Type Less, Find More: Fast Autocompletion Search with a Succinct Index

Holger Bast
Max-Planck-Institut für Informatik
Saarbrücken, Germany
bast@mpi-inf.mpg.de

Ingmar Weber
Max-Planck-Institut für Informatik
Saarbrücken, Germany
iweber@mpi-inf.mpg.de

ABSTRACT

We consider the following full-text search autocompletion feature. Imagine a user of a search engine typing a query. Then with every letter being typed, we would like an instant display of completions of the last query word which would lead to good hits. At the same time, the best hits for any of these completions should be displayed. Known indexing data structures that apply to this problem either incur large processing times for a substantial class of queries, or they use a lot of space. We present a new indexing data structure that uses no more space than a state-of-the-art compressed inverted index, but with 10 times faster query processing times. Even on the large TREC Terabyte collection, which comprises over 25 million documents, we achieve, on a single machine and with the index on disk, average response times of one tenth of a second. We have built a full-fledged, interactive search engine that realizes the proposed autocompletion feature combined with support for proximity search, semi-structured (XML) text, subword and phrase completion, and semantic tags.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing Methods; H.3.3 [**Content Analysis and Indexing**]: Retrieval Models; H.5.2 [**User Interfaces**]: Theory and Methods

General Terms

Algorithms, Design, Experimentation, Human Factors, Performance, Theory

Keywords

Autocompletion, Empirical Entropy, Index Data Structure

1. INTRODUCTION

Autocompletion is a widely used mechanism to get to a desired piece of information quickly and with as little knowledge and effort as possible. One of its early uses was in the Unix Shell, where pressing the tabulator key gives a list of all file names that start with whatever has been typed on the command line after the last space. Nowadays, we find a similar feature in most text editors, and in a large variety of browsing GUIs, for example, in file browsers, in the Microsoft Help suite, or when entering data into a web form. Recently, autocompletion has been integrated into a number of (web and desktop) search engines like Google Suggest or Apple's Spotlight. We discuss more applications in Section 1.2.

In the simpler forms of autocompletion, the list of completions is simply a range from a (typically precomputed) list of words. For the Unix Shell, this is the list of all file names in all directories listed in the PATH variable. For the text editors, this is the list of all words entered into the file so far (and maybe also words from related files). In Google Suggest, completions appear to come from a precompiled list of popular queries. For these kinds of applications we can easily achieve fast response times by two binary or B-tree searches in the (pre)sorted list of candidate strings.

More advanced forms of autocompletion take into account the *context* in which the to-be-completed word has been typed. The problem we propose and discuss in this paper is of this kind. The formal problem definition will be given in Section 2. More informally, imagine a user of a search engine typing a query. Then with every letter being typed, we would like an instant display of completions of the last query word *which would lead to good hits*. At the same time, the best hits for any of these completions should be displayed. All this should preferably happen in less time than it takes to type a single letter. For example, assume a user has typed `conference sig`. Promising completions might then be `sigir`, `sigmod`, etc., but not, for example, `signature`, assuming that, although `signature` by itself is a pretty frequent word, the query `conference signature` leads to only few good hits. See Figure 1 for a screenshot of our search engine responding to that query. For a live demo, see `http://search.mpi-inf.mpg.de/wikipedia`.

1.1 Our results

We have developed a new indexing data structure, named HYB, which uses no more space than a state-of-the-art compressed inverted index, and which can respond to autocompletion queries as described above within a small fraction of a second, even for collection sizes in the Terabyte range.

Our main competitor in this paper is the inverted index, referred to as INV in the following. Other data structures that could be directly applied to our problem either use a lot of space or have other limitations; we discuss these in Section 1.2.

We give a rigorous mathematical analysis of HYB and INV with respect to both space usage and query processing times. Our analysis accurately predicts the real behavior on our test collections.

Concerning space usage, we define a notion of *empirical*

SIGIR'06, August 6–11, 2006, Seattle, Washington, USA.

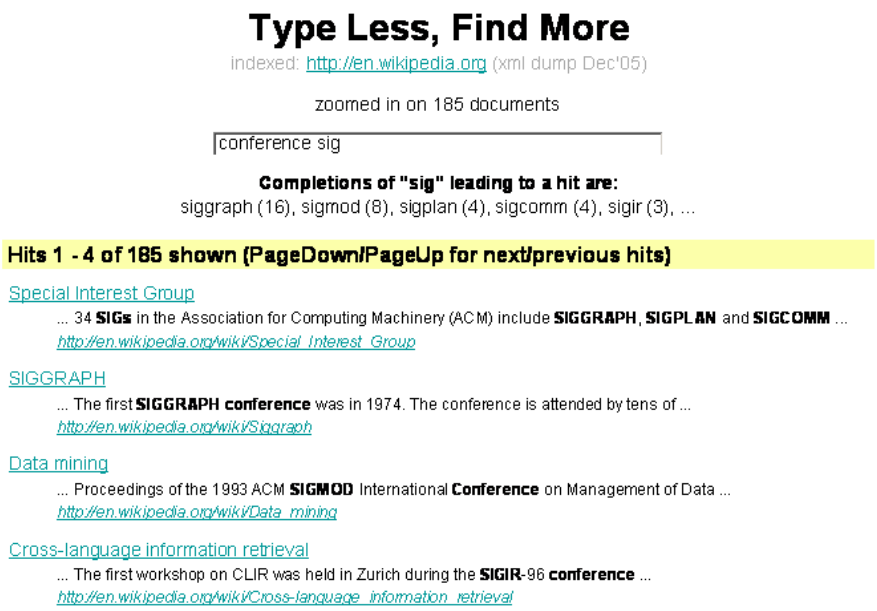

Figure 1: A screenshot of our search engine for the query `conference sig` searching the English Wikipedia. The list of completions and hits is updated automatically and instantly after each keystroke, hence the absence of any kind of search button. The number in parentheses after each completion is the number of hits that would be obtained if that completion where typed. Query words need not be completed, however, because the search engine does an implicit prefix search: if, for example, the user continued typing `conference sig proc`, completions and hits for `proc`, e.g., `proceedings`, would be from the 185 hits for `conference sig`.

