Indexing Text Documents for Fast Evaluation of Regular Expressions

By

Ting Chen

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2012

Date of final oral examination: 07/12/12

The dissertation is approved by the following members of the Final Oral Committee:

AnHai Doan, Associate Professor, Computer Sciences
Jeffrey Naughton, Professor, Computer Sciences
Jignesh Patel, Professor, Computer Sciences
Christopher Re, Assistant Professor, Computer Sciences
Jin-Yi Cai, Professor, Computer Sciences and Mathematics

To my parents, Xiaohua Chen and Xiaozhu Zhang

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

ABSTRACT

Fast regular expression (regex) evaluation over text documents is a fundamental operation in numerous text-centric applications, such as information extraction, search, data mining, exploratory data analysis, and business intelligence. To support this operation, current work builds an inverted index for the k-grams in the documents. Given a regex R, the work analyzes R to infer k-grams that must be present in R, then uses the inverted index to quickly locate documents that are likely to match R.

In this dissertation we significantly advance the above state of the art. First, we develop a new method to build k-gram inverted index that takes far less time, works even when the set of k-grams considered does not fit into memory, and can handle the so-called "zero document" cases. Our index is also "space aware", in that it can make use of extra disk space, if any. Second, we index not just k-grams, but also distance based d-grams and transformed versions of the documents. Third, we show how to analyze a given regex at a much deeper level (than possible in current work) to derive more properties that we can then use to query the index. Taken together, these advances significantly reduce regex evaluation time, as demonstrated with extensive experiments over two real-world data sets. Finally, we develop a novel incremental index update method that greatly improves the index update efficiency. Our index update method can be used in applications that operate on dynamic text corpora.

Chapter 1

Introduction

Regular expressions are used in many applications to detect occurrences of patterns in a collection of text documents. Examples of such applications includes search engines, fraud detection systems and network packet filters. Examples of patterns are entities like emails, credit card numbers and phones.

This dissertation studies *indexing for regular expression evaluation*: the problem of how to efficiently construct, utilize and maintain indexes to speed up regular expression evaluation over large text corpora.

This chapter begins with examples of how real world applications employ regular expressions in their work flows. Then we show that quick evaluation of regular expressions is essential to these applications. Next we review the state-of-the-art indexing solutions to speed up regular expression evaluation and show that it is not satisfactory. We then present our solutions and list the contribution of this thesis. Finally we outline the rest of this dissertation.

1.1 Regular Expressions are Fundamental to Text-centric Applications

Evaluating regular expressions (regexes) is a fundamental operation in many text-centric applications. For example, information extraction (IE) applications often execute myriad regexes when evaluating hand-crafted IE rules, or when transforming training and testing examples into feature vectors [35, 1, 9, 17, 83, 30, 81, 48, 88]. For example, DBLife [83], a Web data analysis application built at the University of Wisconsin database group, uses 20K+ regular expressions to extract various entities like researcher names, publications, institutions and so on from a large collection

of crawled Web pages. These extracted entities are then stitched together by complex IE rules to construct high level semantic concepts like paper co-authorship, researchers' roles in academic conferences and so on. Similarly, in the IBM Avatar project [48] which builds a semantic search engine for IBM's intranet, researchers use regular expressions to annotate important entities such as telephone numbers, addresses, street directions, email addresses and so on. In both projects, regular expressions are used because they can succinctly encode a wide variety of text matches of a pattern.

As another example, in exploratory data analysis, such as finding potential frauds in the Enron email data set, an investigator may want to find emails that contain a phone number close to a bank name. To do so, he or she may want to encode this pattern as a regex, then evaluate it over the email data set. As yet another example, in business intelligence, such as finding negative reactions in the blogosphere to a newly launched product X, a company may want to evaluate a regex (over a large amount of crawled blogs) to find blogs that contain X's name close to "negative" words. Other domains that often require evaluating regexes include Web search, portal building, and text mining (see the related work). Intuitively, regexes are so "pervasive" because many applications must detect certain textual patterns, which in turn can often be encoded using regexes, or composed from lower-level regex-based patterns.

Finally, regular expressions are also used in tools that monitor data centers running Internet scale applications such as Facebook, LinkedIn and Google+. These Web sites can serve global user traffics in the rates of hundreds or even thousands of user requests per second. Behind the scene, all important aspects of a user request are logged. The log entries include the identity of the user, the exact request time, the services involved and for each service, the input and time spent on the service call. These log files can be used for monitoring the health of program operations as well as compliances to legal regulations. Software developers often use special tools to gather statistics from log files. For example, they may need to investigate why a remote service is slow to respond to user requests and causes service breakdown. To find the root cause, they first zoom into a certain period of time to reduce the scope of search. Next, they identify the names of related service calls and finally specify a pattern to extract service running times. Regular expressions are ideal for this

```
1. (?<!(Fax ))\b\d{3}-?\d{3}-\d{4}\b
2. [A-Z]{2,4}\s*@\s*\d{1,5}(\.\d{1,2})?
3. Prof(\.?|essor)\s+(\w+)\s+
4. [a-zA-Z\d_]+[a-zA-Z\d_\-:&=?/~.<>@:]+(\.com/|\.edu/|\.org/)
   [a-zA-Z\d_&?~.<>@:][a-zA-Z\d_\-:&=?/~.<>@:]+[a-zA-Z\d_\-:&=?/~]{2,100}
5. \-{5}\s*Message\sfrom\s*.{5,30}\s.{6,40}\s*on\s*[A-Z][a-z]
    {2,8},\s*\d{1,2}\s*[A-Z][a-z]{2,8}\s*\d{4}(.|\n){10,18}\s*\-{5}
```

Figure 1.1: Sample regexes that find phone numbers, stock symbol prices, persons, URLs and email headers, respectively.

kind of ad-hoc tasks that perform keyword, date and numerical pattern extraction. Indeed, many of today's log analysis tools use regular expressions to find the patterns before running user defined statistical functions over the matched data.

1.2 The Importance of Fast Regular Expression Evaluation

For the above applications, it is important that we can quickly evaluate regexes over the text corpora. First, many applications such as exploratory analysis are by nature *interactive*, in that users want answer to their regex-based queries quickly (e.g., in few seconds). Second, debugging regexes is often *iterative*. Consider for instance writing a regex to find phone numbers. We may start with a regex R_1, evaluate it over the corpus, realize that R_1 is not quite accurate (e.g., it also returns fax numbers), refine R_1 into R_2, evaluate R_2 over the corpus, and so on. Here fast regex evaluation helps speed up iterative debugging (and thus application development). Finally, many applications evaluate batches of many regexes (e.g., tens or hundreds of thousands) over large document corpora [83]. Here fast regex evaluation can drastically slash the total run time.

In fact, all three of the above scenarios arose in the two real-world applications that we have been deploying: Avatar, a semantic search engine at IBM, and DBLife, a Web data analysis application at the University of Wisconsin database group. Figure 1.1 shows sample regexes that we encountered in these applications, which originally motivated our project. In general, as the number of text-centric applications grows, so does the critical need for fast regex evaluation.

1.3 Using Indexes to Speed Up the Regular Expression Evaluation

Consequently, the problem of fast regular expression evaluation has received increasing attention (see the related work in Chapter 6). A promising current solution, as embodied by the Free system [22], is to index the documents. We illustrate Free's major components in Figure 1.2. Free builds an inverted index I for the k-grams in the documents (for all k up to a pre-specified value, say 10). Given a regex such as R = `David\s+\Smith`, Free analyzes R to extract k-grams that must occur in strings matching R (e.g., 2-grams "da", "av", ..., "th"), looks up the k-grams in I to locate a (hopefully small) set S of documents that potentially match R, then evaluates R only over S. Intuitively, if a k-gram occurs in strings matching R, then it must occur in documents that match R.

As described, Free is highly promising. But it is limited in several major ways, and in this dissertation we describe the Trex solution that removes these limitations. First, we show that Free is fundamentally not "space aware", in that it fails to use extra disk space (if any) for the index. In fact, it appears to show some counter-intuitive behaviors: adjusting a system parameter to increase the index size may actually shrink the index. Second, Free takes a long time to build the index, largely because it scans the corpus multiple times. Third, Free suffers from the zero document problem, in that when queried with a k-gram s not in the corpus, it will return all documents (and the regex will have to be evaluated on all of them). Fourth, Free indexes only k-grams. We show that indexing other document features can significantly improve regex evaluation. Fifth, at evaluation time, given a regex R, Free analyzes R to infer relevant k-grams (i.e., those that must occur in strings matching R). But this analysis is limited in that it focuses only on the syntactic structure of R. Finally, many real-world applications regularly add and delete documents from the text corpora. Free has to rebuild indexes from scratch when new documents are added or deleted and are thus inefficient.

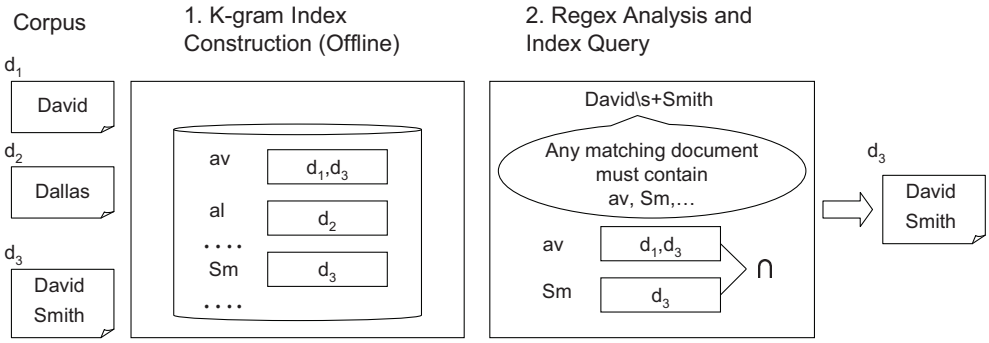


Figure 1.2: Overview of Free's index solution for regex evaluation: (1) it builds a k-gram index (here k = 2) for the three documents offline (2) Given a regex, Free finds the k-grams in the regex and queries the index for a set of documents that may contain strings matching the regex.

1.4 Trex: Our Solution to the Indexing Problem

To address the above problems, this dissertation presents Trex, a comprehensive indexing solution that greatly improves the index construction efficiency, introduces novel index features tailored for regexes, provides deeper regex analysis capability and provides incremental index maintenance over large text corpora.

1.4.1 Building the Indexes

Building k-gram indexes for large text corpora turns out to raise significant technical challenges. A straightforward solution is to keep the set S of all k-grams seen so far in memory, then scan the corpus, process each document d and augment S to keep track of which k-grams appear in d. This solution however works poorly because the set S often exceeds main memory. To fix this, we can sort and flush S to disk into runs whenever memory is full, then merge the runs, in the same spirit of sorting external data in RDBMSs. However, even in this case, the runs are still often too large. In response, Trex innovates by keeping track of only certain k-grams in each run, thus producing much smaller runs (and taking far less time). And yet when merging runs, Trex is able to generate entries for all k-grams, thus producing the same index I.

In many cases this index I is still too large. Thus we introduce the notion of (α, β) indexes. Roughly speaking, such an index keeps only α-selective k-grams, i.e., those that appear in no more than $\alpha * |D|$ documents, such that there is no other α-selective k-gram already in the index that is within a β "distance". We show how to build such an index efficiently. While far more compact than a full k-gram index, an (α, β) index I is problematic in that if a regex R contains no k-grams in I, then the index is useless and we end up evaluating R on the entire corpus, an expensive operation. We show how to extend I to handle this case.

Finally, we extend I to index certain powerful non-k-gram text patterns such as distance based d-grams and k-grams over transformed documents. These new index features are well suited for regexes that have no selective k-grams.

