# Output-Sensitive Autocompletion Search

Holger Bast   Christian W. Mortensen
Ingmar Weber

MPI–I–2006–1–007   June 2006

## Authors' Addresses

Holger Bast
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Christian Worm Mortensen
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen
Denmark

Ingmar Weber
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

## Publication Notes

This in an extended version with full proofs of the paper "Output-Sensitive Autocompletion Search", written by the same authors and published by Springer in the proceedings of the 13th Symposium on String Processing and Information Retrieval (SPIRE '06).

## Abstract

We consider the following autocompletion search scenario: imagine a user of a search engine typing a query; then with every keystroke display those completions of the last query word that would lead to the best hits, and also display the best such hits. The following problem is at the core of this feature: for a fixed document collection, given a set $D$ of documents, and an alphabetical range $W$ of words, compute the set of all word-in-document pairs $(w, d)$ from the collection such that $w \in W$ and $d \in D$. We present a new data structure with the help of which such autocompletion queries can be processed, on the average, in time linear in the input plus output size, independent of the size of the underlying document collection. At the same time, our data structure uses no more space than an inverted index. Actual query processing times on a large test collection correlate almost perfectly with our theoretical bound.

## Keywords

Autocompletion Search, Information Retrieval

# 1 Introduction

Autocompletion, in its most basic form, is the following mechanism: the user types the first few letters of some word, and either by pressing a dedicated key or automatically after each key stroke a procedure is invoked that displays all relevant words that are continuations of the typed sequence. The most prominent example of this feature is the tab-completion mechanism in a Unix shell. In the recently launched Google Suggest service frequent queries are completed. Algorithmically, this basic form of autocompletion merely requires two simple string searches to find the endpoints of the range of corresponding words.

## 1.1 Problem definition

The problem we consider in this paper is derived from a more sophisticated form of autocompletion, which takes into account the *context* in which the to-be-completed word has been typed. Here, we would like an (instant) display of only those completions of the last query word which lead to hits, as well as a display of such hits. For example, if the user has typed `search autoc`, context-aware completions might be `autocomplete` and `autocompletion`, but not `autocratic`. The following definition formalizes the core problem in providing such a feature.

**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 set of all word-in-document pairs $(w, d)$ with $w \in W$ and $d \in D$.*

Given an algorithm for solving autocompletion queries according to this definition, we obtain the context-sensitive autocompletion feature as follows:

For the example query `search autoc`, $W$ would be all words from the vocabulary starting with `autoc`, and $D$ would be the set of all hits for the query `search`. The output would be all word-in-document pairs $(w, d)$, where $w$ starts with `autoc` and $d$ contains $w$ as well as a word starting with `search`. [1]

Now if the user continues with the last query word, e.g., `search autoco`, then we can just filter the sequence of word-in-document pairs from the previous queries, keeping only those pairs $(w', d')$, where $w'$ starts with `autoc`. If, on the other hand, she starts a new query word, e.g., `search autoc pub`, then we have another autocompletion query according to Definition 1, where now $W$ is the set of all words from the vocabulary starting with `pub`, and $D$ is the set of all hits for `search autoc`. For the very first query word, $D$ is the set of all documents.

In practice, we are actually interested in the *best* hits and completions for a query. This can be achieved by the following standard approach. Assume we

[1] We always assume an implicit prefix search, that is, we are actually interested in hits for all words *starting* with `search`, which is usually what one wants in practice. Whole-word-only matching can be enforced by introducing a special end of word symbol `$`.

have precomputed scores for each word-in-document pair. Given a sequence of pairs $(w, d)$ according to Definition 1, we can then easily compute for each word $w'$ occurring in that sequence an aggregate of the scores of all pairs $(w', d)$ from that sequence, as well as for each document $d'$ an aggregate of the scores of all pairs $(w, d')$. The precomputation of scores for word-in-document pairs such that these aggregations reflect user-perceived relevance to the given query is a much-researched area in information retrieval [21], and beyond the scope of this paper. It is for these reasons that the ranking issue is factored out of Definition 1.

To answer a series of autocompletion queries, we can obtain the new set of candidate documents $D$ from the sequence of matching word-in-document pairs for the last query by sorting the matching $(w, d)$ pairs. This sort takes time $O((\sum_{w \in W} |D \cap D_w|) \log(\sum_{w \in W} |D \cap D_w|))$ and would in practice be done together with the ranking of the completions and documents. The time for this sort is also included in the running times of our experiments in Section 6, but is dominated by the work to find all matching word-in-document pairs.

## 1.2 Main result

**Theorem 2.** *Given a collection with $n \geq 16$ documents, $m$ distinct words, and $N \geq 32m$ word-in-document pairs* [2]*, there is a data structure* AUTOTREE *with the following properties:*

*(a)* AUTOTREE *can be constructed in $O(N)$ time.*

*(b)* AUTOTREE *uses at most $N\lceil \log_2 n \rceil$ bits of space (which is the space used by an ordinary uncompressed inverted index)*[3].

*(c)* AUTOTREE *can process an autocompletion query $(D, W)$ in time*

$$O\left((\alpha + \beta)|D| + \sum_{w \in W} |D \cap D_w|\right),$$

*where $D_w$ is the set of documents containing word $w$. Here $\alpha = N|W|/(mn)$, which is bounded above by $1$, unless the word range is very large (e.g., when completing a single letter). If we assume that the words in a document with $L$ words are a random size-$L$ subset of all words, $\beta$ is at most $2$ in expectation. In our experiments, $\beta$ is indeed around $2$ on the average and about $4$ in the (rare) worst case. Our analysis implies a general worst-case bound of $\log(mn/N)$.*

Note that for constant $\alpha$ and $\beta$ the running time is asymptotically optimal, as it takes $\Omega(|D|)$ time to inspect all of $D$ and it takes $\Omega(\sum_{w \in W} |D \cap D_w|)$ time to output the

[2]The conditions on $n$ and $N$ are technicalities and are satified for any realistic document collection.

