Efficient Accumulator Initialisation

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

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

Richard O'Keefe
Department of Computer Science
University of Otago
Dunedin, New Zealand
ok@cs.otago.ac.nz

Abstract *IR efficiency is normally addressed in terms of accumulator initialisation, disk I/O, decompression, ranking and sorting. Traditionally, the performance of search engines is dominated by slow disk I/O, CPU-intensive decompression, complex similarity ranking functions and sorting a large number of candidate documents. However, after we have applied a number of optimisation techniques, our search engine is bottlenecked by accumulator initialisation. In this paper, we propose an efficient accumulator initialisation algorithm, which represents the traditional static accumulator array as a logical two dimensional table and uses a number of flags to track the initialisation status of the accumulators. The efficiency of the algorithm is verified by a simulation program and a search engine. The overall performance can be as good as a 93% increase in throughput.*

Keywords Accumulator Initialisation, Efficiency, Postings Pruning.

1 Introduction

Effectiveness and efficiency are two of the main issues in Information Retrieval (IR). Effectiveness has been the main focus of research. In recent years, efficiency has started to draw more attention under the trend of larger document collection sizes.

IR efficiency is normally addressed in terms of accumulator initialisation, disk I/O, decompression, ranking and sorting. A large portion of the performance of search engines is dominated by (1) slow disk read of dictionary terms and the corresponding postings lists, (2) CPU-intensive decompression of postings lists, (3) complex similarity ranking functions and (4) sorting a large number of possible candidate documents. The effect of accumulator initialisation on the performance has almost been ignored.

However, after we applied a number of optimisation techniques, our search engine was bottlenecked by accumulator initialisation. We have deployed space efficient compression algorithms for storing the dictionary and inverted files, disk I/O is completely eliminated by simply storing the index in memory. Instead of storing the traditional ⟨document number, term frequency⟩ pair in postings, we pre-compute and store impact values instead [15, 1]. The search engine simply adds the impact values when ranking.

Proceedings of the 15th Australasian Document Computing Symposium, Melbourne, Australia, 10 December 2010.

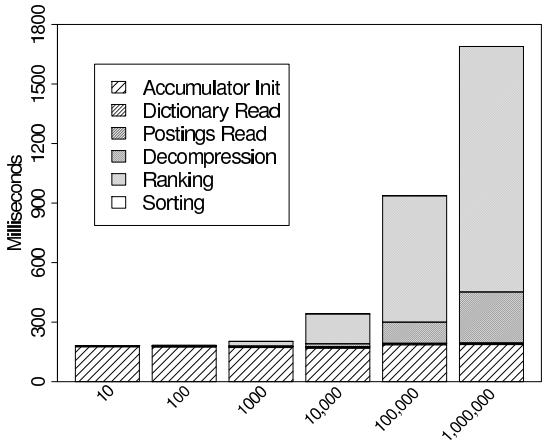

Figure 1: The performance of our optimised search engine using traditional static array for accumulator initialisation.

Postings pruning at query time is a very effective method for reducing the number of postings to be processed and the number of accumulators to be sorted, while still maintaining high precision [8, 14, 23, 17, 1, 22]. Since only part of the postings lists is processed in pruning, only partial decompression of postings lists is required [14, 13, 3, 2]. Our search engine has adopted a heap data structure to keep track of the top k documents and static pruning of postings lists with partial decompression.

Figure 1 shows the performance of our optimised search engine with these optimisations enabled. The document collection and queries are the INEX 2009 Wikipedia collection [18] and the 115 Type-A (short) queries from the INEX 2009 Efficiency Track [19]. Only the top k=15 results are returned and each column in the figure corresponds to a static pruning of 10, 100, 1000, 10 000, 100 000, 1 000 000 postings. When no more than 10 000 postings are processed, the accumulator initialisation takes most of the time,

between 50% and 96% of the total time. When there are 100 000 and 1 000 000 postings (which is one third of the collection size) being processed, the accumulator initialisation still takes 20% and 11% of the total evaluation time respectively.

In this paper, we propose an efficient accumulator initialisation algorithm, using static data structures, for the *term-at-a-time* approach. The algorithm logically partitions the static array of accumulators into a two dimensional table. A flag is created for each logical row to indicate if that row has been initialised. Before processing a new query, only the flags are re-initialised instead of re-initialising all accumulators. Because the algorithm keeps track of all accumulators, there is no loss of precision.