1.4.2 Using the Indexes

Our second set of technical challenges involves using the k-gram index I once it has been built. Given a regex R, in principle we can analyze R to derive an optimal "query" that involves only k-grams in the index I. We then can "execute" this query over I to obtain the minimal set of documents that may contain R. However, we prove that finding and evaluating such an optimal query can be quite expensive (the size of the query can be superpolynomial in the size of the regex R). Thus, for now we consider a greedy solution to find a reasonable query instead. We show how to develop a better solution than the current state of the art, by performing a deeper, dynamic analysis of the regex R, guided by the index I. The above solutions can also be applied to querying k-gram indexes over transformed documents (Chapter 4.6). We also develop methods to analyze regexes and query d-gram indexes (Chapter 4.5).

1.4.3 Maintaining the Indexes

Up until now, we have assumed that the underlying text corpus is static. However, many real-world applications regularly add and delete documents from the text corpora. Free does not consider such cases and thus it always build indexes from scratch. A more efficient approach is to incrementally maintain the index. Incremental index maintenance poses new challenges to Trex:

after adding new documents, the selectivity of k-grams may change and as a result we need to update the set of k-grams and their posting lists kept in the index. A naive solution is to rescan and re-compute the k-gram selectivity and then rebuild the whole index. Such a solution is slow because it rebuilds the index from scratch and thus is not satisfactory for many applications where documents are frequently added and index availability is crucial.

To ensure the index stays on-line during corpus updates, Trex accumulates in memory a separate index I' for the recently added documents besides the existing index I on disk. At query time, Trex queries both I and I' using the same index query method described earlier and combines the results. When the memory is full, Trex flushes I' to disk and merges it with I. To avoid costly corpus rescan, Trex augments the existing index with "extra" corpus information so that it can recover necessary information related to k-grams such as selectivity and posting lists. Using ideas similar to those underlying the index construction algorithm, Trex keeps the set S of all max length k-grams with their posting lists as the "extra" corpus information file. During the index merge, Trex sequentially reads existing the on-disk index I, the "extra" corpus information file and the in-memory index I' and writes the updated index. Because Trex processes structured files instead of rescanning the corpus, it reduces the index update times by factors ranging from 5 to 8 compared to the straightforward re-building from scratch solution.

Finally, we can apply the above idea directly to the maintenance of k-gram indexes over transformed documents. This thesis does not discuss how to maintain d-gram indexes.

1.5 Contributions and Outline of the Dissertation

Overall, in this dissertation we make the following contributions:

- We analyze and discuss the limitations of Free, the state-of-the-art solution for indexing text documents for fast regex evaluation.
- We describe a family of so-called (α, β) inverted indexes that can gracefully adjust to the amount of available disk space.

- We describe Trex, a solution to build (α, β) indexes that remove many fundamental limitations of Free (e.g., not space aware, multiple corpus scan, memory problem, zero document). Trex employs a significantly different solution approach than that of Free.

- We show how Trex can index features outside (α, β) indexes, to further reduce regex evaluation time.

- We define the problem of finding the optimal query to probe the index at evaluation time. We prove that the size of this query is superpolynomial in terms of the regex size, suggesting that a greedy solution may be a practical choice. We then develop a greedy solution that is empirically more efficient than that of Free.

- We describe extensive experiments with two deployed real-world applications (that we are maintaining). The results show that Trex significantly outperforms Free on regex evaluation, by 30-41% overall, and by as much as 94% on certain regexes.

- We develop a novel incremental index maintenance method for (α, β) indexes that greatly reduces the index update time by a factor of 5 to 8 in two real-world applications.

The rest of this dissertation is organized as follows: Chapter 2 defines the problem and reviews Free, the state of the art indexing solution for regular expression evaluation and its limitations. In Chapter 3 we outline our solution developed for the index construction. In Chapter 4 we describe our solution to analyze regexes and query the indexes. Chapter 5 presents Trex 's method for incremental k-gram index maintenance. Chapter 6 surveys the related work in the field of regular expression evaluation and indexing. Chapter 7 concludes the dissertation.

Chapter 2

Preliminaries

In this chapter, we first formally define the indexing problem for regular expression evaluation over large text corpora. Next we present an in-depth analysis of the strengths and limitations of Free: the state of the art solution.

2.1 The Indexing Problem

We begin by describing the problem considered in this dissertation. Let D be a corpus of text documents that are stored on disk. We want to build an index I such that given a regex R, we can use I to quickly locate a (hopefully small) set O of documents in D that potentially contain R (i.e., contain at least one string that matches R). O should include all documents in D that contain R and as few documents that do not contain R as possible.

2.2 The Free Solution

We now describe the Free solution [22] and its limitations. Free builds an inverted k-gram index over documents in D. A k-gram is a sequence of k consecutive characters in a document. To illustrate, consider the three toy documents $d_1 - d_3$ in Figure 2.1.a. Figure 2.1.b shows an inverted k-gram index I_1 that contain all 1-grams and 2-grams of these documents (thus $k \leq 2$).

The first entry of I_1, [\$: $d_1 d_2 d_3$], states that the 1-gram \$ appears in documents with IDs d_1, d_2, d_3 (abusing notation slightly, we use d_1, say, to denote both the document and its ID). Note that we automatically add the special character \$ to the start and the end of each document, to handle 2-grams that appear in those position (a common practice in k-gram generation). The list

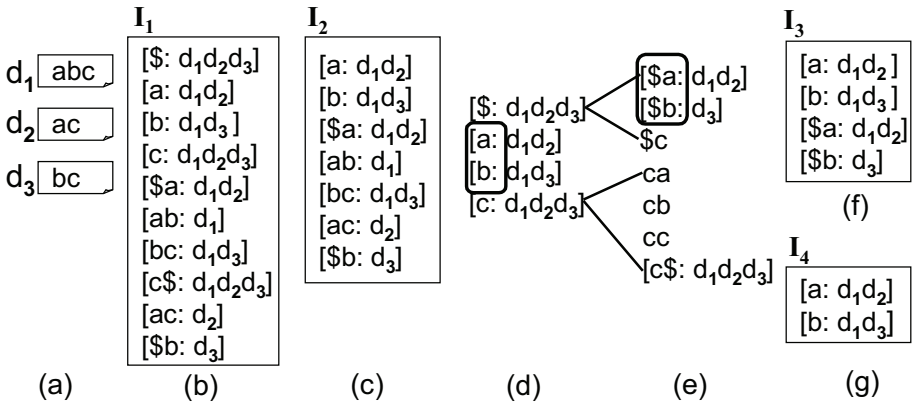


Figure 2.1: An example of how Free builds inverted k-gram indexes

$d_1d_2d_3$ is commonly called the *posting list* of the 1-gram \$. Similarly, the second entry of I_1 states that 1-gram a appears in d_1 and d_2. The last entry states that 2-gram $\$b$ (i.e., character b appearing at the start of a document) appears in d_3. Now, given a regex such as R = (ab)+, clearly ab must occur in any document matching R, and index I_1 shows that ab appears only in d_1. Consequently we only have to evaluate R over d_1 instead of $d_1 - d_3$, thereby saving time.

2.2.1 Building Inverted K-gram Indexes

Free wants to index all k-grams for $k \leq 10$. But this is impractical because there are too many of them (for a typical alphabet size of 200 characters, the number of 10-grams alone is 200^{10}). Consequently, Free defines the following notion [22]:

Definition 2.1 (α-selective k-grams) A k-gram s is α-selective with respect to a corpus D if and only if $0 < freq(s)/|D| \leq \alpha$, where $freq(s)$ is the number of documents in D that s appears in, and $|D|$ is the total number of documents in D. We call $freq(s)$ the frequency of s, and $sel(s) = freq(s)/|D|$ the selectivity of s.

Free selects an $\alpha \in (0, 1)$, then indexes only α-selective k-grams (henceforth *selective k-grams* when there is no ambiguity). Figure 2.1.c shows for example index I_2 of all selective k-grams for $\alpha = 2/3$ and $k \leq 2$, on $d_1 - d_3$. Here a k-gram is selective if and only if it appears in at most 2 documents.

Free sets α to be the value (determined empirically) at which randomly reading $\alpha * |D|$ documents takes as long as a sequential read of the entire corpus D. Intuitively, beyond this point the index is no longer useful.

Unfortunately the set of all selective k-grams can still be quite large. If so, building the index still takes much time and the index itself takes much disk space. Consequently, Free indexes only certain selective k-grams. In particular, after indexing a selective k-gram s, Free will not index any k-gram t that contains s as a prefix (even though by definition t is also selective). To do this, Free builds the index in an *a priori* fashion, in several iterations. For example, to build a k-gram index on $d_1 - d_3$, where $k \leq 2$ and $\alpha = 2/3$, Free proceeds in two iterations. In Iteration 1 (Figure 2.1.d), Free enumerates all 1-grams, scans D to count the frequencies of the 1-grams, then adds those found selective to the index (a and b in this case, as each appears in only 2 documents, see Figure 2.1.d).

In Iteration 2 (Figure 2.1.e), Free creates 2-grams by adding one character to the end of each unselective 1-gram of Iteration 1 (i.e., $ and c), scans D again to count the frequencies of the 2-grams, then add those found selective to the index ($a and $b in this case; note that $c for instance is not added to the index because it does not appear in the corpus). Figure 2.1.f shows the resulting index I_3.

Free then prunes index I_3 further, by removing any gram s a suffix of which is already in the index; see [22] for the detailed algorithm. Figure 2.1.g shows the final index I_4, which is *prefix-* and *suffix-free*.

2.2.2 Using K-gram Indexes

After building the index (e.g., I_4), Free enters the evaluation phase. Briefly, given a regex R such as (ab)+c, Free locates all maximal strings of R (ab and c in this case), generates *substrings* of these maximal strings (e.g., ab, a, b, c), use the substrings to probe the index, then unions or intersects the resulting posting lists as necessary, to obtain a final list L of documents that potentially match R. Free then evaluates R on the documents in L.

2.2.3 Limitations

As described, Free is limited in several important ways. First, it *scans the corpus multiple times*, one per iteration (thus 10 times if we index up to 10-grams [22]). This incurs large indexing time for large corpora.

Second, recall that in each iteration i-th, Free creates a set of i-grams (by adding one character to the end of each unselective gram in the previous iteration). This set can be very large (e.g., contain all possible i-grams). If it exceeds main memory, Free will *thrash* when scanning the corpus to count the frequencies of the i-grams in the set.

Third, Free does not exploit extra disk space (if any). To see this, consider evaluating R = ac. For the example in Figure 2.1, Free will build index I_4, as discussed earlier. Index I_4 contains an entry for a, but not ac. Hence, using I_4, the best we can do is to evaluate R on d_1, d_2 (returned by looking up a in the index). Now suppose we have some extra disk space and can build and store index I_2 (Figure 2.1.c). This index contains an entry for ac. Thus using I_2 we only need to evaluate R on d_2 (returned by looking up ac itself), a clear time saving. Thus, instead of building a "minimal" index as in Free, we should build as large an index as disk space allows (assuming that we can do so efficiently).

Unfortunately, Free appears to be fundamentally unable to exploit extra disk space. Given extra space, one may think that increasing α would admit more k-grams into the index, thereby increasing its size. In fact, the index may shrink. For example, if $\alpha = 1$ (or very close to 1), the index for $d_1 - d_3$ in Figure 2.1.a contains just four 1-grams: $, a, b, c, smaller than say I_2 (at $\alpha = 2/3$, see Figure 2.1.c). The main reason is that Free keeps the index prefix- and suffix-free. Hence, if the above four 1-grams are already in the index, then no other k-grams can be in the index. Clearly, "prefix- and suffix-free" is too strong a pruning criterion, to help generate useful indexes. In Trex we address this problem by introducing a more general pruning criterion called β-optimal (see Section 3.3).