entropy [11] [22], which captures the inherent space complexity of an index independent of a particular compression scheme. We prove that the empirical entropy of HYB is essentially equal to that of INV, and we find that the actual space usage of our implementation of the two index structures is indeed almost equal, for each of our three test collections.

Concerning processing times, we give a precise quantification of the number of operations needed, from which we derive bounds for the worst, best, and average-case behavior of INV and HYB. We also take into account the different latencies of sequential and random access to data [1].

We compare INV and HYB on three test collections with different characteristics. One of our collections has been (semi-)publicly searchable over the last year, so that we have autocompletion queries from real users for it. Our largest collection is the TREC Terabyte benchmark with over 25 million documents [7].

On all three collections and on all the queries we considered, HYB outperforms INV by a factor of $15 - 20$ in worst-case query processing time, and by a factor of $3 - 10$ in average case query processing time. In absolute terms, HYB achieves average query processing of one tenth of a second or less on all collections, on a single machine and with the index on disk (and not in main memory).

We have built a full-fledged search engine that supports autocompletion queries of the described kind combined with support for proximity/phrase search, XML tags, subword and phrase completion, and category information. All of these extensions are described in Section 4.

1.2 Related work

The autocompletion feature as described so far is reminiscent of *stemming*, in the sense that by stemming, too, prefixes instead of full words are considered [23]. But unlike stemming, our autocompletion feature gives the user feedback on which completions of the prefix typed so far would lead to highly ranked documents. The user can then assess the relevance of these completions to his or her search desire, and decide to (i) type more letters for the last query word, e.g., in the query from Figure 1, type `i` and `r` so that the query is then `conference sigir`, or to (ii) start with the next query word, e.g., type a space and then `proc`, or to (iii) stop searching as , e.g., the user was actually looking for one of the hits shown in Figure 1. There is no way to achieve this by a stemming preprocessing step, because there is no way to foresee the user's intent. This kind of user interaction is well known to improve retrieval effectiveness in a variety of situations [21].

While our autocompletion feature is for the purpose of *finding information*, autocompletion has also been employed for the purpose of *predicting user input*, for example, for typing messages with a mobile phone, for users with disabilities concerning typing, or for the composition of standard letters [6] [14] [20] [8] [15]. In [12], contextual information has been used to select promising extensions for a query. Paynter et al. have devised an interface with a zooming-in property on the word level, and based on the identification of frequent phrases [18]. We get a related feature by the subword/phrase-completion mechanism described in Section 4.4.

Our autocompletion problem is related to but distinctly different from *multi-dimensional range searching problems*, where the collection consists of tuples (of some fixed dimension, for example, pairs of word prefixes), and queries are asking for all tuples that match a given tuple of ranges [10] [2] [4] [13]. These data structures could be used for our autocompletion problem, provided that we were willing to limit the number of query words. For fast processing times, however, the space consumption of any of these structures is on the order of N^{1+d}, where N is the size of an inverted index, and $d > 0$ grows (fast) with the dimension. For our au-

tocompletion queries, we can achieve fast query processing times and space efficiency at the same time because we have the set of documents matching the part of the query before the last word already computed (namely when this part was being typed). In a sense, our autocompletion problem is therefore a 1 1/2 - dimensional range searching problem.

Finally, there is a large variety of alternatives to the inverted index in the literature. We have considered those we are aware of with regard to their applicability to our autocompletion problem, but found them either unsuitable or inferior to the inverted index in that respect. For example, approaches that consider *document by document* are bound to be slow due to a poor locality of access; in contrast, both INV and HYB are mostly *scanning* long lists; see Section 3. *Signature files* were found to be in no way superior (but significantly more complicated) to the inverted index in all major respects in [24]. *Suffix arrays* and related data structures address the issue of *full substring search*, which is not what we want here (but see Section 4.4); a direct application of a data structure like [11] would have the same efficiency problems as INV, whereas multi-dimensional variants like [10] require super-linear space, as explained above.

2. FORMAL PROBLEM DEFINITION AND DEFINITION OF EMPIRICAL ENTROPY

The following definition of our autocompletion problem takes neither positional information, nor ranking of the completions or of the documents into account. We will first, in Section 3, analyze our data structures for this basic setting. In Section 4, we then show how to generalize the data structures and their analysis to cope with positional information, ranking, and a number of other useful enhancements. This generalization will be straightforward.

DEFINITION 1. *An autocompletion query is a pair (D, W), where W is a range of words (all possible completions of the last word which the user has started typing) and D is a set of documents (the hits for the preceding part of the query). To process the query means to compute the subset $W' \subseteq W$ of words that occur in at least one document from D, as well as the subset $D' \subseteq D$ of documents that contain at least one of these words.*

For our example `conference sig`, D is the set of all documents containing a word starting with `conference` (computed when the last letter of this word was typed), and W is the range of *all* words from the collection starting with `sig`. For queries with only a single word, e.g., `confer`, D is simply the set of *all* documents.