The remainder of the paper is organised as follows. In Section 2, we discuss the related work. Second 3 describes in detail how the algorithm works and presents a mathematical model. In Section 4, the performance of the algorithm is conducted on a simulation and our search engine. Section 5 concludes.

2 Related Work

Disk I/O involves reading query terms from a dictionary (a vocabulary of all terms in the collection) and the corresponding postings lists for the terms. The dictionary has a small size and can be loaded into memory at start-up. However, due to their large size, postings are usually compressed and stored on disk. A number of compression algorithms have been developed and compared [21, 4]. Another way of reducing disk I/O is caching, either at application level or system level [5, 11]. Since the advent of 64-bit machines with vast amounts of memory, it has become feasible to load both the dictionary and the compressed postings into main memory, thus eliminating all disk I/O. Reading both dictionary and postings lists into memory is the approach taken in our search engine.

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 at query time is a method to eliminate unnecessary processing of postings and thus reduce the number of non-zero accumulators to be sorted. A number of pruning methods have been developed and proved to be efficient and effective [8, 14, 23, 17, 1, 22]. In previous work [22], the *topk* pruning algorithm partially sorts the static array of accumulators using a special version of quick sort [6] and statically prunes postings. Based on this work, we have developed the *heapk* pruning algorithm. Instead of explicitly sorting the accumulators, we uses a heap data structure to keep track of the top documents.

Traditionally, postings are stored in pairs of ⟨document number, term frequency⟩ pairs. However, postings should be impact ordered so that most important postings can be processed first and the less important ones can be pruned using pruning methods [16, 17, 1]. One approach is to store postings in order of term frequency and documents with the same term frequency are grouped together [16, 17]. Each group stores the term frequency at the beginning of the group followed by the compressed differences of the document numbers. The format of a postings list for a term is a list of the groups in descending order of term frequencies. Another approach is to pre-compute similarity values and use these pre-computed impact values to group documents instead of term frequencies [1]. Pre-computed impact values are positive real numbers. In order to better compress these numbers, they are quantised into whole numbers [15, 1]. Three forms of quantisation method have been proposed (*Left.Geom*, *Uniform.Geom*, *Right.Geom*) and each of the methods can better preserve certain range of the original numbers [1]. In our search engine, we use pre-computed BM25 impact values to group documents and the differences of document numbers in each group are compressed using Variable Byte Coding by default. We choose to use the *Uniform.Geom* quantisation method for transformation of the impact values, because the *Uniform.Geom* quantisation method preserves the original distribution of the numbers, thus no decoding is required at query time. Each impact value is quantised into an 8-bit whole number.

Since only partial postings are processed in query pruning, there is no need to decompress the whole postings lists. Skipping [14] and blocking [13] allow pseudo-random access into encoded postings lists and only decompress the needed parts. Further research work [3, 2] represent postings in fixed number of bits, thus allowing full random access. Our search engine partially decompress postings list based on the worst case of the static pruning. Since we know the parameter value of the static pruning and the biggest size of a uncompress impact value (4 bytes), we can multiply these number together to find the cut point for decompression. We can simply hold decompression after that number of postings have been decompressed.

A number of accumulators, usually as a static array, need to be created and initialised for *term-at-a-time* processing [8, 10]. The accumulators hold the intermediate accumulated results for each document. For large collections, a large number of accumulators has to be used. Initialisation of large number of accumulators can take time. One solution to cut down the initialisation time is to use few accumulators, which are allocated using dynamic search structures [15, 14]. Depending on which dynamic structure is used, the memory space required for each accumulator can be several times that of the static array structure. For example, a balanced Red-Black tree structure [9] requires about 20 bytes for each accumulator on 32-bit architectures, and about

32 bytes on 64-bit architectures, compared with only 4 bytes needed in a static array. Only 20% (12.5% for 64-bit) or less of the total number of accumulators should be allocated, otherwise the dynamic structure uses more memory. The more memory is used, the longer it takes to allocate.

Since only a portion of accumulators can be allocated using dynamic search structures, a pruning algorithm has to be used to keep only the top candidates and to prune other less important ones [15, 14]. On the other hand, our search engine allocates all accumulators, and not only keeps track of the top candidates but also updates the less important accumulators. This leaves the possibility for the less important candidates to be among the top ones at the final stage.

The criticism of the *term-at-a-time* approach is the requirement of accumulators in order to hold intermediate results. Alternatively, the *document-at-a-time* approach ranks one document at a time, thus does not need to hold intermediate results [24, 20]. However, the *document-at-a-time* approach requires random scan of postings lists, which takes time [20].