Fourth, Free suffers from the *zero-document* problem. Ideally, if a k-gram s does *not* appear in the corpus, then consulting the index should return *zero* document. The Free index however returns *all* documents. To see this, consider cc. Index I_4 does not contain cc. So either (1) cc is

a selective 2-gram not in I_4, or (2) cc is an unselective 2-gram, or (3) cc does not appear in the corpus. Free has no way to know. Thus, to be safe it will return all documents when asked about cc. So given regex R = cc, Free will evaluate R on the entire corpus $d_1 - d_3$, even though R matches no document. This is clearly wasteful and unsatisfying.

Finally, at evaluation time, given a regex R, Free analyzes R to infer relevant k-grams. But it is clear from Free's description that it looks for k-grams only among the substrings of the maximal strings in R. This set of k-grams can be limited, thereby not fully utilizing the potential of the index.

2.3 Summary

In this chapter, we have defined the problem and performed a detailed analysis of the strengths and limitations of Free: the state of the art k-gram indexing solution for evaluation regex over large text corpora. Overall, Free is a promising solution but leaves huge room for improvements before we can deploy an indexing solution like it for large scale systems. In the rest of the dissertation we show how Trex addresses the above limitations.

Chapter 3

Building the Indexes

In this chapter, we describe how Trex builds indexes over a set of documents for regular expression evaluation. Trex employs a solution that is radically different from Free. First, we show how to build an index of all k-grams (up to a prespecified length k). Next, we introduce the notion of α-selectivity and β-optimality and explain how to keep only α-selective and β-optimal k-grams in the index. We then motivate the need to add unselective k-grams, to handle zero-document lookups. After that, we discuss two useful non-k-gram features to the index: distance based d-grams and indexing transformed documents. Finally we present empirical results on the performance of our indexing methods in comparison with that of Free.

3.1 Generating All K-grams

Recall that Free scans a corpus D multiple times. In contrast, Trex scans D only once to build a relatively small "summary", then uses this summary to build the index.

To explain this idea, consider building the inverted index of *all k-grams*, $k \leq 2$, for the corpus D of documents $d_1 - d_3$ in Figure 3.1.a (copied from Figure 2.1.a). To do this, we initialize an empty inverted index I of 2-grams in memory, then scan D and index its documents, one at a time. First we index d_1: for each distinct 2-gram s of d_1, we add an entry $[s : d_1]$ to I. Next we index d_2. Note that if a 2-gram s of d_2 already exists in I, then we simply add d_2 to the posting list of s (i.e., the list of all doc IDs in which s appear). Then we index d_3, and so on.

As we scan D, index I grows and eventually may exhaust memory. If so, we store I as a file on disk, re-initialize I to be empty, then proceed with the next document. After scanning D, we

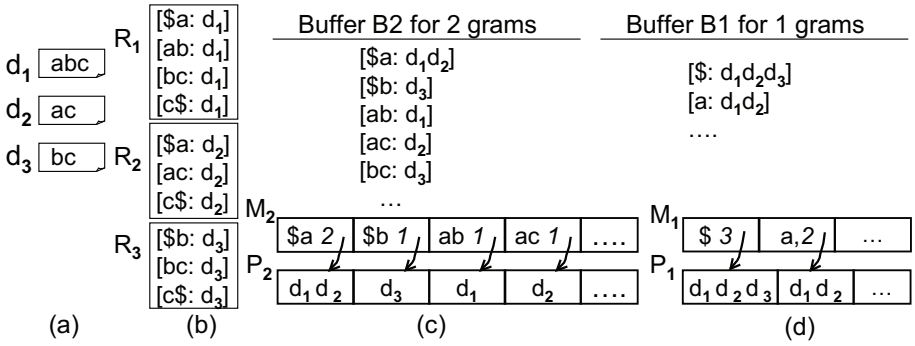


Figure 3.1: An example of how Trex builds inverted k-gram indexes.

thus obtain a set of files called *runs*, each of which can be viewed as an incomplete portion of the 2-gram inverted index. As an example, Figure 3.1.b shows three runs $R_1 - R_3$, each "indexes" one document (in practice a run typically indexes many documents). Note that we sort each run in memory based on its 2-grams before flushing it to disk. So run R_1 for instance stores $a, then ab, bc, etc. in that order (assuming that $ is ranked before a).

Next, we scan the sorted runs $R_1 - R_3$ and merge their entries to build the complete index of 2-grams over D, in a way similar to sort-merge join. Specifically, we consider the top-ranked 2-gram $a, and merge its entries in R_1 and R_2 to create the first index entry [$a: d_1d_2] (at the top of Figure 3.1.c). Next we consider the second-ranked 2-gram $b and create the second index entry [$b: d_3], and so on.

We accumulate the index entries in a memory buffer B_2, then flush them to two files M_2 and P_2 whenever B_2 is full (Figure 3.1.c). To flush the first index entry [$a: d_1d_2], we write posting list d_1d_2 to file P_2, then write "$a 2 p" to file M_2 to states that the posting list of $a has 2 documents and starts at the address p in file P_2. We flush subsequent index entries in a similar fashion (Figure 3.1.c).

Thus, merging runs $R_1 - R_3$ produces the 2-gram index in the two files M_2 and P_2 (Figure 3.1.c). Interestingly, while doing this we can also produce the 1-gram index in the two files M_1 and P_1 (Figure 3.1.d). To see this, let us rewind to the moment when we start to consider the top-ranked 2-gram $a. At this moment we also create 1-gram $, which is a prefix of $a. Then as we read in entries from $R_1 - R_3$ to create the posting lists for 2-grams, we can "graze over" the

entries to collect sufficient information to create the posting list for the 1-gram $. For example, when reading in entry [$a: d_1] of R_1, we can add d_1 to the posting list of $. When reading in [$a: d_2] of R_2, we can add d_2. When reading in [$b: d_3] of R_3, we can add d_3. No other entries of $R_1 - R_3$ contain $ as a prefix. So the posting list of 1-gram $ is $d_1d_2d_3$. We accumulate the entries for 1-grams in a memory buffer B_1, and flush them to disk to create files M_1 and P_1 (Figure 3.1.d), exactly as in the case of creating M_2 and P_2.

It is important to note that when considering a 2-gram s, we consider only *one* 1-gram: the 1-character prefix of s. There is no need to consider any other character t of s, because t will eventually appear as the prefix of some other 2-gram s'. For example, when considering $a, we consider only the 1-gram $. We do not consider a because it will eventually appear as the prefix of ab (in run R_1). By the time we have finished scanning $R_1 - R_3$ to create the 2-gram index M_2, P_2, the above algorithm is guaranteed to have also produced the 1-gram index M_1, P_1.

In fact, the above algorithm can be generalized straightforwardly to any $k \leq n$. That is, while scanning the runs, we keep n buffers $B_n, B_{(n-1)}, \ldots, B_1$ and use them to concurrently build n-gram index, $(n-1)$-gram index, ..., 1-gram index in files $M_n, P_n, M_{n-1}, P_{n-1}, \ldots, M_1, P_1$. By the time we finish scanning, we have obtained the complete k-gram index, $k \leq n$, on disk in the above files. Figure 3.2 shows the pseudo code of the above algorithm.

Discussion: Clearly, Trex scans the original corpus only once to create the runs. Since later Trex uses the runs to generate k-gram indexes for all $k \leq n$, it follows that the runs should capture all n-grams (so that sufficient information is retained to build the indexes). In a sense, Trex works backward from creating n-grams to 1-grams, whereas Free works forward from 1-grams to n-grams. Finally, whenever memory is exhausted, Trex flushes into the runs. So unlike Free, Trex works well even when the set of k-grams exceeds available memory (because it does not keep such a set in memory).

Input: a corpus D, an integer n
Output: all k-grams with length $\leq n$ with their posting lists

1. Initialize an empty inverted index I of grams

2. Scan the corpus D and build runs

   2.1 If all documents are scanned, go to step 3

   2.2 Scan the next doc d

   2.3 Foreach distinct length n k-gram g of d, add $[g, d]$ to I

   2.4 If memory is full

      2.4.1 Sort I on its grams

      2.4.2 Flush I to disk as a new run R

      2.4.3 Re-initialize an empty inverted index I

   2.5 Go to step 2.1

3. Initialize n buffers B_i, n meta files M_i,

   and n posting list files P_i for $i \in [1 \ldots n]$

4. Sort-Merge the runs

   4.1 Read the top-ranked gram g_{min} with its posting lists l

   4.2 Foreach length i ($i \in [1 \ldots n]$) prefix g_i of g_{min} do

      4.2.1 Add g_i and l to B_i

      4.2.2 If B_i is full

         Add all posting lists in B_i to P_i

         Foreach k-gram g' in B_i, add $[g', \#Docs, posting\_list\_address]$ to M_i

   4.3 If some run is unfinished, go to step 4.1

Figure 3.2: Building the inverted index of all k-grams

3.2 Keeping only α-Selective K-grams

Keeping all k-grams in the index can be very expensive, as discussed in Chapter 2. Fortunately, we can easily modify the above algorithm to keep only α-selective k-grams, for a prespecified α. Consider again the example in Figure 3.1. Recall that we accumulate posting lists for 2-grams in buffer B_2, then flushing them to disk whenever B_2 is full (Figure 3.1.c). Since a 2-gram s is selective only if s appears in at most $\alpha * |D|$ documents, the moment the size of a posting list exceeds $\alpha * |D|$, we can simply discard it from B_2, because the corresponding 2-gram cannot be selective. We proceed similarly for 1-grams' entries in buffer B_1. The modified algorithm is guaranteed to produce all and only selective k-grams in the files M_2, P_2, M_1, P_1. It can be generalized straightforwardly to any $k \leq n$.

3.3 Keeping only β-Optimal K-grams

Recall from Chapter 2 that we want to keep as many selective k-grams in the index as space permits. Toward this goal, we introduce the following notion:

Definition 3.1 (β-optimal k-grams) A k-gram s is β-optimal with respect to an index I if and only if there is no substring s' of s such that s' is also in index I and $[sel(s') - sel(s)] < \beta$, where $sel(s)$ is the fraction of documents (in the corpus that I indexes) that contain s.

Note that if s' is a substring of s, then $sel(s') \geq sel(s)$. If $sel(s')$ is "much greater" than $sel(s)$, then we may want to keep both s' and s in the index. Otherwise, it may be better to just keep the smaller string s'. That is the intuition behind the notion of β-optimality. We have

Definition 3.2 ((α, β)-grams and (α, β)-indexes) A k-gram is an (α, β)-gram if it is α-selective and β-optimal. An index is (α, β)-index if it contains only (α, β)-grams.

It is easy to show that for a given α, it makes sense to vary β only from 0 to α. If $\beta = 0$, the resulting $(\alpha, 0)$-index is exactly I_{all}, the index of *all* selective k-grams that the previous subsection builds. If $\beta = \alpha$, the resulting (α, α)-index is exactly I_{free}, the prefix- and suffix-free index that

Free builds. If β is in between 0 and α, the resulting index contains fewer selective k-grams than I_{all}, but more selective k-grams than I_{free}.

Consequently, selecting an α lets us control the number of selective k-grams, then selecting a β between 0 and α lets us control how many of these grams should be in the index. Thus, the pair (α, β) lets us adjust the index size to the available disk space.

With the above in mind, we now describe how to build an (α, β)-index. Given α, we first proceed exactly as in Section 3.2 to build I_{all}. Consider again the example in Figure 3.1, where I_{all} consists of files M_1, P_1, M_2, P_2.

In the next step, we process these files to remove k-grams that are not β-optimal. To do this, we observe the following:

Proposition 3.3 Let s be a k-gram of size n, $maxprefix(s)$ be the prefix (of s) of size $(n-1)$, $maxsuffix(s)$ be the suffix (of s) of size $(n-1)$. Then s is not β-optimal if and only if either (a) $[sel(maxprefix(s)) - sel(s)] < \beta$ or (b) $[sel(maxsuffix(s)) - sel(s)] < \beta$.

