Fast Generation of Result Snippets in Web Search

Andrew Turpin &
Yohannes Tsegay
RMIT University
Melbourne, Australia
aht@cs.rmit.edu.au
ytsegay@cs.rmit.edu.au

David Hawking
CSIRO ICT Centre
Canberra, Australia
david.hawking@acm.org

Hugh E. Williams
Microsoft Corporation
One Microsoft Way
Redmond, WA.
hughw@microsoft.com

ABSTRACT

The presentation of query biased document snippets as part of results pages presented by search engines has become an expectation of search engine users. In this paper we explore the algorithms and data structures required as part of a search engine to allow efficient generation of query biased snippets. We begin by proposing and analysing a document compression method that reduces snippet generation time by 58% over a baseline using the *zlib* compression library. These experiments reveal that finding documents on secondary storage dominates the total cost of generating snippets, and so caching documents in RAM is essential for a fast snippet generation process. Using simulation, we examine snippet generation performance for different size RAM caches. Finally we propose and analyse document reordering and compaction, revealing a scheme that increases the number of document cache hits with only a marginal affect on snippet quality. This scheme effectively doubles the number of documents that can fit in a fixed size cache.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*performance evaluation (efficiency and effectiveness)*;

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

Web summaries, snippet generation, document caching

1. INTRODUCTION

Each result in search results list delivered by current WWW search engines such as `search.yahoo.com`, `google.com` and `search.msn.com` typically contains the title and URL of the actual document, links to live and cached versions of the document and sometimes an indication of file size and type. In addition, one or more *snippets* are usually presented, giving the searcher a sneak preview of the document contents.

Snippets are short fragments of text extracted from the document content (or its metadata). They may be static (for example, always show the first 50 words of the document, or the content of its description metadata, or a description taken from a directory site such as `dmoz.org`) or *query-biased* [20]. A query-biased snippet is one selectively extracted on the basis of its relation to the searcher's query.

The addition of informative snippets to search results may substantially increase their value to searchers. Accurate snippets allow the searcher to make good decisions about which results are worth accessing and which can be ignored. In the best case, snippets may obviate the need to open any documents by directly providing the answer to the searcher's real information need, such as the contact details of a person or an organization.

Generation of query-biased snippets by Web search engines indexing of the order of ten billion web pages and handling hundreds of millions of search queries per day imposes a very significant computational load (remembering that each search typically generates ten snippets). The simple-minded approach of keeping a copy of each document in a file and generating snippets by opening and scanning files, works when query rates are low and collections are small, but does not scale to the degree required. The overhead of opening and reading ten files per query on top of accessing the index structure to locate them, would be manifestly excessive under heavy query load. Even storing ten billion files and the corresponding hundreds of terabytes of data is beyond the reach of traditional filesystems. Special-purpose filesystems have been built to address these problems [6].

Note that the utility of snippets is by no means restricted to whole-of-Web search applications. Efficient generation of snippets is also important at the scale of whole-of-government search services such as `www.firstgov.gov` (c. 25 million pages) and `govsearch.australia.gov.au` (c. 5 million pages) and within large enterprises such as IBM [2] (c. 50 million pages). Snippets may be even more useful in database or filesystem search applications in which no useful URL or title information is present.

We present a new algorithm and compact single-file structure designed for rapid generation of high quality snippets and compare its space/time performance against an obvious baseline based on the *zlib* compressor on various data sets. We report the proportion of time spent for disk seeks, disk reads and cpu processing; demonstrating that the time for locating each document (seek time) dominates, as expected.

SIGIR'07, July 23–27, 2007, Amsterdam, The Netherlands.
Copyright 2007 ACM 978-1-59593-597-7/07/0007 ...$5.00.

As the time to process a document in RAM is small in comparison to locating and reading the document into memory, it may seem that compression is not required. However, this is only true if there is no caching of documents in RAM. Controlling the RAM of physical systems for experimentation is difficult, hence we use simulation to show that caching documents dramatically improves the performance of snippet generation. In turn, the more documents can be compressed, the more can fit in cache, and hence the more disk seeks can be avoided: the classic data compression tradeoff that is exploited in inverted file structures and computing ranked document lists [24].

As hitting the document cache is important, we examine document *compaction*, as opposed to compression, schemes by imposing an *a priori* ordering of sentences within a document, and then only allowing leading sentences into cache for each document. This leads to further time savings, with only marginal impact on the quality of the snippets returned.

2. RELATED WORK

Snippet generation is a special type of extractive document summarization, in which sentences, or sentence fragments, are selected for inclusion in the summary on the basis of the degree to which they match the search query. This process was given the name of *query-biased summarization* by Tombros and Sanderson [20] The reader is referred to Mani [13] and to Radev et al. [16] for overviews of the very many different applications of summarization and for the equally diverse methods for producing summaries.

Early Web search engines presented query-independent snippets consisting of the first k bytes of the result document. Generating these is clearly much simpler and much less computationally expensive than processing documents to extract query biased summaries, as there is no need to search the document for text fragments containing query terms. To our knowledge, Google was the first whole-of-Web search engine to provide query biased summaries, but summarization is listed by Brin and Page [1] only under the heading of future work.