[3]Strictly speaking, an uncompressed inverted index needs even more space, to store the list lengths.

result. The necessary ingredients for the proof of Theorem 2 are developed in the next sections and they are finally assembled in Section 5.

We implemented AUTOTREE , and in Section 6 show that its processing time correlates almost perfectly with the bound from Theorem 2(c) above. In that Section, we also compare it to an inverted index, its presumably closest competitor (see Section 1.4), which AUTOTREE outperforms by a factor of 10 in worst-case processing time (which is key for an interactive feature), and by a factor of 4 in average-case processing time.

## 1.3 Related work

This technical report is an extended version of [3], with full proofs. The problem is derived from a search engine, which we have devised and implemented, and which is described in [4]; for a live demo, see `http://search.mpi-inf.mpg.de/wikipedia`. The emphasis in [4] is on usability (of the autocompletion feature) and on compressibility (of the data), and not on designing an output-sensitive algorithm. The data structures and algorithms in [4] are completely different from those presented in this article.

The most straightforward way to process an autocompletion query $(D, W)$ would be to explicitly search each document from $D$ for occurrences of a word from $W$. However, this would give us a non-constant query processing time per element of $D$, completely independent of the respective $|W|$ or output size $\sum_{w \in W} |D \cap D_w|$. For these reasons, we do not consider this approach further in this paper. Instead, our baseline in this paper is based on an inverted index, the data structure underlying most (if not all) large-scale commercial search engines [21]; see Section 1.4.

Definition 1 looks reminiscent of multi-dimensional search problems, where the collections consists of tuples (of some fixed dimensionality), and queries are asking for all tuples contained in a tuple of given ranges [12, 2, 10, 1]. Provided that we are willing to limit the number of query words, such data structures could indeed be used to process our autocompletion queries. If we want fast processing times, however, any of the known data structures uses space on the order of $N^{1+d}$, where $N$ is the number of word-in-document pairs in the collection, and $d$ grows (fast) with the dimensionality. In the description of our data structures we will point out some interesting analogies to the geometric range-search data structures from [6] and [15].

The large body of work on string searching concerned with data structures such as PAT/suffix tree/arrays [13, 9] is not directly applicable to our problem. Instead, it can be seen as orthogonal to the problem we are discussing here. Namely, in the context of our autocompletion problem these data structures would serve to get from mere prefix search to full substring search. For example, our Theorem 2 could be enhanced to full substring search by first building a suffix data structure like that of [9], and then building our data structure on top of the sorted list of all suffixes (instead of the list of the distinct words).

There is a large body of more applied work on algorithms and mechanisms for *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 [14, 7, 19, 5]. In [11], contextual information has been used to select promising extensions for a query; the emphasis of that paper is on the quality of the extensions, while our emphasis here is on efficiency. An interesting, somewhat related phrase-browsing feature has been presented in [18, 17]; in that work, emphasis was on the identification of frequent phrases in a collection.

### 1.4 The BASIC scheme and outline of the rest of the paper

The following BASIC scheme is our baseline in this paper. It is based on the *inverted index* [21], for which we simply precompute for each word from the collection the list of documents containing that word. For an efficient query processing, these lists are typically sorted, and we assume a sorting by document number.

Having precomputed these lists, BASIC processes an autocompletion query $(D, W)$ very simply as follows: For each word $w \in W$, fetch the list $D_w$ of documents that contain $w$, compute the intersection $D \cap D_w$, and append it to the output.

**Lemma 3.** BASIC *uses time at least* $\Omega(\sum_{w \in W} \min\{|D|, |D_w|\})$ *to process an autocompletion query* $(D, W)$*. The inverted lists can be stored using a total of at most* $N \cdot \lceil \log_2 n \rceil$ *bits, where* $n$ *is the total number of documents, and* $N$ *is the total number of word-in-document pairs in the collection.*

*Proof.* BASIC computes one intersection for each $w \in W$ and any algorithm for intersecting $D$ and $D_w$ has to differentiate between $2^{\min\{|D|,|D_w|\}}$ possible outputs. For the space usage, it suffices to observe that the elements of the inverted lists, are just a rearrangement of the sets of distinct words from all documents, and that each document number can be encoded with $\lceil \log_2 n \rceil$ bits (we do not consider issues of compression in this paper). □

Lemma 3 points out the inherent problem of BASIC : its query processing time depends on the size of both $|D|$ *and* $|W|$, and it can become $|D| \cdot |W|$ in the worst case.

In the following sections, we develop a new indexing scheme AUTOTREE , with the properties given in Theorem 2. A combination of four main ideas will lead us to this new scheme: a tree over the words (Section 2), relative bit vectors (Section 3), pushing up the words (Section 4), and dividing into blocks (Section 5). In Section 6, we will complement our theoretical findings with experiments on a large test collection.

## 2 Building a tree over the words (TREE)

The idea behind our first scheme on the way to Theorem 2 is to *increase the amount of preprocessing by precomputing inverted lists not only for words but also for their prefixes*. More precisely, we construct a complete binary tree with $m$ leaves, where

$m$ is the number of distinct words in the collection. We assume here and throughout the paper that $m$ is a power of two. For each node $v$ of the tree, we then precompute the sorted list $D_v$ of documents which contain at least one word from the subtree of that node. The lists of the leaves are then exactly the lists of an ordinary inverted index, and the list of an inner node is exactly the union of the lists of its two children. The list of the root node is exactly the set of all non-empty documents. A simple example is given in Figure 1.