3 The Algorithm

Instead of using dynamic accumulator structures, we use two static arrays. One array is used to hold all the accumulators (one for each document), 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. Essentially, we turn the one dimensional array of accumulators into a logical two dimensional table as shown in Figure 2. The dimension of the table is represented by $height$ and $width$. The number of flags is the same as the height of the table.

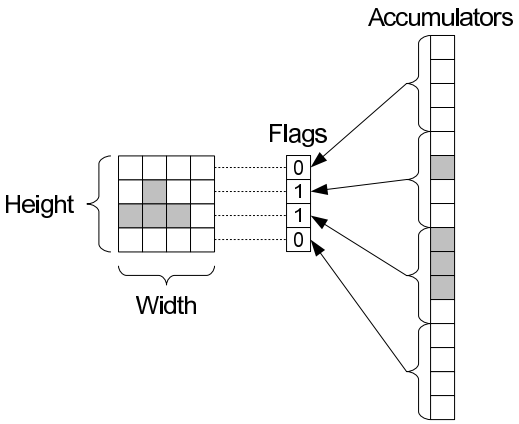

Figure 2: The representation of the accumulators in a logical two dimensional table.

One obvious question is why not just dynamically allocate each row only when needed and replace the flags with pointers pointing to the dynamically allocated rows. There are two efficiency issues when rows are dynamically allocated. First, it is faster to allocate a large chuck of memory in one go, rather than splitting the same of amount of memory into many smaller pieces and allocating one piece at at time. Modern computers come with very large memories, so it is worth to sacrifice small amount of memory for speed. Second, two steps are required to locate each accumulator because rows are not guaranteed to be allocated consecutively in memory (the first step is to locate the row and the second step to find the offset in the row).

Another question is how to determine the dimension of the logical table. In the following sections, a mathematical model is provided to answer this question formally and a simulation program is tested with various sizes of width. For now, let us concentrate on how the algorithm works.

Initially, the flags are initialised to zero, indicating all the accumulators having zero values. When updating an accumulator with a new value, the flag associated with that row of the accumulator is set to 1. For the example shown in Figure 2, The second accumulator in the second logical row has a non-zero value and the associated flag for the row has a value of 1.

The width has to be a whole number at least 2. The height can be calculated according to the width and the size of the document collection, as shown in Algorithm 1. Because the total number of accumulators represented by the logical table can be more than the collection size, extra accumulators (shown as $padding$ in Algorithm 1) are allocated in the accumulator array (this is for efficiency). The number of extra accumulators required is usually small and the worst case is $width - 1$.

Algorithm 1 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 2 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: $acc[doc\_id] \leftarrow acc[doc\_id] + new\_rsv$

Algorithm 2 shows how to update the accumulators. First, the logical row of an accumulator can be obtained by a division operation of the index of the accumulator. Second, the row flag is checked. Two possible cases can happen. If the flag is 0, it set to 1, the associated accumulators are initialised, and the new value is added to the accumulator. If the flag is 1, the new value can be simply aggregated to the accumulator.

3.1 The mathematical model

Let D be the number of documents, Q the number of terms in the query, L the size of the postings list for each term, B the width of each row, $K = D/B$ the number of rows, and U the number of rows that are not used. Zipf's law tells us that postings lists vary enormously in size, but we are considering a system with pruned impact-ordered lists and pessimistically assume all postings lists are of the pruned length L.

Processing the query is a many step process that starts with clearing the row flags. Then for each term, the postings are loaded, decompressed, and processed. That processing in turn involves computing the row number and checking the row flag, if the flag is zero the row is cleared and the flag is set. The accumulator is always increased (and top-k processing is done). At the end of the query the top-k accumulator pointers are sorted.

For only two of these steps does the cost depend on the width and height of the two-dimensional accumulator table:

- c_1K, zeroing the flags for each row;
- $c_2(K - U)B$, zeroing all accumulators in all flagged row (and setting the bits);

where c_1 and c_2 are unknown constant factors. We wish to minimise the cost of initialising the flags, c_1K, plus the cost of initialising all the used rows, $c_2(K - U)B$.

In practice, some documents are more likely to be selected than others (due to the clustering hypothesis), and the user does not expect the terms to be independent. That is, they expect fewer than QL distinct documents to be found. We pessimistically assume each term is independent and identically (randomly) distributed for this analysis.

The postings for a term would normally be sampled without replacement, but if L is big enough and L/D is small enough, that can be approximated by sampling with replacement.