To analyze the inherent space complexity of INV and HYB independently of the specialties of a particular compression scheme, we introduce a notion of *empirical entropy*. Both INV and HYB are essentially a collection of (multi)sets and sequences. The following definition gives a natural notion of entropy for each such building block, and for arbitrary combinations of them (similar definitions have been made in [11] [22]). The reader might first want to skip the following definition and come back to it when it is first used in the analysis that follows.

DEFINITION 2. *We define* empirical entropy *for the following entities, where $\mathcal{H}(p_1, \ldots, p_l) = -\sum_{i=1}^{l}(p_i \cdot \log_2 p_i)$ is the l-ary entropy function.*

(a) For a subset *of size n' with elements from a universe of size n, the empirical entropy is $n \cdot \mathcal{H}(n'/n, 1-n'/n)$ (include each element of the universe into the subset with probability n'/n), which is*

$$n' \cdot \log_2 \frac{n}{n'} + (n - n') \cdot \log_2 \frac{n}{n - n'} .$$

(b) For a multisubset *of size n' with elements from a universe of size n, the empirical entropy is $(n+n') \cdot \mathcal{H}(n'/(n+n'), n/(n+n'))$ (consider a bitvector of size $n+n'$, and let a bit be 0 with probability $n'/(n+n')$ and 1 otherwise; the prefix sums at the 0-bits give the multisubset), which is*

$$n' \cdot \log_2 \frac{n+n'}{n'} + n \cdot \log_2 \frac{n+n'}{n} .$$

(c) For a sequence *of n elements from a universe of size l, where the ith element occurs n_i times ($n_1 + \cdots + n_l = n$), the empirical entropy is $n \cdot \mathcal{H}(n_1/n, \ldots, n_l/n)$ (for each position, pick element i with probability n_i/n), which is*

$$n_1 \cdot \log_2 \frac{n}{n_1} + \cdots + n_l \cdot \log_2 \frac{n}{n_l} .$$

(d) For a collection of l entities with empirical entropies $\mathcal{H}_1, \ldots, \mathcal{H}_l$, the empirical entropy is simply $\mathcal{H}_1 + \cdots + \mathcal{H}_l$.

3. INV, HYB, AND THEIR ANALYSIS

In this section we will describe INV and HYB, and analyze them with respect to their empirical entropy and their processing time for autocompletion queries according to Definition 1. Query processing times will be quantified in terms of all relevant parameters; from this we can easily derive worst-case, best-case, and average-case bounds. Our average-case bounds make simplifying assumptions on the distribution of words in the documents, but nevertheless turn out to predict the actual behavior quite well. Implementation issues and the actual performance of our implementations of INV and HYB will be discussed in Section 5. We briefly comment on index construction times in Section 3.3

3.1 The inverted index (INV)

The inverted index is the data structure of choice for most search applications: it is relatively easy to implement and extend by other features, it can be compressed well, it is very efficient for short queries, and it has an excellent locality of access [23]. In this paper, by INV we mean the following data structure: for each word store the list of all (ids of) documents containing that word, sorted in ascending order. We do not consider enhancements such as skip pointers [17], which we would expect to give similar benefits for both INV and HYB, however at the price of an increased space usage. In the following, we first estimate the inherent space efficiency (empirical entropy) of INV. We then analyze the time complexity of processing autocompletion queries with INV, and point out two inherent problems.

LEMMA 1. *Consider an instance of INV with n documents and m words, and where the ith words occurs in n_i distinct documents (so that $n_1 + \cdots + n_m$ is the total number of word-in-document pairs). Let $\mathcal{H}_{inv}$ be the empirical entropy according to Definition 2. Then*

$$\mathcal{H}_{inv} \;\le\; \sum_{i=1}^{m} \left(n_i \cdot \frac{1}{\ln 2} + n_i \cdot \log_2 \frac{n}{n_i} \right),$$

and for all collections considered in this paper (where most n_i are much smaller than n) this bound is tight up to 2%.

PROOF. According to Definition 2 (a) and (d), we have

$$\mathcal{H}_{inv} \;=\; \sum_{i=1}^{m} \left(n_i \cdot \log_2 \frac{n}{n_i} + (n - n_i) \cdot \log_2 \frac{n}{n - n_i} \right) .$$

To prove the lemma, it suffices to observe that because $1 + x \le e^x$ for any real x,

$$(n - n_i) \cdot \log_2 \frac{n}{n - n_i} = \frac{n - n_i}{\ln 2} \cdot \ln \left(1 + \frac{n_i}{n - n_i} \right) \le \frac{n_i}{\ln 2}.$$

□

Lemma 1 tells us that if the documents in each list were picked uniformly at random, then a *Golomb-encoding of the gaps* [23] from one document id to the next (for list i, the expected size of a gap would be n/n_i) would achieve a space usage very close to $\mathcal{H}_{inv}$ bits. In our implementation, we opted to encode gaps with the Simple-9 encoding from [3], which is easy to implement, yet achieves very fast decompression speeds at the price of only a moderate loss in compression efficacy; details are reported in Section 5.

