Efficient Index-Based Snippet Generation

HANNAH BAST and MARJAN CELIKIK, University of Freiburg

Ranked result lists with query-dependent snippets have become state of the art in text search. They are typically implemented by searching, at query time, for occurrences of the query words in the top-ranked documents. This *document-based* approach has three inherent problems: (i) when a document is indexed by terms which it does not contain literally (e.g., related words or spelling variants), localization of the corresponding snippets becomes problematic; (ii) each query operator (e.g., phrase or proximity search) has to be implemented twice, on the index side in order to compute the correct result set, and on the snippet-generation side to generate the appropriate snippets; and (iii) in a worst case, the whole document needs to be scanned for occurrences of the query words, which could be problematic for very long documents.

We present a new *index-based* method that localizes snippets by information solely computed from the index and that overcomes all three problems. Unlike previous index-based methods, we show how to achieve this at essentially no extra cost in query processing time, by a technique we call *operator inversion*. We also show how our index-based method allows the caching of individual segments instead of complete documents, which enables a significantly larger cache hit-ratio as compared to the document-based approach. We have fully integrated our implementation with the CompleteSearch engine.

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)*; H.2.4 [**Database Management**]: Systems—***Query processing***; H.2.4 [**Database Management**]: Systems—*Textual databases*

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Snippets, document summarization, advanced search, caching, efficiency

ACM Reference Format:
Bast, H. and Celikik, M. 2014. Efficient index-based snippet generation. *ACM Trans. Inf. Syst.* 32, 2, Article 6 (April 2014), 24 pages.
DOI: http://dx.doi.org/10.1145/2590972

1. INTRODUCTION

Ranked result lists with query-dependent document snippets, as shown in Figure 1, are now state of the art in text search. Some of the earlier Web search engines started out with statically precomputed (hence query-independent) document summaries, but by now, all major engines have converged to showing snippets centered around the keywords typed by the user. User studies have shown already quite a while ago that properly selected query-dependent snippets are superior to query-independent summaries concerning the speed, precision, and recall with which users can judge the relevance of a hit without actually having to follow the link to the full document [Tombros and Sanderson 1998; White et al. 2003].

Previous work on snippet generation has mostly focused on the quality aspect: which snippets should be displayed and why. The focus of this work is on the efficiency aspect:

Parts of this article can be found in the Master's thesis of Gabriel Manolache [2008], which was supervised by H. Bast. Gabriel Manolache was involved in the early stages of this work.

Authors' address: H. Bast and M. Celikik (corresponding author), Department for Computer Science, University of Freiburg, Germany; email: celikik@informatik.uni-freiburg.de.

DOI: http://dx.doi.org/10.1145/2590972

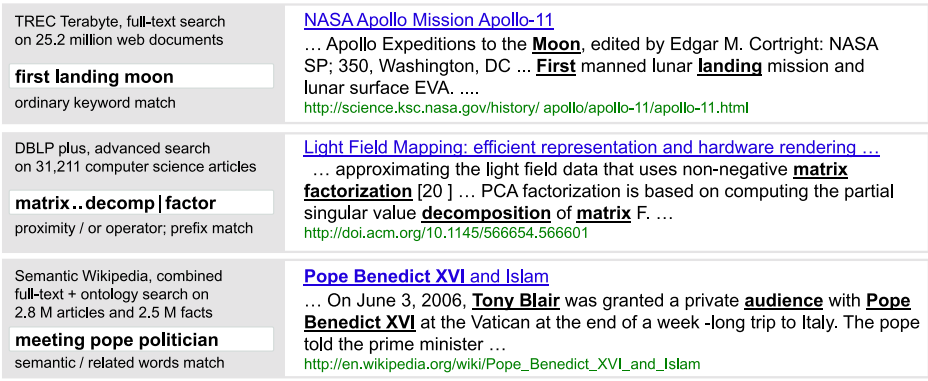

Fig. 1. Three examples of query-dependent snippets. The first example shows a snippet (with two parts) for an ordinary keyword query; this can easily be produced by the document-based approach. The second example shows a snippet for a combined proximity/OR query; in particular, the document-based approach needs to take special care here that only those segments from the document are displayed where the query words indeed occur close to each other. The third example shows a snippet for a semantic query with several nonliteral matches of the query words, for example, Tony Blair matching politician. As explained in Section 1.3, processing this query involves joining information from several documents in which case the document-based approach is not able to identify matching segments based on the document text and the query alone. The index-based approach described in the article can deal with all three cases without additional effort. For ordinary queries, it is at least as efficient as the document-based approach.

how to compute good snippets as fast as possible. We also consider an important engineering aspect: how to avoid the code duplication that is usually associated with implementations of snippet generation for complex query operators.

1.1. Document-Based Snippet Generation

In Section 2, we survey the existing methods for query-dependent snippet generation that have been described in the literature and implemented in (opensource) search engines.[1] These methods differ in which of the potentially-many snippets are extracted and shown, and in how exactly documents are represented so that snippets can be extracted quickly. On a high level, however, the bulk of them follow the same principle two-step approach from Turpin et al. [2007], which we call *document-based*. In the following description and throughout the article, we will assume a precomputed partitioning of the documents into *segments* of bounded lengths. Although it is irrelevant to our approach how this partitioning is done, we will assume that each segment corresponds to a sentence.

The document-based snippet generation can be described by the following principle steps.

(*D0*) For a given query, compute the IDs of the top-ranking documents using a suitable precomputed index data structure.
(*D1*) For each of the top-ranked documents, fetch the (possibly compressed) document text and extract a selection of segments best matching the given query.

Note that we call the first step D0 (instead of D1) because it is not really part of the snippet generation but rather a prerequisite. The same remark goes for I0.

[1]We can only guess how snippet generation is implemented in commercial products or in the big Web search engines.

1.2. Index-Based Snippet Generation

The index-based approach to snippet generation can be described by the following principle steps.

(*I0*) For a given query, compute the IDs of the top-ranking documents. This is just like (D0).
(*I1*) In addition to the information from (I0), also compute, for each query word, the matching positions of that word in each of the top-ranking documents.
(*I2*) For each of the top-ranked documents, given the list of matching positions computed in (I1) and given a precomputed segmentation of the document, compute a list of positions to be output.
(*I3*) Given the list of positions computed in (I2), produce the actual text to be displayed to the user.

Remark. The reader might wonder why we need an additional step (I1) here, given that this information is already contained in a positional index. However, in Section 3.1, we show that efficient incorporation of this information in a result list is far from trivial.

1.3. Document-Based versus Index-Based

In this section, we argue indexed-based snippet generation is superior to document-based snippet generation in several practically relevant respects.[2]

1.3.1. Nonliteral Matches. One of the central problems in text retrieval is the vocabulary mismatch problem, that is, when a relevant document does not literally contain (one of) the query words. A nice list of commented examples from the TREC benchmarks is given in Buckley [2004]. A common way to overcome the vocabulary mismatch problem is to index documents under terms that are related to words in the document but do not literally occur themselves [Bast et al. 2007a; Celikik and Bast 2009]. For example, consider the document segment from the third example in Figure 1: by adding the index term `meeting` for that document (with the same position as the related `audience`), that segment will match `meeting` even though it does not literally contain the term. When that document is returned as a hit for a query containing `meeting`, it is of course desirable that the segment containing `audience` is displayed as one of the snippets, as shown in Figure 1.

We note that another commonly employed technique is to expand the user query by including new terms that are related to the query terms at query time. This kind of query expansion works fine as long as the added query terms are very small in number. However, when many terms are added, computing the resulting disjunction becomes inefficient. Moreover, with this approach, we have a nontrivial query operator to be taken care of both in query processing and in snippet generation; see Section 1.3.2.

There is a variety of advanced search features which call for nonliteral matches: related-words search (as just explained), prefix search (for a query containing `alg`, find `algorithm`), fuzzy search (for a query containing `algorithm`, find the misspelling `algoritm`), semantic search (for a query `musician`, find the entity reference `John Lennon`), etc.

Note that the task of how to enhance the index in order to realize these features is outside the scope of this article. The problem we consider here is given such an index, how to generate snippets containing the corresponding nonliteral matches.

[2]Our argument here assumes that some kind of positional index is available; see Section 3. In scenarios where only a non-positional index is available, the document-based approach is without alternative.

The document-based approach needs additional effort to achieve this. An obvious extension would be to add the additional terms not only to the index but also to the documents. This is indeed how we used to realize this feature in our own search engine prototype so far. This extension has several disadvantages. First, it complicates and slows down the procedure for extracting matching segments from a document. Second, it blows up the space required for each document and, for large documents, can also affect the snippet generation time in case the document is fully read. Third, inserting additional terms into documents after the parsing, as required in many cases, is hard. Fourth, this is an instance of code duplication, as discussed as a separate issue later.

The index-based approach does not have any of these problems. After the top-ranking documents and corresponding positions have been computed in steps (I0) and (I1), steps (I2) and (I3) are completely oblivious of what the query words are, and they only deal with positions.

