TinyLex: Static N-Gram Index Pruning with Perfect Recall

Derrick Coetzee
Microsoft Research, Microsoft Corporation
Redmond, WA (USA)
dcoetzee@microsoft.com

ABSTRACT

Inverted indexes using sequences of characters (n-grams) as terms provide an error-resilient and language-independent way to query for arbitrary substrings and perform approximate matching in a text, but present a number of practical problems: they have a very large number of terms, they exhibit pathologically expensive worst-case query times on certain natural inputs, and they cannot cope with very short query strings. In word-based indexes, static index pruning has been successful in reducing index size while maintaining precision, at the expense of recall. Taking advantage of the unique inclusion structure of n-gram terms of different lengths, we show that the lexicon size of an n-gram index can be reduced by 7 to 15 times without any loss of recall, and without any increase in either index size or query time. Because the lexicon is typically stored in main memory, this substantially reduces the memory required for queries. Simultaneously, our construction is also the first overlapping n-gram index to place tunable worst-case bounds on false positives and to permit efficient queries on strings of any length. Using this construction, we also demonstrate the first feasible n-gram index using words rather than characters as units, and its applications to phrase searching.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods; E.4 [**Coding and Information Theory**]: Data compaction and compression

1. INTRODUCTION

Consider the following *exact search* problem: given a collection of documents, quickly locate all occurrences of some arbitrary string in this collection. The collection may contain any type of data, structured in any manner.

Standard word-based inverted indexes are of little assistance for searching files that have no explicit word structure, such as genome sequences and image data. Effectively searching highly-structured data, such as programming language source files or XML, typically requires specific knowledge of the grammar and an expensive parsing phase; a simple word-based index would not be effective for searching a source code database for `"%s"` or a LaTeX database for `x^2/4`. Word-based indexes are also language-dependent: their stopword lists and stemming algorithms depend specifically on the source language, and for texts written in Asian languages, even word-breaking requires complex language-specific techniques. For these reasons, word-based indexes are not well-suited to a system that needs to search heterogeneous data in a uniform way.

Classical n-gram indexes, which store the location of every

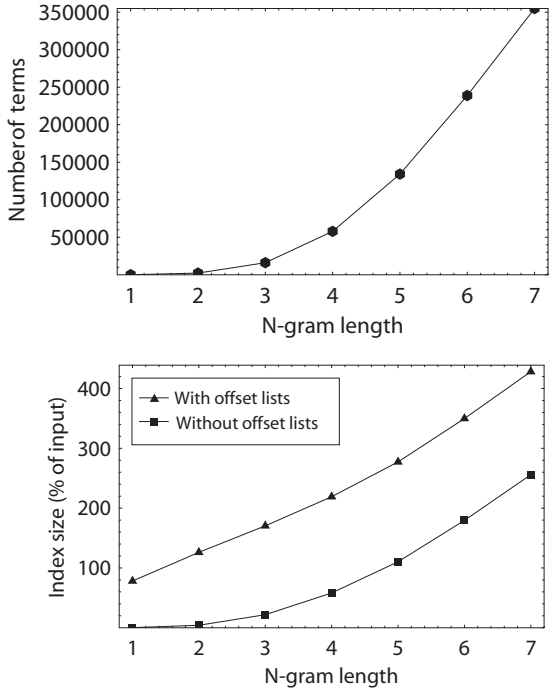


Figure 1: Number of terms and overall size of a compressed n-gram index over a small TREC-1MB collection for various n-gram lengths, with and without posting lists. Number of terms *(top)* is identical for both. Posting lists are compressed by Moffat and Stuiver's interpolative coding [19]; the lexicon is compressed by front coding and Huffman coding.

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.

sequence of n characters in the input for some fixed n, overcome these limitations. They are language-independent and are readily generalizable to approximate matching, which is important for bioinformatics applications as well as querying texts containing minor OCR errors or typos. The choice of n is a design choice based on what is most effective for a particular application: if n is too small, each term will have a long posting list and queries will require large numbers of term lookups, making queries too expensive; but as n increases the number of terms and the compressed index size increases exponentially, as shown in Figure 1.

An important design choice in an n-gram index is whether to store the offset lists of each n-gram in each document, or to store document lists only. If offset lists are stored, then achieving high precision and recall on query results is straightforward, by using relative offset information to align posting lists before intersection; the systems of Adams and Meltzer [1] and Kim et al [13] use this method.

On the other hand, as Figure 1 shows, discarding offset lists provides a simple way of producing much smaller indexes; Mayfield and McNamee's HAIRCUT system [18] and our own system use this approach. Any unqualified references to classical n-gram indexes in the remainder of this paper will refer specifically to n-gram indexes without offset lists.

For small values of n, the index size without offset lists is sublinear, but as n increases the number of terms increases exponentially with n and lexicon storage begins to dominate the index space. Because lexicons are typically stored in main memory, lexicon size directly impacts the load time and memory requirements of the query engine.

Additionally, discarding the offset lists introduces a new problem: queries become conservative overestimates. In the language of query relevancy, they maintain perfect recall, but may have arbitrarily poor precision. For example, a query for "there" over a trigram (3-gram) index may return documents containing the words "were", "then", and "her". These false positives, called *retrieval noise* by Ogawa and Matsuda [20], can be eliminated by a linear scan over the documents occurring in the query result, as first suggested by Adams and Meltzer [1]. This approach takes advantage of the assumption that most applications will, once they locate a term, want to load its document context for further processing; therefore scanning documents that contain the query term involves no additional cost. If precision is high, the overhead of scanning is small.

However, classical n-gram indexes without offset information are not designed to provide any worst-case guarantees on precision; there are natural inputs for which they yield pathologically expensive query times. For example, in one test case we divided the Canterbury Corpus King James Bible [3] into 1000 documents and constructed a classical 3-gram index on this collection. This index returned 81% of all documents when queried for " `the man and his` ", a phrase that occurs in only one document. As another example, an n-gram index on a collection containing many long runs of a single character will tend to yield poor behavior on queries that contain long runs of that character (intuitively, "n-grams can't count"). This unpredictable behavior is a serious problem for interactive applications, where the user would be baffled to encounter long processing delays on queries that produce small result sets.

