Engine Index File\n\0\0`, so that the file type can be identified by a person using the command `head -n 1`. A diagrammatic overview of the index structure is shown in Figure 1.

The first section of the index file is optional, and contains the compressed original documents in the collection. This feature allows for, among other things, focused retrieval and snippet generation. The location of each compressed document is stored in a special term inside the index.

The second section of the index file contains the postings lists for each of the terms. Traditionally postings lists are stored as a sequence of ⟨docid, term frequency⟩ pairs, ordered by docid. In the ATIRE search engine we instead sort on term frequency first [26, 27], then for each term frequency we store the docids in a difference encoded list, terminated with a 0. This ordering on term frequencies first is referred to as impact ordered.

The ATIRE search engine supports the use of precomputed quantised impact scores, where instead of storing the term frequency values we instead precompute the RSV for each term with respect to each document after indexing is complete [1].

In order to better compress these numbers, they are quantised into integers using the *Uniform* method [1], which preserves the original distribution of numbers, so no additional decoding is required at query time [23, 1]. In the ATIRE search engine, these values are scaled to 1–255, so that they may be stored in one byte.

Storing the postings lists in this impact ordered format can be thought of as a form of compression, as fewer integers need be stored. In the worst case where every impact value is used, then in an impact ordered format, $D + 510$ integers need to be stored (due to scaling, quantisation and list termination), as opposed to $2D$, where D is the number of documents that contain the term.

We refer to each list of docids for each impact value as a quantum. Each quantum is stored as a difference encoded list, and the entire impact ordered list is then compressed. The ATIRE search engine is capable of compressing postings lists using different compression schemes (carryover 12, elias delta, elias gamma, golomb, none, relative 10, sigma, simple 9, and variable byte) to minimise the disk space taken by the index. By default, however, ATIRE uses variable byte compression. A diagrammatic representation of this structure is shown in Figure 2.

The third section of the index contains the vocabulary structure that holds all the terms that have been indexed. By default, the ATIRE search engine uses the *embedfixed* algorithm [18] to store vocabulary terms since this algorithm provides a good trade-off between storage space and lookup time. The *embedfixed* algorithm stores the vocabulary in a two level B-tree structure. The first level of which contains the unique first four character prefixes of terms in the vocabulary. Each leaf node in the second layer contains the suffixes of those terms that share the common prefix of the parent node. In essence, it is a form of front-encoding that can be searched efficiently.

As well as storing terms in the vocabulary, a number of variables are also required for each term; they are collection and document frequencies used for ranking purposes, location of the postings list stored on disk and the list length used for retrieval of the postings from disk. These variables are stored in the leaf nodes of the vocabulary B-tree.

Aside from these variables associated with each term, extra variables are introduced for each term: the *postings_length* holds the compressed length of the postings list; the *impacted_length* variable stores the number of integers in the decompressed postings list. These values allow our decompression routines to take the form "decompress n integers from this pointer", and by identifying the longest decom-

pressed postings lists (which is stored in the file footer), allocate a single buffer for decompression purposes at search time.

The suffixes for each term inside the leaf node are stored as a null terminated set of strings at the end of the leaf node block. For this reason the *suffix_position* variable identifies where in this block of suffixes the suffix for this individual term begins. The *local_max_impact* holds the maximum impact value for the term, and is used for early termination and pruning of query evaluation [26, 27]. Figure 3 shows a diagrammatic layout of these variables and the number of bytes assigned to them.

As a further space saving we can store the postings lists directly in the vocabulary structure for terms that occur either once or twice in the collection. This can be done by re-purposing some of the variables in the vocabulary leaves, much like a `union` in the C programming language. How to process these lists can be determined at run time by examining the document frequency for the term. It not only saves the storage space for the postings, but also eliminates the extra storage needed for the impact header and postings list header.

The ATIRE search engine has the option of loading indexes completely into memory at search time. In the case that the index is not loaded completely into memory, the vocabulary root is loaded into memory, and during query evaluation is binary searched. The relevant term leaf is then loaded into memory and binary searched to find the term details, then the document frequency is checked. This technique is a form of pre-fetching [20], and saves an extra disk seek and read. Further details of this method are to be published at a later date.

Lastly, the index file contains a footer that contains variables that describe the index, and are used to minimise the number of memory allocations needed when performing query evaluations, such as the length of the longest postings list. These variables are designed to allow the ATIRE search engine to perform no dynamic memory allocation at search time. Certain variables that are associated with the index that are known at indexing time, such as whether the impact values are pre-calculated RSV scores, are stored within the index itself as special terms.