1.3.2. Code Duplication. A major system's engineering problem of the document-based approach is that each and every search feature which defines what constitutes a hit has to be implemented twice: once for computing the IDs of the best matching documents from the index, and then again for generating appropriate snippets.

For example, consider the proximity query from Figure 1. The proximity operator needs to be considered in step (D0) so that only those documents are considered as hits where the two parts (`matrix` and `decomp|factor`) occur in proximity of each other. Then it needs to be considered again in step (D1) so that we only extract (or at least prefer) snippets where the two parts of the query occur in proximity of each other.

Since (D1) does not have access to the positional information used in (D0)—that is exactly the difference between the document-based and the index-based approach—very similar functionality has to be implemented twice. This is time consuming, error prone, and harder to maintain.

The same holds true for any query operator. This includes simple ones like phrase (requires that two query words are adjacent), proximity (like in our example query), disjunction (finds any one of a group of two or more query words, like in `decomp|factor`), as well as more complex ones like the join operator described in Bast and Weber [2007] (finds all authors which have published in both SIGIR and SIGMOD).

The index-based approach does not have any of these problems. Every query operator has to be implemented once, on the query processing side, such that document IDs and positions are computed which satisfy the requirements of the respective operator. After that, (I2) and (I3) will compute only with those valid positions.

1.3.3. Large Documents. A less severe but still relevant problem of the document-based approach are very large documents. Since step (D1) has no access to the positions of the query words, in the worst case, the whole document has to be scanned to find all required occurrences of the query words. A possible remedy would be to build a small index data structure for every document in order to be able to find occurrences of query words more efficiently. In fact, the Lucene search engine does exactly this (see Section 2), but that incurs additional space and additional implementation work. In fact, it would again mean re-implementing functionality which, in principle, has already been realized for the index computation in step (D0).

The index-based approach does not have that problem, provided that we represent our documents in a way that allows efficient retrieval of those parts covering a given list of positions. As we will see in Section 5, a simple blockwise compression and storage scheme will do the job.

1.4. Our Contribution

The index-based approach is not new; however, a rigorous efficiency study was lacking so far; see Section 2 for a discussion of previous work. In this article, we provide a detailed implementation of each of the steps (I1)–(I3) that performs index-based snippet generation in negligible time, compared to the ordinary query processing in (I0). While the realizations of steps (I2) and (I3) are relatively straightforward, a straightforward implementation of step (I1) almost triples the query processing time. It also requires a modification of the central list data structure; see Section 3.1. Based on an idea which we call *operator inversion*, we show how to realize (I1) in time negligible compared to (I0).

We compare our implementation of the index-based approach with a state-of-the-art implementation of the document-based approach. We first assume that no caching is used in either approach. We then achieve the same performance as the document-based approach, however, with all the additional power that comes with the index-based approach, as explained in Section 1.3. We also compare our implementation against the Wumpus [Buettcher 2007] implementation of the well-known GCL query language [Clarke et al. 1995]. Wumpus supports a variety of structural queries computed solely based on the information contained in the positional index. Our snippet-generation times are an order of magnitude faster than those achieved by Wumpus. For the exact figures, see Section 7.

We further investigate the effect of incorporating straightforward caching in the index-based and the document-based approach. The bulk of the snippet-generation time is spent on disk seeks. This was already shown in Turpin et al. [2007] and is revalidated in Section 7. It is therefore key for high-performance snippet generation to serve as many segments as possible from memory instead of from disk. With straightforward caching, the document-based approach dictates a very coarse caching granularity: for each document, either its whole text is cached or nothing at all. This is because finding the matching segments first requires a scan over the whole document. Getting around this limitation requires explicit schemes for selecting segments from documents to be cached, and along with that, accepting the return of non-optimal snippets for a certain fraction of queries. We do not investigate such caching schemes in this article. Two such caching schemes for the document-based approach are discussed in Section 2 and again in Section 6. The index-based approach naturally permits a much more fine-grained caching: step (I2) produces a request for the text of a particular segment without the need to retrieve parts of the document first. It can therefore be decided individually for each segment whether to cache it or not. This leads to a significantly higher cache-hit ratio of the index-based approach than for the document-based approach, when both are implemented with straightforward caching. This is described and analyzed in Section 6, and our experimental results are provided in Section 7.4.

Finally, we have fully integrated our new method with the CompleteSearch engine [Bast and Weber 2007], where it provides fast snippet-generation compatible with CompleteSearch's various advanced query operators, in particular those which produce nonliteral matches.

2. RELATED WORK

Most of the work dealing with document summarization is concerned either with *query-independent summarization* [Varadarajan and Hristidis 2006] or with snippet selection and ranking. Tombros and Sanderson [1998] presented the first in-depth study showing that properly selected query-dependent snippets are superior to query-independent summaries with respect to speed, precision, and recall with which users can judge the relevance of a hit without actually having to follow the link to the full

document. As usually more segments match the query than could (and should) be displayed to the user, a selection has to be made. Questions of a proper such ranking have been studied by various authors. For example, Varadarajan and Hristidis [2006] have proposed an algorithm for computing segments that are semantically related to each other as much as possible. In Ko et al. [2007], a pseudo relevance feedback is applied to salient sentence extraction to identify the most relevant sentences. A slightly different concept in document summarization is the multi-document summarization paradigm that reduces the document size of the top-matching documents to only a few sentences by retaining the main characteristics of the original documents [Wang et al. 2008].

Note that nowadays all the big search engines have query-dependent snippets, in particular, Google, Yahoo Search, and Bing. In this article, we take the usefulness of query-dependent result snippets for granted and focus on efficiency and feasibility aspects. We come back to the usefulness aspect in our conclusions. For the scoring and ranking of segments, we adopt the simple, yet effective scheme from Turpin et al. [2007], which we briefly recapitulate in Section 7.

Turpin et al. [2007] have presented a first in-depth study of the document-based approach with respect to efficiency (in time and space). They propose the following document representation: all words as well as all separating sequences are replaced by an ID, where the more frequent tokens get lower IDs, that is, the most frequent term gets ID 1, the second most frequent ID 2, etc. (very infrequent tokens remain in plain text). IDs are then universally encoded (with small IDs getting a short code). In addition, they introduce document caching strategies and point out the significance of caching as a highly effective way to increase query throughput.

In a sequel of this work, Tsegay et al. [2009] propose a lossy document compaction strategy (so called *document surrogates*) in order to improve the effectiveness of document caching. Their approach is based on reordering sentences in a document so that sentences that are more likely to be included in a snippet appear near the beginning of the document. The remaining sentences are pruned to reduce their size and hence make a better use of the available memory. In a recent related work, Ceccarelli et al. [2011] use the knowledge stored in query logs to build dynamic concise document surrogates by identifying the subset of snippets in a document that is most likely to be accessed in the future (so called *super-snippets*). They show experimentally that the number of distinct relevant snippets for a given document is in most of the cases very small. As we will see in Sections 6 and 7, our caching approach does this naturally and in a sense optimally.

A variant of the index-based approach has been previously investigated in Clarke et al. [1995] and Clarke and Cormack [2000]. The emphasis of this line of work is on a clean and universal query language model (based on the so-called GCL query algebra) for the retrieval of arbitrary passages of text. The authors do consider questions of efficiency, but even their most efficient algorithm touches every (positional) posting of each of the query words, which is exactly what our operator inversion, explained in Section 3.2, avoids. The GCL query language is implemented in the Wumpus search engine [Buettcher 2007]. In Section 7, we will see that snippet-generation times for this approach indeed outweigh the query-processing times. The issue of fine-granularity caching is not discussed in this line of work.

The open-source search engines that we know of and which provide query-dependent snippet generation follow the document-based approach. In particular, this holds true for Lucene [Cutting 2004], which we have studied in depth for the purpose of this work. Early versions of Lucene used a straightforward implementation of the document-based approach: at query time, each of the top-ranked documents was re-parsed and segments containing the query words were extracted. To address the

obvious inefficiency of this approach, recent versions of Lucene have provided support for storing the sequence of term IDs output by the parser (much in the vein of Turpin et al. [2007]).

Another alternative is to precompute and store for each document a small inverted index for fast location of the query words. Without significant space overhead, the small inverted index will provide us the positions for each of the query words in a document. However, this does not help with avoiding code duplication for nontrivial query operators (like proximity), since it still remains to filter out the positions which obey that query operator. Simply running the query again on this index will not solve the problem, since standard query processing only provides a list of matching documents (in Section 3.1, we show that computing the matching postings in a straightforward way is far from efficient). Therefore, all of the proposed enhancements remain in the realm of the document-based approach, and as such do not address the problems of nonliteral matches, code duplication, and coarse caching granularity pointed out in Sections 1.3 and 1.4.

3. STEP I1: COMPUTING ALL MATCHING POSITIONS

In this and the following two sections, we will describe our realization of steps (I1), (I2), and (I3) of the index-based approach, as described in high level in Section 1.2. We begin by presenting two algorithms for (I1), that is, computing for each query word the list of its positions in each of the top-ranked documents.