Another issue with classical n-gram indexes is that queries on strings shorter than n characters are expensive and complex: they can only be found by retrieving all index terms containing the query and unioning them. Whereas the number of terms retrieved increases linearly as query strings become longer, it increases exponentially as they become shorter, and a secondary index is required on the lexicon to locate these terms. The alternative of including posting lists for all n-grams of length at most n greatly increases index size, typically doubling it.

In this paper we present TinyLex, an n-gram index designed to minimize the lexicon size and eliminate pathological worst-case behavior while keeping both recall and precision competitive with classical n-gram indexes in the typical case. It achieves this by selectively incorporating n-grams of many lengths, effectively providing an arbitrarily large context. Its construction also permits simple, efficient queries on short strings.

Because TinyLex indexes use many lengths of n-gram, they are not parameterized by n-gram length; instead they are parameterized by a threshold value t placing a hard upper bound on the number of false positives in query results. The index is constructed by a multipass procedure that examines one length of n-gram at a time, beginning with the smallest, and computes the query size of every such n-gram. An n-gram is added to the lexicon if its actual posting list is significantly smaller than its query list. By tracking partial query results and pruning the list of candidate n-grams from pass to pass, efficient index construction over a wide range of lengths is made practical. Further details are described in section 3.

Our results, detailed in section 6, show that TinyLex indexes achieve index sizes and median query times comparable to or better than those of classical n-gram indexes with a number of terms that is 7 to 16 times smaller for English text and 1.2 to 13 times smaller for genome sequences. In fact, our pruning is so effective that it renders n-gram indexes feasible on large alphabets such as English words, as described in section 6.4. We also demonstrate scalability of index construction, and exhibit worst-case query sets for which TinyLex query times are dramatically (2–30 times) faster than those of classical n-gram indexes.

The primary limitations of our approach compared to classical n-gram indexes, detailed in section 6.6, include a longer index construction time (about 9.8 times longer) and (with the standard parameter settings) slower query times on queries for which the classical n-gram index would return empty results. Also, as described in section 7, the problem of generalizing our approach to dynamic collections remains open.

In section 2, we describe a number of related works that address the problem of precision in n-gram indexes or deal with similar index structures or applications. Section 3 describes our basic index construction, with details about its encoding in section 4 and query processing in section 5. Performance is evaluated in section 6. Future work and applications are discussed in section 7 and conclusions in section 8.

2. RELATED WORK

A number of elements of our motivation and construction have appeared in previous work.

Wu and Manber's *agrep* [24] and GLIMPSE [16] were motivated by desktop search applications and also store imprecise location information that necessitates a linear scan to

filter out false positives; like n-gram indexes, they enable approximate matching. However, only words are indexed, so they only apply to documents with a word concept. Moreover, the posting lists of all words in the index that match the pattern are searched. This leads to poor precision on queries that span words or contain common words; by indexing terms that span words, n-gram indexes overcome these limitations.

Exact string search is one of the applications targeted by suffix arrays, a compact data structure that lists all offsets into a collection, sorted by the order of the suffix strings occurring at those offsets. Although traditional construction algorithms make poor use of the memory hierarchy and so fail to scale, sophisticated techniques such as those of Dementiev et al. [9] scale effectively to large collections. However, even compact suffix arrays such as those of Mäkinen [15] still tend to be several times larger than our indexes ("less than twice the size of the text" as opposed to 15–60% the size of the text), partly because the number of suffixes is large and partly because their overconstrained ordering prevents them from taking advantage of delta-based list compression.

Because our pruning criteria are based on the document frequency of n-grams, and suffix arrays can be used to efficiently determine the document frequency of all n-grams [25], we considered using them in our index construction procedure; however, for the collections we considered the cost and complexity of generating the suffix array in the first place exceeded that of the simple scanning-based procedure described herein. This approach may be more profitable for very large collections.

The n-gram/2L system of Kim et al. [13] is also designed to reduce the space and time overhead of classical n-gram indexes by using a two-level index. A small front-end index maps small m-grams to larger n-grams containing them, and the back-end index maps these larger n-grams to document offsets. The primary advantage of this approach is that only offsets equal to zero mod $n - m + 1$ need to be stored in the back-end index. However, their approach depends critically on the presence of offset information, which we eschew to reduce index size, and n-gram/2L exhibits poor worst-case behavior in cases where the front-end index yields a large set of n-grams to query.

Ogawa and Matsuda [20] constructed a classical n-gram index with offset lists for Japanese queried using a subset of nonoverlapping terms. They address the false positives (which they call "retrieval noise") introduced by their reduced query term set with a simple query planner that predicts which set of nonoverlapping terms will produce the least noise, and then adds one additional term for further differentiation. Although like our approach this work attempts to address the issue of false positives, their index also depends on offset information and only attempts to address noise introduced by the query planning process.

Mayfield and McNamee's HAIRCUT system [18] shares our design choice of eliminating offset lists to reduce index size. Although HAIRCUT focused on ranked search, an application we do not consider in this paper, the trends exhibited by HAIRCUT's lexicon size and index size over a range of n-gram lengths are consistent with our baseline measurements for classical n-gram indexes, and we expect this system to exhibit poor worst-case query times as described in section 1.

Static index pruning on word-based indexes [7][6] also focuses on shrinking the lexicon, but because word terms don't share the rich substring structure of n-gram terms, it is not able to maintain perfect recall or provide guarantees on precision; instead, its goal is to maintain relevance in highly-ranked results. In exchange for this loss of information, this type of pruning is more successful in decreasing overall index size.

Mah and D'Amore pointed out the issue of retrieval noise in n-gram indexes and modelled it statistically via a similarity function and a multinomial distribution [14][8] However, their goal was also retrieval of relevant documents, rather than complete enumeration of matching documents. Mah and D'Amore also augmented their classical n-gram indexes with larger n-grams in order to improve precision by adding n-grams which are extensions of existing frequent index terms. However, this *ad hoc* approach does not generalize well to many lengths of n-grams and included many n-grams that are strictly unnecessary, such as unique extensions of frequent n-grams.