As shown in Table 1, the ATIRE search engine is capable of producing compact indexes that are a fraction of the size of the original collection, the rate of which depends largely on the ratio between indexable and non-indexable content. The ClueWeb09 Category A index was constructed with spam filtering set to discard the 70% most spammiest documents, as suggested by Cormack et. al. [10], with the number of documents included in the index shown in brackets in Table 1.

Table 5 shows some comparisons for indexing and searching times across the ClueWeb09 Category A and B collections. Each index was constructed without quantisation and searching was performed using a single thread, with none of the optimisations discussed below, across queries 101–150.

Collection	Index Time (hh:mm:ss)	Search Time Per Query	Search MAP
ClueWeb09 Cat. B	4:10:20	11.9s	0.1216
ClueWeb09 Cat. A (excl. 70% spam)	20:30:38	30.7s	0.1028

Table 5: Comparison of timings for indexing and searching across the ClueWeb09 collection

4. QUERY EVALUATION

There are two main query evaluation methods used in information retrieval systems, *document-at-a-time* and *term-at-a-time* processing. The document-at-a-time approach completely evaluates one document at a time before moving to the next, while the term-at-a-time approach process one query term at a time.

There are advantages and disadvantages to the two approaches; (1) term-at-a-time requires an array of intermediate accumulators (one for each document) to hold the accumulated results between the evaluation of each term, while document-at-a-time only needs to hold the top n documents (where n is the number of documents to return). Turtle & Flood [33] state that document-at-a-time is more cost efficient than term-at-a-time based on the assumption that the intermediate accumulators are stored on disk. They state that the performance of the two methods would be equivalent if the accumulators could be stored in memory. (2) Document-at-a-time requires a random scan of postings lists for all the query terms in order to fully evaluate a document. This scan takes time especially if all postings lists cannot be held in memory. Skipping [21] and blocking [22] were introduced to allow pseudo-random access into postings lists. However, there is an extra overhead to build skipping and blocking, and the index size increases. Broder et al. [6] addressed this random scan problem by introducing a new document-at-a-time query processing algorithm called WAND which can smartly skip some unnecessary postings for fast scanning. Ding & Suel [12] further extended the WAND algorithm and introduced Block-Max WAND which can further skip more unnecessary postings. The skipping criteria for both of the algorithms are based on the runtime calculation of current thresholds of the maximum impacts for all query terms. (3) Postings lists for document-at-a-time must be longer because the postings lists are sorted on doc id and are not impact ordered. (4) Intuitively, document-at-a-time is more suitable for conjunctive search while term-at-a-time for disjunctive search.

Most criticisms aimed at term-at-a-time approach are towards the requirement of the intermediate accumulators and the need to sort the accumulators to return the top documents. However, we believe that it is more difficult to manage the memory for all postings lists and efficiently random scan the postings lists for document-at-a-time. We are not concluding that term-at-a-time is better than document-at-a-time, or vice versa. Instead we have built a baseline using the term-at-a-time approach in the ATIRE search engine and will use this baseline to compare and investigate the document-at-a-time approach in future work.

The rest of this section discusses how the issues associate with the term-at-a-time approach are addressed in ATIRE for query evaluation.

4.1 Ranking Functions

By default, ATIRE uses a modified BM25 ranking function. This variant does not result in negative IDF values [1] and is

[1]We thank Shlomo Geva for this contribution.

defined as:

$$RSV_d = \sum_{t \in q} \log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1+1)\, tf_{td}}{k_1\left((1-b)+b \times \left(\frac{L_d}{L_{avg}}\right)\right) + tf_{td}}$$

Here, N is the total number of documents, and df_t and tf_{td} are the number of documents containing the term t and the frequency of the term in document d, and L_d and L_{avg} are the length of document d and the average length of all documents. The empirical parameters k_1 and b have been set to 0.9 and 0.4 respectively by training on the INEX 2008 Wikipedia collection.

There are a number of other ranking functions supported in ATIRE as well, for example: Bose-Einstein GL2, Divergence from randomness, Terrier DPH and DFRee, Language Models, and Pregen [30].