Throughout this work, we assume that we have a *positional index* with posting lists that conceptually look as follows.

doc ids	D401	D1701	D1701	D1701	D1807
word ids	W173	W113	W94	W202	W516
positions	5	3	17	51	10
scores	0.3	0.7	0.4	0.3	0.2

For example, the third posting says that the word with ID W94 occurs in the document with ID D1701 at position 17, and was assigned a score of 0.4 (individual scores are aggregated to per-document scores, according to which the documents are eventually ranked). In an actual implementation, these lists would be stored in compressed format, but we need not consider this level of detail.

A standard inverted index will have one such list precomputed for each word (i.e., all word IDs are the same in each such list). Given a query, the basic operation will be to either intersect the lists for each of the query words (to obtain the list of IDs of all documents that contain all query words; such queries are called conjunctive) or to compute their union (to obtain the list of IDs of all documents that contain at least one of the query words; such queries are called disjunctive). We will later consider more query operators; see Section 3.2.

Pruning techniques are often used to avoid a full scan of the index lists involved, especially in the case of disjunctive queries; for example, see Anh and Moffat [2006] and Bast et al. [2006]. In Bast and Weber [2006], index lists are computed not for individual words but for ranges of words. In Bast and Weber [2007], a join operation on these lists is introduced, which forms the base of the semantic queries introduced in Bast et al. [2007b].

The following considerations are valid for all of these list representations and operations.

Table I. Elapsed Query-Processing Times

Scheme	all	one-word	two-word	≥ three-word
Ordinary Lists	144.0 ms	17.9 ms	46.7 ms	174.9 ms
Extended Lists	342.6 ms	29.7 ms	114.5 ms	415.7 ms
Operator Inversion	+ 0.4 ms	+ 0.1 ms	+ 0.2 ms	+ 0.5 ms

Note: With ordinary lists and with extended lists, and the additional time taken by the operator inversion (only step I1) on TREC GOV2 (the numbers in the third tow are in addition to the first row). The first column gives the average over 10,000 queries. The other columns provide the averages for fixed numbers of query words (description on the hardware used is given in Section 7).

3.1. Extended Posting Lists

Step (I1) asks for the computation of the positions of each of the query words in each of the top-ranked documents. A positional index, as already described, obviously contains this information. A straightforward way to incorporate this information in a result list would be to enhance each posting by an additional entry, telling from which query word it stems. Conceptually, it would look like this.

doc ids	D1701	D1701	D1701	D1701	D1701
word ids	W173	W173	W173	W179	W179
positions	5	9	12	54	92
scores	0.3	0.7	0.4	0.3	0.2
query word	1	1	1	2	2

For example, the third posting from the right, now "knows" that it stems from the first query word (when reading a list from disk, the query word entry would be set to some special value). It is not hard to see that with this enhancement, the information required for (I1) can be computed for all of the operations previously described.

There are two major disadvantages with this approach, however. The first is that we modified the central data structure, the sanctuary of every search engine. Unless the search engine was written with such modifications in mind, this is usually not tolerable. For our own research prototype, we would expect numerous undesirable side effects and bugs following such a modification.

The second major problem is efficiency. Consider an intersection of two query words. The extended result lists would now contain postings from both input lists, that is, it at least doubles in size compared to the corresponding simple result list. This effect aggravates as the number of query words grows. Table I shows that this indeed affects the processing time. The problem is that per-query-word positions are computed for all matching documents, before the ranking is done. Note that this also happens when pruning techniques are involved, because, again, large numbers of postings (namely, candidates for one of the top-ranked doc IDs) are involved in the intermediate calculations. In fact, the results from Table I pertain to such a pruning technique for disjunctive queries, following the ideas of Bast et al. [2006].

3.2. Operator Inversion

We propose to implement (I1) by what we call *operator inversion*. Our approach works for (but is not limited to) all of the following query operators: *boolean search*, *phrase search*, *proximity search*, *prefix search*, *range search*, *fuzzy search*, and *semantic search*.

We first explain our approach by an example by using a simplifying assumption that our query can be evaluated from left to right.[3] Then in Lemma 3.2, we will formalize our approach for arbitrary expression trees.

[3]Note that this is often the case in practice.

Example 3.1. Consider the query `efficient snippet..generation`, which searches for documents that contain the word `efficient` and the two words `snippet` and `generation` in (some predefined) proximity to each other. Assume that in step (I0) we have already computed the set D of IDs of the top-ranked documents. Let L_1 be the list of postings of the matching occurrences of `efficient` in D, and let L_2 and L_3 be the corresponding lists for `snippet` and `generation`, respectively. Our operator inversion algorithm then goes in two phases as follows.

For the first phase, let I_1, I_2, I_3 be the index lists for the three query words, that is, the list of all postings of all occurrences of `efficient`, `snippet`, and `generation`, respectively, in the whole document collection. Let L'_1 be the list of all postings from I_1 with a document ID in D. This can be computed via a simple intersection of the two (sorted) lists of document ids. Similarly, compute L'_2 as the list of all postings from I_2 with a document ID in D. For L'_3, we do something slightly different. Namely, let L'_3 be the list of all postings from I_3 with a document ID d and a position i such that there is a posting from L'_2 with the same document ID d and a position j, such that i and j are within the predefined proximity to each other. If the posting lists are sorted by document ID and postings for the same document ID are sorted by position, this can also be computed with a straightforward variant of the standard intersection algorithm between I'_2 and I_3.

This ends the first phase. Note that now L'_1 is already the desired list L_1, but all we can say about L'_2 at this point is that it contains all the desired postings from L_2. It may also contain other postings though, namely, those pertaining to the occurrence of `snippets` in documents from D that are not in proximity to occurrences of `generation`. The last list computed in this phase, L'_3, is exactly the desired L_3 again.

In the second phase, we compute L_i for those L'_i which are not yet L_i. In our example, this is only L'_2. We compute L_2 as the list of all postings from L'_2 with a document ID d and a position i such that there is a posting from L'_3 with the same document ID d and a position j, such that i and j are within the predefined proximity to each other. This is the same intersection operation as between L'_2 and I_3 in phase 1, except that L'_2 is now the other of the two lists. Technically, we apply the *inverse* of the proximity operator now, that is, we compute $(L'_2..L_3)^{-1}$ (which happens to be the same operator, if we define proximity as $|i-j| \leq \delta$, for some fixed δ).

Since D is very short, each of L'_1, L'_2, L'_3, and L_2 will be very short too, and all of the involved intersections can be computed extremely fast. Indeed, our experiments in Section 7.3 will show that the time required for (I1) using operator inversion is negligible compared to both the time for (I0) (the query processing that has to be done anyway) and the time needed for (I2–I3) (getting the snippet text from the documents). This ends our example.

Generalizing from this example, let D be the sorted list of IDs of the top-k matching documents for an arbitrary query with l keywords and a given k. Let $L'(q_i)$ be the posting list that contains all occurrences of the ith query word in D, for $i = 1, \ldots, l$. In the simplest case, this could be the inverted index list of q_i.[4] If the fuzzy search operator is used on q_i, then $L'(q_i)$ is a fuzzy inverted list of q_i, etc. Assume that each query operator has an inverse operator (e.g., the proximity operators and all binary boolean operators except OR are self-inverse).

[4]With top-k techniques like those from Anh and Moffat [2006] or Bast et al. [2006], it can be a subset of this inverted list.

As in Example 3.1, whenever an operator is applied on two posting lists, the resulting list will contain positions from the second posting list only. We consider the expression tree for a given query Q. Each internal node in the tree contains pointers to its two children, the operator and a list of postings (documents and positions). Each leaf node corresponds to a query word and initially contains its posting list. For example, the leaf node that corresponds to the query word q_i initially contains the list of postings $L'(q_i)$. As in the example, the algorithm goes in two phases. In the first phase, the expression tree is evaluated in a bottom-up fashion starting from the root, as shown with the following pseudocode:

```
eval-bottom-up(node):
    If isLeafNode(node)
        node.list = L'(node.q_i);
    else
        node.list = node.op(eval-bottom-up(node.left),
                            eval-bottom-up(node.right));
```

where node.*op* is the operator of the current internal node. The bottom-up evaluation of the expression tree is analogous to the left-to-right evaluation in the example. In the second phase, the evaluation is top-down, where each node updates the lists of its children starting from the root as shown with the following pseudocode:

```
eval-top-down(node):
    If isInternalNode(node)
        node.right.list = node.list;
        node.left.list = node.op-inv(node.left.list, node.right.list);
        eval-top-down(node.left);
        eval-top-down(node.right);
```

where node.*op-inv* is the inverse operator of the current internal node (in many cases the same operator). The top-down evaluation of the expression tree is analogous to the right-to-left evaluation in the example.