N-gram models in language modelling (with words rather than characters as units) routinely use variable-length models created by a process called *pruning* [22]. In this process, a large model is constructed using n-grams for a large value of n, then all n-grams that can be removed without significantly altering the model's predictions are removed. In fact, the most similar work to ours comes from language modelling, where Siivola and Pellom [21] describe a method for "growing" an n-gram language model beginning with small n-grams and adding larger ones whenever this decreases the data coding length; they point out the advantage, shared by our method, that "[v]ery high order n-grams can be used" because the complete set of high-order n-grams never needs to be stored.

One important application of our technique is efficient phrase searching using an auxiliary index of word n-grams, as discussed in section 6.4. Bahle et al. [4] discuss a similar technique for phrase searching using an auxiliary "nextword index" that is pruned by only including phrases where common words appear first, similar to the approach of Mah and D'Amore but at the word n-gram level. For the same reason, it includes some strictly unnecessary terms; for example, if a two-word term occurs very frequently, it may be added to the nextword index, even if the two terms never occur separately. Our scheme excludes terms like this and includes important longer word n-grams of length 3 and 4; we show that overall space overhead is comparable. It is difficult to draw a conclusive comparison however, since we do not measure query times in this work.

The tradeoff between size and precision exhibited by our index is a classical one that appeared as early as 1970 in Bloom filters [5], where the average number of false positives per query increases as the table size is reduced. Zobel et al. [26] discuss the repercussions of this tradeoff in signature file indexes, which are also hash-based; our implementation, however, does not use hashing.

3. CONSTRUCTING THE INDEX

3.1 Definitions

We begin by considering a simplified problem: suppose we are given a document collection and a threshold t, and we wish to ensure that, for any string of length between $n_{\min}$

and $n_{\max}$ occurring in the document collection, its query result has at most t false positives. In other words, the query result on strings not occurring in the collection is (for the moment) irrelevant.

Formally, suppose the alphabet is Σ, that $S : \Sigma^* \to \mathscr{P}(\Sigma^*)$ yields the set of all substrings of a given string, and the document collection is viewed as a collection of strings D_i over Σ^* ($i \in \mathbb{N}$). Any index implies a query function $Q : \Sigma^* \to \mathscr{P}(\mathbb{N})$ producing the query result for a given string, and $P(s) = \{n : s \in S(D_n)\}$ is the actual posting list for s. We seek an index yielding a Q satisfying

$$\forall s \in \Sigma^* \qquad (n_{\min} \le |s| \le n_{\max} \wedge |P(s)| > 0) \Rightarrow |Q(s)| - |P(s)| \le t. \tag{1}$$

As in a typical inverted index, we select a lexicon $L \in \mathscr{P}(\Sigma^*)$ and the index explicitly stores $P(s)$ for all $s \in L$. In a typical inverted index, Q is defined by:

$$Q(q) = \begin{cases} \bigcap_{s \in S(q)} P(s) & \text{if } T(q) \subseteq L, \\ \emptyset & \text{otherwise.} \end{cases}$$

where $T(q)$ is the *term set* of q, a set of potential index terms extracted from q. For word-based indexes these are the words of q (typically stemmed and with stopwords removed). For classical n-gram indexes these are the substrings of q of a fixed length.

Since $s \in S(q)$ implies $P(s) \supseteq P(q)$, we have $Q(q) \supseteq P(q)$. Terms not in the lexicon are asserted implicitly to occur nowhere. However, in a variable-length index such as ours, $T(q) = S(q)$, and any L such that $S(q) \subseteq L$ for all queries q returning results would be too large for an explicit representation. To avoid this issue, we instead assume that terms not in the lexicon implicitly occur everywhere, and define:

$$Q(q) = \bigcap_{s \in (S(q) \cap L)} P(s).$$

3.2 Simple scanning construction

We aim to satisfy our requirement using a variable-length n-gram index containing terms of many lengths. To motivate the construction procedure, suppose the index contains the n-gram "jugg" and we wish to determine whether we should add "juggl" to the index. In many collections, "juggl" will have exactly the same posting list as "jugg"; any query string containing "juggl" also contains "jugg", so adding "juggl" eliminates no additional documents, and it should not be added. More generally, if $|Q(s)|$ is close to $|P(s)|$, adding s to L will add little information.

To exploit this, we construct a multipass greedy algorithm, detailed in Figure 3. The procedure makes a pass over the collection for each value of n from $n_{\min}$ to $n_{\max}$, beginning with the smallest, and computes the posting list of every n-gram in the collection (lines 3–6). At the end of each pass, the partially constructed index is queried for each n-gram encountered during that pass (line 8); any n-gram exhibiting t or more false positives is added to the index (lines 9–12).

The resulting index will trivially satisfy (1) because we have visited every string satisfying the condition $n_{\min} \le |s| \le n_{\max} \wedge |P(s)| > 0$ and added it to L if it failed to satisfy $|Q(s)| - |P(s)| \le t$, and $Q(s) = P(s)$ for every $s \in L$.

```
1   function ConstructIndexSimple(D, n_min, n_max, t):
2       for n from n_min to n_max:
3           for d from 1 to |D|:
4               for i from 1 to |D_d|:
5                   s ← D_d[i..i + n − 1]
6                   Add d to P(s)
7           for each s where |s| = n and |P(s)| > 0:
8               Q(s) ← ⋂ P(s′) over substrings s′ of s in L
9               if |Q(s)| − |P(s)| ≤ t
10                  Discard P(s)
11              else
12                  L ← L ∪ {s}
13      return {L, P}
```

Figure 3: Pseudocode for simple index construction. D is the document collection and P contains the posting lists of n-grams that have been added to the index.

Moreover, this set is minimal in the sense that the Q implied by any proper subset of L fails to satisfy (1): because only terms of larger or equal size are added after a given n-gram s is added, and no such term can be a substring of s, they do not affect $|Q(s)|$. Hence if s satisfied $|Q(s)| - |P(s)| > t$ when it was added, it will again satisfy it if s itself is removed.

On the other hand, this construction does not necessarily produce the smallest possible L satisfying (1). For example, suppose there are three documents 0, 1, 2 and five distinct n-grams of length between 1 and 2 with $P(a) = \{0, 1\}$, $P(b) = \{0, 2\}$, $P(aa) = P(ab) = P(ba) = \{0\}$. Setting $t = 1$, the construction produces $L = \{aa, ab, ba\}$, but $L = \{a, b\}$ also satisfies (1). We do not attempt to minimize $|L|$ globally.