Most of the experimental work using query-biased summarization has focused on comparing their value to searchers relative to other types of summary [20, 21], rather than efficient generation of summaries. Despite the importance of efficient summary generation in Web search, few algorithms appear in the literature. Silber and McKoy [19] describe a linear-time lexical chaining algorithm for use in generic summaries, but offer no empirical data for the performance of their algorithm. White et al [21] report some experimental timings of their WebDocSum system, but the snippet generation algorithms themselves are not isolated, so it is difficult to infer snippet generation time comparable to the times we report in this paper.

3. SEARCH ENGINE ARCHITECTURES

A search engine must perform a variety of activities, and is comprised of many sub-systems, as depicted by the boxes in Figure 1. Note that there may be several other sub-systems such as the "Advertising Engine" or the "Parsing Engine" that could easily be added to the diagram, but we have concentrated on the sub-systems that are relevant to snippet generation. Depending on the number of documents that the search engine indexes, the data and processes for each sub-system could be distributed over many machines, or all occupy a single server and filesystem, competing with each other for resources. Similarly, it may be more efficient to combine some sub-systems in an implementation of the diagram. For example, the meta-data such as document title and URL requires minimal computation apart from highlighting query words, but we note that disk seeking is likely to be minimized if title, URL and fixed summary information is stored contiguously with the text from which query biased summaries are extracted. Here we ignore the fixed text and consider only the generation of query biased summaries: we concentrate on the "Snippet Engine".

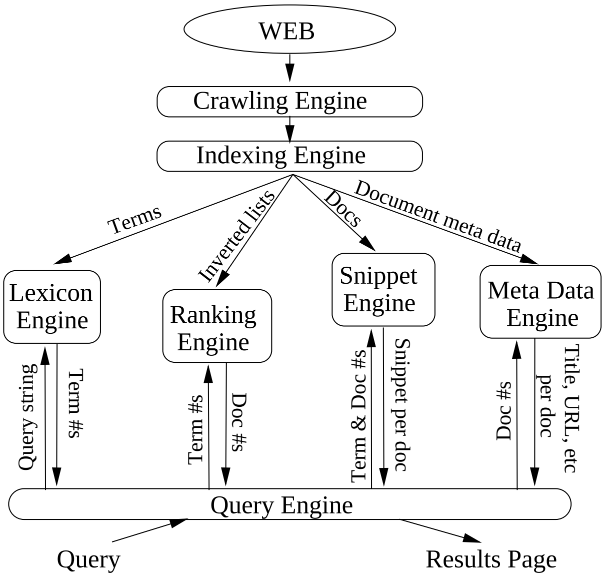

Figure 1: An abstraction of some of the sub-systems in a search engine. Depending on the number of documents indexed, each sub-system could reside on a single machine, be distributed across thousands of machines, or a combination of both.

In addition to data and programs operating on that data, each sub-system also has its own memory management scheme. The memory management system may simply be the memory hierarchy provided by the operating system used on machines in the sub-system, or it may be explicitly coded to optimise the processes in the sub-system.

There are many papers on caching in search engines (see [3] and references therein for a current summary), but it seems reasonable that there is a *query cache* in the Query Engine that stores precomputed final result pages for very popular queries. When one of the popular queries is issued, the result page is fetched straight from the query cache. If the issued query is not in the query cache, then the Query Engine uses the four sub-systems in turn to assemble a results page.

1. The Lexicon Engine maps query terms to integers.
2. The Ranking Engine retrieves inverted lists for each term, using them to get a ranked list of documents.
3. The Snippet Engine uses those document numbers and query term numbers to generate snippets.
4. The Meta Data Engine fetches other information about each document to construct the results page.

```
IN   A document broken into one sentence per line,
     and a sequence of query terms.
1    For each line of the text, $\mathcal{L} = [w_1, w_2, \ldots, w_m]$
2        Let $h$ be 1 if $\mathcal{L}$ is a heading, 0 otherwise.
3        Let $\ell$ be 2 if $\mathcal{L}$ is the first line of a document,
         1 if it is the second line, 0 otherwise.
4        Let $c$ be the number of $w_i$ that are query
         terms, counting repetitions.
5        Let $d$ be the number of distinct query terms
         that match some $w_i$.
6        Identify the longest contiguous run of query
         terms in $\mathcal{L}$, say $w_j \ldots w_{j+k}$.
7        Use a weighted combination of $c$, $d$, $k$, $h$
         and $\ell$ to derive a score $s$.
8        Insert $\mathcal{L}$ into a max-heap using $s$ as the key.
OUT  Remove the number of sentences required from
     the heap to form the summary.
```

Figure 2: Simple sentence ranker that operates on raw text with one sentence per line.