LEMMA 3.2. *For a query with keywords $q_1, \ldots, q_\ell$, let D be the IDs of the top-k matching documents, and let $L'(q_i)$ be the posting list that contains all (fuzzy, if that operator is used on q_i) occurrences of the ith query word in D. Assume that each query operator can be computed in time $O(x \cdot \log y)$, where x and y are the lengths of the inverted lists of the two operands. Then the preceding algorithm is correct and computes the set of matching positions for each query word q_i in each of the documents from D in time $O(k \cdot \sum_{i=1}^{\ell} \log |L'(q_i)|)$.*

Remark. Note that query operators are typically realized via list intersection, or variants of it with the same running time. The assumption of the lemma is then fulfilled, because we can simply binary-search the elements from the first list in the second list. In the proof, the first list will always be small, namely, of size k. Hence the factor of k in the final running time of our algorithm, which is just a sequence of intersections of D with (subsets of the) appropriate inverted lists of the query words.

PROOF. First we will show that the preceding algorithm works correctly. We consider the expression tree of a query Q. Observe that each node in the expression tree corresponds to a subquery of Q. Let Q_u be the subquery that corresponds to a node u, and let Q_v and Q_w be the subqueries that correspond to the left and the right child of u, respectively (see Figure 2, left). Let last(Q_u) be the index of the last query word in Q_u, where last(q_i) $= i$. Let $E(Q_u)$ be the result list of Q_u in form of an extended posting list (defined in the previous section), and let $L'(Q_u)$ be the result list of Q_u

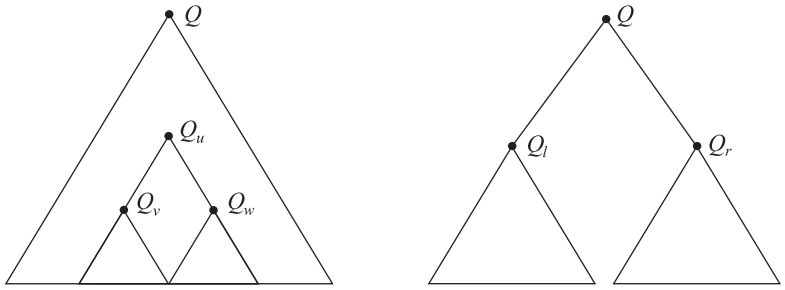

Fig. 2. Schematic representation of the expression tree for a query Q with subtrees that correspond to subqueries.

that contains only the postings of the last query word, that is, $L'(Q_u) = \{p \in E(Q_u) \mid \text{ind}(p) = \text{last}(Q_u) \wedge \text{doc}(p) \in D\}$, where $\text{ind}(p)$ is the index of the query word in the extended posting list from which the posting p stems and $\text{doc}(p)$ is the doc ID of p. For each query word q_i, we have $L'(q_i) = \{p \in E(q_i) \mid \text{doc}(p) \in D\}$, assuming that $E(q_i)$ is the resulting list of postings after applying the unary operator on q_i (if any). We would like to compute the list $L(Q_u) = \{p \in E(Q) \mid \text{ind}(p) = \text{last}(Q_u) \wedge \text{doc}(p) \in D\}$ for each subquery Q_u. Recall that each operator $\circ$ defines a binary relation over the set of all postings. Let R° be the relation defined by the operator $\circ$. Whenever $(p_1, p_2) \in R^\circ$, we will write $p_1 \circ p_2 = \top$, where $\top$ is the "true" symbol. $L'(Q_v) \circ L'(Q_w)$ is then defined as $\{p_w \in L'(Q_w) \mid \exists p_v \in L'(Q_v), p_v \circ p_w = \top\}$, where $\circ$ is the query operator for Q_v and Q_w. Analogously, $L'(Q_v) \circ^{-1} L'(Q_w)$ is defined as $\{p_v \in L'(Q_v) \mid \exists p_w \in L'(Q_w), p_v \circ p_w = \top\}$. By using a simple induction and applying the definition of $L'(Q_v) \circ L'(Q_w)$ in the induction step, we obtain $L'(Q_u) = L'(Q_v) \circ L'(Q_w)$. This is what the first phase of our algorithm computes. Observe that $L(Q)$ is computed already at the end of the first phase of the algorithm, since $L'(Q) = \{p \in E(Q) \mid \text{ind}(p) = \text{last}(Q) \wedge \text{doc}(p) \in D\} = L(Q)$. Let l and r be the left and the right child, respectively, of the root of the expression tree, such that $L'(Q) = L'(Q_l) \circ L'(Q_r)$ (see Figure 2, right). Since $\text{last}(Q) = \text{last}(Q_r)$, it follows that $L(Q_r) = \{p \in E(Q) \mid \text{ind}(p) = \text{last}(Q_r) \wedge \text{doc}(p) \in D\} = \{p \in E(Q) \mid \text{ind}(p) = \text{last}(Q) \wedge \text{doc}(p) \in D\} = L(Q)$. Let the list $L''(Q_l)$ be defined as $L''(Q_l) = L'(Q_l) \circ^{-1} L'(Q_r)$, and let $p_l \in L''(Q_l)$. Since $p_l \in L'(Q_l)$ such that $\exists p_r \in L'(Q_r)$ with $p_l \circ p_r = \top$, the latter is equivalent to $p_l \in E(Q)$. Since $\text{ind}(p_l) = \text{last}(Q_l)$ and $\text{doc}(p_l) \in D$, by using the definition of $L(Q_l)$, the latter is equivalent to $p_l \in L(Q_l)$. Hence, $L''(Q_l) = L(Q_l)$. This is what the second phase of the algorithm computes in top-down fashion. The correctness can be shown by using induction on the depth in the expression tree. Namely, if we assume that all desired lists of nodes up to depth d are correctly computed, then by using these arguments, we can correctly compute the lists of nodes at depth $d + 1$ from their parents.

The running time of our algorithm is now easily bounded. By the assumption of the lemma (see also the preceding remark, after the statement of the lemma), each operator can be evaluated in time $O(x \cdot \log y)$, where x and y are the lengths of the two inverted lists. Since $|D| = k$, the first phase can be completed in time $O(k \cdot \sum_{i=1}^{\ell} \log |L'(q_i)|)$. The second phase requires at most that amount of time, and typically less. This is because in that phase, the same number of operators are evaluated (the difference is that now the reverse operators are considered), and the size of lists involved can only be smaller than in the first phase, never larger. □

Table I shows that our operator inversion approach adds less than 1% to the query processing time, and unlike the extended posting lists, it does not require any modification of the internal posting list data structure.

4. STEP I2: COMPUTING THE SNIPPET POSITIONS

This section is about computing and ranking the actual snippets once the positions of each query word in each of the top-ranked documents have been computed. We start by showing how to compute the positions of all matching snippets given the positions of each query word and a partitioning of the document into segments. The produced snippets can be then ranked similarly as in the document-based snippet generation.

4.1. Algorithm for Computing the Snippet Positions

Computing the *positions* of the snippets (not the actual text, which is done in I3) to be output becomes an instance of the following simple problem, to be solved for each of the top-ranked documents.

(*I2*) Given ℓ sorted lists of positions $L_1, \ldots, L_\ell$ pertaining to the ℓ query words in a document, and a sorted list of positions $S = s_1, s_2, \ldots$ marking the segment beginnings in that document, compute a list of s tuples $(\text{seg}_i, \text{Pos}_i)$, sorted by the seg_i, where seg_i is the index in S of the ith segment containing a position from one of the q_i, and Pos_i is the list of those positions from $L_1, \ldots, L_\ell$ which fall into that segment, together with the information from which of the lists each position has come. In simple terms, for each segment, we would like to compute the set of positions from $L_1, \ldots, L_\ell$ (together with the corresponding query term indices).

Example 4.1. Consider a two-word query and a fixed document with five segments. Let $L_1 = \{3, 8, 87\}$ be the positions of the first query word, $L_2 = \{13, 79\}$ be the positions of the second query word, and $S = \{1, 17, 43, 67, 98\}$ be the first positions of each segment. Then the correct output of I2 would be the two tuples $(1, (3, 1), (8, 1), (13, 2))$—segment 1 with three occurrences of the query words—and $(4, (79, 2), (87, 1))$—segment 4 with two occurrences of the query words.

LEMMA 4.2. *For each hit displayed to the user, step (I2) can be executed in time* $O\big((|L_1| + \cdots + |L_\ell| + |S|) \cdot \log \ell\big)$.

PROOF. (I2) can be implemented by a straightforward variant of an l-way merge, followed by a simple sort. Here we assume that a segment can be scored in linear time in its length. □

Our experiments show that the computation involved in (I2) takes negligible time. This is understandable, since L_i, S, and ℓ are always small, and we are only manipulating positions (numbers) in this step. The bulk of the time in the index-based approach is spent on disk seeks to access those parts of the documents that are used as snippets. For (I2), disk seeks are needed to fetch the list of segment boundaries.