The distribution of n-gram lengths in the lexicon produced by this algorithm is nontrivial and follows a predictable pattern: the number of n-grams rapidly increases to some peak value, depending on the collection, then decreases to a tail of small values before falling to zero. Two example distributions are shown for an English text collection and a genome sequence collection in Figure 2.

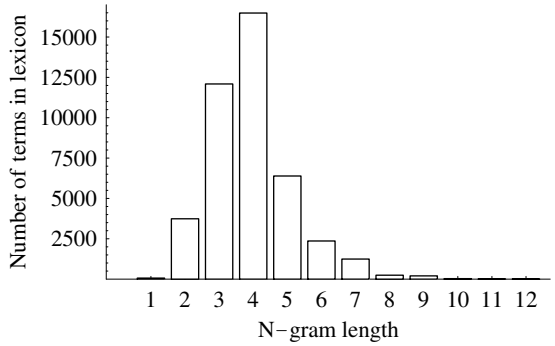
3.3 Generalizing to handle all query strings

The index described so far operates only a limited set of queries, and construction rapidly becomes infeasible as $n_{\max}$ increases. The reasons for this are twofold. The first is that it requires tracking the posting list of *every* n-gram in the collection; for a large value of n, most n-grams are unique and occur in one place, so storing these lists typically requires storage several times the size of the collection, even though almost all of them will not pass the condition for inclusion in L. The second is that as the size of a query string q grows, queries become more expensive because $S(q) \cap L$ grows.

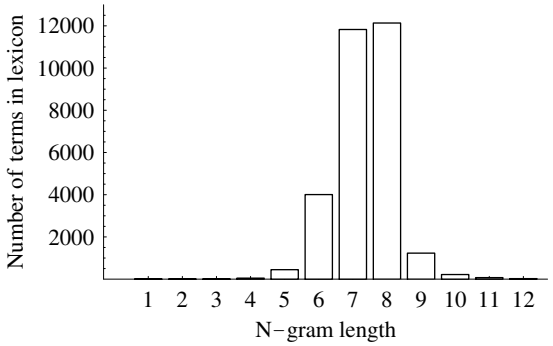
To address these, we note that if $|P(s)| > 0$, the condition $|Q(s)| - |P(s)| \le t$ is implied by the stronger condition:

$$|Q(s)| \le t + 1 \tag{2}$$

We call this the *heritable condition* because it has the useful property that if it holds for s, it holds for all superstrings of s. We exploit it in two ways. First, while scanning the collection, we ignore any n-grams satisfying the heritable condition and do not track their posting lists. If they are encountered multiple times, they are queried each time. For



(a) Distribution of n-gram lengths on TREC-100MB collection, a typical English-language text, with 5% threshold.



(b) Distribution of n-gram lengths on chr14 collection, a genome sequence, with 5% threshold. Omitted are 157 n-grams of length > 12.

Figure 2: Typical distributions of n-gram lengths in a TinyLex lexicon.

large n, this makes sense, because most n-grams will be excluded by the heritable condition and will also occur rarely.

Second, while scanning the collection, if we encounter an n-gram that fails the heritable condition at offset i, we do not need to visit offsets $i-1$ or i in the next pass; by maintaining a list of offsets into each document and pruning it during the scan, each pass becomes successively shorter. This also provides a convenient stopping condition: once the offset list becomes empty, we have enumerated all possible n-grams in the collection that might have t or more false positives, allowing efficient index construction for $n_{\max} = \infty$. If we also set $n_{\min} = 1$, the guarantee is provided over all substrings in the collection, and we can remove the condition $n_{\min} \le |s| \le n_{\max}$ from (1):

$$\forall s \in \Sigma^* \quad |P(s)| > 0 \Rightarrow |Q(s)| - |P(s)| \le t. \tag{3}$$

An important question is whether the condition $|P(s)| > 0$ can also be removed. It's natural for users to present queries for strings absent from the collection, and the lexicon produced by this construction can yield large numbers of false positives for such queries. The key lies in the contrapositive of (3): if $|Q(s)| - |P(s)| \ge t+1$, then $|P(s)| = 0$. In other words, if we query for s and then scan $t+1$ documents from the query result without observing s, we can safely conclude that s does not occur in the remaining documents either. This allows us to provide a strong unconditional worst-case guarantee:

$$\forall s \in \Sigma^* \quad |Q'(s)| - |P(s)| \le t+1, \tag{4}$$

where $Q'(s)$ with $P(s) \subseteq Q'(s) \subseteq Q(s)$ is the set of documents read by the linear scan phase of query processing.

3.4 Further improvements

To further accelerate index construction, we note that for any s with $|s| = n$, $Q(s)$ can be computed as the intersection of $Q(s_1)$ and $Q(s_2)$, where s_1 and s_2 are the two length $n-1$ substrings of s. Here, $Q(s_i)$ will either be $P(s_i)$, if s_i was added to the lexicon, or else will be its query list generated in the previous pass. Thus we retain all query results from the previous pass; this does not require prohibitive memory, since we need only retain query results for strings passing the heritable condition.

Combining this optimization with the heritable condition optimizations described above, we obtain the construction procedure in Figure 4. In this procedure, each query result is constructed by intersecting two previously computed query results (line 9); if an n-gram s was added to the index, $Q(s)$ will equal its posting list (line 20). N-grams failing the heritable condition are discarded and ignored (lines 10–11). The offsets list, initially containing all offsets in all documents (line 3) is pruned whenever an n-gram fails the heritable condition (line 12).