4. THE SNIPPET ENGINE

For each document identifier passed to the Snippet Engine, the engine must generate text, preferably containing query terms, that attempts to summarize that document. Previous work on summarization identifies the sentence as the minimal unit for extraction and presentation to the user [12]. Accordingly, we also assume a web snippet extraction process will extract sentences from documents. In order to construct a snippet, all sentences in a document should be ranked against the query, and the top two or three returned as the snippet. The scoring of sentences against queries has been explored in several papers [7, 12, 18, 20, 21], with different features of sentences deemed important.

Based on these observations, Figure 2, shows the general algorithm for scoring sentences in relevant documents, with the highest scoring sentences making the snippet for each document. The final score of a sentence, assigned in Step 7, can be derived in many different ways. In order to avoid bias towards any particular scoring mechanism, we compare sentence quality later in the paper using the individual components of the score, rather than an arbitrary combination of the components.

4.1 Parsing Web Documents

Unlike well edited text collections that are often the target for summarization systems, Web data is often poorly structured, poorly punctuated, and contains a lot of data that do not form part of valid sentences that would be candidates for parts of snippets.

We assume that the documents passed to the Snippet Engine by the Indexing Engine have all HTML tags and JavaScript removed, and that each document is reduced to a series of word tokens separated by non-word tokens. We define a word token as a sequence of alphanumeric characters, while a non-word is a sequence of non-alphanumeric characters such as whitespace and the other punctuation symbols. Both are limited to a maximum of 50 characters. Adjacent, repeating characters are removed from the punctuation.

Included in the punctuation set is a special end of sentence marker which replaces the usual three sentence terminators "?!.". Often these explicit punctuation characters are missing, and so HTML tags such as <br> and <p> are assumed to terminate sentences. In addition, a sentence must contain at least five words and no more than twenty words, with longer or shorter sentences being broken and joined as required to meet these criteria [10].

Unterminated HTML tags—that is, tags with an open brace, but no close brace—cause all text from the open brace to the next open brace to be discarded.

A major problem in summarizing web pages is the presence of large amounts of promotional and navigational material ("navbars") visually above and to the left of the actual page content. For example, "The most wonderful company on earth. Products. Service. About us. Contact us. Try before you buy." Similar, but often not identical, navigational material is typically presented on every page within a site. This material tends to lower the quality of summaries and slow down summary generation.

In our experiments we did not use any particular heuristics for removing navigational information as the test collection in use (WT100G) pre-dates the widespread take up of the current style of web publishing. In WT100G, the average web page size is more than half the current Web average [11]. Anecdotally, the increase is due to inclusion of sophisticated navigational and interface elements and the JavaScript functions to support them.

Having defined the format of documents that are presented to the Snippet Engine, we now define our Compressed Token System (CTS) document storage scheme, and the baseline system used for comparison.

4.2 Baseline Snippet Engine

An obvious document representation scheme is to simply compress each document with a well known adaptive compressor, and then decompress the document as required [1], using a string matching algorithm to effect the algorithm in Figure 2. Accordingly, we implemented such a system, using *zlib* [4] with default parameters to compress every document after it has been parsed as in Section 4.1.

Each document is stored in a single file. While manageable for our small test collections or small enterprises with millions of documents, a full Web search engine may require multiple documents to inhabit single files, or a special purpose filesystem [6].

For snippet generation, the required documents are decompressed one at a time, and a linear search for provided query terms is employed. The search is optimized for our specific task, which is restricted to matching whole words and the sentence terminating token, rather than general pattern matching.

4.3 The CTS Snippet Engine

Several optimizations over the baseline are possible. The first is to employ a semi-static compression method over the entire document collection, which will allow faster decompression with minimal compression loss [24]. Using a semi-static approach involves mapping words and non-words produced by the parser to single integer tokens, with frequent symbols receiving small integers, and then choosing a coding scheme that assigns small numbers a small number of bits. Words and non-words strictly alternate in the compressed file, which always begins with a word.

In this instance we simply assign each symbol its ordinal number in a list of symbols sorted by frequency. We use the vbyte coding scheme to code the word tokens [22]. The set of non-words is limited to the 64 most common punctuation sequences in the collection itself, and are encoded with a flat 6-bit binary code. The remaining 2 bits of each punctuation symbol are used to store capitalization information.

The process of computing the semi-static model is complicated by the fact that the number of words and non-words appearing in large web collections is high. If we stored all words and non-words appearing in the collection, and their associated frequency, many gigabytes of RAM or a B-tree or similar on-disk structure would be required [23]. Moffat et al. [14] have examined schemes for pruning models during compression using large alphabets, and conclude that rarely occurring terms need not reside in the model. Rather, rare terms are spelt out in the final compressed file, using a special word token (ESCAPE symbol), to signal their occurrence.

During the first pass of encoding, two move-to-front queues are kept; one for words and one for non-words. Whenever the available memory is consumed and a new symbol is discovered in the collection, an existing symbol is discarded from the end of the queue. In our implementation, we enforce the stricter condition on eviction that, where possible, the evicted symbol should have a frequency of one. If there is no symbol with frequency one in the last half of the queue, then we evict symbols of frequency two, and so on until enough space is available in the model for the new symbol.