LEMMA 2. *With INV, an autocompletion query (D, W) can be processed in the following time, where D_w denotes the inverted list for word w:*

$$|D| \cdot |W| + \sum_{w \in W} |D_w| + \sum_{w \in W} |D \cap D_w| \cdot \log |W|.$$

Assuming that the elements of W, D, and the D_w are picked uniformly at random from the set of m words and the set of n documents, respectively, this bound has an expected value of

$$|D| \cdot |W| + \frac{|W|}{m} \cdot N + \frac{|D|}{n} \cdot \frac{|W|}{m} \cdot N \cdot \log |W|.$$

Remark. By picking the elements of a set S at random from a set U, we mean that each subset of U of size $|S|$ is equally likely for S. We are *not* making any randomness assumption on the *sizes* of W, D, and D_w above.

PROOF SKETCH. The obvious way to use an inverted index to process an autocompletion query (D, W) is to compute, for each $w \in W$, the intersections $D \cap D_w$. Then, W' is simply the set of all w for which the intersection was non-empty, and D' is the union of all (non-empty) intersections. The intersections can be computed in time linear in the total input volume $\sum_{w \in W}(|D| + |D_w|)$.[1] The union can be computed by a $|W|$-way merge, which requires on the order of $\log |W|$ time per element scanned. With the randomness assumptions, the expected size of D_w is N/m, and the expected size of $|D \cap D_w|$ is $|D|/n \cdot N/m$. □

Lemma 2 highlights two problems of INV. The first is that the term $|D| \cdot |W|$ can become prohibitively large: in the worst case, when D is on the order of n (i.e., the first part of the query is not very discriminative) and W is on the order of m (i.e., only few letters of the last query word have been typed), the bound is on the order of $n \cdot m$, that is, quadratic in the collection size. The second problem is due to the required merging. While the volume $\sum_{w \in W} |D \cap D_w|$ will typically be small once the first query word has been completed, it will be large for the first query word, especially when only few letters have been typed. As we will see in Section 5, INV frequently takes seconds for some queries, which is quite undesirable in an interactive setting, and is exactly what motivated us to develop a more efficient index data structure.

3.2 Our new data structure (HYB)

The basic idea behind HYB is simple: *precompute inverted lists for unions of words.* Assume an autocompletion query (D, W), where the union of all lists for word range W have been precomputed. We would then get D' with a single intersection (of D with the precomputed list). However, from this precomputed list alone we can no longer infer the set W' of completions leading to a hit. Since W can be an arbitrary word range, it is also not clear which unions should be precomputed, especially when we do not want to use more space than an (optimally compressed) inverted index.

The analysis given in this section suggests the following approach: group the words in blocks so that the lengths of the inverted lists in each block sum to (approximately) $c \cdot n$, for some constant $c < 1$ (we will later choose $c \approx 0.2$). For each block, store the union of the covered inverted lists as a compressed *multiset*, using an effective gap encoding scheme just as done for INV (repetitions of the same element in the multiset correspond to a gap of zero). In parallel to each multiset, for each element x store the id of the word that led to the inclusion of (this occurrence of) x in the multiset. This gives a sequence of word ids, the length of which is exactly the size of the multiset. Encode these word ids with code length (approximately) $\log_2((n_1 + \cdots + n_l)/n_i)$ for the ith word, where n_i is the number of documents containing the ith word, and l is the number of words in the respective block.

Here is an example. Let one of the blocks comprise four words A, B, C, and D, with inverted lists

A	:	3, 5, 6, 8, 9, 11, 12, 15
B	:	5, 11
C	:	3, 7, 11, 13
D	:	3, 8

We would then like to store, in compressed form, the multiset (of document ids) and the sequence (of word ids)

3	3	3	5	5	6	7	8	8	9	11	11	11	12	13	15
A	C	D	A	B	A	C	A	D	A	A	B	C	A	C	A

The optimal encoding of the words A, B, C, D would use code lengths $\log_2(16/8) = 1$, $\log_2(16/2) = 3$, $\log_2(16/4) = 2$, $\log_2(16/2) = 3$, respectively, for example $A = 0$, $B = 110$, $C = 10$, $D = 111$. An optimal encoding of the four gaps 0, 1, 2, 3 that occur in the above multiset of document ids would be 0, 10, 110, 111, respectively. What we actually store are then the two bit vectors (where the | are solely for better readability; the codes in this example are prefix-free)

111|0|0|110|0|10|10|10|0|10|110|0|0|10|10|110
0|10|111|0|110|0|10|0|111|0|0|110|10|0|10|0

Note that due to the two different encodings the two lists end up having different lengths in compressed form, and this is also what will happen in reality.

The following analysis will make very clear that (i) one should choose blocks of equal list volume (and not, for example, of equal number of words), (ii) this volume should be a small but substantial fraction of the number of documents (and neither smaller nor larger), and (iii) the lists of document ids should be "gap-encoded" while the lists of word ids should be "entropy-encoded".

As for the space usage, we will first derive a very tight estimate of the entropy of HYB, and then show that, somewhat surprisingly, if we only choose the block volume to be a small enough fraction of the number of documents, the entropy of HYB is almost exactly that of INV.

We will then show how HYB, when the blocks are chosen of sufficiently large volume, can be used to process autocompletion queries in time linear in the number of documents, for any reasonable word range. Since HYB essentially scans long lists, without the need for any merging, except when the word range is huge, it also has an excellent locality of access.

LEMMA 3. *Consider an instance of HYB with n words and m documents, where the ith word occurs in n_i documents, and where for each block the sum of the n_i with i from that block is $c \cdot n$, for some $c > 0$. Then the empirical*

[1] There are asymptotically faster algorithms for the intersection of two lists [5], but in our experiments, we got the best results with the simple linear-time intersect, which we attribute to its compact code and perfect locality of access.

entropy $\mathcal{H}_{hyb}$, defined according to Definition 2, satisfies

$$\mathcal{H}_{hyb} \;\le\; \sum_{i=1}^{m} \left(n_i \cdot \frac{1+c/2}{\ln 2} + n_i \cdot \log_2 \frac{n}{n_i} \right),$$

and the bound is tight as $c \to 0$.