4.2 Pruning

The processing (decompression and similarity ranking) of postings and subsequent sorting of accumulators can be computationally expensive, especially when queries contain frequent terms. Processing of these frequent terms not only takes time, but also has little impact on the final ranking results. Postings pruning is a method to eliminate unnecessary processing of postings and provide partial scores for top-k documents. Postings pruning can be done at either index time or query time. Pruning at index time reduces the physical size of the index file [8, 25, 5]. However it is a lossy compression; pruned postings are not kept for access at query time.

Pruning at query time does not modify the index, but prunes postings at run-time during query evaluation. It allows different criteria at query time to be applied to keep track of top k documents. A number of pruning methods have been developed and proved to be efficient [7, 15, 24, 21, 32, 27, 1, 31, 17, 19]. ATIRE supports both pruning at index time and at query time, and pruning at query time is discussed in this section.

In ATIRE, the *heapk* pruning algorithm [17, 19] is used to keep track of the top-k documents. There are two stages in the algorithm. The first stage is the initialisation stage, as shown in Algorithm 1. N is the number of documents in the collection. *top_k* is the number of top documents (specified as a command-line parameter) to be returned. *result_list* keeps track of the number of current top candidate documents during evaluation. *acc* is the accumulator array which hold the intermediate similarity scores for each document. *heapk* is an array of pointers which will be used by the minimum heap to keep track of current top documents. *top_bitstring* is an array of bits (one bit for each document) to track if the document is marked as one of the top candidate documents.

Algorithm 1 Heapk Initialisation

Require: $N > 0$ and $lower\_k > 0$
1: $N \leftarrow total\_documents\_in\_collection$
2: $top\_k = lower\_k$
3: $result\_list = 0$
4: $acc \leftarrow$ **new** $array[N]$
5: $heapk \leftarrow$ **new** $array[N]$
6: $top\_bitstring \leftarrow$ **new** $array[N]$

The second stage is the update stage, shown in Algorithm 2. There are four steps. First (lines 1 to 3), the score is updated for the accumulator. Second (lines 4 to 12), if the number of the current top candidate documents is less than the required ($result\_list < top\_k$), it means the heap is not full. A new document (if $old\_value = 0$) can be simply added to the heap and the corresponding bit is set. When the heap is full ($result\_list = top\_k$), it is required to build the minimum heap on the *heapk* array. Third (lines 13 to 14), if *result_list* is no less than *top_k* and the current document is marked as set (*top_bitstring*[*index*]), it means the document which is already in the top gets updated. Updating one of the top document could violate the properties of the minimum heap. It is necessary to call *min_update*() to partially fix the heap. Last (lines 15 to 18), if *result_list* is no less than *top_k* and the score is greater than the smallest score in the heap (which is *heapk*[0]), it means the document, which was not in the top, should now be inserted into the top to replace the smallest score. The bit of the smallest document in heap should be unset. The new document is inserted into the heap by calling *min_insert*.

Instead of repeatedly re-building the minimum heap for update and insertion operations, two special functions are implemented for efficiency optimisation. Every time one of the top candidate documents gets updated, the *min_update*() function is called. It first linearly scans the *heapk* array to locate the right pointer and then partially traverses down the subtree of the pointer for proper update of the minimum heap. The linear scan is required because the minimum heap is not a binary search tree. Every time a new document is going to be inserted into the minimum heap, the *min_insert*() function is called. It first replaces the document with the smallest score and then partially traverses down the tree for proper update of the minimum heap.

Algorithm 2 Heapk Update

Require: $index \geq 0$ and $score > 0$
1: $old\_value \leftarrow$ get the current value of $acc[index]$
2: $acc[index] \leftarrow acc[index] + score$
3: $new\_value \leftarrow$ get the current value of $acc[index]$
4: **if** $result\_list < top\_k$ **then**
5: **if** $old\_value = 0$ **then**
6: $heapk[result\_list] \leftarrow$ address of $acc[index]$
7: $result\_list \leftarrow result\_list + 1$
8: set the bit of $top\_bitstring[index]$
9: **end if**
10: **if** $result\_list = top\_k$ **then**
11: build the minimum heap on *heapk*
12: **end if**
13: **else if** $top\_bitstring[index]$ is set **then**
14: *min_update*() to update the *heapk*
15: **else if** $new\_value >$ the value of $heapk[0]$ **then**
16: unset the bit of $top\_bitstring[heapk[0]]$
17: $min\_insert(acc[index])$
18: set the bit of $top\_bitstring[index]$
19: **end if**

The value of *lower_k* can be specified from command-line, used to tell the *heapk* pruning algorithm how many top documents to keep track of.