The second pass of encoding replaces each word with its vbyte encoded number, or the ESCAPE symbol and an ASCII representation of the word if it is not in the model. Similarly each non-word sequence is replaced with its codeword, or the codeword for a single space character if it is not in the model. We note that this lossless compression of non-words is acceptable when the documents are used for snippet generation, but may not be acceptable for a document database. We assume that a separate sub-system would hold cached documents for other purposes where exact punctuation is important.

While this semi-static scheme should allow faster decompression than the baseline, it also readily allows direct matching of query terms as compressed integers in the compressed file. That is, sentences can be scored without having to decompress a document, and only the sentences returned as part of a snippet need to be decoded.

The CTS system stores all documents contiguously in one file, and an auxiliary table of 64 bit integers indicating the start offset of each document in the file. Further, it must have access to the reverse mapping of term numbers, allowing those words not spelt out in the document to be recovered and returned to the Query Engine as strings. The first of these data structures can be readily partitioned and distributed if the Snippet Engine occupies multiple machines; the second, however, is not so easily partitioned, as any document on a remote machine might require access to the whole integer-to-string mapping. This is the second reason for employing the model pruning step during construction of the semi-static code: it limits the size of the reverse mapping table that should be present on every machine implementing the Snippet Engine.

4.4 Experimental assessment of CTS

All experiments reported in this paper were run on a Sun Fire V210 Server running Solaris 10. The machine consists of dual 1.34 GHz UltraSPARC IIIi processors and 4GB of RAM. All source code was compiled using gcc4.1.1 with -O9 optimisation. Timings were run on an otherwise unoccupied machine and were averaged over 10 runs, with memory flushed between runs to eliminate any caching of data files.

	WT10G	WT50G	WT100G
No. Docs. ($\times 10^6$)	1.7	10.1	18.5
Raw Text	10,522	56,684	102,833
Baseline(*zlib*)	2,568 (24%)	10,940 (19%)	19,252 (19%)
CTS	2,722 (26%)	12,010 (21%)	22,269 (22%)

Table 1: Total storage space (Mb) for documents for the three test collections both compressed, and uncompressed.

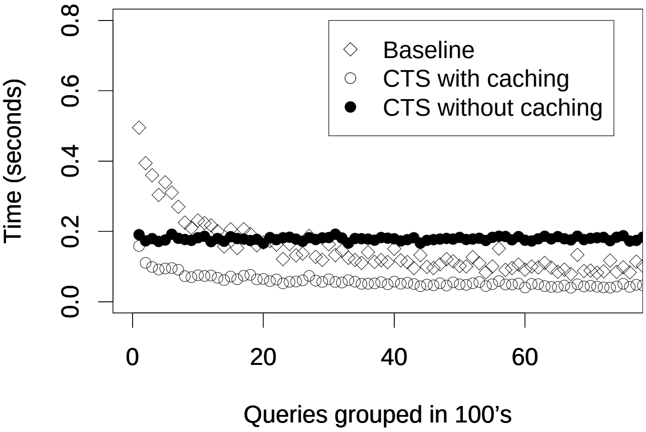

Figure 3: Time to generate snippets for 10 documents per query, averaged over buckets of 100 queries, for the first 7000 Excite queries on WT10G.

In the absence of evidence to the contrary, we assume that it is important to model realistic query arrival sequences and the distribution of query repetitions for our experiments. Consequently, test collections which lack real query logs, such as TREC Ad Hoc and .GOV2 were not considered suitable. Obtaining extensive query logs and associated result doc-ids for a corresponding large collection is not easy. We have used two collections (WT10G and WT100G) from the TREC Web Track [8] coupled with queries from Excite logs from the same (c. 1997) period. Further, we also made use of a medium sized collection WT50G, obtained by randomly sampling half of the documents from WT100G. The first two rows of Table 1 give the number of documents and the size in Mb of these collections.

The final two rows of Table 1 show the size of the resulting document sets after compression with the baseline and CTS schemes. As expected, CTS admits a small compression loss over *zlib*, but both substantially reduce the size of the text to about 20% of the original, uncompressed size. Note that the figures for CTS do not include the reverse mapping from integer token to string that is required to produce the final snippets as that occupies RAM. It is 1024 Mb in these experiments.

The Zettair search engine [25] was used to produce a list of documents to summarize for each query. For the majority of the experiments the Okapi BM25 scoring scheme was used to determine document rankings. For the static caching experiments reported in Section 5, the score of each document

	WT10G	WT50G	WT100G
Baseline	75	157	183
CTS	38	70	77
Reduction in time	49%	56%	58%

Table 2: Average time (msec) for the final 7000 queries in the Excite logs using the baseline and CTS systems on the 3 test collections.

is a 50:50 weighted average of the BM25 score (normalized by the top scoring document for each query) and a score for each document independent of any query. This is to simulate effects of ranking algorithms like PageRank [1] on the distribution of document requests to the Snippet Engine. In our case we used the normalized Access Count [5] computed from the top 20 documents returned to the first 1 million queries from the Excite log to determine the query independent score component.