This observation suggests an algorithm that first removes all k-grams that satisfy condition (a) of Proposition 3.3, then removes all k-grams that satisfy condition (b).

1. Removing K-grams that Satisfy Condition (a): To do this, we start by reading file M_2. Consider the first 2-gram of that file: $a. Checking if $a satisfies condition (a) requires computing the selectivity of $ (i.e., $maxprefix($a)). This in turn requires computing the number of documents in which $ appears. This number (which is 3 in this case) can be found at the start of M_1.

In general, since M_2 and M_1 are sorted, as we scan M_2 and M_1 concurrently, for any 2-gram s in M_2 we will be able to quickly consult M_1 to find the number of documents in which $maxprefix(s)$ appears. This in turn allows us to check if s satisfies condition (a). If yes, s is a non-β-optimal k-gram, and we can immediately remove it from M_2. Note that here we cannot yet check for condition (b) because that requires knowing the number of documents in which $maxsuffix(s)$ appears. This number unfortunately is most likely "somewhere further down" in file M_1, not yet in the part of M_1 that we have read in so far.

By processing files M_2, M_1 in this fashion, we obtain files M_2', M_1' that contain only selective k-grams that do not satisfy condition (a).

2. Removing K-grams that Satisfy Condition (b): We now remove all k-grams that satisfy condition (b) from M_2', M_1'. To do so, we first use sorting to reverse the order of the k-grams in these files. Next, we proceed exactly as in Step 1, but considering $maxsuffix$ instead of $maxprefix$ of the k-grams. It is not difficult to prove that since the k-gram orders have been reversed, processing $maxsuffix$ can proceed exactly as described for $maxprefix$ in Step 1.

After processing, we obtain files M_2'', M_1'' that contain selective k-grams that satisfy neither condition (a) nor condition (b) and thus are β-optimal. We now process P_2, P_1 to obtain P_2'', P_1'' that contain only the posting lists of those β-optimal k-grams in M_2'', M_1''. The set of files $M_2'', P_2'', M_1'', P_1''$ form the desired (α, β)-index. And the above algorithm can be generalized straightforwardly to any $k \leq n$.

3.4 Adding Unselective K-grams

We now consider the zero-document problem, i.e., determining if a given k-gram t does not appear in the corpus D. The above (α, β)-index I_s of selective k-grams cannot solve this problem. Indeed, if t does not appear in I_s, we can only infer that either (a) t is selective but not in I_s, or (b) t is unselective, or (c) t does not appear in D. It is not possible to make a more precise inference.

To address this problem, we propose to keep a separate index I_u that lists all *unselective k-grams* in D (with k up to a pre-specified value, as usual). It is not difficult to show that

Proposition 3.4 A k-gram t does not appear in a corpus D if and only if (a) neither t nor any of its substrings appears in index I_s, and (b) t does not appear in index I_u.

Given this result, we can modify the lookup algorithm as follows. Given a k-gram t, if t appears in I_s then return its posting list. Otherwise generate all of its substrings, look them up in I_s, then return the intersection of all resulting posting lists (if any). If none of t's substrings appears in I_s, then check if t appears in I_u. If yes, t is an unselective k-gram, return the list of all documents in D. Otherwise, t does not appear in D, return a list of zero documents.

It turns out that the algorithm to build the (α, β)-index described earlier can be trivially modified to also generate I_u. Recall from Section 3.2 that the moment the size of a posting list exceeds $\alpha * |D|$, we remove that posting list from the buffer. Now, in addition to doing that, we also write the corresponding k-gram (which is unselective) to I_u.

One may wonder if the set of unselective k-grams is large, thereby creating a large I_u. Our experiments have in fact shown the opposite. Perhaps not surprisingly, k-grams follow a Zipfian distribution. It turned out that most k-grams fall into the tail of this distribution. That is, they appear in a relatively few documents and are selective. Thus, I_u remains quite small (taking only a few tens of Mbyte in our experiments).

3.5 Complexity Analysis

The cost of generating the index of selective k-grams is S_D (size of corpus) + 2*S_R (total size of the runs) + S_M (total size of the M_i files) + S_P (total size of the P_i files). While creating the above index, we also generate the index I_U of unselective k-grams, at the cost S_U (where U is the size of I_U). Finally, we process the M_i, P_i files to remove non-β-optimal k-grams. It is not hard to show that the cost of that step is bounded above by 8*S_M + 2*S_P.

So the total cost of the index construction algorithm so far is $S_D + 2 * S_R + 9 * S_M + 3 * S_P + S_U$. Out of these, S_D is significant (one scan of the corpus). We have found that M and U are quite small and so $9 * S_M + S_U$ is negligible. R and P can be of significant size, but are usually just a fraction of D. Thus, the total cost is usually no more than 2-3 sequential scans of the original corpus, which is supported by our experiments in Section 3.8.

3.6 Indexing D-grams

We now discuss indexing non-k-gram features of text documents. We consider in particular two such features: character distance based Minspan/Maxspan indexes and indexing transformed document. In this section, we discuss the first feature and in Section 4.6 we will explain the second one in detail.

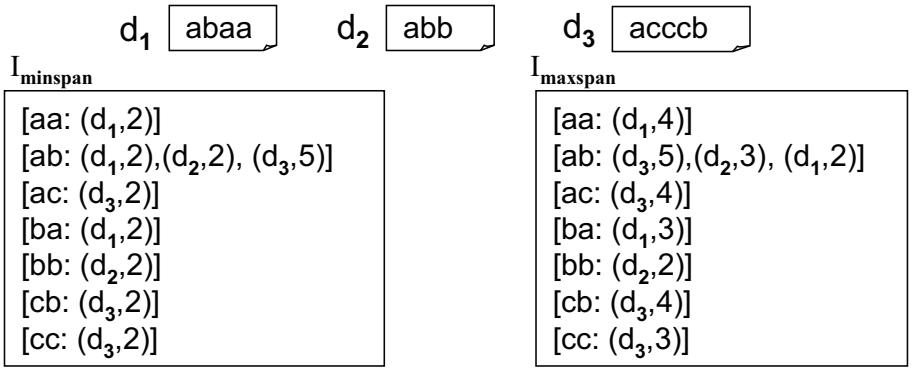


Figure 3.3: An example of $I_{minspan}$ and $I_{maxspan}$.

3.6.1 Motivating Minspan and Maxspan Indexes

A d-gram of size n is a *discontiguous* sequence of characters $c_1c_2 \cdots c_n$ in a document d. By "discontiguous" we mean some characters may appear between any two characters c_i and c_{i+1} of the sequence. Consider for example documents $d_1 - d_3$ in Figure 3.3 (we have modified $d_1 - d_3$ slightly, for ease of exposition). In d_1, ab is a d-gram of size 2, and aba is a d-gram of size 3. In d_3, ab is a d-gram of size 2 (with string ccc appearing between a and b).

As a first step, in this paper we consider only d-grams of size 2, to keep the number of d-grams manageable. Considering d-grams of greater size is an ongoing work. For d-grams of size 2 (henceforth just *d-grams*), we define the following notions:

Definition 3.5 (Minspan and Maxspan) The minspan of a d-gram xy in a document d, denoted $minspan(\mathsf{xy}, d)$, is the size of the smallest string in d that starts with x and ends with y. The similar notion $maxspan(\mathsf{xy}, d)$ is the size of the largest string in d that starts with x and ends with y. The size of a string is the total number of characters in the string.

For example, $minspan(\mathsf{aa}, d_1) = 2$, $maxspan(\mathsf{aa}, d_1) = 4$, $minspan(\mathsf{ab}, d_3) = 5$, $maxspan(\mathsf{ab}, d_3) = 5$.

We can now define inverted indexes $I_{minspan}$ and $I_{maxspan}$ that store minspan and maxspan values of d-grams for the documents, respectively. Figure 3.3 shows these indexes for $d_1 - d_3$. Consider $I_{minspan}$ in that figure. The first index entry $[\mathsf{aa} : (d_1, 2)]$ states that d-gram aa appears only in d_1, with minspan value 2. The next entry states that d-gram ab appears in d_1, d_2, d_3, with

minspan values 2, 2, and 5, respectively. Note that here posting lists such as $(d_1, 2), (d_2, 2), (d_3, 5)$ are sorted in *increasing* value of minspan (we explain why soon below). Index $I_{maxspan}$ is defined similarly, except that its posting lists are sorted in *decreasing* value of maxspan.

We now show how these indexes can be used to locate documents for regex evaluation. Consider R = `a\w{1,2}b`. Any string s matching R starts with a, followed by 1 or 2 alphabetic characters, then ends with b. Thus, a and b will be at most 2 characters apart in s. Given this, if a document d matches R, then clearly $minspan(\texttt{ab}, d) \leq 4$.

We can exploit the above constraint to locate documents as follows. First we locate the posting list of ab in $I_{minspan}$, which is $(d_1, 2), (d_2, 2), (d_3, 5)$ in this case (see Figure 3.3). Next, we scan the list to obtain all documents with minspan value not exceeding 4 (this explains why we sort this list in increasing value of minspan). Finally we return this list, which is d_1, d_2 in this case.

We can exploit $I_{maxspan}$ in a similar fashion. Consider again R = `a\w{1,2}b`. It is not difficult to see that if document d matches R, then $maxspan(\texttt{ab}, d) \geq 3$. So we can locate the posting list of ab in $I_{maxspan}$, which is the list $(d_3, 5), (d_2, 3), (d_1, 2)$ in this case. Next, we scan the list to obtain all documents with maxspan at least 3 (hence the need to sort the list in decreasing value of maxspan). Finally we return the list of obtained documents, which is d_3, d_2 in this case.

Observe that intersecting the two returned lists produces d_2. Thus, by consulting $I_{minspan}$ and $I_{maxspan}$, we only need to evaluate R on d_2, a significant saving.

In Section 4.5 we show how to analyze a given regex R to infer constraints of the form $minspan(s, d) \leq v$ and $maxspan(s, d) \geq v$. Indexes $I_{minspan}$ and $I_{maxspan}$ can then use these constraints to locate documents. It is not difficult to show that the indexes cannot use constraints of the form $mixspan(s, d) \geq v$ and $maxspan(s, d) \leq v$ for this purpose.

3.6.2 Building Minspan and Maxspan Indexes

We now discuss how to build the above indexes. Consider building $I_{minspan}$. To do this, we process the corpus one document at a time. Consider the first document d_1 (Figure 3.3) and suppose that the alphabet has only three characters a, b, and c. Then we begin by creating a 3x3 matrix M whose entries are indexed by the characters. Our goal is to scan d_1, and by the time we finish

scanning to have filled in M's entries such that $M[\mathsf{a}][\mathsf{b}]$ for example stores $minspan(\mathsf{ab}, d_1)$ (or -1 if ab does not appear as a d-gram in d).

To do this, the key idea is as follows. Suppose right now we are scanning a character a at position 3 in d_1. And suppose that the last time we saw a character b was at position 2 in d_1. Then these two characters form a d-gram ba with "span" 3-2+1=2. Clearly, we should update $M[b][a]$, which stores the best $minspan(\mathsf{ba}, d_1)$ found so far. If this best $minspan(\mathsf{ba}, d_1)$ is smaller than 2 (but not -1), then we leave it alone. Otherwise, we set $M[b][a]$ to be 2.

To implement the above idea, while scanning d_1 we keep an array A whose entries are indexed by the characters. An entry such as $A[\mathsf{b}]$ stores the last position at which we saw b (not including the position of the character currently under scan). Recall that d_1 = abaa. So if we currently scan the second a (counting from the start), then $A[\mathsf{a}] = 1, A[\mathsf{b}] = 2, A[\mathsf{c}] = -1$. We then use the current character (under scan) and array A to update M, as discussed earlier.