4.2. Scoring of Snippets

Each tuple of positions computed by the algorithm in the previous section corresponds to a potential segment to be output. Now all that remains is to score the segments and return those with the m-largest scores, for some user-defined m. In our experiments, we take $m = 3$ and score segments by a simple but effective scheme (described in Figure 2 of Turpin et al. [2007]). The scheme is based on a weighted sum of the number of query words in the segment, the number of distinct query words in the segment, the longest contiguous run of query words, and a few other similar quantities. Note that our index-based approach is compatible with any other scoring mechanism that works on the sentence level (or any other precomputed segmentation of the text). For

example, many of the more complex approaches in the literature [Goldstein et al. 1999; Ko et al. 2007; Tombros and Sanderson 1998; Varadarajan and Hristidis 2006; White et al. 2002] are naturally based on scoring individual sentences. Hence, the same (document-based) scoring schemes could be easily applied here with no difference in the quality of the produced snippets.[5] We note that a sophisticated snippet scoring scheme is not in the scope of this work.

5. STEP I3: FROM SNIPPET POSITIONS TO SNIPPET TEXT

Given a list of positions, namely, the positions of the top-ranked segments computed in (I2), we want to obtain the actual text at these positions, which will then be displayed to the user. In this section, we show how to represent the documents so that this can be done efficiently. We begin by describing our block representation of documents that allows a trade-off between the amount of text scanned and the space overhead imposed by storing the document partitioning into segments.

We stress again that this final step in which the actual text gets generated is completely oblivious of the query words that have given rise to the position lists. As explained in Section 1.3, this has the valuable advantage that any kind of advanced search feature that defines what constitutes a match has to be implemented only once, on the query processing side.

5.1. Block Representation

In the document-based approach, the whole compressed document needs to be read from disk and scanned in order to determine which segments match the query. For large documents, this is an efficiency problem. As a heuristic, Turpin et al. [2007] suggest rearranging the documents such that segments that are likely to score high are at the beginning, and then consider only the head of each document.

For our approach, we could in principle read exactly the amount of text needed for the snippets and no more. There would be a price for this, however: first, we would (at least in theory) incur one disk seek per segment in large documents, and second, it would require us to store for every segment its offset in the document. To balance these conflicting goals, we take the following approach.

Each document is divided into blocks, where each block covers a fixed number of p positions. We can easily identify the blocks containing the desired positions, given the output of step (I2). Every block is stored compressed on disk together with the relative offsets in the document. This minimizes the size of our document database, as well as the time to read blocks from disk. Smaller p would imply less text to be read and scanned, however, higher space overhead. Since our blocks are small, there is little value in encoding each token separately, as proposed in Turpin et al. [2007]. Instead, we compress each block as a whole by using zlib (with the default compression level 6). As shown in Section 7.2, this gives us slightly better compression than the compression achieved in Turpin et al. [2007].

To achieve a higher compression ratio, instead of zlib, one could use a compression method based on the Lempel-Ziv (LZ) family that takes advantage of global repetitive properties of the document collection. Hoobin et al. [2011] propose a compression scheme for fast random access that is an alternative to the blocked representation. This method is based on a string data structure called relative Lempel-Ziv factorization and allows for much higher compression ratios, yet faster document retrieval. We would like to note that novel compression schemes for fast document retrieval are not in the scope of this work.

[5]Note that this might sometimes require performing the segment scoring in step (I3), where the actual segment text is produced, instead of doing it in step (I2) which operates on positions only.

5.2. Snippet Text and Highlighting Matches

After each of the relevant blocks is read and decompressed, the text corresponding to the given segment positions is easily obtained by a simple scan over these blocks. The actual highlighting is done at the end of (I3), when the snippet text has been fully reproduced. In order to use the same word boundary definition as the parser, each indexed word is marked with a special character produced at index time. This provides the exact information for every word position. Note that the tuples corresponding to segment positions computed in (I2) contain the information whether a certain position is matching.

6. SNIPPET CACHING

In this section, we investigate the effect of incorporating straightforward caching in the index-based and the document-based approach. As we will see in Section 7, the times for steps (I2) and (I3) are dominated by the disk seeks. More precisely, these disk seeks are required to seek the list of the first positions of the segments, and then seek the blocks to be read from disk. A similar observation has been made in Turpin et al. [2007] for the document-based approach. They suggest two caching strategies, one static and one dynamic, with the goal of avoiding as many disk seeks as possible. Both of these strategies cache whole documents because, as discussed earlier, finding the matching segments requires a scan over the whole document.

Tsegay et al. [2009] and Ceccarelli et al. [2011] proposed more advanced caching schemes, which explicitly select segments from the documents for caching, and along with that accept returning non-optimal snippets for a fraction of the queries. We already discussed those schemes in Section 1.4 and Section 2 and briefly come back to them again in Section 6.2.2.

We start by describing our segment caching algorithm. We then compare it against whole-document caching and show that under certain assumptions, snippet caching achieves substantially higher cache hit-ratio when the cache sizes are equal. We confirm this experimentally in Section 7.4.

6.1. Segment Cache

Our approach naturally allows caching of individual segments instead of whole documents. While any segmentation would work with our approach, for our experiments, we consider a simple segmentation into sentences. This is because (I2) outputs the indices of segments (which were obtained from the inverted index) to be output before we ever look at the actual documents. When a segment with index j from a document with ID i is requested for the first time, it is fetched from disk and stored in memory under the key (i, j) using a hash map. Whenever the segment with key (i, j) is requested again, we fetch it directly from memory. When a prespecified amount of memory (i.e., the capacity of the cache) is exceeded, we evict the least recently used (LRU) segment(s) from the cache until there is enough space again.

The segment cache has two main advantages over the document cache. First, it allows higher throughput, since it involves copying significantly less amount of text to memory. Second, and even more importantly, it makes much better use of the available memory than whole-document caching. In the following, we provide theoretical as well as experimental evidence to support this claim.

Note that storing only selected segments would be of no use for the document-based approach, because for a new query-document pair, we do not know which segments to retrieve before we scan the entire document. Turpin et al. [2007] and Tsegay et al. [2009] address the problem of storing whole documents in the cache by moving segments with likely high scores to the beginning of the documents. This, however,

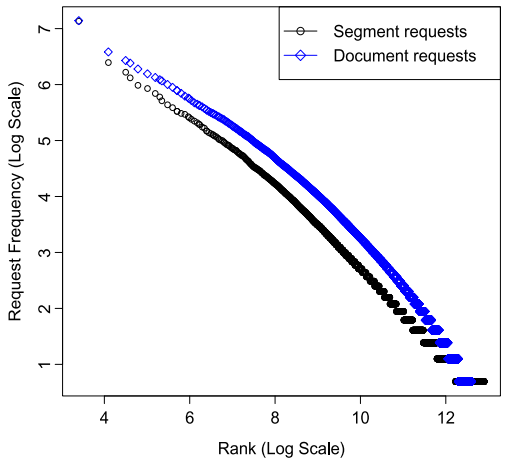

Fig. 3. A log-log plot of the frequency of document and segment requests versus document and segment ranking on the TREC GOV2 collection (see Section 7.1).

risks an occasional loss of relevant segments, which may not be tolerable in some applications.

6.2. Distribution of Document and Segment Requests

In this section, we provide experimental evidence to show that the top segment requests exhibit distribution similar to that of the document requests.

6.2.1. Distribution of Document Requests. Previous studies suggest that document requests follow an approximate *Zipfian distribution*, also known as *Zipf's law* [Almeida et al. 1996; Breslau et al. 1999]. Zipf's law states that the probability of a request for the ith most frequently seen document is equal to $H_{n,\alpha}^{-1}/i^{\alpha}$, where $\alpha > 0$ is a constant, n is the total number of documents, and $H_{n,\alpha}$ is the generalized harmonic number of order n of α defined as $\sum_{i=1} 1/i^{\alpha}$. For a fixed n, a larger α means a higher locality of reference of the document requests. A large n, on the other hand, means a lower locality of reference of the document requests. This is because the probability mass is spread among a larger number of documents.

The page request distribution seen by Web caches is investigated by Breslau et al. [1999] by using traces from variety of sources. According to their study, typical values of α range in the interval 0.7 up to 0.8 (i.e., we consider $0 < \alpha < 1$). The plot of the Zipfian distribution has a long tail and it is represented by a straight line (with a slope equal to $-\alpha$) when both axes are in log scale.

6.2.2. Distribution of Segment Requests. Figure 3 shows a log-log plot of the distribution of document and segment request for the TREC GOV2 collection. It is clear that the plots have similar slopes. Figure 4 shows the segment request frequency of the segments requested in the top-100 documents on two of our test collections. For example, the frequency of the ith ranked segment is calculated as an average frequency of all segments with rank i (segments with average frequency of less than 1 are not shown). The left side of Figure 4 shows that, unlike the Zipfian distribution, the distribution of segment requests does not have a long tail. In fact, most of the requests are concentrated on a single segment per document. More precisely, 85% of all segment requests on the DBLP collection and 78% of all segment requests on the TREC GOV2 collection are requests for a single segment per document. The plots in log-log scale show that the distribution of segment requests is more skewed than the Zipfian distribution.