Points on Figure 3 indicate the mean running time to generate 10 snippets for each query, averaged in groups of 100 queries, for the first 7000 queries in the Excite query log. Only the data for WT10G is shown, but the other collections showed similar patterns. The x-axis indicates the group of 100 queries; for example, 20 indicates the queries 2001 to 2100. Clearly there is a caching effect, with times dropping substantially after the first 1000 or so queries are processed. All of this is due to the operating system caching disk blocks and perhaps pre-fetching data ahead of specific read requests. This is evident because the baseline system has no large internal data structures to take advantage of non-disk based caching, it simply opens and processes files, and the speed up is evident for the baseline system.

Part of this gain is due to the spatial locality of disk references generated by the query stream: repeated queries will already have their document files cached in memory; and similarly different queries that return the same documents will benefit from document caching. But when the log is processed after removing all but the first request for each document, the pronounced speed-up is still evident as more queries are processed (not shown in figure). This suggests that the operating system (or the disk itself) is reading and buffering a larger amount of data than the amount requested and that this brings benefit often enough to make an appreciable difference in snippet generation times. This is confirmed by the curve labeled "CTS without caching", which was generated after mounting the filesystem with a "no-caching" option (directio in Solaris). With disk caching turned off, the average time to generate snippets varies little.

The time to generate ten snippets for a query, averaged over the final 7000 queries in the Excite log as caching effects have dissipated, are shown in Table 2. Once the system has stabilized, CTS is over 50% faster than the Baseline system. This is primarily due to CTS matching single integers for most query words, rather than comparing strings in the baseline system.

Table 3 shows a break down of the average time to generate ten snippets over the final 7000 queries of the Excite log on the WT50G collection when entire documents are processed, and when only the first half of each document is processed. As can be seen, the majority of time spent generating a snippet is in locating the document on disk ("Seek"): 64% for whole documents, and 75% for half documents. Even if the amount of processing a document must undergo is halved, as in the second row of the Table, there is only a 14% reduction in the total time required to generate a snippet. As locating documents in secondary storage occupies such a large proportion of snippet generation time, it seems logical to try and reduce its impact through caching.

% of doc processed	Seek	Read	Score & Decode
100%	45	4	21
50%	45	4	11

Table 3: Time to generate 10 snippets for a single query (msec) for the WT50G collection averaged over the final 7000 Excite queries when either all of each document is processed (100%) or just the first half of each document (50%).

5. DOCUMENT CACHING

In Section 3 we observed that the Snippet Engine would have its own RAM in proportion to the size of the document collection. For example, on a whole-of-Web search engine, the Snippet Engine would be distributed over many workstations, each with at least 4 Gb of RAM. In a small enterprise, the Snippet Engine may be sharing RAM with all other sub-systems on a single workstation, hence only have 100 Mb available. In this section we use simulation to measure the number of cache hits in the Snippet Engine as memory size varies.

We compare two caching policies: a static cache, where the cache is loaded with as many documents as it can hold before the system begins answering queries, and then never changes; and a least-recently-used cache, which starts out as for the static cache, but whenever a document is accessed it moves to the front of a queue, and if a document is fetched from disk, the last item in the queue is evicted. Note that documents are first loaded into the caches in order of decreasing query independent score, which is computed as described in Section 4.4.

The simulations also assume a query cache exists for the top Q most frequent queries, and that these queries are never processed by the Snippet Engine.

All queries passed into the simulations are from the second half of the Excite query log (the first half being used to compute query independent scores), and are stemmed, stopped, and have their terms sorted alphabetically. This final alteration simply allows queries such as "red dog" and "dog red" to return the same documents, as would be the case in a search engine where explicit phrase operators would be required in the query to enforce term order and proximity.

Figure 4 shows the percentage of document access that hit cache using the two caching schemes, with Q either 0 or 10,000, on 535,276 Excite queries on WT100G. The x-axis shows the percentage of documents that are held in the cache, so 1.0% corresponds to about 185,000 documents. From this figure it is clear that caching even a small percentage of the documents has a large impact on reducing seek time for snippet generation. With 1% of documents cached, about 222 Mb for the WT100G collection, around 80% of disk seeks are avoided. The static cache performs surprisingly well (squares in Figure 4), but is outperformed by the LRU cache (circles). In an actual implementation of LRU, however, there may be fragmentation of the cache as documents are swapped in and out.

The reason for the large impact of the document cache is

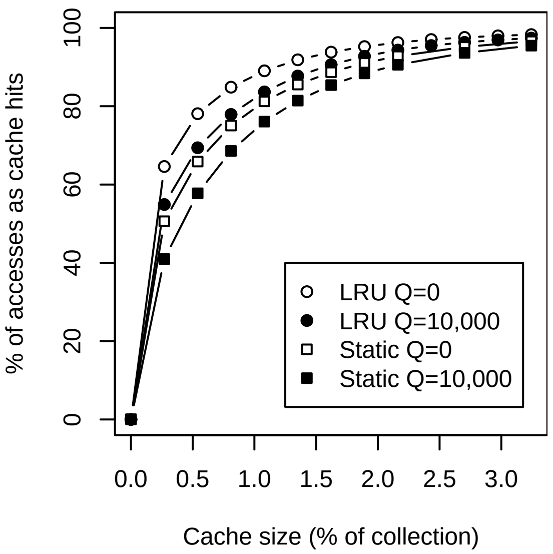

Figure 4: Percentage of the time that the Snippet Engine does not have to go to disk in order to generate a snippet plotted against the size of the document cache as a percentage of all documents in the collection. Results are from a simulation on WT100G with 535,276 Excite queries.