The performance of the *heapk* pruning algorithm was investigated in INEX 2010 and the results showed that the algorithm is not only CPU cost efficient but also effective. For details of the experiments and results, see our previous work [17].

4.3 Accumulator Initialisation

The term-at-a-time approach uses a number of accumulators, usually as a static array, to hold the intermediate accumulated results for each document. For large collections, there can be a large number of accumulators and it takes time to initialise them. One way to avoid this problem is to use few accumulators allocated using dynamic search structures [24, 21]. However, dynamic structures require more memory space for each accumulator. For example, a balanced Red-Black tree structure [11] uses about 20 and 32 bytes for each accumulator on 32- and 64-bit architectures respectively. Compared with only 4 bytes required in a static array, only 20% for 32-bit (12.5% for 64-bit) or less of the total number of accumulators should be allocated, otherwise the Red-Black tree structure uses more memory than a static array.

For ATIRE, a new efficient accumulator method has been developed. It not only keeps tracks of the top candidates (using the *heapk* algorithm) but also updates the less important accumulators. This allows initially low scoring candidates be to among the top ones at the final stage. The method uses two static arrays. One array is used to hold all accumulators (one for each document) and the other to hold a number of flags. Every flag is associated with a particular subset of the accumulators, indicating the initialisation status for that set of accumulators (either initialised or not). Essentially, we turn the one dimensional array of accumulators into a logical two dimensional table as shown in Figure 4. The dimension of the table is defined by *height* and *width*, and the number of the flags is the same as the height of the table.

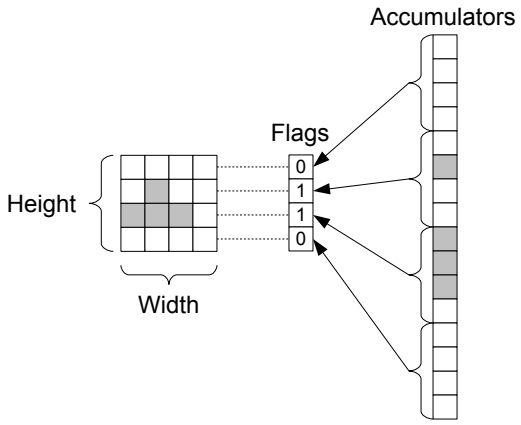

Figure 4: The representation of the accumulators in a logical two dimensional table

As shown in Algorithm 3, the width of the table has to be a whole number (at least 2), and the height can be calculated dynamically by referencing the width and the size of the document collection. Extra accumulators (shown as *padding* in the algorithm) are used to fill the gaps when the number of accumulators is not evenly divisible by the height. The allocation of the extra accumulators are so that we can perform block operation on whole rows. The number of extra accumulators required is usually small (the worst case is $width - 1$).

The update operation for the accumulators is shown in Algorithm 4. First, the index of an accumulator is divided to locate the logical row of the accumulator. Second, the status of the row flag is checked and two outcomes can happen; (1) If the flag has a value of 0, the associated accumulators in the row are initialised and the new value is then added to the accumulator. (2) If the flag has a value of 1, the new value can be simply added to the accumulator.

Algorithm 3 Accumulator Initialisation

Require: $width \geq 2$
1: $N \leftarrow total\_documents\_in\_collection$
2: $height \leftarrow (N/width) + 1$
3: $init\_flags \leftarrow$ **new** $array[height]$
4: initialise $init\_flags$
5: $padding \leftarrow (width * height) - N$
6: $acc \leftarrow$ **new** $array[N + padding]$

Algorithm 4 Accumulator Update

Require: $doc\_id \geq 0$ and $doc\_id < N$
1: $row \leftarrow doc\_id/width$
2: **if** $init\_flags[row] == 0$ **then**
3: $\quad init\_flags[row] \leftarrow 1$
4: $\quad$ initialise the row of the accumulators in acc
5: **end if**
6: $acc[doc\_id] \leftarrow acc[doc\_id] + new\_rsv$

In order to find the optimal solution for the width of the table, a mathematical model for the algorithm was described and a simulation was performed. For a detailed discussion of the mathematical model, the experiments and results, please see Jia et al. [19].