P(row k is unused)
= P(document $d \subseteq kB \ldots kB + B - 1$ not chosen)
= P(document d is not chosen)^B
= P(document d is not in postings for term $t \subseteq t_1..t_Q$)^B
= P(document d is not in postings for term t)^{QB}
= $(1 - L/D)^{QB}$

The postings are randomly distributed in the postings list and so represent independent trials, there are K rows and the probability of a single row being unused is $(1 - L/D)^{QB}$. So the distribution of U is Binomial$(K, (1 - L/D)^{QB}$. As mean(Binomial$(n, p)) = np$, the mean of the number of unused rows is $K(1 - L/D)^{QB}$. It depends on the number of terms in the query and the number of postings for each term.

We wish to minimise $c_1K + c_2(K - U)B$. Since $B = D/K$ we get

$$\begin{aligned} & c_1K + c_2(K - U)B \\ = \; & c_1K + c_2\tfrac{(K-U)D}{K} \\ = \; & c_1K + c_2\tfrac{KD-UD}{K} \\ = \; & c_1K + c_2\tfrac{KD}{K} - c_2\tfrac{UD}{K} \\ = \; & c_1K + c_2D - c_2\tfrac{UD}{K} \end{aligned}$$

Recall that U is Binomial$(K, (1 - L/D)^{QB}$ and the mean value is $K(1 - L/D)^{QB}$. We get

$$\begin{aligned} & c_1K + c_2D - c_2\tfrac{UD}{K} \\ = \; & c_1K + c_2D - c_2\tfrac{K(1-\frac{L}{D})^{BQ}D}{K} \\ = \; & c_1K + c_2D - c_2(1 - \tfrac{L}{D})^{BQ}D \end{aligned}$$

Applying the binomial theorem to $(1 - \frac{L}{D})^{QB}$ and truncatingat the first two terms, we get

$$\begin{aligned} & c_1K + c_2D - c_2(1 - \tfrac{L}{D})^{BQ}D \\ \approx \; & c_1K + c_2D - c_2(1 - \tfrac{L}{D}BQ)D \\ \approx \; & c_1K + c_2D - c_2(D - \tfrac{L}{D}BQD) \\ \approx \; & c_1K + c_2D - c_2D + c_2LBQ \\ \approx \; & c_1K + c_2LBQ \\ \approx \; & c_1\tfrac{D}{B} + c_2LBQ \end{aligned}$$

Fermat's stationary point theorem tells us that a differentiable function has its maxima and minima only on the boundaries or where the first derivative is zero. The second derivative tells whether an extreme point is a maximum or a minimum.

$$\begin{aligned} & \tfrac{d}{dB}(c_1\tfrac{D}{B} + c_2LBQ) \\ = \; & \tfrac{d}{dB}(c_1\tfrac{D}{B}) + \tfrac{d}{dB}(c_2LBQ) \\ = \; & -c_1\tfrac{D}{B^2} + c_2LQ \end{aligned}$$

$$\begin{aligned} & \tfrac{d^2}{dB^2}(c_1\tfrac{D}{B} + c_2LBQ) \\ = \; & \tfrac{d}{dB}(-c_1\tfrac{D}{B^2} + c_2LQ) \\ = \; & c_1\tfrac{D}{B^3} \end{aligned}$$

By setting the first derivative to zero, we get

$$\begin{aligned} c_1\tfrac{D}{B^2} &= c_2LQ \\ B^2 &= \tfrac{c_1D}{c_2LQ} \\ B &= \sqrt{\tfrac{c_1D}{c_2LQ}} \end{aligned}$$

Since the second derivative is greater than zero, this must be a local minimum.

4 Experiments

We conducted all our experiments on a system with dual quad-core Intel Xeon E5410 2.3 GHz, DDR2 PC5300 8 GB main memory, Seagate 7200 RPM 500 GB hard drive, and running Linux with kernel version 2.6.30.

For both the simulation and our search engine, we set the width of the table to a power of 2. This allows us to look up the row efficiently using a hashing function that is just a bit shift. A shift operation is considerably faster than a division.

First, we tried a number of simulation tests to inspect the behaviour of the algorithm. Then we integrated the algorithm into our search engine and investigated its performance using the INEX 2009 Wikipedia collection [18] and queries from the INEX 2009 Efficiency Track [19].

4.1 Simulation

We wrote a simulation program to test both the traditional static array allocation and our proposed logical two dimensional table algorithm.