Finally, another useful trick to accelerate index construction and limit memory usage is to separate each pass into two passes, the first counting the document frequencies of each

```
1   function ConstructIndex(D, n_min, n_max, t):
2       Q(ε) ← {1, 2, ..., |D|}
3       O_i ← {1, 2, ..., |D_i|} for all 1 ≤ i ≤ |D|
4       for n from n_min to n_max (or until ∀i O_i = ∅):
5           for d from 1 to |D|:
6               for each i in O_d:
7                   s ← D_d[i..i + n − 1]
8                   if Q(s) has not been assigned
9                       Q(s) ← Q(s[1..n − 1]) ∩ Q(s[2..n])
10                      if Q(s) ≤ t + 1
11                          Discard Q(s)
12                          O_i ← O_i − {i − 1, i}
13                  if Q(s) has been assigned
14                      Add d to P(s)
15          for each s where |s| = n and |P(s)| > 0:
16              if |Q(s)| − |P(s)| ≤ t
17                  Discard P(s)
18              else
19                  L ← L ∪ {s}
20                  Q(s) ← P(s)
21          Discard Q(s) for all s with |s| < n
22      return {L, P}
```

Figure 4: Optimized pseudocode for index construction. D is the document collection, O maintains a list of offsets in each document, Q contains temporary query results, and P contains the posting lists of n-grams that have been added to the index. In practical implementations, D, O, and $P(s)$ for $|s| < n-1$ are stored compressed in persistent storage.

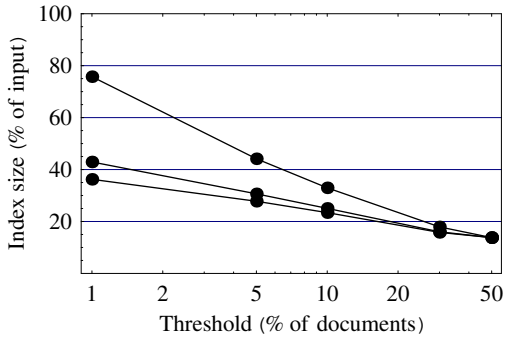


Figure 5: The impact of subset encoding on the size of the compressed posting lists for various values of the threshold parameter. From top to bottom the curves show the index size with no subset encoding, with rarest-substring subset encoding, and with full subset encoding. Data is the TREC 10MB collection. Whether or not subset compression is used, all posting lists are compressed by interpolative coding.

term passing the heritable condition, using this information to prune the terms, then making another pass to build the posting lists of the remaining terms. This optimization is not shown in the pseudocode, but is straightforward to implement.

4. ENCODING THE INDEX

Although our method could be the basis for an in-memory index, for example for searching a document in a text editor, our experiments in this work are limited to disk-based indexes; consequently, query time is dominated by disk accesses, allowing us to emphasize space-efficient codes over ones that are fast to decode. All codes that we evaluated were built on Moffat and Stuiver's interpolative coding[19], rather than a faster-to-decode but more complex alternative such as the word-aligned codes of Anh and Moffat.[2]

In section 3 we noted that, typically, the closer $|Q(s)|$ is to $|P(s)|$, the less information is conveyed by adding s to the index. Therefore, the bits that we invest in encoding $P(s)$ should be proportional to $|Q(s)| - |P(s)|$ (herein $Q(s)$ is understood to be the query against the index omitting the n-gram s from the lexicon). A simple way to do this is using a *subset encoding*: having determined both $Q(s)$ and $P(s)$ we can compute the *index set* $I(P(s), Q(s))$, which contains $n \in \mathbb{N}$ if and only if the nth largest element of $Q(s)$ is in $P(s)$. This set is the same size as $P(s)$ but has a smaller range and a smaller average gap, resulting in significantly better compression by standard list compression methods such as Golomb coding and interpolative coding. Figure 5 demonstrates the compression achieved by subset compression on the TREC 10MB collection for various threshold values.

Unfortunately, although highly effective at compressing the index, subset encoding makes decoding expensive for long queries because a roughly quadratic number of posting lists must be retrieved to compute the result, and a quadratic number of intersections must be performed in the dynamic programming algorithm that computes $Q(s)$ for each substring s of the query. As a compromise, we experimented with a simplified method where each posting list was encoded as a subset of the posting list of its rarest substring that was also in the index; we call this *rarest-substring subset encoding*. This achieves good compression and much more practical query times, but its query times are still not competitive with the independently-encoded posting lists of classical n-gram indexes.

In the end, we relied on independent encoding of each term's posting list in order to keep query times competitive with classical n-gram indexes that also use independent encoding. Subset encoding may still prove useful in scenarios like desktop search where small indexes and predictable query times are more important than average query times. Also, if there is no access to the original document collection, and so no ability to perform a linear scan phase, subset encoding makes practical the extreme threshold setting of $t = 0$, where all query results are exact.

5. QUERY PROCESSING

Although our construction algorithm is significantly different from that of classical n-gram indexes, the result is still a standard inverted index, and query processing is performed in the same manner as on any other inverted index, with the noted exception that n-grams not occurring in the lexicon are assumed to occur in all documents.

Ogawa and Matsuda [20] highlighted the importance of *query planning* in n-gram indexes: the set of terms retrieved by the query affects both query time and precision. A simple query planning strategy, used as early as 1993 by Adams and Meltzer [1], is to separate a query of length q into $\lceil q/n \rceil$ non-overlapping segments and retrieve only these terms; two non-overlapping terms tend to have a smaller intersection than two overlapping terms, and in indexes with offset information, this plan is sufficient to ensure perfect precision. Ogawa and Matsuda used more sophisticated query planning to select a subset of these nonoverlapping terms.

A simple but effective query planning mechanism is to store in the lexicon the document frequency of each term, then retrieve the terms relevant to a query in order by their document frequency. As each term is retrieved it is successively intersected with a running query result. If at any point the result becomes a single document, we terminate early. Since long queries tend to occur in a single document and contain rare substrings, only a fraction of all terms typically needs to be retrieved. In our experiments this query planner was used for both types of indexes being compared.

Additionally, our queries do not retrieve terms that occur as proper substrings of terms already retrieved, because $s' \in S(s)$ implies $P(s') \supseteq P(s)$.