4.4 Quantum At a Time

Instead of the traditional approaches of term-at-a-time and document-at-a-time, we propose a new query evaluation approach called *quantum-at-a-time*. Before the start of a query evaluation, all the quanta of the query terms are sorted on their impact values so that the highest impact quanta can be evaluated first and then the next highest, and so on until some of the remaining quanta cannot cause a change to the top-k documents.

The quantum-at-a-time approach is a mixture between term-at-a-time and document-at-a-time. This new approach is similar to score-at-time [2, 3] and Block-Max WAND [12]. The differences are that term ranks are used in score-at-a-time instead of the impact values to sort the quanta, and a block in Block-Max WAND can have postings with different impact values and Block-Max WAND is for document-at-a-time processing.

The quantum-at-a-time approach is targeted for efficient and effective pruning of postings, and better parallel processing of postings lists on multi-core architectures. We will discuss this work in future publications once we have completed it.

5. TERM PROXIMITY

ATIRE does not currently support positional indexes. We have built several search engines in the past and our experiences suggest that positional indexes are not effective under current academic IR evaluation methods that use a

binary relevance model. If the precision improvement of a positional index cannot yet be demonstrated in a recognized forum such as TREC or INEX then it is difficult to justify having one.

Our informal reasoning for this is as follows. If the user enters a two word phrase then for a document to contain that phrase it must also contain both words. For a document to contain that phrase many times it must also contain both those words many times. That is, a document that would rank highly for the phrase would also rank highly for both words not as a phrase – and typically they do.

Further, examining the precision-at-1 (P@1) score for both approaches; if phrase searching is more effective than term searching then a specific set of conditions must be met: (1) the term search must not put a relevant document at position 1, and (2) the phrase search must do so. Simply replacing one relevant document with another has no effect on precision; and nor does replacing a non-relevant document with another non-relevant document.

The circumstances necessary for an improvement are hard to meet; but we accept that they can be so. If both words in the phrase are seen frequently in a document, but never as a phrase, then phrase searching should increase precision – in this case the phrase acts as a noise filter. An example of such a query is "The Who". A second example is when all the words are seen but not as the phrase. Again the phrase acts as a filter. An example of this can be seen when searching for the musician "Lisa Lisa" on the Apple iTunes Store.

We believe these examples are pathological and can be handled by storing n-grams in the vocabulary and welcome an evaluation forum running a phrase search track. Such an experiment was conducted at INEX 2009 but was inconclusive ("competitive, but not superior" [13]).

6. RELEVANCE FEEDBACK AND DIVERSIFICATION

The ATIRE search engine currently supports the use of pseudo-relevance feedback. Pseudo-relevance feedback makes the assumption that the top n returned documents are relevant to the query and inspects those documents to identify new and relevant keywords.

In ATIRE we use the KL-divergence for terms inside these top documents to identify the terms which are more likely to be used inside these top documents than would be expected by examining the entire collection. These identified terms are then added to the original query according to Rocchio's algorithm [28]. Terms that were added to the query with this method are given an equal weighting with, and may duplicate, the original terms. The ATIRE search engine allows for other methods for term selection to be incorporated, although currently only KL-divergence is supported.

Although we perform relevance feedback by identifying those terms that are used more frequently in relevant documents than one would expect, it can be thought of as a result of clustering the documents on topic. Relevance feedback promotes those documents that belong to clusters that contain documents that have been identified as relevant.

When a search engine is presented with an ambiguous query, such as "apple", then it can employ diversification in order to help maximise the usefulness of the returned results to the user. Diversification aims to select documents that are related to different possible interpretations of the original query (continuing the above example: the computer company, fruit, record company, etc.) so that each interpretation is given weight according to its likelihood.

One such method for diversification is to cluster the documents by topic, and then select documents from clusters that contain no previously selected documents. Currently the ATIRE search engine does not explicitly diversify results lists, but this is an active research area for us.

When presented as results of clustering documents, relevance feedback and diversification are juxtaposed against each other. Each method uses an opposing criteria to select documents, with diversification exploring cluster-space, and relevance feedback exploiting it. However, both of these ideas seem to improve the results.

7. BUT WAIT THERE'S MORE!

In addition to all the above discussed features, the ATIRE search engine also supports: stemming including Krovetz and Porter as well as soundex and metaphone; topsig; snippet generation and focused retrieval.

The ATIRE search engine can natively read assessment formats, evaluate queries against a large number of metrics, and produce runs for evaluation forums.