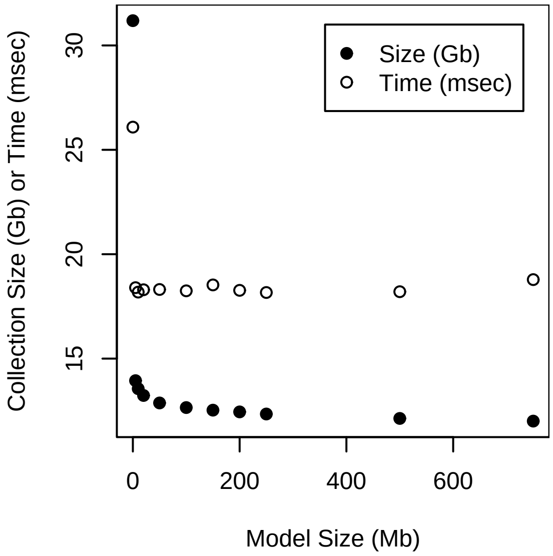

Figure 5: Collection size of the WT50G collection when compressed with CTS using different memory limits on the model, and the average time to generate single snippet excluding seek time on 20000 Excite queries using those models.

that, for a particular collection, some documents are much more likely to appear in results lists than others. This effect occurs partly because of the approximately Zipfian query frequency distribution, and partly because most Web search engines employ ranking methods which combine query based scores with static (*a priori*) scores determined from factors such as link graph measures, URL features, spam scores and so on [17]. Documents with high static scores are much more likely to be retrieved than others.

In addition to the document cache, the RAM of the Snippet Engine must also hold the CTS decoding table that maps integers to strings, which is capped by a parameter at compression time (1 Gb in our experiments here). This is more than compensated for by the reduced size of each document, allowing more documents into the document cache. For example, using CTS reduces the average document size from 5.7 Kb to 1.2 Kb (as shown in Table 1), so a 2 Gb RAM could hold 368,442 uncompressed documents (2% of the collection), or 850,691 documents plus a 1 Gb decompression table (5% of the collection).

In fact, further experimentation with the model size reveals that the model can in fact be very small and still CTS gives good compression and fast scoring times. This is evidenced in Figure 5, where the compressed size of WT50G is shown in the solid symbols. Note that when no compression is used (Model Size is 0Mb), the collection is only 31 Gb as HTML markup, JavaScript, and repeated punctuation has been discarded as described in Section 4.1. With a 5 Mb model, the collection size drops by more than half to 14 Gb, and increasing the model size to 750 Mb only elicits a 2 Gb drop in the collection size. Figure 5 also shows the average time to score and decode a a snippet (excluding seek time) with the different model sizes (open symbols). Again, there is a large speed up when a 5 Mb model is used, but little improvement with larger models. Similar results hold for the WT100G collection, where a model of about 10 Mb offers substantial space and time savings over no model at all, but returns diminish as the model size increases.

Apart from compression, there is another approach to reducing the size of each document in the cache: do not store the full document in cache. Rather store sentences that are likely to be used in snippets in the cache, and if during snippet generation on a cached document the sentence scores do not reach a certain threshold, then retrieve the whole document from disk. This raises questions on how to choose sentences from documents to put in cache, and which to leave on disk, which we address in the next section.

6. SENTENCE REORDERING

Sentences within each document can be re-ordered so that sentences that are very likely to appear in snippets are at the front of the document, hence processed first at query time, while less likely sentences are relegated to the rear of the document. Then, during query time, if k sentences with a score exceeding some threshold are found before the entire document is processed, the remainder of the document is ignored. Further, to improve caching, only the head of each document can be stored in the cache, with the tail residing on disk. Note that we assume that the search engine is to provide “cached copies” of a document–that is, the exact text of the document as it was indexed–then this would be serviced by another sub-system in Figure 1, and not from the altered copy we store in the Snippet Engine.

We now introduce four sentence reordering approaches.

1. Natural order The first few sentences of a well authored document usually best describe the document content [12]. Thus simply processing a document in order should yield a quality snippet. Unfortunately, however, web documents are often not well authored, with little editorial or professional writing skills brought to bear on the creation of a work of literary merit. More importantly, perhaps, is that we are producing query-biased snippets, and there is no guarantee that query terms will appear in sentences towards the front of a document.