PROOF. Consider a fixed block of HYB, and let n_i denote the number of documents containing the ith word belonging to that block. Throughout this proof, let $\sum_i n_i$ denote the sum over all these n_i (so that the sum over all $\sum_i n_i$ from all blocks gives the $\sum_{i=1}^{m} n_i$ from the lemma). According to Definition 2 (b), (c), and (d), the empirical entropy of this block is then

$$\textstyle\sum_i n_i \cdot \log_2 \frac{n+\sum_i n_i}{\sum_i n_i} + n \cdot \log_2 \frac{n+\sum_i n_i}{n} + \sum_i n_i \log_2 \frac{\sum_i n_i}{n_i}.$$

Now adding the first and the last term, the arguments of the logarithms partially cancel out (!), and we get

$$\textstyle\sum_i n_i \cdot \log_2 \frac{n+\sum_i n_i}{n_i} \;+\; n \cdot \log_2 \frac{n+\sum_i n_i}{n}.$$

Now using that, by assumption, $\sum_i n_i = c \cdot n$, we obtain

$$\textstyle\sum_i n_i \left((1+1/c)\log_2(1+c) + \log_2 \frac{n}{n_i} \right).$$

Since $(1+1/c)\ln(1+c) \le 1+c/2$ for all $c > 0$ (not obvious, but true), we can upper bound this (tightly, as $c \to 0$) by

$$\textstyle\sum_i n_i \left(\frac{1+c/2}{\ln 2} + \log_2 \frac{n}{n_i} \right).$$

This bounds the empirical entropy of a single block of HYB (the sum goes over all words from that block). Adding this over all blocks gives us the bound claimed in the lemma. □

Comparing Lemma 3 with Lemma 1, we see that if we let the blocks of HYB be of volume at most $c \cdot n$, for some small fraction c, then the empirical entropy of HYB is essentially that of an inverted index. In Section 4.2, we will see that when we take positional information into account, the empirical entropy of HYB actually becomes *less* than that of INV, for any choice of block volumes.

In our implementation of HYB, we compress the lists of document ids by a Simple-9 encoding of the gaps, just as described for INV above. For the lists of word ids, entropy-optimal compression could be achieved by arithmetic encoding [23], but for efficiency reasons, we compress word ids as follows: assuming that the word frequencies in a block have a Zipf-like distribution, it is not hard to see that a *universal encoding* with $\sim \log x$ bits for number x [17] of the *ranks* of the words, if sorted in order of descending frequency, is entropy-optimal, too. We again opted for Simple-9 encoding of these ranks, which gives us a reasonable compression and very fast decompression speed, without the need for any large codebook. We take block sizes as $n/5$, but also take word/prefix boundaries into account such that frequent prefixes like `pro`, `com`, `the` get a block on their own. This is to avoid that a query unnecessarily spans more than one block.

LEMMA 4. *Using HYB with blocks of volume N', autocompletion queries (D, W) can be processed in the following time, where D_w is the inverted list for word w*

$$\sum_{w \in W} |D_w| \cdot (1 + |D|/N') + \sum_{w \in W} |D \cap D_w| \cdot \log \left(\sum_{w \in W} |D_w|/N' \right).$$

For $N' = \Theta(n)$ and $|W| \le m \cdot n/N$, and assuming that the elements of D, D_w, and W are picked uniformly at random from the set of all n documents or all m words, respectively, the expected *processing time is bounded by $O(n)$.*

PROOF SKETCH. According to Definition 1, we have to compute, given (D, W), the set W' of words from W contained in documents from D, as well as the set D' of documents containing at least one such word. For each block B, a straightforward intersection of the given D with the list of document-word pairs from B, gives us the set W'_B of all words from W' from block B, as well as the set D'_B of all document from D' which contain a word from B. From these, D' can be computed by a k-way merge, where k is the number of blocks that contain a word from W, and W' can be computed by a simple linear-time sort into W buckets (because W is a range). The number k of blocks is $\sum_w |D_w|/N'$, which is $O(1)$ in expectation, given the randomness assumptions stated in the lemma. □

3.3 Index construction time

While getting from a collection of documents (files) to INV is essentially a matter of one big external sort [23], HYB does not require a full inversion of the data. For our experiments, however, we built the compressed indices for both INV and HYB from an intermediate fully inverted text version of the collection, which takes essentially the same time for both.

4. EXTENSIONS

In this section, we describe a number of extensions of the basic autocompletion facility we have described and analyzed so far. The first (ranking) is essential for practical usability, the second (proximity search) greatly widens the spectrum of search tasks for which autocompletion can be useful, and the others (support for XML tags, subword and phrase completion, and semantic tags) give advanced search facilities to the expert searcher.

4.1 Ranking

So far, we have considered the following problem (from Definition 1): while the user is typing a query, compute after each keystroke the list of *all* completions of the last query word that lead to at least one hit, as well as the list of *all* hits that would be obtained by any of these completions.

In practice, only a *selection* of items from these lists can and will be presented to the user, and it is of course crucial that the most relevant completions and hits are selected.

A standard approach for this task in ad-hoc retrieval is to have a precomputed *score* for each word-in-document pair, and when a query is being processed, to aggregate these scores for each candidate document, and return documents with the highest such aggregated scores [23].