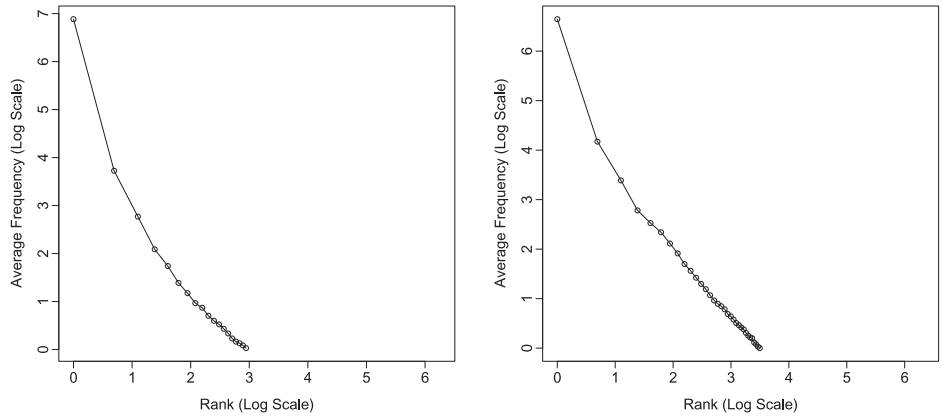

Fig. 4. Average frequency of segment requests in a document, shown in log-log scale (the base of the logarithm is e on both axes). The frequencies have been calculated as averages over the 100 most frequently requested documents from the TREC GOV2 (left) and the DBLP collection (right) (see Section 7.1).

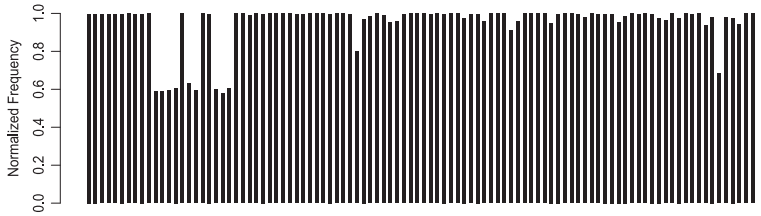

Fig. 5. The fraction of segment requests which belong to the ten most frequently requested segments in a document shown for the 100 most frequently requested documents from the DBLP collection. Each bar corresponds to a single document.

Figure 5 shows the fraction of top-10 segment requests from all segment requests in the top-100 documents on the DBLP collection. In general, 95% of all segment requests are concentrated on the 10 most frequently requested segments per document.

Ceccarelli et al. [2011] try to answer a similar question, namely, how many different snippets have to be generated for a single document. They come to a similar conclusion as we do, namely, that the number of different snippets for a given document is small. In particular, according to their experimental analysis, 99.96% of all documents have less than ten snippets associated with them, and about 92.5% have only a single snippet. In this work, instead of caching whole documents, the authors construct a so-called *super-snippet* that contain the most frequently requested snippets of that document. It is not hard to see that our fine-granularity cache does this naturally and transparently by caching only the relevant snippets. After a warm-up period, our segment cache will automatically contain only the snippets with high likelihood of being requested again in the future.

6.3. Segment vs. Document Cache

In this section, we compare the asymptotic cache hit-ratios (i.e., the hit-ratio for an infinite number of document requests) and the convergence rates of the document and the segment cache by using the observations from the previous section.

First, we would like to point out that given the fact that only a small number of snippets are generated from each document, it is intuitive that the snippet cache will make better use of the available memory than the document cache, and hence, achieve

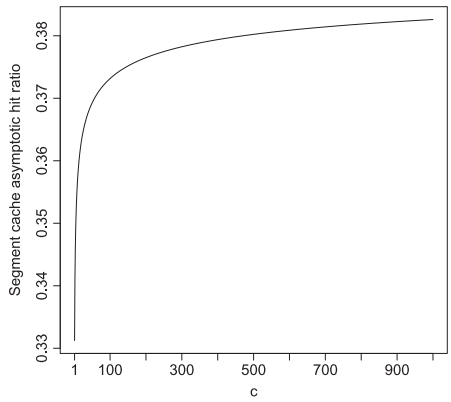

Fig. 6. Asymptotic hit-ratio of the segment cache ($h(c,f)$) for $f = 1$ and increasing c ($\alpha = 0.8; m = 1{,}000$; $n = 100{,}000$).

a higher cache hit-ratio. To show this more formally, we consider a finite cache with a capacity of m documents that receives a stream of independent requests. In addition, we make a number of simplifying assumptions. First, we assume a Perfect-LFU removal policy, that is, a cache that always contains the m most frequently requested documents (segments). Second, we assume that all documents have the same size, and third, we assume that only the top-ranked segment is cached and shown to the user. The asymptotic hit-ratio of the Perfect-LFU replacement policy (see Breslau et al. [1999]) is then given by

$$\sum_{i=1}^{m} \frac{1}{H_{n,\alpha}} \cdot \frac{1}{i^{\alpha}} = \frac{H_{m,\alpha}}{H_{n,\alpha}}. \tag{1}$$

Let $c > 1$ be the average number of segments per document. Our segment cache could then fit roughly $c \cdot m$ out of $c \cdot n$ segments. Let $0 < f \leq 1$ be the fraction of all segments that contribute to the overall distribution of segment requests, that is, we assume that on average, $f \cdot c$ relevant segments are generated from each document. Let α_D and α_S be the parameters of the Zipfians that correspond to the document and the segment requests, respectively. It is not hard to see that $\alpha_D \geq \alpha_S$, since each segment request is preceded by a request of the document that contains that segment. However, from the discussion in the previous section, we will assume that $\alpha_D \approx \alpha_S$.

Under these assumptions, it is not hard to show the following.

LEMMA 6.1. *(i) The segment cache has higher asymptotic hit-ratio than the document cache; (ii) the document cache converges faster than the segment cache.*

PROOF. To show (*i*), we consider the asymptotic hit-ratio of the segment cache given by

$$h(c,f) = \frac{H_{c \cdot m,\alpha}}{H_{f \cdot c \cdot n,\alpha}}, \tag{2}$$

as a function of c and f, where $h(1,1)$ is equal to the asymptotic hit-ratio of the document cache given in Equation (1) (Figure 6). By approximating $H_{n,\alpha}$ by the integral $\int_1^n 1/x^{\alpha}\, dx$, it is a matter of simple calculus to show that $h(c,f)$ is a monotically increasing function of $c > 1$ and hence $h(c,f) > h(1,1)$. Figure 7 compares the asymptotic hit-ratios of the segment and the document cache for different values of m and f.

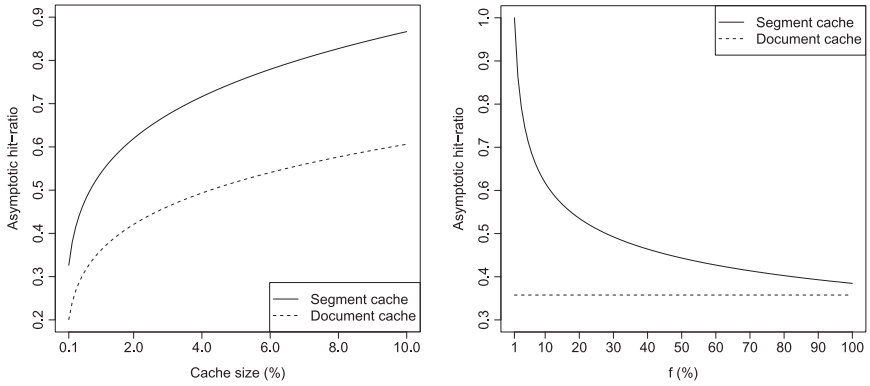

Fig. 7. Asymptotic hit-ratio of the segment and the document cache for $n = 1{,}000{,}000$ (one-million document collection) when the cache size varies (left); and when f (the percentage of relevant segments) varies and $m = 10{,}000$ documents (right).