The program takes five parameters; (1) *collection_size*, (2) *postings_list_length*, (3) *num_of_terms*, (4) *width_in_bits* and (5) *num_of_repeats*. The first two parameters allow the program to simulate different collection sizes with various lengths of postings lists. The third parameter tries to simulate real world queries with different number of terms. The *width_in_bits* parameter sets the size of the width for the logical two dimensional table. The last parameter simply tells the program to repeat a number of times with the same settings.

Each posting only contains an index number (the document number) for indexing into the accumulators. All the index numbers are randomly generated using the Mersenne Twister 64-bit random generator [12]. Every run of the simulation is guaranteed to execute with a different seed for the random number generator and each run was repeated 20 times. For each run, the same lists of postings are used by both the accumulator initialisation techniques.

Figure 3 shows the results of simulating a collection of three million documents. The horizontal axes represents the width in bits and the vertical axes are the performance ratio of the two accumulator initialisation techniques (the total time taken by the logical two dimensional table over the total time taken by the static array structure). A ratio below 100% means the performance increase in our proposed algorithm again the static array approach and above 100% means performance decrease. Four sets of simulation were conducted, each with 1, 2, 3 and 10 terms. For each set, various lengths of postings lists were used, ranged from 10 to 3 million.

As shown in Figure 3, the four sets of simulation showed the same pattern. When the length of the postings lists was short, smaller width for the logical table showed better performance, while larger width shower better performance for long postings lists. On average, bits between 8 and 12 showed good performance for both short and long postings lists.

One thing to note is the caching effects for simulations with a small number of postings. An example of the results being affected by caching is when there are two logical rows ($width\_in\_bit$ is 1) and only 10 postings processed. The size of the static accumulator array is 6 million bytes (the collection size times the byte size of a single accumulator) and the size of the flag array is 1.5 million bytes (we used one byte for each flag). By hand calculation, the performance ratio should be more than 25% because the size of the flag array is 25% of the static accumulator array plus the initialisation of 10 rows for the 10 postings (the worst case). However, the simulation showed that the performance ratio is only about 8.5%. This further suggests that the logical two dimensional table is more efficient because the structure has a smaller memory footprint and thus can be better cached by the CPU (the CPU has 6 MB of L2 cache).

Overall, our proposed logical two dimensional table for accumulator initialisation shows good performance, especially when postings lists are short. This suggests that the algorithm can be better used together with postings pruning techniques, which only processes partial postings.

4.2 Search Engine

As discussed in Section 2, our search engine supports reading of the dictionary and postings directly in memory, impact-ordered postings, partial decompression and postings pruning at query time. In this section we discuss how the proposed accumulator initialisation algorithm performs in our search engine.

In the previous *topk* pruning implementation [22], an extra array of pointers was used and the size of the array was k. The pointers were used to keep track of the top documents in the static array of accumulators. At the final stage, the top k pointers are sorted instead of sorting the static accumulator array directly.

The implementation of our *heapk* pruning algorithm is also based on the pointer array. The *heapk* algorithm simply uses the same pointer array to keep track of top documents. The minimum accumulator among the top k is always pointed to by the first pointer in the heap. During the update of an accumulator, the new value of the accumulator is checked against the minimum. If the new value is greater than the minimum, the minimum pointer simply points to the new value and the heap structure is re-built. If the new value is less than the minimum, then there is nothing to be done. At the final stage, the documents which are tracked by the heap data structure are sorted and returned.

A modified BM25 is used for ranking. This variant does not result in negative IDF values and is 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.

We used the INEX 2009 Wikipedia collection [18] and the 115 Type-A (short) queries for the INEX 2009 Efficiency Track [19]. The collection was indexed with