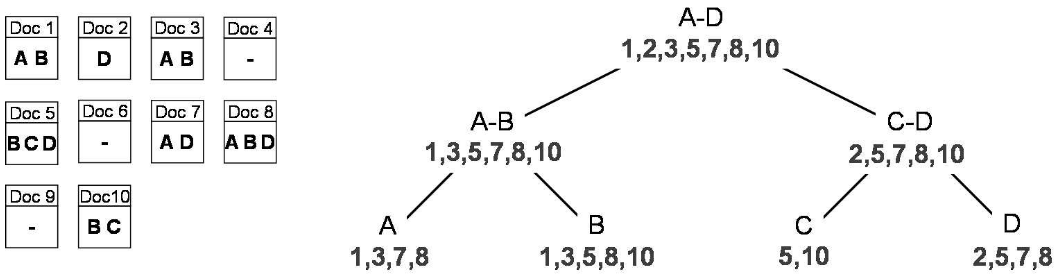

Figure 1: Toy example for the data structure of scheme TREE with 10 documents and 4 different words.

Given this tree data structure, an autocompletion query given by a word range $W$ and a set of documents $D$ is then processed as follows.

1. Compute the unique minimal sequence $v_1, \ldots, v_\ell$ of nodes with the property that their subtrees cover exactly the range of words $W$. Process these $\ell$ nodes from left to right, and for each node $v$ invoke the following procedure.

2. Fetch the list $D_v$ of $v$ and compute the intersection $D \cap D_v$. If the intersection is empty, do nothing. If the intersection is non-empty, then if $v$ is a leaf corresponding to word $w$, report for each $d \in D \cap D_v$ the pair $(w, d)$. If $v$ is not a leaf, invoke this procedure (step 2) recursively for each of the two children of $v$.

Scheme TREE can potentially save us time: If the intersection computed at an inner node $v$ in step 2 is empty, we know that none of the words in the whole subtree of $v$ is a completion leading to a hit, that is, *with a single intersection we are able to rule out a large number of potential completions*. However, if the intersection at $v$ is non-empty, we know nothing more than that there is *at least one word* in the subtree which will lead to a hit, and we will have to examine both children recursively. The following lemma shows the potential of TREE to make the query processing time depend on the output size instead of on $W$ as for BASIC . Since TREE is just a step on the way to our final scheme AUTOTREE , we do not give the exact query processing time here but just the number of nodes visited, because we need exactly this information in the next section.

**Lemma 4.** *When processing an autocompletion query $(D, W)$ with TREE, at most $2(|W'|+1)\log_2|W|$ nodes are visited, where $W'$ is the set of all words from $W$ that occur in at least one document from $D$.*

*Proof.* A node at height $h$ has at most $2^h$ nodes below it. So each of the nodes $v_1, \ldots, v_l$ has height at most $\lfloor \log_2 |W| \rfloor$. Further, no three nodes from $v_1, \ldots, v_l$ have identical height, which implies that $l \leq 2\lfloor \log |W| \rfloor$. Similarly, for each word in $W'$ we need to visit at most two additional nodes, each at height below $\lfloor \log |W| \rfloor$. $\square$

The price TREE pays in terms of space is large. In the worst case, each level of the tree would use just as much space as the inverted index stored at the leaf level, which would give a blow-up factor of $\log_2 m$.

## 3 Relative Bitvectors (TREE+BITVEC)

In this section, we describe and analyze TREE+BITVEC, which reduces the space usage from the last section, while maintaining as much as possible of its potential for a query processing time depending on $W'$, the set of matching completions, instead of on $W$. *The basic trick will be to store the inverted lists via relative bit vectors*. The resulting data structure turns out to have similarities with the static 2-dimensional orthogonal range counting structure of Chazelle [6].

In the root node, the list of all non-empty documents is stored as a bit vector: when $N$ is the number of documents, there are $N$ consecutive bits, and the $i$th bit corresponds to document number $i$, and the bit is set to 1 if and only if that document contains at least one word from the subtree of the node. In the case of the root node this means that the $i$th bit is 1 if and only if document number $i$ contains any word at all.

Now consider any one child $v$ of the root node, and with it store a vector of $N'$ bits, were $N'$ is the number of 1-bits in the parent's bit vector. To make it interesting already at this point in the tree, assume that indeed some documents are empty, so that not all bits of the parent's bit vector are set to one, and $N' < N$. Now the $j$th bit of $v$ corresponds to the $j$th 1-bit of its parent, which in turn corresponds to a document number $i_j$. We then set the $j$th bit of $v$ to 1 if and only if document number $i_j$ contains a word in the subtree of $v$.

The same principle is now used for every node $v$ that is not the root. Constructing these bit vectors is relatively straightforward; it is part of the construction given in Section 4.1.

**Lemma 5.** *Let $s_{tree}$ denote the total lengths of the inverted lists of algorithm TREE. The total number of bits used in the bit vectors of algorithm TREE+BITVEC is then at most $2s_{tree}$ plus the number of empty documents (which cost a 0-bit in the root each).*

*Proof.* The lemma is a consequence of two simple observations. The first observation is that wherever there was a document number in an inverted list of algorithm

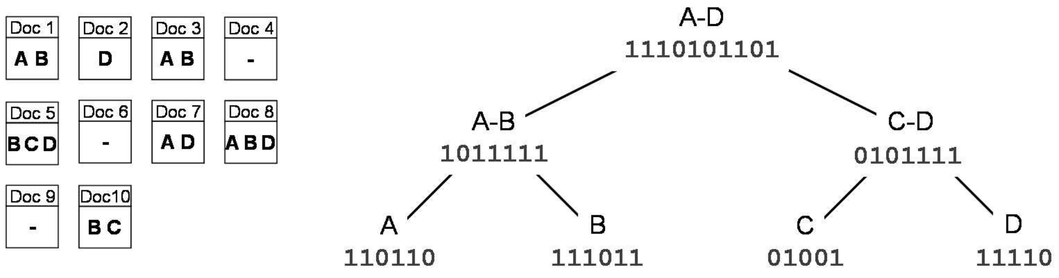

Figure 2: The data structure of TREE+BITVEC for the toy collection from Figure 1.