To show (*ii*), we make use of the following simplification from Breslau et al. [1999]. We consider a cache of infinite size so that all previously requested pages remain in the cache. We assume that R documents have been already requested and consider the cache hit-ratio at the $R + 1$'st request. Let the $R + 1$'st request be a request for the document with rank i. The probability for a cache hit is then equal to the probability that this document has been already requested, which in turn is equal to

$$1 - \left(1 - \frac{H_{n,\alpha}^{-1}}{i^{\alpha}}\right)^{R}.$$

Consider the event that the $R + 1$'st document request causes a cache hit. By the law of total probability, the probability of this event is equal to

$$p(R,n) = \sum_{i=1}^{n} \frac{H_{n,\alpha}^{-1}}{i^{\alpha}} \left(1 - \left(1 - \frac{H_{n,\alpha}^{-1}}{i^{\alpha}}\right)^{R}\right). \tag{3}$$

Analogously, the probability for a cache hit at the $R + 1$'st segment request is equal to $p(R, f \cdot c \cdot n)$. We will show that for any fixed R, $p(R,n)$ is a monotonically decreasing function of n, which would imply that $p(R, f \cdot c \cdot n) \leq p(R,n)$. By dropping i^{α} from the second term in Equation (3), we obtain

$$\begin{aligned} & p(R,n) \\ \leq & \left(1 - \left(1 - \frac{1}{H_{n,\alpha}}\right)^{R}\right) \cdot \sum_{i=1}^{n} \frac{H_{n,\alpha}^{-1}}{i^{\alpha}} \\ = & \, 1 - \left(1 - \frac{1}{H_{n,\alpha}}\right)^{R}. \end{aligned}$$

This expression is a monotonically decreasing function of n, since $H_{n,\alpha}$ is a monotonically decreasing function of n. Hence, $p(R,n)$ must be a monotonically decreasing function of n. □

7. EXPERIMENTS

We ran various experiments to evaluate our implementation of index-based snippet generation and compared it against the document-based approach from Turpin et al.

[2007] as well as against the query algebra approach from Clarke et al. [1995] as implemented by the Wumpus search engine; see Section 2.

All our experiments were run on a machine with two dual-core AMD Opteron processors (2.8GHz and 1MB cache each), 16GB of main memory, running Linux 2.6.20 in 32-bit mode. All our code is single-threaded, that is, each run used only one core at a time. All our code is in C++ and was compiled with g++ version 4.1.2, with option −O3. Before each run of an experiment, we repeatedly read a 16GB file (that was otherwise unrelated to our experiments) from disk, until read times suggested that it was entirely cached in main memory (and thus all previous contents flushed from there).

7.1. Collections and Queries

DBLP Plus. 31,211 computer science articles listed in DBLP, with full text and meta data, and 20,000 queries extracted from a real query log, for example, `matrix..factor`. The index was built with support for error correction and prefix search, as explained in Section 1.1. All queries were run as proximity queries, as in the example of Figure 1, where `..` means that the query words must occur within five words of each other.

This collection was chosen to test our approach on advanced query operators, in this case, proximity search and boolean OR, and for some forms of query expansion, in this case, prefix search and error correction.

For the cache efficiency experiment on this collection, we used another query log of roughly 150,000 boolean queries.

Semantic Wikipedia. 2,843,056 articles from the English Wikipedia with an integrated ontology, as described in Bast et al. [2007a] and 500 semantic queries from Bast et al. [2007b], for example, `finland city`, where `city` does not only match the literal word but also all instances of that class. To compute these matches, information from several documents needs to be *joined*; for details, we refer to [Bast et al. 2007a, 2007b].

This collection was chosen to test our approach for queries that require joining information from two or more documents for each hit. As explained in Section 1.1, the document-based approach would not be able to produce query-dependent snippets in this case, because the hit document does not contain all the required information to assess which words or phrases did actually match the query.

TREC GOV2. 25,204,103 documents from the TREC GOV2 corpus[6], and 420,316 queries from the TREC 2006 efficiency track and all queries from the AOL query log which lead to at least one hit. An example query is `first landing moon`.

This collection was chosen to test the efficiency of our approach on a very large collection and for a large number of queries. Note that Turpin et al. experimented with the somewhat smaller wt100g collection (102GB of raw data) combined with the somewhat larger Excite query log (535,276 queries).

7.2. Space Consumption

We built the document database as described in Section 5, with each block covering 1000 positions, which amounts to an average of around 4 KB of compressed text. This gave the best snippet generation times overall.

Table II shows that the size of our document database is roughly equal to the size of the index used for query processing (which is of a different kind for each of the collections, see Section 7.1).

[6]`http://www-nlpir.nist.gov/projects/terabyte`

Table II. Size of the Data

Collection	Size	Index	Docs DB	Turpin
DBLP Plus	8.7 GB	0.47 GB	0.35 GB	0.42 GB
Wikipedia	12.7 GB	4.52 GB	3.41 GB	3.53 GB
GOV2	132.9 GB	38.2 GB	36.3 GB	45.8 GB

Note: With all tags, mark-up, and binary data (e.g., pdf) removed, size of the index (used for query processing), and size of the document databases (for snippet generation) for our implementation of the index-based approach (fourth column), and for Turpin et al.'s implementation of the document-based approach (fifth column), on all three test collections.

Table III. Breakdown of the Elapsed Snippet Generation Times of Our Implementation of the Index-Based Approach by Steps I1, I2, and I3

Collection	I0	I1	I2	I3
DBLP Plus	45 ms	0.2 ms	6.0 ms	13.7 ms
Wikipedia	113 ms	0.1 ms	51.6 ms	8.7 ms
GOV2	144 ms	0.9 ms	37.5 ms	12.2 ms

Note: Step I0 is the ordinary query processing, which provides the IDs of the top-ranked documents. All entries are averages per query, that is, the times in the last three columns relate to the generation of ten snippets at a time.

Our document database is slightly smaller than that described in Turpin et al. [2007]. This is because in our index-based approach, every block is compressed as a whole (using zlib), which compresses the original text[7] to about 27%. The ID-wise compression of Turpin et al. [2007], which we briefly described in Section 2, achieves a compression ratio of only about 35%. A part of our document database is auxiliary data needed to locate the document data within the single big file, as well as information on the segment bounds and the block offsets of each document (as before, for our experiments, we consider a simple segmentation into sentences). This auxiliary data takes about 5% of the total size of the document database for each of our three collections.

For comparison, the RLZ compression scheme from Hoobin et al. [2011] mentioned in Section 5.1 achieves a compression ratio of 9–16% and simultaneously somewhat faster random segment access.

7.3. Snippet-Generation Time

We ran our index-based snippet-generation implementation on all three test collections, for the queries described in Section 7.1. On the GOV2 collection, we compared it to the document-based implementation as well as to the GCL query algebra approach implemented in the Wumpus search engine. All experiments were carried out by computing the snippets of the top-10 retrieved documents.

The main observation from Table III is that the operator inversion from step (I1), which provides the basis for all other steps in the index-based approach, takes negligible time compared to all other steps. The bulk of the time spent for (I2) and (I3) is disk seek time, that is, the cost of our snippet generation is essentially the cost of the required disk seeks. Turpin et al. [2007] have come to the same conclusion for their document-based approach.

In fact, the table shows that, unlike query-processing times, snippet-generation times do not depend on the size and nature of the collection, but rather on the number of queries. We also measured a slight dependency on the average document length

[7]We measured the compression ratio with respect to the plain text, with all mark-up and tags removed. The compression ratio in Turpin et al. [2007] has been measured with respect to the size of the original html, pdf, etc. files.

Table IV. Elapsed Snippet Generation Times (without the use of a cache)

Ours	Turpin	Wumpus
46.6 ms	49.5 ms	396.7 ms

Note: Of our implementation of the index-based approach compared to Turpin et al.'s implementation of the document-based approach and to the GCL query algebra approach implemented in the Wumpus search engine on the TREC GOV2 collection. The reader may notice that I1 + I2 + I3 from the GOV2 row in Table III sums to 50.6 ms, which is 8.5% more than the 46.6 ms reported in Table IV. This is because the two running times were obtained from two different runs. Variation in that range is typical for IO-heavy code. In particular, the difference to the running time of Turpin et al.'s implementation should therefore be considered insignificant. In this context, recall that the index-based approach is more powerful than the document-based approach (see Section 1.3), but very slow in its straightforward implementation (see the results for Extended Lists in Table I and the results for Wumpus in Table IV). Our goal is to make the index-based approach as fast as the document-based approach as implemented by Turpin et al. (but not necessarily faster).

(which is largest for DBLP Plus: 32 kB, and smallest for GOV2: 10 kB), which influences the length of the list of segment bounds read in step (I2).

Table IV shows that for ordinary keyword queries, our implementation of the index-based approach is slightly faster than the document-based approach. This is because the index-based approach does not have to read the whole document from disk, but rather only those blocks containing the positions computed in steps I1 and I2. Note, however, that apart from caching, there is no way to be much faster for basic keyword search, but that the document-based approach simply cannot be used as is for more complex queries like those for DBLP Plus and Wikipedia.

Worst-case snippet-generation times are much higher for the document-based approach, for example, 1,453 of the 420,316 queries on GOV2 required more than 1 second (due to very large documents being involved), whereas for our index-based approach, snippet generation never exceeded 100 milliseconds.

The snippet-generation times for Wumpus were significantly higher than that of our implementation of the index-based approach. As explained in Section 2, this is because the underlying algorithm touches every posting of each query word, which is exactly what our operator inversion avoids. Put simply, one could thus say that we achieve the power of the index-based approach (also realized in Wumpus) at the computational cost that can be achieved by the (less powerful) document-based approach.