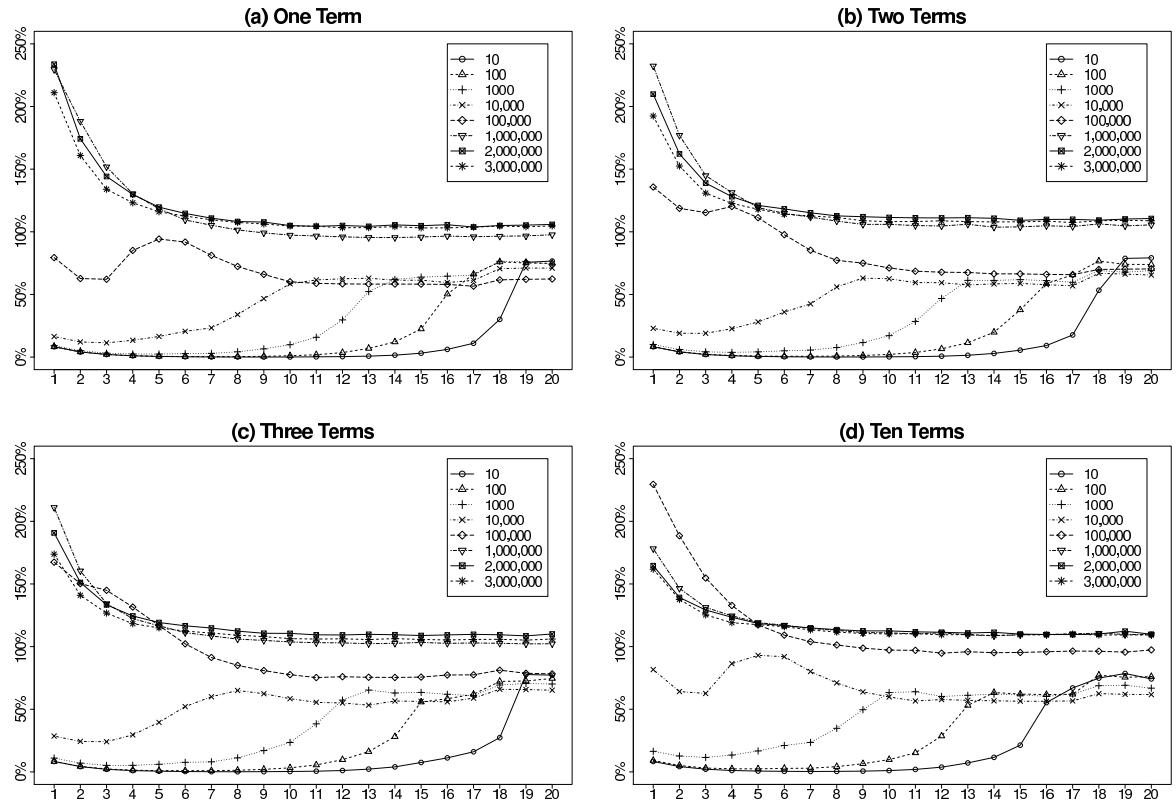

Figure 3: Simulation results on a collection of three million documents.

Collection Size	50.7 GB
Documents	2666190
Average Document Length	881 words
Unique Words	11393924
Total Words	2348343176
Postings Size	1.68 GB
Dictionary Size	269 MB

Table 1: Summary of INEX 2009 Wikipedia Collection.

no words stopped and stemming was not used. Table 1 shows a summary of the document collection.

Initially our search engine used the traditional static array approach for accumulators. To see the performance of the accumulator initialisation with regards to the overall runtime, the 115 queries were evaluated with all the optimisation options enabled (in-memory dictionary and postings, impact-ordered postings, partial decompression and the *heapk* pruning). The *heapk* pruning method was used with a value of 15 for the number of top k documents and various values of 10, 100, 1000, 10 000, 100 000, 1 000 000 for static pruning. The total run-times are shown in Figure 1. When no more than 10 000 postings were processed, the accumulator initialisation took between 50% and 96% of the total evaluation time. When 100 000 and 1 000 000 postings were processed, it took 20% and 11% respectively. As shown in previous work [22], query pruning is not only efficient, but also very effective, even when as few as 1000 postings are processed.

Now our search engine is bottlenecked by the accumulator initialisation. We need a solution for efficient accumulator initialisation. The logical two dimensional table has been integrated into the *heapk* pruning algorithm. The integration requires only changes to a few lines of code. Before processing each query, the flags in the logical two dimensional table are initialised. During the update of an accumulator, the flag is checked to see if it is necessary to initialise the logical row first.

In order to compare the performance again the static array approach, we performed the same set of tests on the logical two dimensional table. The width for the logical two dimensional table was chosen to be 8 bits. The results are shown in Figure 4(a). The only performance differences between these two structures are the accumulator initialisation, the ranking and the total times. The logical two dimensional table took about zero time for the accumulator initialisation since it was very fast to initialise 11719 flags (the height of the table calculated as shown in Algorithm 1). However, the logical two dimensional table added small amount of overhead for ranking due to the extra operations required to keep track of the flag status (as shown in Algorithm 2). When comparing the total evaluation time, the logical two dimensional table outperformed the static array structure in all runs. The best performance increase is 93% when 10 postings