Both INV and HYB can be easily adapted to implement any such scoring and aggregation scheme: store by each word-in-document pair its precomputed score, and when intersecting, aggregate the scores. A decision has to be made on how to reconcile scores from different completions within the same document. We suggest the following: when merging the intersections (which gives the set D' according to Definition 1), compute for each document in D' the *maximal* score achieved for some completion in W' contained in that document, and compute for each completion in W' the *maximal* score achieved for a hit from D' achieved for this completion.

Asymptotically, the inclusion of ranking does not affect the time bounds derived in Lemmas 2 and 4, and our experiments show that ranking never takes more than half of the total query processing time; see Section 5.4. The increase in space usage depends on the selected scoring scheme, and is the same for INV and HYB. It is for these reasons, that we factored out the ranking aspect from our basic Definition 1

and from our space and time complexity analysis in Section 3.

4.2 Proximity/Phrase searches

With a properly chosen scoring function, such as BM25, mere ranking by score aggregation often gives very satisfactory precision/recall behavior [19]. There are many queries, however, where the decisive cue on whether a particular document is relevant or not lies in the fact whether certain of the query words occur *close to each other* in that document. See [16] for a recent positive result on the use of proximity information in ad-hoc retrieval.

Our autocompletion feature increases the benefits of a proximity operator, because the use of this operator will strongly narrow down the list of completions displayed to the user, which in turn makes it easier for the user to filter out irrelevant completions. For example, when searching the Wikipedia collection the most relevant completion for the non-proximity query `max pl` would be `place` (because `max` and `place` are both frequent words), but for the proximity query `max..pl` it is `planck`. Here the two dots `..` indicate that words should occur within x words of each other, for some user-definable parameter x.

It is not hard to extend both INV and HYB to support proximity search: in the document lists (INV and HYB) as well as in the word lists (HYB only), we duplicate each entry as many times as it occurs in the corresponding document, and store the positions in a parallel array of the same size. Word and document lists are compressed just as before, and the lists of positions are gap-encoded by Simple-9, just like the lists of document ids. The intersection routine is adapted to consider a proximity window as an additional parameter.

As we will see in Section 5.3, the position lists increase the index size by a factor of 4-5, for both INV and HYB (without any kind of stopword removal). We can extend our analysis from Section 3 to predict this factor as follows. If we replace n_i, the number of documents containing the ith word, by N_i, the total number of occurrences of the ith words, and n, the number of documents, by N, the total number of word occurrences, we can show that (details omitted)

$$\mathcal{H}_{hyb^*} \;\leq\; \mathcal{H}_{inv^*} \;\leq\; \sum_{i=1}^{m} \left(N_i/\ln 2 + N_i \cdot \log_2(N/N_i)\right),$$

where $\mathcal{H}_{inv^*}$ and $\mathcal{H}_{hyb^*}$ denote the empirical entropy of INV and HYB, respectively, with positional information. That is, with positional information, HYB is always more space-efficient than INV, *irrespectively of how we divide into blocks*. It can be shown that $N_i \cdot \log_2(N/N_i) \approx 2N_i \cdot \log_2(n/n_i)$, and since on average a word occurs about 2-3 times in a document, this is just 4-5 times $n_i \log_2(n/n_i)$, which was the corresponding term in the entropy bound for INV or HYB without positional information (Lemmas 1 and 3).

4.3 Semistructured (XML) text

Many documents contain semantic information in the form of tag pairs. We briefly sketch how we can make good use of such tags in our autocompletion scenario. Assume that in the archive of a mailing list, the subject of a mail is enclosed in a `<subject>` ... `<\subject>` tag pair. We can then easily implement an operator `=` (in a way very similar to our implementation of the proximity operator), such that for a query `subject=sig` only those completions of `sig` are displayed which actually occur in the subject line of a mail, and only such documents are displayed as hits.

4.4 Completion to subwords and phrases

Another simple yet often useful extension to the basic autocompletion feature is to consider as potential matches not only the words as they occur in the collection, but also meaningful subwords and phrases. An example involving a subword: for the query `normal..vec` we might want to see `eigenvector` as one of the relevant completions. An example involving a phrase: for the query `max planck` we might want to see the phrase `max planck institute` as one of the relevant completions. It is not hard to see, that the autocompletion according to Definition 1 will automatically provide this feature if only we *add* the corresponding subwords/phrases to the index.

4.5 Category information

Our autocompletion feature can be combined with a number of other technologies that enhance the semantics of a corpus. To give just one more example here, assume we have *tagged* all conference names in a collection. Then assume we duplicate all conference names in the index, with an added prefix of, say, `conf:`, e.g., `conf:sigir`. By the way our autocompletion works, for the query `seattle conf:` we would then get a list of all names of conferences that occur in documents that also mention Seattle.

5. EXPERIMENTS

We implemented both INV and HYB in compressed format, as described in Sections 3.1 and 3.2. Each index is stored in a single file with the individual lists concatenated and an array of list offsets at the end. The vocabulary (which is the same for INV as for HYB) is stored in a separate file. All our code is in C++. All our experiments were run on a Dual Opteron machine, with 2 Intel Xeon 3 GHz processors, 8 GB of main memory, and running Linux. We ensured that the index was not cached in main memory.