Thus, after one scan of d_1, we have filled out matrix M. Next, we use M to update the entries of an inverted index I on d-grams. Whenever I is full, we sort it based on d-grams, then flush it to disk into a run, then resume processing subsequent documents and rebuilding I. After processing all documents in D, we have obtained a set of runs on disk. We then merge the runs to build $I_{minspan}$, in a way similar to building the k-gram inverted index in Section 3.1.

We build $I_{maxspan}$ in a similar fashion. The key idea here is that while scanning a document, say d_1, we keep two arrays F and L. $F[\mathsf{a}]$ for example stores the first position that we see a in d_1, and $L[\mathsf{b}]$ stores the last position that we see b in d_1. Clearly, we then have $maxspan(\mathsf{ab}, d_1) = L[\mathsf{b}] - F[\mathsf{a}]$ (or -1 if this value is negative, meaning d-gram ab does not appear in d_1).

Finally, we note that building $I_{minspan}$ and $I_{maxspan}$ can be easily folded into the step of scanning the corpus of the k-gram algorithm in Section 3.1. The complete algorithm can be found in Figure. 3.4.

3.7 Indexing Transformed Documents

Most current solutions index only the *original* documents. In contrast, Trex also indexes *transformed versions* of the documents. Specifically, given a document d, we remove all lower-case

Input: a corpus D, an alphabet A
Output: the d-grams of length 2 with their posting lists

1. Initialize an empty inverted index I of d-grams

2. For each document d in D

   2.1 Initialize an integer matrix Min of size $|A| \times |A|$: all entries are assigned with maximum values

   2.1 Initialize an integer matrix Max of size $|A| \times |A|$: all entries are assigned with maximum values

   2.2 Initialize an integer array R of size $|A|$: all entries are assigned with min values

   2.3 Initialize an integer array F of size $|A|$: all entries are assigned with max values

   2.4 For the ith character c of d (i from 0 to $|d|$)

      2.4.1 for each char j

      $Min[j][c] = i - R[j]$ if $R[j]$ is not min and $(i - R[j]) < M[j][c]$

      2.4.2 $R[c] = i$

      2.4.3 $F[c] = i$ if $F[c] > i$

   2.5 For the pair of chars i and j: $Max[i][j] = R[j] - F[i]$

   2.6 Append the two matrix Min and Max to the in-memory indices

   2.7 When the memory is full, sort the Minspan index ascendingly and

   the Maxspan descendingly on the distance and flush to the disk

3. Merge the runs on the disk to build the final Minspan/Maxspan index

Figure 3.4: Building minspan/maxspan indexes

characters, thus transforming it into a document d' that retains only capital-, numeric-, and special characters. For example, if d = Bob@abc.xyz.com, then d' = B@... Let D' be the transformed version of the original corpus D. We then build a k-gram inverted index I_{trans} for D', in the same way of building the k-gram inverted index I_{kgram} for D. Building I_{trans} can be done at the same time of building k-gram indexes for the original documents. That is, whenever we read in a document d and process it to build the k-gram indexes, we also transform d into d' and build I_{trans}. Figure 3.5 gives the details of the transformation procedure.

Index I_{trans} can handle novel cases that indexes I_{kgram} on the original corpus cannot. To see this, consider R = `@\w*`. Here, I_{kgram} does not work well because it would be used mostly to look up @ and ., two fairly common k-grams in real-world corpora. I_{trans} however could be useful because we can derive the k-gram @. from R, then look it up in I_{trans} (to find candidate documents such as Bob@abc.xyz.com).

An index over transformed documents is smaller in size than a normal k-gram index because it ignores certain sets of characters. Using the earlier example of removing all lower-case characters, our experiments on two real-world datasets show that the resulted index sizes are only about 8% of that of the normal k-gram indexes. Thus, one can define several transformation rules (e.g. ignoring all lower and upper-case word characters) and build the corresponding indexes and use them to compliment k-gram indexes. We believe that this idea of indexing transformed documents has much richer applications. For example, given a real-world corpus such as the Enron email data set, many regexes look for common entities such as emails, phone numbers, social security numbers, zip codes, etc. In such cases, we can identify a set of patterns that regexes commonly use (e.g., `\d{5}` for zip codes and `\d{3}-\d{2}-\d{4}` for social security numbers), preprocess the documents to identify strings matching these patterns and replacing the strings with special k-grams (e.g., $Z for zip codes and $S for social security numbers), then index the transformed documents. Now if we are given a regular expression such as R = `my soc sec is \d{3}-\d{2}-\d{4}`, we can analyze R to detect common patterns and replace them with corresponding special k-grams (e.g., replacing `\d{3}-\d{2}-\d{4}` with $S in this case), then proceed as in the current Trex to

Input: a corpus D, a set R of transformation rules
Output: all "transformed" k-grams with length $\leq n$ and their posting lists

1. Initialize an empty inverted index I of grams

2. For each document d in D

   2.1 Initialize an empty string d'

   2.2 For each character C of d

      2.2.1 If C is a lower-case word char, ignore C

      2.2.2 Otherwise append C to d'

3. Construct the k-gram index on d' using the same algorithm as in Figure 3.2

Figure 3.5: Building index over transformed documents

locate documents on which to evaluate R. By "collapsing" potentially complex common regex patterns into much smaller special k-grams, we enable our indexes (considered so far) to index more complex patterns (than they can on the original documents). Since parsing the special k-grams is presumably much faster than the original complex common regex patterns, we can also speed up regex evaluation. We are currently examining this idea in an ongoing work.

3.8 Empirical Evaluation

We now empirically evaluate the index building performance of Trex. We used two real-world data sets: Enron and DBLife. Enron contains 251,748 emails (i.e., documents) of the former Enron Corp., at the average size of 4K / email, for a total size of 986M. DBLife contains 10,702 HTML pages (i.e., documents) crawled from database research related websites, at the average size of 18K / page, for a total size of 180M. We will use Web to denote the DBLife dataset in the rest of this section because the source of the data is from the Web.

These two data sets come from the two real-world applications that we are maintaining: Avatar, a semantic search engine at IBM and DBLife, a Web data analysis application at University of Wisconsin Database group. From Avatar we collected a snapshot of 106 regexes that users have posed over the Enron data set (to extract directions, addresses, emails, phones, etc.), and from

DBLife we collected 29 regexes that were posed by two application modules in a single-day period. Our experiments were conducted with these regexes over the above data sets, running Java on a standard 3GHz PC.

The work [22] used Free to index up to 10-grams (the Free 10 version). We observed that using Free to index up to only 5-grams (the Free 5 version) cuts index construction time in half and yet reduces regex evaluation time only slightly. Hence, we used Trex to index only up to 5-grams, and compare Trex with both Free 10 and Free 5. We set α to 0.2 and 0.23 on Enron and Web, respectively, using the method described in [22]. (Later in this section we examine the effect of varying α.)

3.8.1 Size of Indexes

We begin by examining index construction performance. Figure 3.6.a shows the size of the resulting I_{kgram} (the index of k-grams) for various methods on Enron, as we vary the size of the corpus (in the thousands of documents) on the X axis. A method such as Trex 0.05 means that we used Trex at β = 0.05 (while α is set at 0.2, as mentioned earlier).

The figure shows that at β = 0.15, 0.1, and 0.05, Trex's index size is roughly the same as Free 5's index size (we omit Free 10 from Figure 3.6.a-b, but will compare it with Free 5 shortly). As we gradually reduce β to 0, Trex's index size increases, but remains well below the corpus size (the top line of the figure). These results suggest that (a) adjusting β is an effective way to control Trex's index size, and (b) Trex's index size is competitive with that of Free 5. Figure 3.6.b shows the size of I_{kgram} for Web, with similar conclusions.

Figures 3.7.a-b show what happened when we also add the non-k-gram indexes ($I_{minspan}$, $I_{maxspan}$ and I_{trans}) and the set of unselective k-grams. The figures show that these indexes increase the total index size of Trex only moderately (e.g., by 8.3-16% for Enron at 250K documents and Web at 10K documents, respectively, for Trex 0).

3.8.2 Construction Time of Indexes

Figures 3.8 shows the k-gram index construction time for the various methods described earlier. The figures show that Free 5 took far more time than Trex methods, e.g., 28% more on Enron (at 250K) and 51% more on Web (at 10K). All Trex methods took roughly the same time, suggesting that processing the files $M_1, P_1, M_2, P_2, \ldots$ (Section 3) takes negligible time.

Figures 3.9 show what happened when we also add the time of constructing non-k-gram indexes. The times of Trex methods increase, but still stay below those of Free. Overall, the results suggest that even when building larger indexes with more features, Trex still takes less time than Free.

3.8.3 Free 5 vs. Free 10

Interestingly, the index size of Free 10 is just slightly larger than that of Free 5 (hence we do not show Free 10 on the figures). This suggests that most selective k-grams where $5 < k \leq 10$ do not make it into the index (because their substrings are already in the index). At the same time, we observed that building Free 10's indexes take twice as much time compared to Free 5, and reduces regex evaluation time by only 0.9-2% (see below). These results suggest that Free 5 is highly competitive with Free 10, at a smaller index construction cost.

3.8.4 Sensitivity and Scalability Analysis

Now we consider the effect of varying major parameters. We have considered the effect of varying β earlier. We now consider varying α. Figure 3.10 and Figure 3.11 show that as we vary α from 0.1 to 0.5, index size (for Trex 0 on the full data sets) increases, but gracefully. The next two figures show that index construction time increases too, but by relatively little. The results thus suggest that Trex's index construction method is robust to changes in α.

Finally, Figure 3.12 shows the index size and construction time as we scale the Web data set to 4.4G (we did not run this experiment for Enron as we do not have this much data on Enron). The figure shows that size and time scale linearly with the data set size.

3.9 Summary

The chapter presents in details what Trex indexes, how Trex builds its indexes and finally evaluates the indexing performance of our proposed methods. The indexer of Trex has the following major advantages compared with the state-of-the-art k-gram indexer of Free:

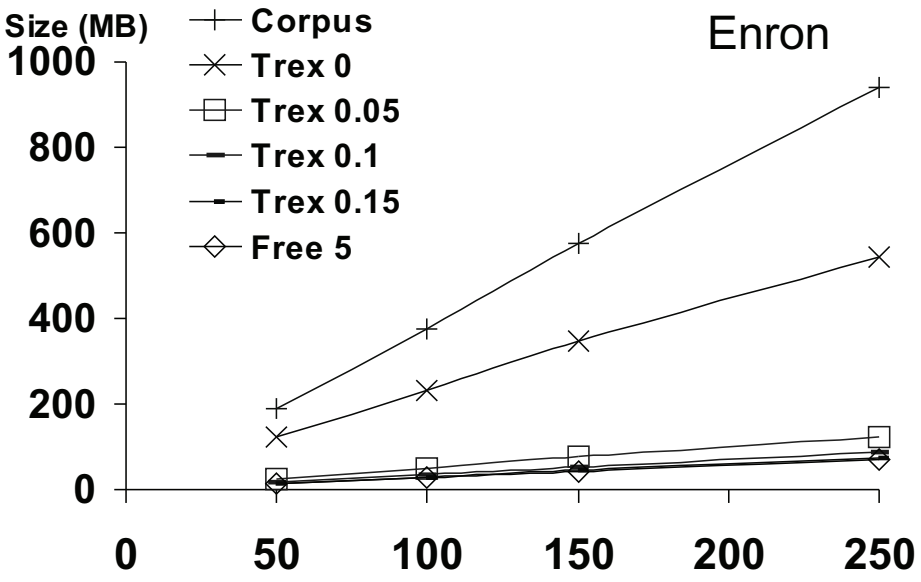
1. Trex exploits the notions of α-selectivity and β-optimality to adjust the index size based on available disk space: selecting an α lets us control the number of selective k-grams, then selecting a β between 0 and α lets us further control how many of these selective k-grams should be in the index.