2. Significant terms (ST) Luhn introduced the concept of a significant sentence as containing a cluster of significant terms [12], a concept found to work well by Tombros and Sanderson [20]. Let $f_{d,t}$ be the frequency of term t in document d, then term t is determined to be significant if

$$f_{d,t} \geq \begin{cases} 7 - 0.1 \times (25 - s_d), & \text{if } s_d < 25 \\ 7, & \text{if } 25 \leq s_d \leq 40 \\ 7 + 0.1 \times (s_d - 40), & \text{otherwise,} \end{cases}$$

where s_d is the number of sentences in document d. A *bracketed section* is defined as a group of terms where the leftmost and rightmost terms are significant terms, and no significant terms in the bracketed section are divided by more than four non-significant terms. The score of a bracketed section is the square of the number of significant words falling in the section, divided by the total number of words in the entire sentence. The a priori score for a sentence is computed as the maximum of all scores for the bracketed sections of the sentence. We then sort the sentences by this score.

3. Query log based (QLt) Many Web queries repeat, and a small number of queries make up a large volume of total searches [9]. In order to take advantage of this bias, sentences that contain many past query terms should be promoted to the front of a document, while sentences that contain few query terms should be demoted. In this scheme, the sentences are sorted by the number of sentence terms that occur in the query log. To ensure that long sentences do not dominate over shorter qualitative sentences the score assigned to each sentence is divided by the number of terms in that sentence giving each sentence a score between 0 and 1.

4. Query log based (QLu) This scheme is as for QLt, but repeated terms in the sentence are only counted once.

By re-ordering sentences using schemes ST, QLt or QLu, we aim to terminate snippet generation earlier than if Natural Order is used, but still produce sentences with the same number of unique query terms (d in Figure 2), total number of query terms (c), the same positional score ($h + \ell$) and the same maximum span (k). Accordingly, we conducted experiments comparing the methods, the first 80% of the Excite query log was used to reorder sentences when required, and the final 20% for testing.

Figure 6 shows the differences in snippet scoring components using each of the three methods over the Natural Order method. It is clear that sorting sentences using the Significant Terms (ST) method leads to the smallest change in the sentence scoring components. The greatest change over all methods is in the sentence position ($h + \ell$) component of the score, which is to be expected as their is no guarantee that leading and heading sentences are processed at all after sentences are re-ordered. The second most affected component is the number of distinct query terms in a returned sentence, but if only the first 50% of the document is processed with the ST method, there is a drop of only 8% in the number of distinct query terms found in snippets.

Depending how these various components are weighted to compute an overall snippet score, one can argue that there is little overall affect on scores when processing only half the document using the ST method.

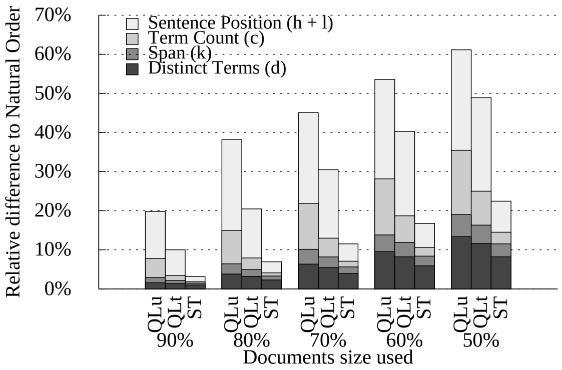

Figure 6: Relative difference in the snippet score components compared to Natural Ordered documents when the amount of documents processed is reduced, and the sentences in the document are reordered using Query Logs (QLt, QLu) or Significant Terms (ST).

7. DISCUSSION

In this paper we have described the algorithms and compression scheme that would make a good Snippet Engine sub-system for generating text snippets of the type shown on the results pages of well known Web search engines. Our experiments not only show that our scheme is over 50% faster than the obvious baseline, but also reveal some very important aspects of the snippet generation problem. Primarily, caching documents avoids seek costs to secondary memory for each document that is to be summarized, and is vital for fast snippet generation. Our caching simulations show that if as little as 1% of the documents can be cached in RAM as part of the Snippet Engine, possibly distributed over many machines, then around 75% of seeks can be avoided. Our second major result is that keeping only half of each document in RAM, effectively doubling the cache size, has little affect on the quality of the final snippets generated from those half-documents, provided that the sentences that are kept in memory are chosen using the Significant Term algorithm of Luhn [12]. Both our document compression and compaction schemes dramatically reduce the time taken to generate snippets.

Note that these results are generated using a 100Gb subset of the Web, and the Excite query log gathered from the same period as that subset was created. We are assuming, as there is no evidence to the contrary, that this collection and log is representative of search engine input in other domains. In particular, we can scale our results to examine what resources would be required, using our scheme, to provide a Snippet Engine for the entire World Wide Web.

We will assume that the Snippet Engine is distributed across M machines, and that there are N web pages in the collection to be indexed and served by the search engine. We also assume a balanced load for each machine, so each machine serves about N/M documents, which is easily achieved in practice. Each machine, therefore, requires RAM to hold the following.

- The CTS model, which should be 1/1000 of the size of the uncompressed collection (using results in Figure 5 and Williams et al. [23]). Assuming an average uncompressed document size of 8 Kb [11], this would require $N/M \times 8.192$ bytes of memory.