5.1 Test collections

We compared the performance of INV and HYB on three collections of different characteristics. The first collection is a mailing-list archive plus several encyclopedias on homeopathic medicine (`www.homeonet.org`). This collection has been searchable via our engine over the past year by an audience of several hundred people. The second collection consists of the complete dumps of the English and German Wikipedia from December 2005 (`search.mpi-inf.mpg.de/wikipedia`). The third collection is the large TREC Terabyte collection [7], which served as a stress test for our index structures (and for the authors as well). Details about all three collection are given in Table 1, where the "Raw size" of a collection is the total size of the original, uncompressed files in their original formats (e.g., HTML or PDF).

5.2 Queries

For the Homeopathy collection, we picked 5,732 maximal queries (that is, queries, which are not a true prefix of another query) from a fixed time slice of our query log for that collection. From each of these maximal queries, a sequence of autocompletion queries was generated by "typing" the query from left to right, with a minimal prefix length of 3. Like that, for example, the maximal query `acidum phos` gives rise to the 6 autocompletion queries `aci`, `acid`, `acidu`, `acidum`, `acidum pho`, and `acidum phos`. For the Wikipedia collection, autocompletion queries were generated in the same manner from a set of 100 randomly generated queries, with a distribution of the number of query words and of the term frequency similar to that of the real queries for the Homeopathy collection. For the Terabyte collection, autocompletion queries were generated, in again the same way but with a minimal prefix length of 4, from the (stemmed) 50 ad-hoc queries of the Robust Track Benchmark [7], e.g., `squirrel control protect`. For all three collections, we removed queries containing words that had no completion at all in the respective collection.

For Homeopathy and Wikipedia, all queries were run as proximity queries (using a full positional index according to Section 4.2), while for Terabyte, they were executed as ordinary document-level queries. For all collections, both completions and hits were ranked as we described it in Section 4.1 (details of the aggregation function are omitted here).

Each autocompletion query was processed according to Definition 1, e.g., for `acidum pho`, we compute all completions of `pho` that occur in a document which also contains a word starting with `acidum`, as well as the set of all such documents. The result for each autocompletion query is remembered in a *history*, so that we do not need to recompute the set of documents matching the first part of the query. E.g., when processing `acidum pho`, we can take the set of documents matching `acidum` from the history; see the explanation following Definition 1.

The autocompletion queries with minimal prefix lengths, like `aci` and `acidum pho` for Homeopathy, are the most difficult ones. All other queries can be easily processed by what we call *filtering*. For example, both the completions and hits for the query `acidum phos` can be obtained by retrieving the list of matching word-in-document pairs for the previously processed query `acidum pho` from the history, and by filtering out, in a linear scan over that list, all those pairs, where the word starts with `phos`. In practice, this is always faster than processing such queries as full autocompletion queries according to Definition 1. Note that this filtering is identical for INV and HYB. We nevertheless include the filtered queries in our experiments, because in reality we will always get a mix of both kinds of queries. Table 3 will provide figures for just the difficult (unfiltered) queries. We remark that the history is useful also for caching purposes, but in our experiments we used it solely for the purpose of filtering.

5.3 Index space

Table 1 shows that INV and HYB use essentially the same space on all three test collections, and that HYB is slighter more compact than INV for a full positional index. This is exactly what Lemmas 1 and 3, and the derivation in Section 4.2 predicted! The sizes for both INV and HYB exceed that predicted by the empirical entropy by about 50%. This is due to our use of the Simple-9 compression scheme, which trades very fast decompression time for about this increase in space usage [3]. A combination of Golomb and arithmetic encoding would give us a space usage closer to the empirical entropy. However, decompression would then become the computational bottleneck for almost all queries, see Table 3. We remark that, by the way we did our analysis, any new compression scheme with improved compression ratio/decompression speed profile, would immediately yield a corresponding improvement for both INV and HYB.

5.4 Query processing time

Table 2 shows that in terms of query processing time, HYB outperforms INV by a large margin on all collections. With respect to maximum processing time, which is especially critical for an interactive application, the improvement is by a factor of 15-20. With respect to average processing time, which is critical for throughput in a high-load scenario, the improvement is by a factor of 3-10.

Table 3 gives interesting insights into where exactly INV loses against HYB. The table shows a breakdown of the running times of those queries for the Terabyte collection, which were not answered by filtering as discussed above. (Note that the breakdown of the filtered queries would be identical for both methods.) The table differentiates between 1-*word queries* like `squi`, `squir`, etc. and *multi-word queries* like `squirrel contr` or `squirrel control prot`.

For the 1-word queries, no intersections have to be computed for either INV or HYB. According to Lemma 2, the merging of the intersections then dominates for INV, and this indeed shows in the first column of Table 3. For multi-word queries, the result volume $\sum_w |D \cap D_w|$ (Lemmas 2 and 4) goes down, and, according to Lemma 2, the intersection costs dominate for INV, which shows in the third column of Table 3. In contrast, columns two and four demonstrate that HYB achieves a better balance of the costs for reading, uncompressing, and intersecting, and none of these essential operations becomes the bottleneck. HYB avoids merging altogether since, by construction, the potential completions from the given word range W always lie within a single block.