2. By adding unselective k-grams to the index, Trex is able to detect k-grams that do not occur the corpus. Trex can then skip regex evaluation if it finds the regex contains such non-existent k-grams. In contrast, Free will have to evaluate all documents.

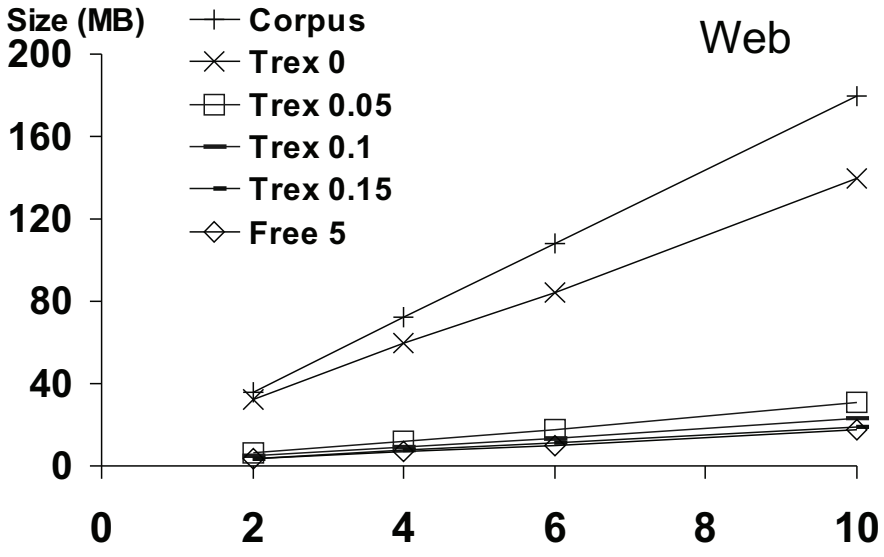
3. Trex indexes non-k-gram features such as minspan/maxspan of d-grams and transformed documents.

4. During index construction, Trex scans the corpus only once. It then utilizes a compact summary structure built during the corpus scan to construct the full index. Compared with the approach of Free which scans the corpus multiple times, Trex demonstrates much faster index construction time.

5. Unlike Free, the indexing performance of Trex will not degrade when the size of k-gram lexicon is greater than the available memory and thus causes throttling. As a result, the indexing algorithm of Trex is more scalable for huge text corpora found in many of today's application environments.

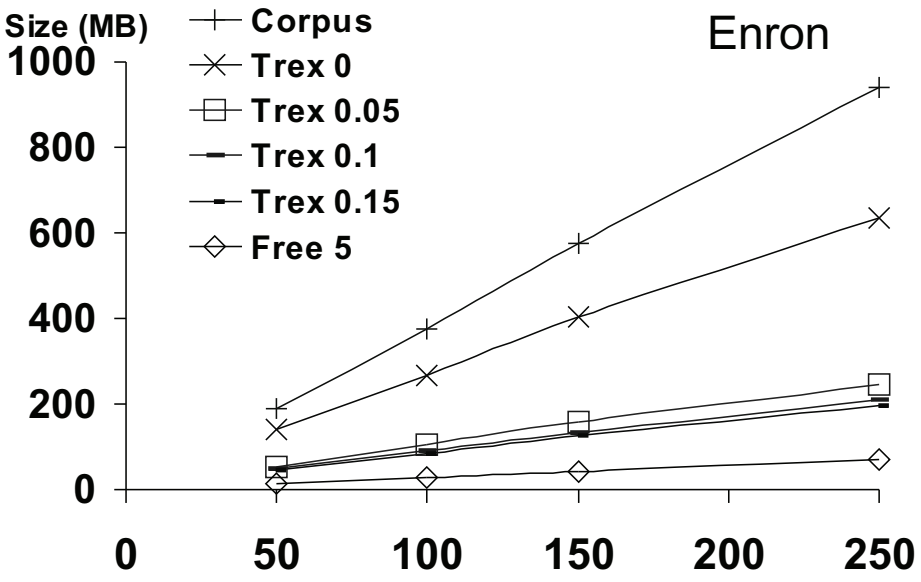


(a) Size of k-gram index

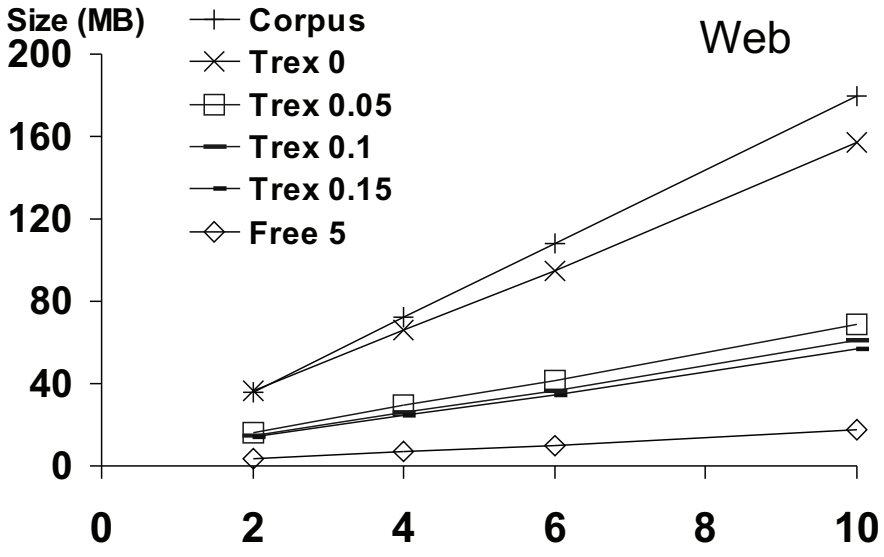


(b) Size of k-gram index

Figure 3.6: Comparison of k-gram index sizes on Enron and DBLife(Web) datasets

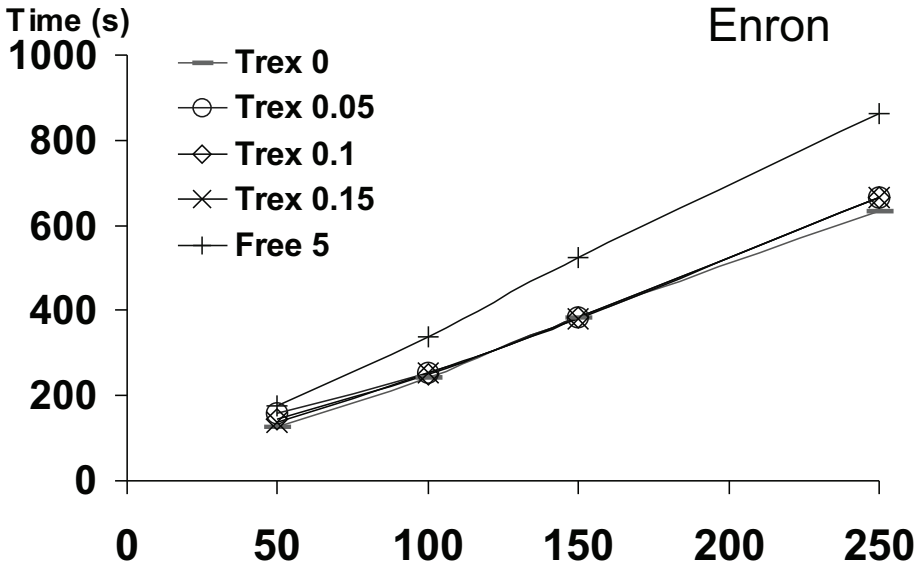


(a) Size of complete index

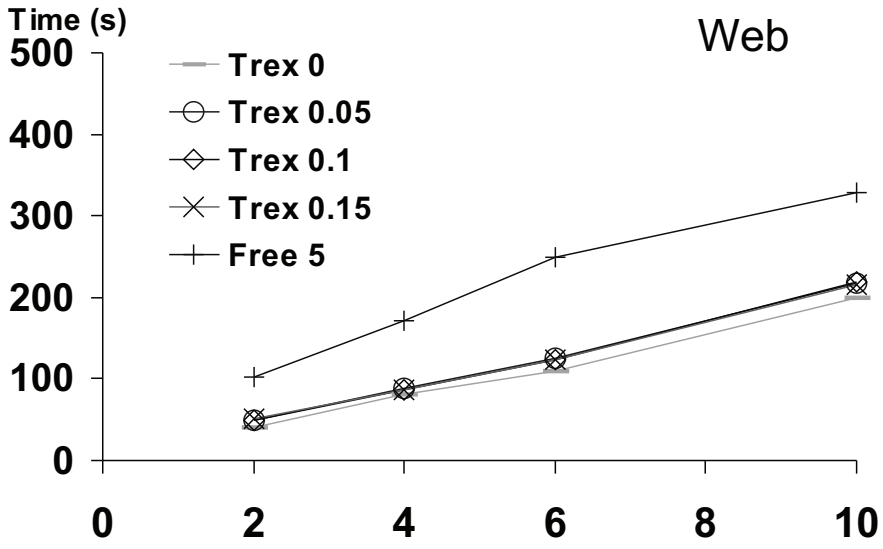


(b) Size of complete index

Figure 3.7: Comparison of complete index sizes on Enron and DBLife(Web) datasets

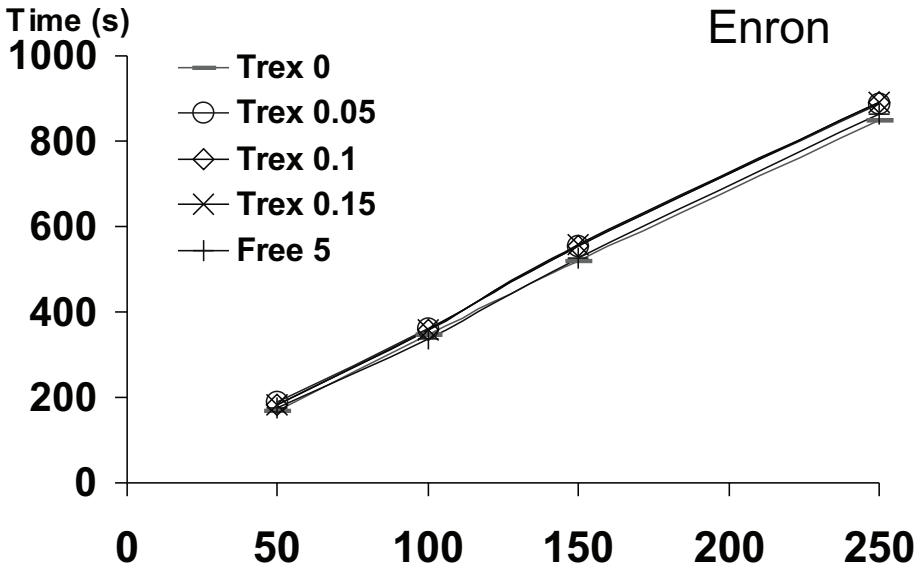


(a) Build time of k-gram index

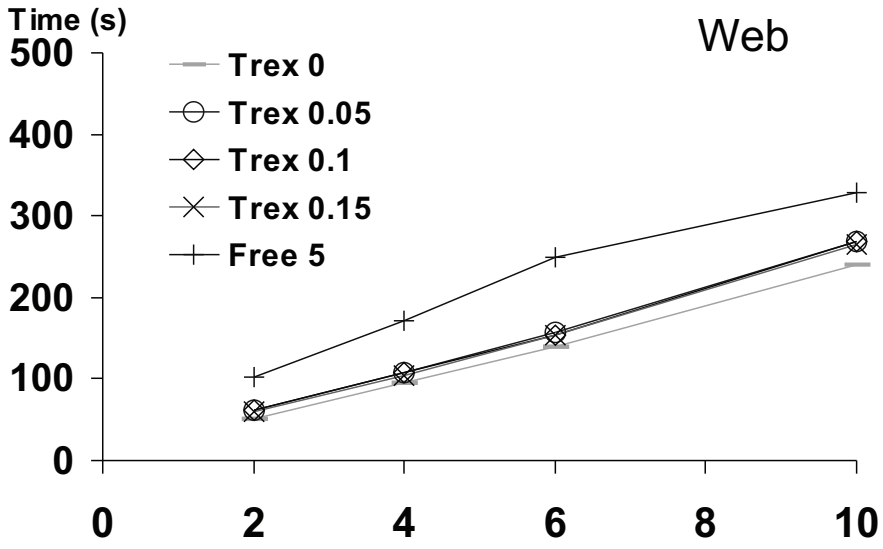


(b) Build time of k-gram index

Figure 3.8: Comparison of k-gram index construction times on Enron and DBLife(Web) datasets