6. EXPERIMENTAL RESULTS

6.1 Methodology

Previous sections used smaller TREC-1MB and TREC-10MB collections, based on non-normalized article bodies from the first document and the first 11 documents in the TREC WSJ (Wall Street Journal) collection, respectively. In this section we prefer larger collections: a TREC-100MB collection based on the first 101 documents from the TREC WSJ collection, and an 86MB chr14 collection created by dividing the human chromosome 14 genome sequence into 4000-byte chunks overlapping by 20 bytes; this was derived

Parameters	Terms	Size (MB)	Query time (ms) Mean	Median
3-grams	52 173	16	569	484
TinyLex 50%	7 023	13	358	308
4-grams	305 692	37	139	52
TinyLex 7%	34 620	36	82	47
5-grams	1 094 542	62	84	37
TinyLex 2%	76 167	57	66	35
6-grams	2 767 914	93	67	31
TinyLex 0.5%	167 855	85	54	31

(a) Comparison of TinyLex and classical n-gram indexes on the TREC-100MB English text data set; percentage values are threshold parameters as a percentage of all documents. Query strings are drawn uniformly at random from the collection. Lexicon size is improved by 7.4, 8.8, 14, and 16 times.

Parameters	Terms	Size (MB)	Query time (ms) Mean	Median
7-grams	16 384	29	106	66
TinyLex 20%	13 164	29	88	57
8-grams	65 536	52	68	47
6% threshold	25 930	51	65	47
9-grams	262 037	75	60	31
3% threshold	45 269	73	61	32
10-grams	1 032 550	100	40	31
1% threshold	79 500	100	42	31

(b) Comparison of TinyLex and classical n-gram indexes on the chr14 genome data set; percentage values are threshold parameters as a percentage of all documents. Query strings are drawn uniformly at random from the collection. Lexicon size is improved by 1.2, 2.5, 5.8, and 13 times.

Figure 6: Comparison of classical n-gram indexes and TinyLex indexes of comparable index size. TinyLex demonstrates comparable index size and query performance but much smaller lexicons.

from NCBI Build 36.1 [11] as distributed by the UCSC Genome Browser project [12].

For each corpus, we constructed a series of classical n-gram indexes without offset lists; for TREC-100M, n ranged from 3 to 6, and for chr14 n ranged from 7 to 10. For each classical n-gram index, we constructed a corresponding TinyLex index with a threshold value selected so that the final index sizes would be close to that of the classical indexes (all threshold values are specified as a percentage of all documents).

To eliminate the effects of caching, all reads of posting lists and documents were done against a 7200 RPM raw disk (no filesystem) with no buffering or caching other than an on-disk 8MB buffer. All experiments were performed on a single 2.0 GHz CPU core.

All query times include all phases of query processing: list retrieval, list processing, and false positive scanning. Median and mean query times are both given because the distribution of query times is not consistently symmetric and may contain outliers.

6.2 Lexicon size

Our primary goal was to demonstrate that we can achieve index size and query time comparable to classical n-gram indexes using a much smaller lexicon. For each index and each query length, 30 seconds of queries were performed and the mean and median query time taken. Queries were drawn from the collection uniformly at random. To ensure that processing of documents containing the query did not dominate query time, the query times shown here are for large queries of length 30–50 characters.

As shown in Figures 6(a) and 6(b), TinyLex achieves a dramatically smaller lexicon with no penalty to median query time or overall index size. The index size is comparable overall, rather than smaller, because smaller n-grams have larger posting lists. On the TREC-100M set, the mean query times of the TinyLex index are significantly less than the classical n-gram index because its query distribution is less skewed; that is, there are less outlying queries that take an unusually long time. We suspect this is because n-grams in English text tend to follow a form of Zipf's law, as shown by Egghe [10], whereas n-grams in genome sequences do not. Although the times shown here are for long queries, the TinyLex indexes never exceeded 2.7 times the classical n-gram median query time for any query length.

Note that although in these tables we focus on TinyLex indexes with a few discrete threshold values, one of the important advantages of our index is that is more finely tunable than the classical n-gram index: the index size varies gradually as the threshold parameter is varied, allowing it to be fine-tuned to meet specific resource constraints.

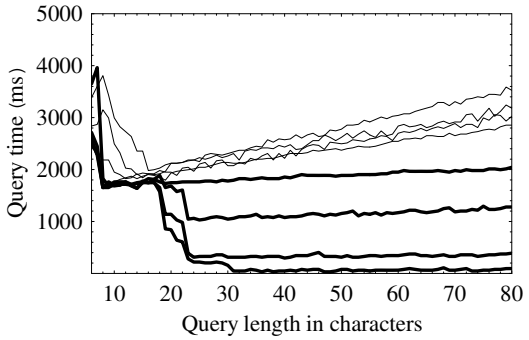
6.3 Worst-case query performance

In section 1, we explained that classical n-gram indexes can exhibit pathologically expensive query behavior on certain inputs, and that our construction avoids this. To demonstrate this, we generate a *bad set* of query strings that produces large numbers of false positives in query results. The procedure for generating the strings starts with a short, frequently-occurring string and iteratively expands it while keeping its query set large. These sets are intentionally artificial and meant to illustrate worst-case performance.

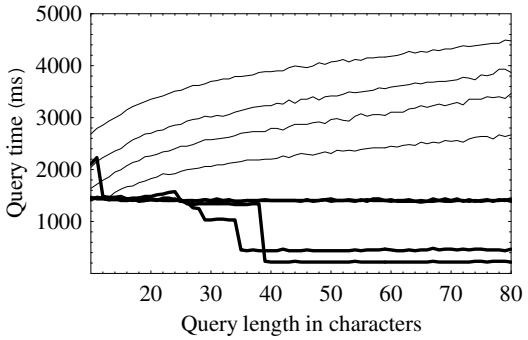
Each bad set generated contains one string of every length, each formed by repeating a single generator string enough times to fill the query. For the TREC-100MB collection, the generator string was " `the company said`", and for the chr14 collection, the the generator string was "t". We measured the query performance of each of the indexes used in the comparison in section 6.2. As shown in Figure 7, the difference between the TinyLex and classical indexes on these queries is dramatic: even the smallest TinyLex indexes outperform the largest classical n-gram indexes by at least 30% on long queries, despite being 7.2 and 3.5 times smaller. The best TinyLex indexes achieve query times close to zero, about 15–30 times faster than the classical n-gram index of comparable size. The failure of even highly precise classical n-gram indexes to tackle natural queries like these is a marked disadvantage that our construction corrects.