The read time of HYB is about 50% larger than that of INV, because HYB always reads a whole block of size $\Theta(n)$, even for small word ranges. This also partially explains why HYB spends more time decompressing than INV; the other factor is that decompression of the word ids is more expensive than decompression of the document ids. As we remarked in Section 4.1, the absolute time for ranking is

Collection	Homeopathy	Wikipedia	Terabyte
Raw size	**452 MB**	**7.4 GB**	**426 GB**
#documents	44,015	2,866,503	25,204,103
#words	263,817	6,700,119	25,263,176
#items	12 [27] million	0.3 [0.8] billion	3.5 billion
Vocabulary	2.9 MB	73 MB	239 MB
Entropy	6.6 [13.1] bits	9.1 [14.0] bits	8.4 bits
INV			
index size	**13 [70] MB**	**0.5 [2.2] GB**	**4.6 GB**
-per item	9.3 [21.5] bits	12.8 [23.2] bits	11.0 bits
HYB			
index size	**14 [62] MB**	**0.5 [2.0] GB**	**4.9 GB**
-per item	9.4 [19.2] bits	13.0 [20.7] bits	11.6 bits
-per doc	3.9 [15.4] bits	4.3 [14.8] bits	5.9 bits
-per word	5.5 [3.8] bits	8.7 [5.9] bits	5.7 bits

Table 1: Properties of our three test collections, and the space consumption of INV versus HYB. The entries in square brackets are for a full positional index, without any word whatsoever removed.

Collection	Method	mean	90%	99%	max
Homeopathy	INV	0.033	0.038	0.384	12.27
	HYB	0.003	0.008	0.026	0.065
Wikipedia	INV	0.171	0.143	2.272	22.66
	HYB	0.055	0.158	0.492	1.085
Terabyte	INV	0.581	0.545	16.83	28.84
	HYB	0.106	0.217	0.865	1.821

Table 2: Average, 90%-ile, 99%-ile and maximum processing times in seconds for INV versus HYB on our three test collections.

the same for both methods. Ranking takes more time on average for the 1-word queries, because these tend to have larger result sets. The comparison with the time needed for the maintenance of the history, which is nothing but memory allocation and copying, shows that all of HYB's operation are essentially fast list scans.

Query size	1-word		multi-word	
Index type	INV	HYB	INV	HYB
average time	0.340 secs	0.244 secs	2.404 secs	0.207 secs
read	.024 7%	.048 20%	.032 1%	.045 22%
decompress	.011 3%	.032 13%	.023 1%	.045 22%
intersect	——	——	2.27 94%	.030 14%
merging	.165 48%	——	.010 .4%	——
ranking	.081 24%	.083 34%	.007 .3%	.007 4%
history	.058 17%	.082 32%	.062 3%	.079 38%

Table 3: Breakdown of average processing times for INV and HYB, for the difficult (unfiltered) queries on Terabyte.

6. CONCLUSIONS

We have introduced an autocompletion feature for full-text search, and presented a new compact indexing data structure for supporting this feature with very fast response times. We have built a full-fledged search engine around this feature, and we have given arguments, why we believe it to be practically useful. Given the interactivity of this engine, the next logical step following this work would be to conduct a *user study* for verifying that belief. We also see potential for a further speed-up of query processing time by applying techniques from top-k query processing [9], in order to display the most relevant hits and completions without first computing and ranking *all* of them.

7. ACKNOWLEDGEMENTS

Many thanks to our mentor David Grossman for his encouragement and many valuable comments.

8. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *41st Symposium on Foundations of Computer Science (FOCS'00)*, pages 198–207, 2000.

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8:151–166, 2005.

[4] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *18th Symposium on Principles of Database Systems (PODS'99)*, pages 346–357, 1999.

[5] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. *Lecture Notes in Computer Science*, 3109:400–408, 2004.

[6] S. Bickel, P. Haider, and T. Scheffer. Learning to complete sentences. In *16th European Conference on Machine Learning (ECML'05)*, pages 497–504, 2005.

[7] C. L. A. Clarke, N. Craswell, and I. Soboroff. The TREC terabyte retrieval track. *SIGIR Forum*, 39(1):25, 2005.

[8] J. J. Darragh, I. H. Witten, and M. L. James. The reactive keyboard: A predictive typing aid. *IEEE Computer*, pages 41–49, 1990.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[10] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. *Journal of Computer and System Science*, 66(4):763–774, 2003.

[11] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[12] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin. Placing search in context: The concept revisited. In *10th International World Wide Web Conference (WWW10)*, pages 406–414, 2001.

[13] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[14] K. Grabski and T. Scheffer. Sentence completion. In *27th Conference on Research and Development in Information Retrieval (SIGIR'04)*, pages 433–439, 2004.

[15] M. Jakobsson. Autocompletion in full text transaction entry: a method for humanized input. In *Conference on Human Factors in Computing Systems (CHI'86)*, pages 327–323, 1986.

[16] D. Metzler, T. Strohman, H. Turtle, and W. B. Croft. Indri at TREC 2004: Terabyte track. In *13th Text Retrieval Conference (TREC'04)*, 2004.

[17] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

[18] G. W. Paynter, I. H. Witten, S. J. Cunningham, and G. Buchanan. Scalable browsing for large collections: A case study. In *5th Conference on Digital Libraries (DL'00)*, pages 215–223, 2000.

[19] S. E. Robertson, S. Walker, M. M. Beaulieu, M. Gatford, and A. Payne. Okapi at TREC-4. In *4th Text Retrieval Conference (TREC'95)*, pages 73–96, 1995.

[20] T. Stocky, A. Faaborg, and H. Lieberman. A commonsense approach to predictive text entry. In *Conference on Human Factors in Computing Systems (CHI'04)*, pages 1163–1166, 2004.

[21] E. M. Voorhees. Query expansion using lexical-semantic relations. In *17th Conference on Research and Development in Information Retrieval (SIGIR'94)*, pages 171–180, 1994.

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

[23] I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edition*. Morgan Kaufmann, 1999.

[24] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.