8. CONCLUSION AND FUTURE WORK

In future work we aim to change the storage format of postings lists in order to allow quantum-at-a-time processing discussed earlier in Section 4.4. In order to do this, we need to identify where each quantum is stored within a postings list, which will require the use of an impact header. This header structure is in development and we aim to also include support for incremental index updates.

With the ATIRE search engine, we have tackled optimisation of the term-at-a-time processing approach in several areas; (1) The index structure has been optimised. An impact header is created for each postings list for easy manipulation of those lists (sorted on impact values). (2) The *heapk* pruning algorithm is used to keep track of the top-k documents, thus eliminating the need to sort accumulators. (3) The cost of the accumulator initialisation has been minimised by using the logical two dimensional table.

In future work, we will use this baseline to compare with the document-at-a-time and quantum-at-a-time approaches. We will also continue our experiments in relevance feedback, diversification, focused and snippet retrieval in INEX. We hope one of the evaluation forums will run term proximity in the near future.

9. REFERENCES

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

[2] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 226–233, New York, NY, USA, 2005. ACM.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*,

SIGIR '06, pages 372–379, New York, NY, USA, 2006. ACM.

[4] P. Bailey, N. Craswell, and D. Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Inf. Process. Manage.*, 39(6):853–871, Nov. 2003.

[5] R. Blanco and A. Barreiro. Probabilistic static pruning of inverted files. *ACM Trans. Inf. Syst.*, 28(1):1–33, 2010.

[6] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 426–434, New York, NY, USA, 2003. ACM.

[7] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. pages 97–110, 1985.

[8] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. pages 43–50, 2001.

[9] C. Clarke, N. Craswell, and I. Soboroff. Overview of the trec 2004 terabyte track. In *Proceedings of TREC*, volume 2004, 2004.

[10] G. Cormack, M. Smucker, and C. Clarke. Efficient and effective spam filtering and re-ranking for large web datasets. *Information retrieval*, 14(5):441–465, 2011.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithm*. The MIT Press, 1990.

[12] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM.

[13] S. Geva, J. Kamps, M. Lethonen, R. Schenkel, J. Thom, and A. Trotman. Overview of the inex 2009 ad hoc track. *Focused Retrieval and Evaluation*, pages 4–25, 2010.

[14] D. Harman. *Overview of the third text retrieval conference (TREC-3)*, volume 500. Diane Pub Co, 1995.

[15] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41:581–589, 1990.

[16] D. Hawking, N. Craswell, and P. Thistlewaite. Overview of trec-7 very large collection track. *NIST SPECIAL PUBLICATION SP*, pages 93–106, 1998.

[17] X.-F. Jia, D. Alexander, V. Wood, and A. Trotman. University of otago at inex 2010. In *INEX '10: Pre-Proceedings of the INEX*. ACM, 2010.

[18] X.-F. Jia, A. Trotman, and J. Holdsworth. Fast search engine vocabulary lookup. In *ADCS '11*, 2011.

[19] X.-F. Jia, A. Trotman, and R. O'keefe. Efficient accumulator initialisation. In *ADCS '10: Proceedings of the Fifteenth Australasian Document Computing Symposium*, 2010.

[20] X.-F. Jia, A. Trotman, R. O'Keefe, and Z. Huang. Application-specific disk I/O optimisation for a search engine. In *PDCAT '08*, pages 399–404, 2008.

[21] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[22] A. Moffat, J. Zobel, and S. T. Klein. Improved inverted file processing for large text databases. pages 162–171, 1995.

[23] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

[24] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

[25] E. S. d. Moura, C. F. d. Santos, B. D. s. d. Araujo, A. S. d. Silva, P. Calado, and M. A. Nascimento. Locality-based pruning methods for web search. *ACM Trans. Inf. Syst.*, 26(2):1–28, 2008.

[26] M. Persin. Document filtering for fast ranking. pages 339–348, 1994.

[27] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[28] J. Rocchio. Relevance feedback in information retrieval. 1971.

[29] R. Schenkel, F. Suchanek, and G. Kasneci. YAWN: A semantically annotated wikipedia xml corpus. March 2007.

[30] N. Sherlock and A. Trotman. Efficient sorting of search results by string attributes. In *ADCS '11*, 2011.

[31] A. Trotman, X.-F. Jia, and S. Geva. Fast and effective focused retrieval. volume 6203 of *Lecture Notes in Computer Science*, pages 229–241. 2010.

[32] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. pages 987–990, 2007.

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

[34] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.