TREE there is now a 1-bit in the bit vector of the same node, and this correspondence is $1-1$. The total number of 1-bits is therefore $s_{tree}$.

The second observation is that if a node $v$ that is not the root has a bit corresponding to some document number $i$, then the parent node also has a bit corresponding to that same document, and that bit of the parent is set to 1, since otherwise node $v$ would not have a bit corresponding to that document.

It follows that the nodes, which have a bit corresponding to a particular fixed document, form a subtree that is not necessarily complete but where each inner node has degree 2, and where 0-bits can only occur at a leaf. The total number of 0-bits pertaining to a fixed document is hence at most the total number of 1-bits for that same document plus one. Since for each document we have as many 1-bits at the leaves as there are words in the documents, the same statement holds without the plus one.

$\square$

The procedure for processing a query with TREE+BITVEC is, in principle, the same as for TREE. The only difference comes from the fact that the bit vectors, except that of the root, can only be interpreted relative to their respective parents.

To deal with this, we ensure that whenever we visit a node $v$, we have the set $\mathcal{I}_v$ of those positions of the bit vector stored at $v$ that correspond to documents from the given set $D$, as well as the $|\mathcal{I}_v|$ numbers of those documents. For the root node, this is trivial to compute. For any other node $v$, $\mathcal{I}_v$ can be computed from its parent $u$: for each $i \in \mathcal{I}_u$, check if the $i$th bit of $u$ is set to 1, if so compute the number of 1-bits at positions less than or equal to $i$, and add this number to the set $\mathcal{I}_v$ and store by it the number of the document from $D$ that was stored by $i$. With this enhancement, we can follow the same steps as before, except that we have to ensure now that whenever we visit a node that is not the root, we have visited its parent before. The lemma below shows that we have to visit an additional number of up to $2\log_2 m$ nodes because of this.

**Lemma 6.** *When processing an autocompletion query $(D, W)$ with TREE+BITVEC, at most $2(|W'|+1)\log_2|W| + 2\log_2 m$ nodes are visited, with $W'$ defined as in Lemma 4.*

*Proof.* By Lemma 4, at most $2(|W'|+1)\log_2|W|$ nodes are visited in the subtrees of the nodes $v_1,\ldots,v_l$ that cover $W$. It therefore remains to bound the total number of nodes contained in the paths from the root to these nodes $v_1,\ldots,v_l$.

First consider the special case, where $W$ starts with the leftmost leaf, and extends to somewhere in the middle of the tree. Then each of the $v_1,\ldots,v_l$ is a left child of one node of the path from the root to $v_l$. The total number of nodes contained in the $l$ paths from the root to each of $v_1,\ldots,v_l$ is then at most $d-1$, where $d$ is the depth of the tree. The same argument goes through for the symmetric case when the *range ends with the rightmost leaf.*

In the general case, where $W$ begins at some intermediate leaf and ends at some other intermediate leaf, there is a node $u$ such that the leftmost leaf of the range is contained in the left subtree of $u$ and the rightmost leaf of the range is contained in the right subtree of $u$. By the argument from the previous paragraph, the paths from $u$ to those nodes from $v_1,\ldots,v_l$ lying in the left subtree of $u$ then contain at most $d_u-1$ different nodes, where $d_u$ is the depth of the subtree rooted at $u$. The same bound holds for the paths from $u$ to the other nodes from $v_1,\ldots,v_l$, lying in the right subtree of $u$. Adding the length of the path from the root to $u$, this gives a total number of at most $2d-3$

$\square$

## 4 Pushing Up the Words (TREE+BITVEC+PUSHUP)

The scheme TREE+BITVEC+PUSHUP presented in this section gets rid of the $\log_2|W|$ factor in the query processing time from Lemma 6. *The idea is to modify the TREE+BITVEC data structure such that for each element of a non-empty intersection, we find a new word-in-document pair $(w,d)$ that is part of the output.* For that we store with each single 1-bit, which indicates that a particular document contains a word from a particular range, one word from that document and that range. We do this in such a way that each word is stored only in one place for each document in which it occurs. When there is only one document, this leads to a data structure that is similar to the priority search tree of McCreight, which was designed to solve the so-called 3-sided dynamic orthogonal range-reporting problem in two dimensions [15].

Let us start with the root node. Each 1-bit of the bit vector of the root node corresponds to a non-empty document, and we store by that 1-bit the *lexicographically smallest* word occurring in that document. Actually, we will not store the word but rather its number, where we assume that we have numbered the words from $0,\ldots,m-1$.

More than that, for all nodes at depth $i$ (i.e., $i$ edges away from the root), we omit the leading $i$ bits of its word number, because for a fixed node these are all identical and can be computed from the position of the node in the tree. However, asympotically this saving is not required for the space bounds in Theorem 2 as dividing the words into blocks will already give a sufficient reduction of the space needed for the word numbers.