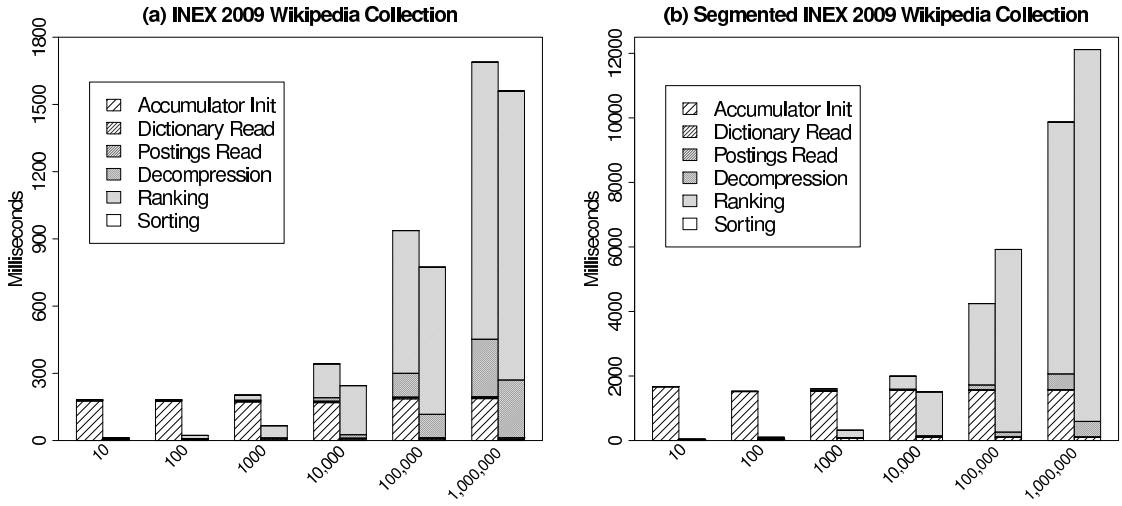

Figure 4: Comparison between the static array and logical two dimensional table structures for accumulator initialisation. The first bar in each group shows the results for the static array structure while the second shows the results for the logical two dimensional table.

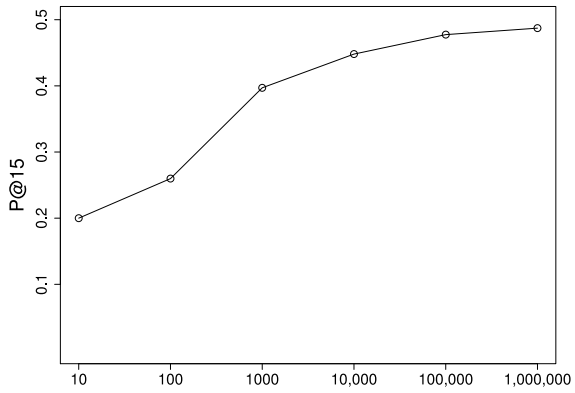

Figure 5: Precision for P@15 using assessments from the INEX 2009 Efficiency Track [19].

are processed. The performance increases for 100, 1000, 10 000, 100 000 are 86%, 67%, 28% and 17% respectively. However, there is only a 2% performance increase when 1 000 000 postings are processed.

The effectiveness of the *heapk* pruning algorithm is shown in Figure 5 using precision at 15 for various values of postings. The highest precision is 0.487 when 1 000 000 were processed. There were small precison drops 2% and 8% when 100 000 and 10 000 postings were processed respectively. The precisions were dramaticly dropped 18%, 47% and 59% when 1000, 100, 10 postings were processed. Overall, it shows that postings pruning is very effective.

We also tested the same set of experiments on a larger document collection provided by the University of Queensland. The collection has 23 million documents and is created by splitting each section of the documents as a single document in the INEX 2009 Wikipedia collection [18]. Since there is no evaluation for this collection, we cannot show precision.

Figure 4(b) shows the results. It took about 1700 milliseconds for initialising the static accumulator array, compared with zero time for the logical two dimensional table. There was a overall performance increase of 97%, 93%, 80% and 25% when 10, 100, 1000, 10 000 postings were processed respectively. However, there was a overhead of 40%, 23% for processing 100 000, 1 000 000 postings respectively.

Compared with the original Wikipedia collection, the processing of 100 000 and 1 000 000 postings caused more overhead for the logical two dimension table in the segmented collection. In the original collection, few terms have more than 1 000 000 postings. However, when documents are segmented by sections, more lists are closer to or longer than 100 000 and 1 000 000.