(a) Build time of complete index



(b) Build time of complete index

Figure 3.9: Comparison of complete index construction times on Enron and DBLife(Web) datasets

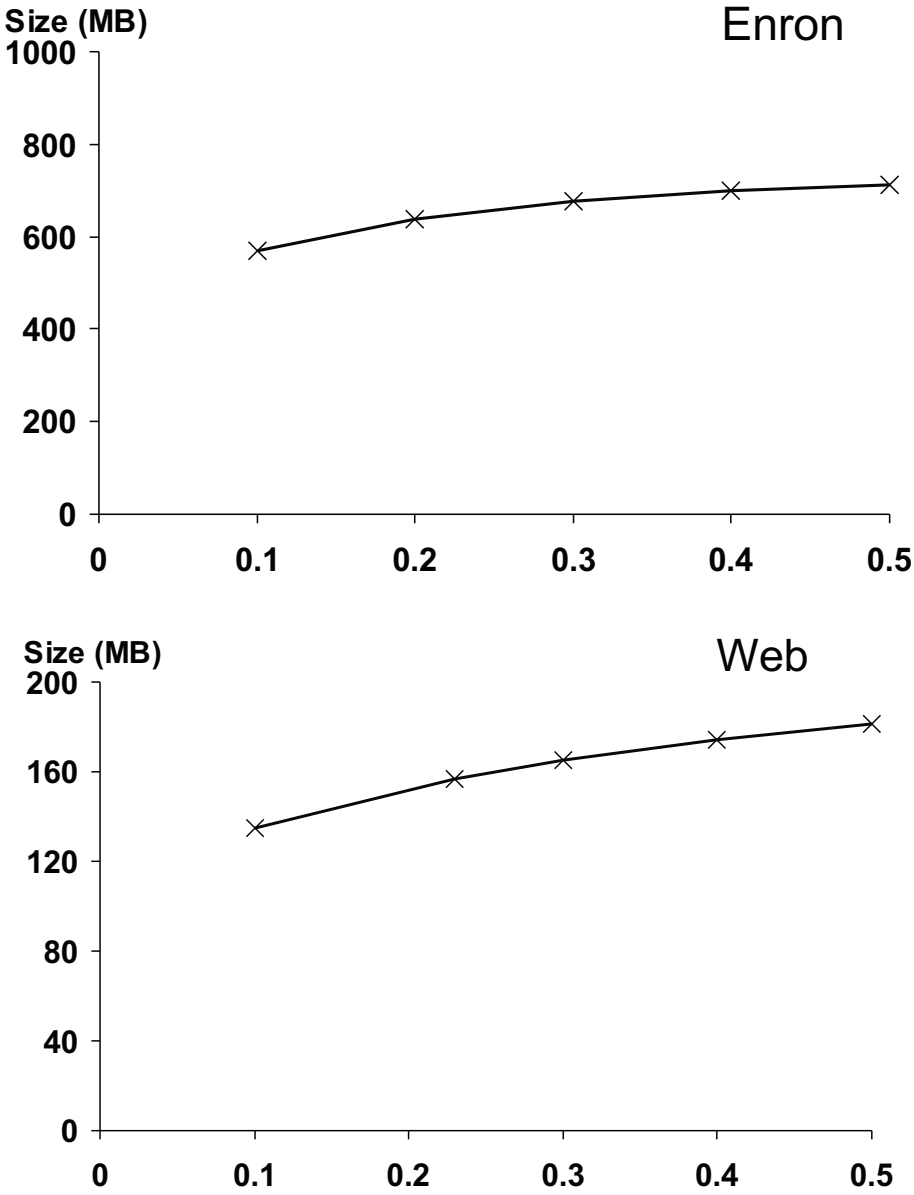


Figure 3.10: Effects of varying α on index sizes

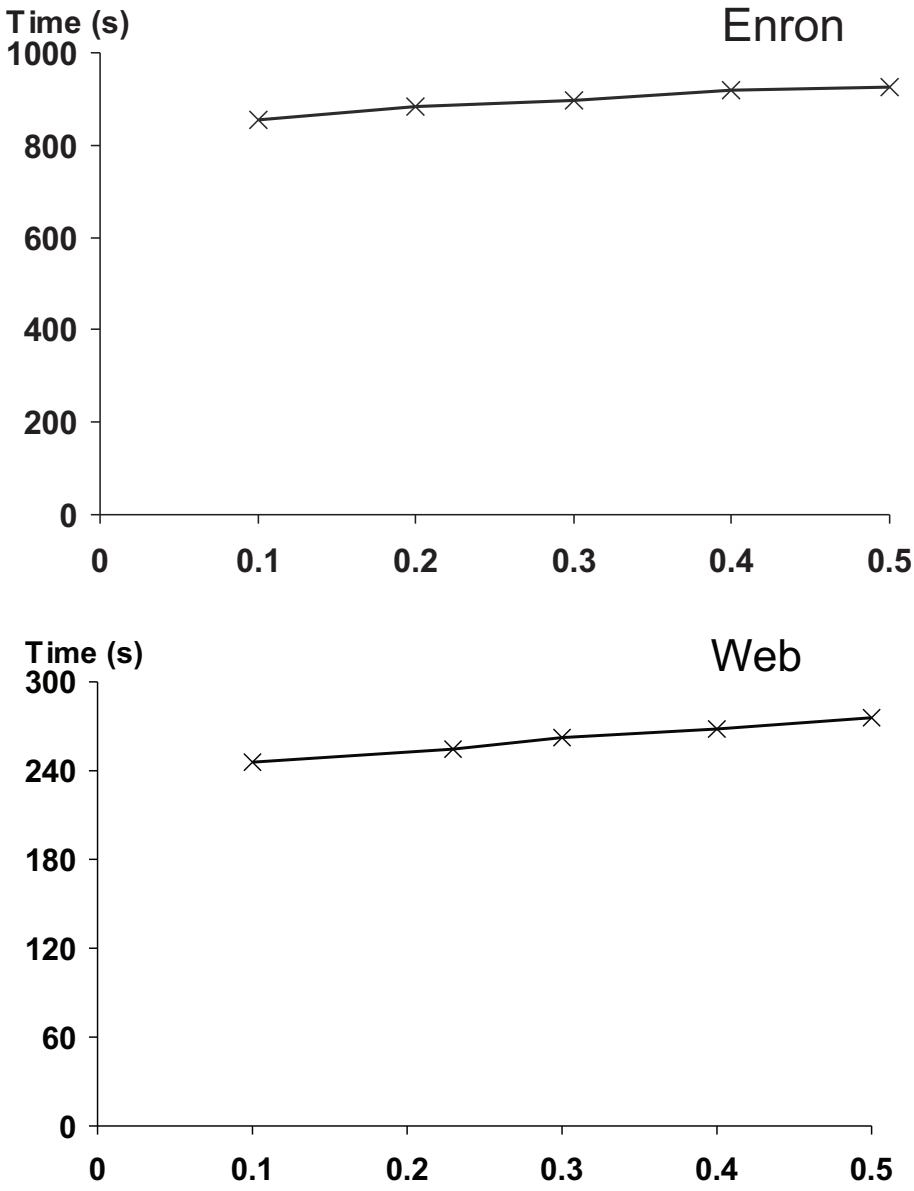
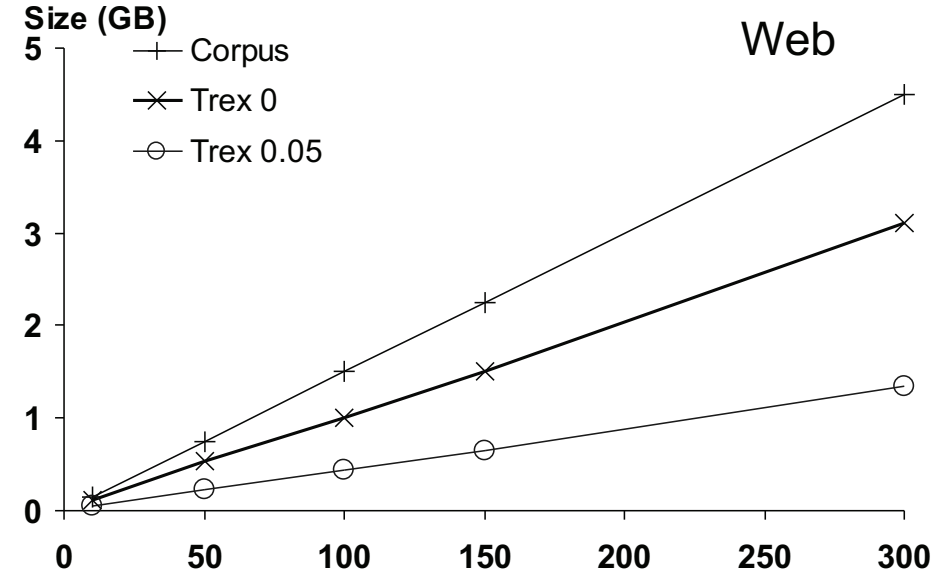
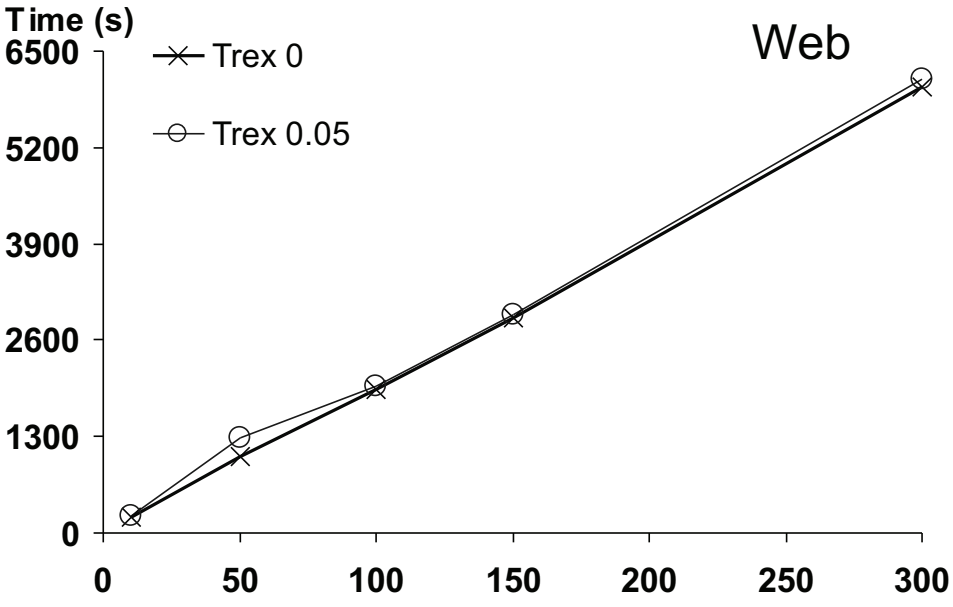


Figure 3.11: Effects of varying α on index construction time



(a) size vs no. docs (x1000)



(b) time vs no. docs (x1000)

Figure 3.12: Scalability analysis for the DBLife(Web) data set

Chapter 4

Using the Indexes

In the previous chapter, we have discussed how to create various indexes I_{kgram}, $I_{minspan}$, $I_{maxspan}$ and I_{trans}. Given a regex R, we now discuss how to use these indexes to locate documents on which R is to be evaluated.