6.4 A word n-gram index for phrase search

As mentioned in section 2, it's common in language modelling to create variable-length n-gram models using words rather than characters as units. The ubiquitous *phrase search* feature of search engines can be cast as a query for docu-



(a) TREC-100M. From top to bottom, classical indexes (thin lines) are 3-, 4-, 5-, 6-grams, TinyLex indexes (thick lines) are 50%, 7%, 2%, 0.5% threshold. Query: (`the company said`)*



(b) chr14. From top to bottom, classical indexes (thin lines) are 7-, 8-, 9-, 10-grams, TinyLex indexes (thick lines) are 30%, 6%, 3%, 1% threshold. Query: (`t`)*

Figure 7: The query time required for the bad sets of query strings on both classical (thin lines) and TinyLex (thick lines) n-gram indexes.

ments containing a particular word n-gram; the simple word-based index is the special case of this with n-gram length equal to one.

All of the problems with n-gram indexes described in section 1 apply equally to this type of index: unless offset lists are stored for each index term, phrase queries can easily produce large numbers of false positive documents that contain all the words in the phrase but not the phrase. For example, the words "be" and "with" occur in every document of the TREC-10MB collection, but the phrase "be with" occurs in only one.

Storing offsets to perform phrase searches is expensive in storage cost and query processing time. The *nextword indexes* of Williams et al. [23] take a different approach, storing the set of all word bigrams and using standard compression techniques to reduce the index size; but these indexes are still very large, about 60% of the collection size, due to a large number of terms. For example, in the TREC-10MB collection with about 38,000 unique words, there are over 500,000 unique bigrams. A similar problem occurs at the character level in documents featuring large alphabets, such as Asian-language texts.

The hybrid indexes of Bahle et al. [4] reduce storage requirements by pruning the set of bigrams stored, and accelerate queries by using a query planning technique based on the document frequencies of the constituent words. Their pruning criterion retains only pairs where the first word of the bigram is among the most k common words. By applying the TinyLex construction to word n-grams, and using our query planning scheme that retrieves terms of low frequency first, we expect to achieve similar advantages, while performing better on queries where bigrams are insufficient, or where the second word (rather than the first) is the common one.

We encoded each document in the TREC-10MB collection as a sequence of word symbols, and then used our index construction algorithm directly on these sequences. The portion of the index describing 1-grams is effectively a simple word-based index, and was used as our base index size. As shown in Table 1, the additional space requirement of adding larger n-grams is comparable to the hybrid indexes of Bahle et al. in typical parameter ranges. Although query performance remains to be determined in future work, this offers a promising alternative to existing methods of phrase searching.

Threshold	2-grams	3-grams	Size (% of input)
TinyLex 1%	137 002	3910	19%
TinyLex 5%	31 743	467	6.4%
TinyLex 10%	11 530	165	3.0%
TinyLex 20%	3 202	57	1.4%
TinyLex 50%	574	0	0.4%
Hybrid 3	15 669	0	2.1%
Hybrid 6	32 892	0	4.4%
Hybrid 254	164 827	0	20%
Offsets	—	—	19%

Table 1: Size of an auxiliary TinyLex phrase index over the TREC-10M collection, which has 38,000 distinct words, for a variety of threshold values. Size does not include unigrams. The Hybrid k schemes of Bahle et al. [4] retain only bigrams where the first word was among the most k frequent by document frequency. The final row shows the overhead of adding offset lists.

An important advantage of TinyLex over nextword indexes is that it includes a small number of 3-grams and 4-grams that would cause a large number of false positives in a nextword index. An example from our experiments is "of the first", a phrase which occurs in only 1% of documents, even though each bigram in it occurs in at least 21% of documents. Even an unpruned nextword index, which requires many times more storage, is susceptible to this limitation.

6.5 Other results

A number of works such as Adams and Meltzer [1] and Mayfield and McNamee's n-gram stemming [17] have used a two-phase approach to reduce index size where first words are extracted from the collection and then n-grams are extracted from each word. This prevents queries that span

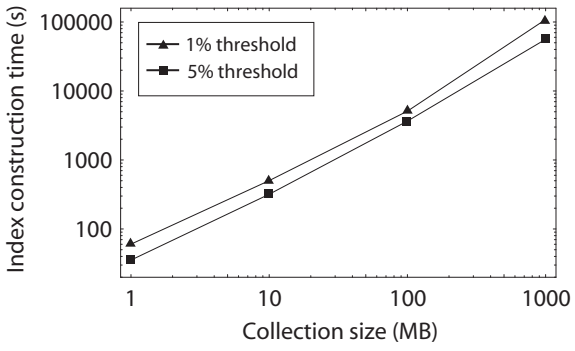


Figure 8: Scalability of TinyLex construction time at 1% and 5% threshold values over TREC collections of various sizes; scaling is roughly linear, tending up for the largest collection.

words, but still permits word and partial word queries. We can adapt our approach to use this technique by simply excluding any n-grams that contain internal spaces; the result is an n-gram index that operates as a functional replacement for a word-based index with a much smaller lexicon size. For example, a 7%-threshold TinyLex index over the TREC-100MB collection yields a 15 MB index with only 13548 terms in its lexicon; a word-based index over the same collection has 100449 terms.

One interesting result we found is that for a fixed threshold percentage, the TinyLex index size is largely independent of the document size – as documents become smaller, the number of documents grows but the lexicon shrinks because shorter n-grams are selected by the construction. This is useful because it allows us to freely select a document size close to the size of a unit of transfer, such as a disk block or cache line, keeping the index as small as possible while still assigning unit cost to each document scan. It also demonstrates the ability of the index to tune itself to different types of collections; in classical n-gram indexes approximating a similar invariance of index size over different document sizes requires the parameter n to be systematically varied. In this experiment we varied the size of the documents into which we divided the human chromosome 14 genome sequence used for the chr14 collection, maintaining an overlap of 20 bytes, and computed 5% threshold indexes on each. The results are shown in Table 2.

Finally, we demonstrated scalability of index construction by completing a 1% and a 5% threshold index on the TREC-1MB, TREC-10MB, and TREC-100MB collections, and a TREC-1GB collection drawing from the TREC AP (Associated Press), WSJ (Wall Street Journal), and FR (Federal Register) collections. Results are shown in Figure 8.