- A cache of 1% of all N/M documents. Each document requires 2 Kb when compressed with CTS (Table 1), and only half of each document is required using ST sentence reordering, requiring a total of $N/M \times 0.01 \times 1024$ bytes.
- The offset array that gives the start position of each document in the single, compressed file: 8 bytes per N/M documents.

The total amount of RAM required by a single machine, therefore, would be $N/M(8.192 + 10.24 + 8)$ bytes. Assuming that each machine has 8 Gb of RAM, and that there are 20 billion pages to index on the Web, a total of $M = 62$ machines would be required for the Snippet Engine. Of course in practice, more machines may be required to manage the distributed system, to provide backup services for failed machines, and other networking services. These machines would also need access to 37 Tb of disk to store the compressed document representations that were not in cache.

In this work we have deliberately avoided committing to one particular scoring method for sentences in documents. Rather, we have reported accuracy results in terms of the four components that have been previously shown to be important in determining useful snippets [20]. The CTS method can incorporate any new metrics that may arise in the future that are calculated on whole words. The document compaction techniques using sentence re-ordering, however, remove the spatial relationship between sentences, and so if a scoring technique relies on the position of a sentence within a document, the aggressive compaction techniques reported here cannot be used.

A variation on the semi-static compression approach we have adopted in this work has been used successfully in previous search engine design [24], but there are alternate compression schemes that allow direct matching in compressed text (see Navarro and Mäkinen [15] for a recent survey.) As seek time dominates the snippet generation process, we have not focused on this portion of the snippet generation in detail in this paper. We will explore alternate compression schemes in future work.

Acknowledgments

This work was supported in part by ARC Discovery Project DP0558916 (AT). Thanks to Nick Lester and Justin Zobel for valuable discussions.

8. REFERENCES

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW7*, pages 107–117, 1998.

[2] R. Fagin, Ravi K., K. S. McCurley, J. Novak, D. Sivakumar, J. A. Tomlin, and D. P. Williamson. Searching the workplace web. In *WWW2003*, Budapest, Hungary, May 2003.

[3] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[4] J-L Gailly and M. Adler. Zlib Compression Library. www.zlib.net. Accessed January 2007.

[5] S. Garcia, H.E. Williams, and A. Cannane. Access-ordered indexes. In V. Estivill-Castro, editor, *Proc. Australasian Computer Science Conference*, pages 7–14, Dunedin, New Zealand, 2004.

[6] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *SOSP '03: Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[7] J. Goldstein, M. Kantrowitz, V. Mittal, and J. Carbonell. Summarizing text documents: sentence selection and evaluation metrics. In *SIGIR99*, pages 121–128, 1999.

[8] D. Hawking, Nick C., and Paul Thistlewaite. Overview of TREC-7 Very Large Collection Track. In *Proc. of TREC-7*, pages 91–104, November 1998.

[9] B. J. Jansen, A. Spink, and J. Pedersen. A temporal comparison of altavista web searching. *J. Am. Soc. Inf. Sci. Tech. (JASIST)*, 56(6):559–570, April 2005.

[10] J. Kupiec, J. Pedersen, and F. Chen. A trainable document summarizer. In *SIGIR95*, pages 68–73, 1995.

[11] S. Lawrence and C.L. Giles. Accessibility of information on the web. *Nature*, 400:107–109, July 1999.

[12] H.P. Luhn. The automatic creation of literature abstracts. *IBM Journal*, pages 159–165, April 1958.

[13] I. Mani. *Automatic Summarization*, volume 3 of *Natural Language Processing*. John Benjamins Publishing Company, Amsterdam/Philadelphia, 2001.

[14] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *Knowledge and Data Engineering*, 9(2):302–313, 1997.

[15] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 2007. To appear.

[16] D. R. Radev, E. Hovy, and K. McKeown. Introduction to the special issue on summarization. *Comput. Linguist.*, 28(4):399–408, 2002.

[17] M. Richardson, A. Prakash, and E. Brill. Beyond pagerank: machine learning for static ranking. In *WWW06*, pages 707–715, 2006.

[18] T. Sakai and K. Sparck-Jones. Generic summaries for indexing in information retrieval. In *SIGIR01*, pages 190–198, 2001.

[19] H. G. Silber and K. F. McCoy. Efficiently computed lexical chains as an intermediate representation for automatic text summarization. *Comput. Linguist.*, 28(4):487–496, 2002.

[20] A. Tombros and M. Sanderson. Advantages of query biased summaries in information retrieval. In *SIGIR98*, pages 2–10, Melbourne, Aust., August 1998.

[21] R. W. White, I. Ruthven, and J. M. Jose. Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes. In *SIGIR02*, pages 57–64, 2002.

[22] H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comp. J.*, 42(3):193–201, 1999.

[23] H.E. Williams and J. Zobel. Searchable words on the Web. *International Journal on Digital Libraries*, 5(2):99–105, April 2005.

[24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, second edition, May 1999.

[25] The Zettair Search Engine. www.seg.rmit.edu.au/zettair. Accessed January 2007.