Now consider anyone child $v$ of the root node, which has exactly one half $H$ of all words in its subtree. The bit vector of $v$ will still have one bit for each 1-bit of its parent node, but the definition of a 1-bit of $v$ is slightly different now from that for TREE+BITVEC. Consider the $j$th bit of the bit vector of $v$, which corresponds to the $j$th set bit of the root node, which corresponds to some document number $i_j$. Then this document contains at least one word — otherwise the $j$th bit in the root node would not have been set — and the number of the lexicographically smallest word contained is stored by that $j$th bit. Now, if document $i_j$ contains other words, and at least one of these *other* words is contained in $H$, only then the $j$th bit of the bit vector of $v$ is set to 1, and we store by that 1-bit *the lexicographically smallest word contained in that document that has not already been stored in one of its ancestors* (here only the root node).

Figure 3 explains this data structure by a simple example. The construction of the data structure is relatively straightforward and can be done in time $O(N)$. Details are given in Section 4.1.

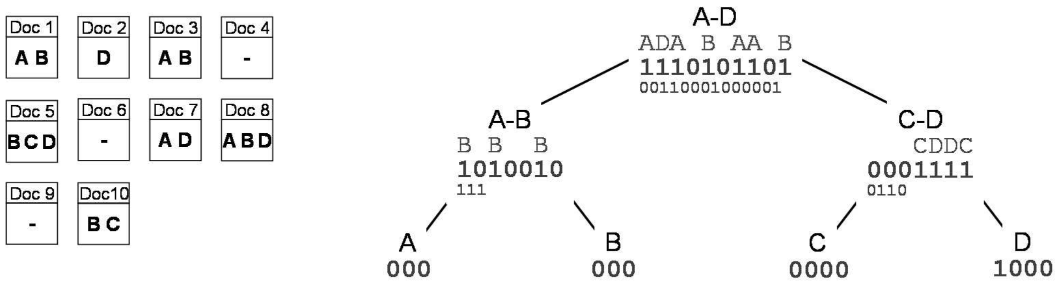

Figure 3: The data structure of TREE+BITVEC+PUSHUP for the example collection from Figure 1. The large bitvector in each node encodes the inverted list. The words stored by the 1-bits of that vector are shown in gray on top of the vector. The word list actually stored is shown below the vector, where A=00, B=01, C=10, D=11, and for each node the common prefix is removed, e.g., for the node marked C-D, C is encoded by 0 and D is encoded by 1. A total of 49 bits is used, not counting the redundant 000 vectors and bookkeeping information like list lengths etc.

To process a query we start at the root. Then, we visit nodes in such an order that whenever we visit a node $v$, we have the set $\mathcal{I}_v$ of exactly those positions in the bit vector of $v$ that correspond to elements from $D$ (and for each $i \in \mathcal{I}_v$ we know its corresponding element $d_i$ in $D$). For each such position with a 1-bit, we now check whether the word $w$ stored by that 1-bit is in $W$, and if so output $(w, d_i)$. This can be implemented by random lookups into the bit vector in time $O(|\mathcal{I}_v|)$ as follows. First, it is easy to intersect $D$ with the documents in the root node, because we can simply lookup the document numbers in the bitvector at the root. Consider then a child $v$ of the root. What we want to do is to compute a new set $I_v$ of document indices, which gives the numbering of the document indices of $D$ in terms of the numbering used in $v$. This amounts to counting the number of 1-bits in the bitvector of $v$ up to a given sequence of indices. Each of these so-called *rank* computations can be performed in constant time with an auxiliary data structure that uses space sublinear in the size of the bitvector [16].

Consider again the check whether a word $w$ stored by a 1-bit corresponding to a document from $D$ is actually in $W$. This check can only fail for relatively few nodes, namely those with at least one leaf not from $W$ in their subtree. These checks do not contribute an element to the output set, and are accounted for by the factor $\beta$ mentioned in Theorem 1, and Lemmas 7 and 9 below.

**Lemma 7.** *With TREE+BITVEC+PUSHUP, an autocompletion query* $(D, W)$ *can be processed in time* $O\big(|D| \cdot \beta + \sum_{w \in W} |D \cap D_w|\big)$, *where* $\beta$ *is bounded by* $\log_2 m$ *as well as by the average number of distinct words in a document from D. For the special case, where* $W$ *is the range of* all *words, the bound holds with* $\beta = 1$.

*Proof.* As we noticed above, the query processing time spent in any particular node $v$ can be made linear in the number of bits inspected via the index set $\mathcal{I}_v$. Recall that each $i \in \mathcal{I}_v$ corresponds to some document from $D$. Then for reasons identical to those that led to the space bound of Lemma 5, for any fixed document $d \in D$, the set of all visited nodes $v$ which have an index in their $\mathcal{I}_v$ corresponding to $d$ form a binary tree, and it can only happen for the leaves of that tree that the index points to a 0-bit, so that the number of these 0-bits is at most the number of 1-bits plus one.

Let again $v_1, \ldots, v_l$ denote the at most $2 \log_2 m$ nodes covering the given word range $W$ (see Section 2). Observe that, by the time we reach the first node from $v_1, \ldots, v_l$, the index set $\mathcal{I}_v$ will only contain indices from $D'$, as all the 1-bits for these nodes correspond to a word in $W'$. Strictly speaking, this is only guaranteed after the intersection with this node, which accounts for an additional $D$ in the total cost. Thus, each distinct word $w$ we find in at least one of the nodes can correspond to at most $|D \cap D_w|$ 1-bits met in intersections with the bitvectors of other nodes in the set, and each 1-bit leads to at most two 0-bits met in intersections. Summing over all $w \in W$ gives the second term in the equation of the lemma.

The remaining nodes that we visit are all ancestors of one of the $v_1, \ldots, v_l$, and we have already shown in the proof of Lemma 6 that their number is at most $2 \log_2 m$. Since the processing time for a node is always bounded by $O(|D|)$, that fraction of the query processing time spent in ancestors of $v_1, \ldots, v_l$ is bounded by $O(|D| \log_2 m)$.