7.4. Caching

As explained in Section 6, the index-based approach allows caching on the level of individual segments, instead of on the level of whole documents, as is the case for the document-based approach. To evaluate the potential of this finer-granularity caching, we carried out experiments on both caching methods by using various cache sizes on the (relatively large) GOV2 and the (relatively small) DBLP collection. For each method, we used the first half of the queries to fill the cache and then measured the hit-ratio for the second half of the queries. The cache eviction strategy was LRU, as explained in Section 6.

Table V shows that the fine-grained segment caching results in significantly better cache hit-ratio than the coarse document caching, as predicted in Section 6.3. This is especially pronounced for small cache sizes for two reasons: first, our model predicts that the gap between the cache hit-ratios of the two caching methods becomes larger when the size of the cache is small; and second, on larger cache sizes, the top-segment cache requires a longer request stream than the whole-document cache to converge and reach its asymptotic hit-ratio, also predicted by our model. The number of queries in our case was simply too small to make full use of the segment cache with larger cache sizes. Namely, only 196MB were sufficient to store the snippets returned for all

Table V. Cache Hit-Ratios for the Two Caching Strategies and Four Different Cache Capacities on the TREC GOV2 and the DBLP Collection

Collection	Method	128 MB	256 MB	512 MB	1024 MB
GOV2	Segment cache	0.69 (128 MB)	0.74 (196 MB)	0.74 (196 MB)	0.74 (196 MB)
	Document cache	0.20 (128 MB)	0.28 (256 MB)	0.40 (512 MB)	0.56 (1024 MB)
DBLP	Segment cache	0.87 (40 MB)	0.87 (40 MB)	0.87 (40 MB)	0.87 (40 MB)
	Document cache	0.53 (128 MB)	0.64 (256 MB)	0.82 (512 MB)	1.00 (1024 MB)

Note: The numbers in parentheses give the total size of the cached items after all queries have been processed.

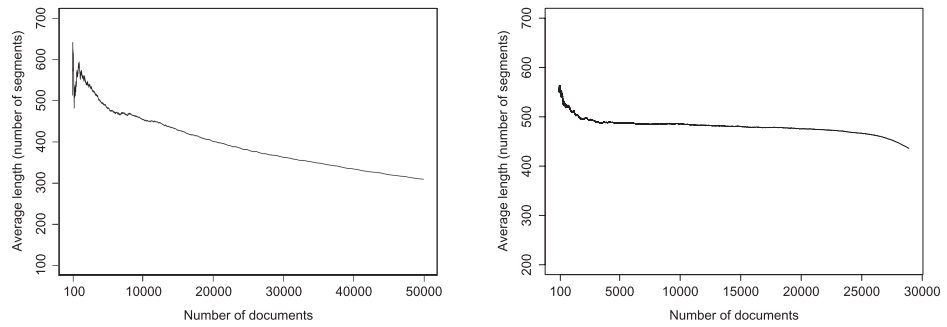

Fig. 8. Average document size of the first n most frequently accessed documents given in number of segments (sentences) on the TREC GOV2 collection (left) and the Wikipedia collection (right).

top-10 hits for all of our 420,316 queries on the TREC GOV2. For larger query logs, we would expect the cache hit-ratio of the segment cache to go up further.

Document ranking has an additional negative effect on document caching. To see this, recall that one of the assumptions in our model is that all documents have roughly the same size. However, Figure 8 shows that more frequently accessed documents are more likely to have larger sizes than less frequently accessed documents. This effect is especially pronounced for the TREC GOV2 collection.[8] As a result, the document cache fills up more quickly. Note that this is in accordance with the scope hypothesis in information retrieval stating that the likelihood of document relevance increases with document size [Losada et al. 2008]. For the segment cache, the distribution of document sizes is not an issue, since the segment sizes are independent of the document sizes.

8. CONCLUSIONS

We showed how to implement index-based snippet generation efficiently at the cost of only a slight increase in query time compared to computing the IDs of the relevant hits based on a standard positional index. We argue how the index-based approach is superior to the document-based approach in three practically relevant aspects: (i) nonliteral matches of query words with document text are not a problem, (ii) advanced query operators need not be re-implemented on the snippet-generation side, and (iii) snippet-generation times do not degenerate for large documents, because only those parts of the documents, containing the to-be-displayed snippets will be read.

We pointed out that index-based snippet generation allows for a more fine-grained caching than the standard document-based approach. We found that this more fine-grained caching gives a significantly larger cache hit-ratio, and we gave a simple

[8]We used a standard BM25 ranking.

theoretical explanation. It would be interesting to test the limits of the fine-grained caching on much larger query logs and collections.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their encouraging, competent, and constructive comments.

REFERENCES

Almeida, V., Bestavros, A., Crovella, M., and deOliveira, A. 1996. Characterizing reference locality in the WWW. Tech. rep., Boston University.

Anh, V. N. and Moffat, A. 2006. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 372–379.

Bast, H. and Weber, I. 2006. Type less, find more: Fast autocompletion search with a succinct index. In *Proceedings of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 364–371.

Bast, H. and Weber, I. 2007. The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration. In *Proceedings of the 3rd Conference on Innovative Data Systems Research (CIDR)*. VLDB Endowment, 88–95.

Bast, H., Majumdar, D., Schenkel, R., Theobald, M., and Weikum, G. 2006. IO-Top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 475–486.

Bast, H., Chitea, A., Suchanek, F., and Weber, I. 2007a. ESTER: Efficient search on text, entities, and relations. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 671–678.

Bast, H., Majumdar, D., and Weber, I. 2007b. Efficient interactive query expansion with CompleteSearch. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM)*. ACM, New York, NY, 857–860.

Breslau, L., Cue, P., Cao, P., Fan, L., Phillips, G., and Shenker, S. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*. IEEE Computer Society Press, Los Alamitos, CA, 126–134.

Buckley, C. 2004. Why current IR engines fail. In *Proceedings of the 27th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 584–585.

Buettcher, S. 2007. The Wumpus search engine. http://www.wumpus-search.org/.

Ceccarelli, D., Lucchese, C., Orlando, S., Perego, R., and Silvestri, F. 2011. Caching query-biased snippets for efficient retrieval. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*. ACM, New York, NY, 93–104.

Celikik, M. and Bast, H. 2009. Fast error-tolerant search on very large texts. In *Proceedings of the Symposium of Applied Computing (SAC)*. ACM, New York, NY, 1724–1731.

Clarke, C. L., Cormack, G. V., and Burkowski, F. J. 1995. An algebra for structured text search and a framework for its implementation. *Comput. J. 38*, 1, 43–56.

Clarke, C. L. A. and Cormack, G. V. 2000. Shortest-substring retrieval and ranking. *Trans. Inf. Syst. 18*, 1, 44–78.

Cutting, D. 2004. Lucene. http://lucene.apache.org/.

Goldstein, J., Kantrowitz, M., Mittal, V., and Carbonell, J. 1999. Summarizing text documents: Sentence selection and evaluation metrics. In *Proceedings of the 22nd Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 121–128.

Hoobin, C., Puglisi, S. J., and Zobel, J. 2011. Relative Lempel-Ziv factorization for efficient storage and retrieval of Web collections. *Proc. VLDB Endow. 5*, 3, 265–273.

Ko, Y., An, H., and Seo, J. 2007. An effective snippet generation method using the pseudo relevance feedback technique. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 711–712.

Losada, D. E., Azzopardi, L., and Baillie, M. 2008. Revisiting the relationship between document length and relevance. In *Proceedings of the 17th Conference on Information and Knowledge Management (CIKM)*. ACM, New York, NY, 419–428.

Manolache, G. 2008. Index-based snippet generation. M.S. thesis, Saarland University.

Tombros, A. and Sanderson, M. 1998. Advantages of query biased summaries in information retrieval. In *Proceedings of the 21st Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 2–10.

Tsegay, Y., Puglisi, S. J., Turpin, A., and Zobel, J. 2009. Document compaction for efficient query-biased snippet generation. In *Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval (ECIR)*. Lecture Notes in Computer Science, vol. 5478. Springer-Verlag, Berlin, 509–520.

Turpin, A., Tsegay, Y., Hawking, D., and Williams, H. E. 2007. Fast generation of result snippets in Web search. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 127–134.

Varadarajan, R. and Hristidis, V. 2006. A system for query-specific document summarization. In *Proceedings of the 15th Conference on Information and Knowledge Management (CIKM)*. ACM, New York, NY, 622–631.

Wang, D., Li, T., Zhu, S., and Ding, C. 2008. Multi-document summarization via sentence-level semantic analysis and symmetric matrix factorization. In *Proceedings of the 31st Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 307–314.

White, R. W., Ruthven, I., and Jose, J. M. 2002. Finding relevant documents using top ranking sentences: An evaluation of two alternative schemes. In *Proceedings of the 25th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, New York, NY, 57–64.

White, R. W., Jose, J. M., and Ruthven, I. 2003. A task-oriented study on the influencing effects of query-biased summarisation in Web searching. *Inf. Process. Manage. 39*, 5, 707–733.

Received August 2012; revised February 2013, December 2013; accepted December 2013