5 Conclusion and Future Work

In this paper we have proposed an efficient accumulator initialisation algorithm, especially when used together with postings pruning. The algorithm represents the traditional static accumulator array as a logical two dimensional table and uses an array of flags to keep track of the initialisation status of the accumulators. Before processing queries, the array of flags is initialised instead of initialising the accumulator array. During the update of accumulators, a hashing function (shift) is used to locate the accumulator.

Using two dimensional structure is not new. It has been used in other areas of Computer Science, including paging and file systems [7]. In paging and file systems, multiple dimensions (multiple level of indexing) are required in order to address large amount of memory or very big files. We will explore the use of multiple dimension structures for efficient addressing large doc-

ument collections, like the ClueWeb 2009 collection (1 billion documents) in future work.

We have explained how we have integrated the logical two dimensional table into our *heapk* pruning algorithm. In future work, we will examine the efficiency of the structure in other pruning algorithms. One example is the any-time stopping pruning algorithm [1]. It uses an array of lists (indexed by quantised impact values) to keep track of the current top candidates stored in the static accumulator array. The logical two dimensional table can be integrated into this pruning algorithm without affecting the original algorithm.

For both of our simulation and search engine, we defined the width to be a power of 2 and used a shift hashing function for the logical two dimensional table. A shift hashing function is considerably faster a division. However, we have not explored how far away from the optimum ($B = \sqrt{\frac{c_1 D}{c_2 LQ}}$) this is. If the gap between the optimum and a power of 2 is very large, which means a lot un-necessary accumulators are initialised, it might be more efficient to define the width as the optimum and use division for hashing.

References

[1] Vo Ngoc Anh, Owen de Kretser and Alistair Moffat. Vector-space ranking with effective early termination. pages 35–42, 2001.

[2] Vo Ngoc Anh and Alistair Moffat. Compressed inverted files with reduced decoding overheads. pages 290–297, 1998.

[3] Vo Ngoc Anh and Alistair Moffat. Random access compressed inverted files. *Australian Computer Science Comm.: Proc. 9th Australasian Database Conf., ADC*, Volume 20, Number 2, pages 1–12, February 1998.

[4] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, Volume 8, Number 1, pages 151–166, 2005.

[5] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 183–190, New York, NY, USA, 2007. ACM.

[6] Jon L. Bentley and M. Douglas Mcilroy. Engineering a sort function, 1993.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, November 2005.

[8] Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. pages 97–110, 1985.

[9] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithm*. The MIT Press, 1990.

[10] Donna Harman and Gerald Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, Volume 41, pages 581–589, 1990.

[11] Xiang-fei Jia, Andrew Trotman, Richard O'Keefe and Zhiyi Huang. Application-specific disk I/O optimisation for a search engine. In *PDCAT '08: Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 399–404, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, Volume 8, Number 1, pages 3–30, 1998.

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

[14] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, Volume 14, Number 4, pages 349–379, 1996.

[15] Alistair Moffat, Justin Zobel and Ron Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, Volume 30, Number 6, pages 733–744, 1994.

[16] Michael Persin. Document filtering for fast ranking. pages 339–348, 1994.

[17] Michael Persin, Justin Zobel and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, Volume 47, Number 10, pages 749–764, 1996.

[18] Ralf Schenkel, Fabian Suchanek and Gjergji Kasneci. YAWN: A semantically annotated wikipedia xml corpus. March 2007.

[19] Ralf Schenkel and Martin Theobald. Overview of the inex 2009 efficiency track. In Shlomo Geva, Jaap Kamps and Andrew Trotman (editors), *Focused Retrieval and Evaluation*, Volume 6203 of *Lecture Notes in Computer Science*, pages 200–212. Springer Berlin / Heidelberg, 2010.

[20] Martin Theobald, Ralf Schenkel and Gerhard Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. pages 242–249, 2005.

[21] Andrew Trotman. Compressing inverted files. *Inf. Retr.*, Volume 6, Number 1, pages 5–19, 2003.

[22] Andrew Trotman, Xiang-Fei Jia and Shlomo Geva. Fast and effective focused retrieval. In Shlomo Geva, Jaap Kamps and Andrew Trotman (editors), *Focused Retrieval and Evaluation*, Volume 6203 of *Lecture Notes in Computer Science*, pages 229–241. Springer Berlin / Heidelberg, 2010.

[23] Yohannes Tsegay, Andrew Turpin and Justin Zobel. Dynamic index pruning for effective caching. pages 987–990, 2007.

[24] Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, Volume 31, Number 6, pages 831 – 850, 1995.