□

**Lemma 8.** *The bit vectors of TREE+BITVEC+PUSHUP require a total of at most* $2N + n$ *bits. The auxiliary data structure (for the constant-time rank computation) requires at most* $N$ *bits.*

*Proof.* Just as for TREE+BITVEC, each 1-bit can be associated with the occurrence of a particular word in a particular document, and that correspondence is $1-1$. This proves that the total number of 1-bits is exactly $N$, and since word numbers are stored only by 1-bits and there is indeed one word number stored by each 1-bit, the total number of word numbers stored is also $N$. By the same argument as in Lemma 5, the number of 0-bits is at most the number of 1-bits plus 1 for each document plus the number of 0-bits in the root node.

□

### 4.1 The index construction for TREE+BITVEC+PUSHUP

The construction of the tree for algorithm TREE+BITVEC+PUSHUP is relatively straightforward and takes *constant amortized time* per word-in-document occurrence (assuming each document contains its word sorted in ascending order).

1. Process the documents in order of ascending document numbers, and for each document $d$ do the following.

2. Process the distinct words in document $d$ in order of ascending word number, and for each word $w$ do the following. Maintain a *current node,* which we initialize as an artificial parent of the root node.

3. If the current node does not contain $w$ in its subtree, then set the current node to its parent, until it does contain $w$ in its subtree. For each node left behind in this process, append a 0-bit to the bit vector of those of its children which have not been visited.

   *Note: for a particular word, this operation may take non-constant time, but once we go from a node to its parent in this step, the old node will never be visited again. Since we only visit nodes, by which a word will be stored and such nodes are visited at most three times, this gives constant amortized time for this step.*

4. Set the current node to that one child which contains $w$ in its subtree. Store the word $w$ by this node. Add a 1-bit to the bit vector of that node.

## 5 Divide into Blocks (TREE+BITVEC+PUSHUP+BLOCKS)

This section is our last station on the way to our main result, Theorem 2.

For a given $B$, with $1 \leq B \leq m$, we divide the set of all words in blocks of equal size $B$. We then construct the data structure according to TREE+BITVEC+PUSHUP for each block separately. As we only have to consider those blocks, which contain any words from $W$, this gives a further speedup in query processing time. An autocompletion query given by a word range $W$ and a set of documents $D$ is then processed in the following three steps.

1. Determine the set of $\ell$ (consecutive) blocks, which contain at least one word from $W$, and for $i = 1, \ldots, \ell$, compute the subrange $W_i$ of $W$ that falls into block $i$. Note that $W = W_1 \dot{\cup} \cdots \dot{\cup} W_\ell$.

2. For $i = 1, \ldots, \ell$, process the query given by $W_i$ and $D$ according to TREE+BITVEC+PUSHUP, resulting in a set of matches $M_i := \{(w, d) \in C : w \in W_i, d \in D\}$, where $C$ is the set of of word-in-document pairs.

3. Compute the union of the sets of matching word-in-document pairs $\cup_{i=1}^{\ell} M_i$ (a simple concatenation).

**Lemma 9.** *With TREE+BITVEC+PUSHUP+BLOCKS and block size $B$, an auto-completion query $(D, W)$ can be processed in time $O\big(|D| \cdot (\alpha + \beta) + \sum_{w \in W} |D \cap D_w|\big)$, where $\alpha = |W|/B$ and $\beta$ is bounded by $\log_2 B$ as well as by the average number of distinct words from $W_1 \cup W_\ell$ (the first and the last subrange from above) in a document from $D$.*

*Proof.* Let $W_i$ denote the subset of $W$ pertaining to block $i$. Since each block contains at most $B$ words, according to Lemma 7, we need time at most $O(|D| \log_2 B + \sum_{w \in W_i} |D \cap D_w|)$ for a block $i$. However, for all but at most two of these blocks (the first and the last) it holds that all words of the blocks are in $W$, so that according to the special case in Lemma 7, the query processing time for each of the at most $|W|/B$ inner blocks is actually $O(|D| + \sum_{w \in W_i} |D \cap D_w|)$ . Summing these up gives us the bound claimed in the lemma.

□

**Lemma 10.** *TREE+BITVEC+PUSHUP+BLOCKS with block size $B$ requires at most $3N + n \cdot \lceil m/B \rceil$ bits for its bit vectors and at most $N\lceil \log_2 B \rceil$ bits for the word numbers stored by the 1-bits. For $B \geq mn/N$, this adds up to at most $N(4 + \lceil \log_2 B \rceil)$ bits.*

*Proof.* To count the number of bits in the relative bitvectors, we use the same argument as for Lemma 8: there is exactly one 1-bit for each word-in-document occurrence. The total number of 0-bits is exactly one more than the total number of 1-bits, plus the number of 0-bits in the bit vectors of the roots of the trees. The latter can be bounded by $n \cdot \lceil m/B \rceil + n \cdot N/n$, since there are $n$ documents, and $\lceil m/B \rceil$ blocks of size $B$ and at most $N/n$ blocks of volume larger than $n$. To encode a particular word within a block, $\lceil \log_2 B \rceil$ bits are obviously sufficient.

□

With all the required machinery in place, we can now prove Theorem 2. Part (a) of Theorem 2 is established by the construction given in Section 4.1. Part (b) of Theorem 2 follows from Lemma 10 by choosing $B = \lceil nm/N \rceil$. This choice of $B$ minimizes the space bound of Lemma 10, and we call the corresponding data structure AUTOTREE . Part (c) of Theorem 2 follows from Lemma 9 and the following remarks. If the words in a document with $L$ words are a random size-$L$ subset of all words, then the average number of words per document that fall into a fixed block is at most 1. In our experiments, the average value for $\beta$ was 2.2.