6.6 Limitations

Our work has two primary limitations as compared to classical n-gram indexes. First, because index construction is a multipass algorithm performing a large number of queries, index construction is more expensive in time and space than for classical n-gram indexes, which need only make a single pass over the collection followed by a sort. Using the index pairs from section 6.2 on the TREC-100MB collection, construction of the TinyLex index required 9.8 times longer on average than a classical n-gram index of comparable size.

Document size	Terms	Index size (MB)	Peak term length
500	4 167	63	6
1000	7 244	55	6
2000	16 170	60	7
4000	30 165	56	8
8000	63 392	59	8
16000	121 150	57	9

Table 2: A TinyLex 5% index on the chr14 collection with various document sizes. As the document size is varied, the compressed index size remains roughly constant. The peak term length is the most frequently-occurring length of n-gram in the lexicon, and increases with document size.

Second, classical n-gram indexes are able to return empty query results for a large class of queries not existing in the document (namely, those containing a substring of length n that does not occur in the document); due to our modified query computation, we produce nonempty results for these queries, and may have to scan up to $t + 1$ documents to determine that the query does not occur. In applications where this is important, it may be preferable to set $n_{\min} > 1$ and to return empty query results for queries containing a string of length $n_{\min}$ not in the lexicon. We chose $n_{\min} = 1$ in our experiments because it results in smaller lexicons, slightly smaller indexes, and enables queries on arbitrarily short strings.

Finally, as detailed below, the problem of generalizing our approach to dynamic collections such as those used by web search engines remains open.

7. FUTURE WORK

Techniques for incremental update on ordinary inverted indexes do not apply to TinyLex indexes in a straightforward manner, partly because some of the information needed for update is not available explicitly in the lexicon, and partly because of strong dependencies between index terms. For example, suppose we delete or modify a document that happened to be one of the false positive documents for the query on s, before s was added to the index. This may make $|Q(s)| - |P(s)| \leq t$ where it was not before, necessitating the removal of s from L; but not only is there no efficient way to identify all such s from the removed document contents, but removing s from L may force superstrings of s in other documents to be *added* to L (because their query size has increased), which may in turn require removal of other terms, and so on. Adding documents is even more problematic because the new document may be a new false positive for a term that was *not* previously included in the index, but should now be – there is no clear way to identify such terms.

Future evaluations include: evaluating phrase search query times for word n-gram indexes; evaluating the improvement our technique can yield for n-gram indexes that use offset lists and nonoverlapping queries; evaluations on large-alphabet natural-language texts such as Asian texts; and evaluations on larger collections.

8. CONCLUSIONS

In this work we described a method for generating variable-length n-gram indexes that have query performance and size comparable to classical n-gram indexes, but with dramatically smaller lexicons, enabling query engines to load more quickly and use less memory. It provides strong worst-case guarantees on the precision of query results, leading to more predictable query times, and permits queries on strings of any length. We anticipate that this type of n-gram index will make new types of interactive queries available on memory-constrained devices and will open the door to future research exploiting small lexicons.

9. ACKNOWLEDGEMENTS

I would like to thank Ken Church, Susan Dumais, Hugh Williams, and Michael Cameron for useful discussion and for recommending relevant resources, reviewers, and venues; and Nick Lester, Galen Hunt, Orion Hodson, Ed Nightingale, and anonymous reviewers for their reviews, which were critical in effectively developing this paper.

10. REFERENCES

[1] E. S. Adams and A. C. Meltzer. Trigrams as index elements in full text retrieval: Observations and experimental results. In *ACM Conference on Computer Science*, pages 433–439, 1993.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[3] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *DCC '97: Proceedings of the Conference on Data Compression*, page 201, Washington, DC, USA, 1997. IEEE Computer Society.

[4] D. Bahle, H. E. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 215–221, New York, NY, USA, 2002. ACM.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 182–189, New York, NY, USA, 2006. ACM.

[7] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 43–50, New York, NY, USA, 2001. ACM.

[8] R. J. D'Amore and C. P. Mah. One-time complete indexing of text: theory and practice. In *SIGIR*, pages 155–164, 1985.

[9] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *ALENEX/ANALCO*, pages 86–97, 2005.

[10] L. Egghe. The distribution of n-grams. *Scientometrics*, 47(2):237–252, 2000.

[11] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[12] W. Kent, C. W. Sugnet, T. S. Furey, K. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The Human Genome Browser at UCSC. *Genome Res*, 12(6):996–1006, 2002.

[13] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[14] C. P. Mah and R. J. D'Amore. Complete statistical indexing of text by overlapping word fragments. *SIGIR Forum*, 17(3):6–16, 1982.

[15] V. Mäkinen. Trade off between compression and search times in compact suffix array. In *ALENEX '01: Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, pages 189–201, London, UK, 2001. Springer-Verlag.

[16] U. Manber and S. Wu. GLIMPSE: a tool to search through entire file systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 4–4, Berkeley, CA, USA, 1994. USENIX Association.

[17] J. Mayfield and P. McNamee. Single n-gram stemming. In *SIGIR*, pages 415–416, 2003.

[18] P. McNamee, J. Mayfield, and C. Piatko. Haircut: a system for multilingual text retrieval in java. *J. Comput. Small Coll.*, 17(3):8–22, 2002.

[19] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[20] Y. Ogawa and T. Matsuda. An efficient document retrieval method using n-gram indexing. *Systems and Computers in Japan*, 33(2):54–63, 2002.

[21] V. Siivola and B. Pellom. Growing an n-gram language model. 2005.

[22] A. Stolcke. Entropy-based pruning of backoff language models. In *Proceedings of the DRAPA News Transcription and Understanding Workshop*, pages 270–274, 1998.

[23] H. E. Williams, J. Zobel, and P. Anderson. What's next? index structures for efficient phrase querying. In *Australasian Database Conference*, pages 141–152, 1999.

[24] S. Wu and U. Manber. Agrep: A fast approximate pattern-matching tool. In *Proc. of the Winter 1992 USENIX Conference*, pages 153–162, San Francisco, California, 1991.

[25] M. Yamamoto and K. W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, 2001.

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