As mentioned just before Lemma 7, $\beta$ counts the number of bitvector lookup operations for a candidate document in $D$, which do not contribute any element to the result set. If the wordrange $W$ spans multiple tree blocks of size $B = \lceil nm/N \rceil$, then such "useless" bitvector lookups can also occur at the root nodes of the intermediate tree blocks. However, these comparisons are accounted for by the factor $\alpha$, which bounds the number of such intermediate blocks, and $\beta$ only counts such bitvector lookups in the boundary blocks, which also contain at least one word not in $W$.

Formally, $\beta$ is defined as the number of bitvector lookups that need to be performed in the boundary blocks (of which there are at most two) for a candidate document in $D$ until either (a) this document can be ruled out as an element of $D'$ (as it contains no valid completions) or (b) a relevant completion is reported from this document (at which point the total number of additional bitvector lookups is bounded by twice the number of matching output elements for this document). A small, constant $\beta$ thus indicates a strong output-sensitive behaviour of the algorithm.

## 6 Experiments

We tested both AUTOTREE and our baseline BASIC on the corpus of the TREC 2004 Robust Track (ROBUST '04), which consists of the documents on TREC disks 4 and 5, minus the Congressional Record [20]. We implemented AUTOTREE with a block size of 4096, which is the optimal block size according to Section 5, rounded to the nearest power of two. The following table gives details on the collection and on the space consumption of the two schemes; as we can see, AUTOTREE does indeed use no more space (and for this collection, in fact, significantly less) than BASIC , as guaranteed by Theorem 2.

| | | | | | | bits per word-in-doc pair | |
|---|---|---|---|---|---|---|---|
| Collection | raw size | $n$ | $m$ | $N/n$ | $B^*$ | BASIC | AUTOT |
| ROBUST '04 | 1.9 GB | 528,025 | 771,189 | 219.2 | 4,096 | 19.0 | 13.9 |

Table 1: The characteristics of our test collection: $n$ = number of documents, $m$ = number of distinct words, $N/n$ = average number of distinct words in a document, $B^*$ = space-optimal choice for the block size. The last two columns give the space usage of BASIC and AUTOT(REE) in bits per word-in-document pair.

Queries are derived from the 200 "old" [4] queries (topics 301-450 and 601-650) of the TREC Robust Track in 2004 [20], by "typing" these queries from left to right, taking a minimum word length of 4 for the first query word, and 2 for any query word after the first. From these autocompletion queries we further omitted those, which would be obtained by simple filtering from a prefix according to the explanation following Definition 1. This filtering procedure is identical for AUTOTREE and BASIC and takes only a small fraction of the time for the autocompletion queries

[4]They are "old" as they had been used in previous years for TREC.

processed according to Definition 1, which is why we omitted it from consideration in our experiments. To give an example, for the ad hoc query `world bank criticism`, we considered the autocompletion queries `worl`, `world ba`, and `world bank cr`. We considered a total number of 512 such autocompletion queries.

We implemented BASIC and AUTOTREE in C++ and measured query processing times on a Dual Opteron machine, with 2 Intel Xeon 3 GHz processors, 8 GB of main memory, running Linux. We measured the time for producing the output according to Definition 1. The time for scoring and ranking would be identical for AUTOTREE and BASIC , and would, according to a number of tests, take only a small fraction of the aforementioned processing time. We therefore excluded it from our measurements. For BASIC , we implemented a fast linear-time intersect, which, on average, turned out to be faster than its asymptotically optimal relatives [8].

| Scheme | Max | Mean | StdDev | Median | 90%-ile | 95%-ile | Correl. |
|---|---|---|---|---|---|---|---|
| BASIC | 6.32secs | 0.19secs | 0.55secs | 0.034secs | 0.41secs | 0.93secs | 0.996 |
| AUTOT | 0.63secs | 0.05secs | 0.06secs | 0.028secs | 0.11secs | 0.14secs | 0.973 |

Table 2: Processing times statistics of BASIC and AUTOT(REE) for all 512 autocompletion queries. The 6th and 7th column show the $k$th worst processing time, where $k$ is $10\%$ and $5\%$, respectively, of the number of queries. The last column gives the correlation factor between query processing times and total list volume $\sum_{w \in W}(|D| + |D_w|)$ for BASIC , and input size plus total output volume $|D| + 5\sum_{w \in W}|D \cap D_w|$ for AUTOTREE .

The results from Table 2 conform nicely to our theoretical analysis. Four main observations can be made: (i) with respect to maximal query processing time, which is key for an interactive application, AUTOTREE improves over BASIC by a factor of 10; (ii) in average processing time, which is significant for throughput in a high-load scenario, the improvement is still a factor of $4$; (iii) processing times of AUTOTREE are sharply concentrated around their mean, while for BASIC they vary widely (in both directions as we checked); (iv) the almost perfect correlation between query processing times and our analytical bounds (explained in the caption of Figure 2) demonstrates both the soundness of our theoretical modelling and analysis as well as the accuracy of our implementation.

Table 3, finally, breaks down query processing times by the number of query words. As we can see, BASIC is significantly faster than AUTOTREE for the 1-word queries, however, not because AUTOTREE is slow, but because BASIC is extremely fast on these queries. This is so, because BASIC does not have to compute any intersections for 1-query but merely has to copy all relevant lists $D_w$ to the output, whereas AUTOTREE has to extract, for each output element, bits from its (packed) document id and word id vectors. On multi-word queries, BASIC has to process a

|  | 1-word |  | multi-word |  |
|---|---|---|---|---|
| Scheme | Max | Mean | Max | Mean |
| BASIC | 0.10secs | 0.01secs | 6.32secs | 0.30secs |
| AUTOTREE | 0.37secs | 0.09secs | 0.63secs | 0.02secs |

Table 3: Breakdown of query processing for BASIC and AUTOTREE by number of query words.

much larger volume than AUTOTREE , and we see essentially the situation discussed above for the overall figures.

## References

[1] 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.

[2] 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.

[3] H. Bast, C. W. Mortensen, and I. Weber. Output-sensitive autocompletion search. In *13th Symposium on String Processing and Information Retrieval (SPIRE'06)*, pages 150–162, 2006.

[4] H. Bast and I. Weber. Type less, find more: Fast autocompletion search with a succinct index. In *29th Conference on Research and Development in Information Retrieval (SIGIR'06)*, 2006.

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

[6] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

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

[8] E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *11th Symposium on Discrete Algorithms (SODA'00)*, pages 743–752, 2000.

[9] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

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

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

[13] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *32nd Symposium on the Theory of Computing (STOC'00)*, pages 397–406, 2000.

[14] 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.

[15] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[16] J. I. Munro. Tables. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, pages 37–42, 1996.

[17] C. G. Nevill-Manning, I. Witten, and G. W. Paynter. Lexically-generated subject hierarchies for browsing large collections. *International Journal of Digital Libraries*, 2(2/3):111–123, 1999.

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

[19] 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.

[20] E. Voorhees. Overview of the trec 2004 robust retrieval track. In *13th Text Retrieval Conference (TREC'04)*, 2004. http://trec.nist.gov/pubs/trec13/papers/ROBUST.OVERVIEW.pdf.

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

| | | |
|---|---|---|
| MPI-I-2005-4-002 | O. Schall, A. Belyaev, H. Seidel | Sparse Meshing of Uncertain and Noisy Surface Scattered Data |
| MPI-I-2005-4-001 | M. Fuchs, V. Blanz, H. Lensch, H. Seidel | Reflectance from Images: A Model-Based Approach for Human Faces |
| MPI-I-2005-2-003 | H.d. Nivelle | Using Resolution as a Decision Procedure |
| MPI-I-2005-2-002 | P. Maier, W. Charatonik, L. Georgieva | Bounded Model Checking of Pointer Programs |
| MPI-I-2005-2-001 | J. Hoffmann, C. Gomes, B. Selman | Bottleneck Behavior in CNF Formulas |
| MPI-I-2005-1-008 | C. Gotsman, K. Kaligosi, K. Mehlhorn, D. Michail, E. Pyrga | Cycle Bases of Graphs and Sampled Manifolds |
| MPI-I-2005-1-007 | I. Katriel, M. Kutz | A Faster Algorithm for Computing a Longest Common Increasing Subsequence |
| MPI-I-2005-1-003 | S. Baswana, K. Telikepalli | Improved Algorithms for All-Pairs Approximate Shortest Paths in Weighted Graphs |
| MPI-I-2005-1-002 | I. Katriel, M. Kutz, M. Skutella | Reachability Substitutes for Planar Digraphs |
| MPI-I-2005-1-001 | D. Michail | Rank-Maximal through Maximum Weight Matchings |
| MPI-I-2004-NWG3-001 | M. Magnor | Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae |
| MPI-I-2004-NWG1-001 | B. Blanchet | Automatic Proof of Strong Secrecy for Security Protocols |
| MPI-I-2004-5-001 | S. Siersdorfer, S. Sizov, G. Weikum | Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning |
| MPI-I-2004-4-006 | K. Dmitriev, V. Havran, H. Seidel | Faster Ray Tracing with SIMD Shaft Culling |
| MPI-I-2004-4-005 | I.P. Ivrissimtzis, W.-. Jeong, S. Lee, Y.a. Lee, H.-. Seidel | Neural Meshes: Surface Reconstruction with a Learning Algorithm |
| MPI-I-2004-4-004 | R. Zayer, C. Rssl, H. Seidel | r-Adaptive Parameterization of Surfaces |
| MPI-I-2004-4-001 | J. Haber, C. Schmitt, M. Koster, H. Seidel | Modeling Hair using a Wisp Hair Model |
| MPI-I-2004-2-007 | S. Wagner | Summaries for While Programs with Recursion |
| MPI-I-2004-2-002 | P. Maier | Intuitionistic LTL and a New Characterization of Safety and Liveness |
| MPI-I-2004-2-001 | H. de Nivelle, Y. Kazakov | Resolution Decision Procedures for the Guarded Fragment with Transitive Guards |
| MPI-I-2004-1-006 | L.S. Chandran, N. Sivadasan | On the Hadwiger's Conjecture for Graph Products |
| MPI-I-2004-1-005 | S. Schmitt, L. Fousse | A comparison of polynomial evaluation schemes |
| MPI-I-2004-1-004 | N. Sivadasan, P. Sanders, M. Skutella | Online Scheduling with Bounded Migration |
| MPI-I-2004-1-003 | I. Katriel | On Algorithms for Online Topological Ordering and Sorting |
| MPI-I-2004-1-002 | P. Sanders, S. Pettie | A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching |
| MPI-I-2004-1-001 | N. Beldiceanu, I. Katriel, S. Thiel | Filtering algorithms for the Same and UsedBy constraints |
| MPI-I-2003-NWG2-002 | F. Eisenbrand | Fast integer programming in fixed dimension |
| MPI-I-2003-NWG2-001 | L.S. Chandran, C.R. Subramanian | Girth and Treewidth |
| MPI-I-2003-4-009 | N. Zakaria | FaceSketch: An Interface for Sketching and Coloring Cartoon Faces |
| MPI-I-2003-4-008 | C. Roessl, I. Ivrissimtzis, H. Seidel | Tree-based triangle mesh connectivity encoding |