To the best of our knowledge, there is no prior work on how to effectively use inverted indexes for regular expression evaluation. In this chapter, we first formally define the notion of "minimal superset query" over k-gram indexes. Next we prove that deriving such "minimal superset query" requires superpolynomial time and thus is computationally expensive. After that we review the state-of-the-art greedy query generation algorithm by Free on k-gram indexes. We then present our query generation method in detail and demonstrate its advantages over that of Free. After that we discuss how to generate queries over $I_{minspan}$, $I_{maxspan}$ and I_{trans} indexes respectively. Finally in this chapter, we present empirically results of our index querying methods.

4.1 Using I_{kgram} Indexes

For now, we restrict the discussion to using the I_{kgram} index. Given a k-gram index I over a corpus D, and a regex R, we want to pose a query Q over I to obtain a set of documents. We then evaluate R over the set of documents.

First, let us define the notion of index queries. We say that an atomic query $L(s)$ looks up k-gram s in the index and returns the posting list of s (or zero documents if s does not appear in the corpus). A composite query then unions and intersects the results of atomic or other composite queries, such as $Q = [L(s) \cup L(t)] \cap L(u)$. Executing Q means looking up s, t, u in the index, then

unioning and intersecting their posting lists appropriately. Henceforth we refer to both atomic and composite queries as simply queries.

Next, let us restrict the set of queries that we consider. Given a regular expression R, let $S(I, R)$ be the set of documents in I that contain substrings matching R. Let $exec(Q)$ be the result of executing a query Q. We say a query Q is a superset query if and only if $S(I, R) \subseteq exec(Q)$. Furthermore, a query Q is said to be a valid query if and only if Q is a superset query regardless of how the index I changes. Because we can not access the index I during the query formulation, we consider only valid superset queries. For example, given the regex $abcd$ and a k-gram index of length 2 k-grams over the alphabet $\{a, b, c, d\}$, $L(ab)$ is a valid query. On the other hand, $L(ac)$ is not a valid query because it will not be a superset query if there is a document in D whose text is simply the string $abcd$.

Ideally we want a minimal superset query that returns as few documents as possible. We say a query $Q*$ is a minimal superset query if $Q*$ is a valid superset query and for any other valid superset query Q, $exec(Q*) \subseteq exec(Q)$.

The following theorem states that the computational cost of deriving the minimal superset query can be prohibitive.

Theorem 4.1 Let the size n of a regex R be the number of characters in the string representing R. Then the worst-case size of the minimal superset query $Q*$ is $e^{\Omega[n/logn]}$, where the size of $Q*$ is the number of union and intersection operators in $Q*$.

The proof relates a sequence of posting list intersection and union operations to a monotone boolean circuit with only AND and OR gates. By doing so, we can leverage the well known superpolynomial lower bound result [79] on monotone circuit complexity. We demonstrate that common regexes (e.g. `@\w*.` where `\w` is the set of lower- and upper-case characters) exhibit the lower bound.

The quantity $e^{\Omega[n/logn]}$ is superpolynomial in n. This suggests that finding and evaluating $Q*$ can be expensive. The next section gives the complete proof.

4.2 Proof of Theorem 4.1

In this section, we present the proof of Theorem 1. The proof explicitly constructs (in Section 4.2.3) a regular expression that exhibits the superpolynomial lower bound. Techniques (e.g. DNF minimization in Section 4.2.2) in the proof can also be used in practical k-gram index query minimization.

4.2.1 Preliminaries and Proof Outline

Let D be the set of all document IDs. A k-gram index I for D consists of all k-grams of length up to K together with their *posting lists*. Depending on index design, indexes may select different sets of k-grams. However as we will see shortly these choices do not affect the complexity results.

Given a regular expression R, let S^* be the set of all doc IDs whose corresponding documents contain a substring matching R. Our task is to issue a sequence of queries to I so that the final query result S satisfies $S^* \subseteq S$. It is clear that the closer S is to S^*, we have more saving in regex matching. Our index can only answer queries of the following form: given a k-gram s of length less or equal to K, returns the posting list of s if s is indexed. An important assumption is that we can not access I and read a string's posting list contents when constructing an query sequence due to high access cost. Formally, we define

Definition 4.2 (Query Sequence) A query sequence is a finite sequence of atomic queries to I and union/intersection of earlier query results. $Q(\phi)$ denotes the set of doc ids which is the result of the last query in ϕ.

In the remaining part of this proof we will simply use *query* instead of *query sequence*. Because we can not access posting list contents of I when constructing an query sequence, this means that we should construct a query sequence correctly for all possible indexes whose max k-gram length is K. Formally,

Definition 4.3 (Valid Query) We call a query *valid* if it always return a superset of $S*$ for R on all possible indexes which has max k-gram length K.

Two different queries are equivalent if the results of both queries are equal on all indexes with the same K. The size of a query is its number of union/intersection operators.

Definition 4.4 (Minimal Superset Query) A valid query ϕ is a *minimal superset query* of R and K if $Q(\phi)$ is a subset of $Q(\phi')$ of any other $valid$ query ϕ' on any index (with max k-gram length of K).

Ideally we want to obtain a query which prunes the maximum number of documents. Formally,we define our problem as:

Definition 4.5 (Minimal superset query derivation) Given a regular expression R, an integer K which is the max length of k-grams indexed by I and an alphabet σ, derive the minimal superset query $\phi*$ among all Union/Intersection only queries.

The size of a regular expression is the number of characters used in the string representing the regular expression. Our proof proceeds in two steps. First, given any regex R, we show how to construct its minimal superset query $\phi*$ in a special form called Disjunctive Normal Form (DNF). Next we construct a regular expression R and prove that there is *NO* polynomial size valid query which is equivalent to its DNF form minimal superset query.

4.2.2 Minimal Superset Query in DNF Form

We call a query in *Disjunctive Normal Form (or DNF)* if it is a union of intersections of atomic queries. We call a query in *Conjunctive Normal Form (or CNF)* if it is an intersection of unions of atomic queries. For example, $a \cup (b \cap c)$ is in DNF whereas $a \cap (b \cup c)$ is in CNF. Both DNF and CNF queries have at most two levels. In boolean logic, a clause (e.g. $b \cap c$ in the earlier example) in DNF is called a $term$. Similarly a clause in CNF is also called a $term$. A term whose set of atomic queries are not a proper superset of any other term is called in minterm (in DNF) or maxterm (in CNF). It is clear that all non-min(max) terms can be safely eliminated from a query. For example, $a \cup (a \cap b) = a$ so the term $(a \cap b)$ is redundant.

Lemma 4.6 Given a string s with $|s| \geq K$, the query $\phi^* : s_1 \cap s_2 \cap \ldots s_{|s|-K+1}$ is the minimal superset query for s and K where s_i for $i = 1$ to $|s| - K + 1$ is the length K substring of s starting at the ith character of s.

We prove the claim by contradiction. Suppose the claim is false. Then by definition there is another valid query ϕ' whose resulting superset is not a proper superset of that of ϕ^* on some index. Because both ϕ' and ϕ^* are valid queries, $\phi^+ = \phi' \cap \phi^*$ is also a valid query. Now we can use distribution laws to transform ϕ' in ϕ^+ into CNF form. Notice that ϕ^* is already in CNF form. After the transformation, we can simplify ϕ^+ by combining two terms s_i and $s_i \cup s' \cup \ldots$ into one term s_i and the resulting query is still equivalent to ϕ^+. The reason is that ϕ^+ takes intersection of the two terms. After the simplification, there must be a term t left in ϕ^+ which contains none of s_i for $i = 1$ to $|s| - K + 1$ because otherwise $\phi^+ = \phi^*$. However this means ϕ^+ will filter out a matching document consisting of the string s only because s does not satisfy the term t. This contradicts the fact that ϕ^+ is a valid query.

A regular expression R encodes a list of strings. Suppose $r_1, \ldots, r_n$ are the list of matching strings of R. Next we show that the union of the minimal superset queries of these strings is a minimal superset query of R.

Lemma 4.7 Given a regular expression R, we claim that the following DNF query is the minimal superset query ϕ^*: $c_1 \cup c_2 \cup \ldots \cup c_n$ where c_i is the CNF minimal superset query of the matching string r_i of R.

Although there could be infinite many members of a regular expression, the number of *distinct* terms in ϕ^* is finite. The reason is that there could be only a finite set S of strings of length K and thus the power set of 2^S is also finite.

We prove the claim by contradiction. Suppose the claim is false. Then by definition there is another valid query ϕ' whose superset is not a proper superset of that of ϕ^* on some index I. This means there exists a document d such that $d \in Q(\phi^*)$ but not in $Q(\phi')$. Suppose $d \in Q(c_i)$ for some term c_i of ϕ^*. Now consider the query $c_i' = (c_i \cap \phi')$. Suppose r_i is the string c_i corresponds

to. c_i' is a valid sequence for r_i and filters out more documents than c_i. The reason is that $d \in Q(c_i)$ but not in $Q(c_i')$. This contradicts to the fact that c_i is the minimal superset query for r_i.

We will show in the next section that the minimal superset query ϕ^* in DNF has exponential size. Interestingly the size lower bound is still exponential even we consider arbitrary union/intersection sequence (i.e. not only DNF form).

4.2.3 Superpolynomial Lower Bound on ϕ^*

In this section, we explicitly construct a regular expression R. We then demonstrate that R's minimal superset query ϕ^* does not have equivalent queries of polynomial size and thus establish the lower bound.

To start, we note that a query with only intersection and union operations corresponds to a monotone boolean circuit with only AND/OR gates by converting $union$ to OR and $intersection$ to AND and an atomic query to a boolean variable. After that we can prove the query size lower bound of ϕ^* by showing its corresponding monotone boolean circuit has exponential complexity.

Consider the regular expression $R = a(b_1|b_2|\ldots|b_n)^*c$ and $K = 2$ where $a, b_1, b_2, \ldots, b_n, c$ are distinct characters. Notice that regular expressions with structure like R are ubiquitous. Examples include `@\w*.` where `\w` is a union of all word characters.

Let $I = \{ab_i, b_ib_j, b_ic|i, j \in \{1, \ldots, n\}\}$ be a set of boolean variables. For all strings $s \in R$, let $X_s = \{x \in I|\exists y, z \in \{a, b_i, c\}^* s.t.\ s = yxz\}$. In other words, X_s is the set of indexed k-grams in the string s. Let $T = \{X_s|s \in R\}$. Even though R could have infinite number of matching strings, the set T is finite because $X_s \subset I$.

Define the function $f : I \to \{0, 1\}$

$$f(x|x \in I) = \bigvee_{X_s \in T} \bigwedge_{x \in X_s} x$$

as a boolean function on the set of variables in I. Note f corresponds to the DNF form minimal query of R.

Lemma 4.8 f has monotone complexity $e^{\Omega(n/logn)}$.

That is, any AND/OR boolean expression of f takes exponential size. Note f is a function on $n^2 + 2n$ variables but the exponential lower bound still holds. We apply a previous known lower-bound result by Jukna for monotone circuit complexity to prove the lemma. The result states

Theorem 4.9 [50][51] A monotone boolean function $f(X)$ of n variables $X = \{x_1, \ldots, x_n\}$ requires monotone circuits of size exp($\Omega(t/logt)$) if there is a set $\mathcal{F} \subseteq 2^X$ such that: (i) each set in $\mathcal{F}$ is either a minterm or maxterm of f and (ii) $D_k(\mathcal{F})/D_{k+1}(\mathcal{F}) \geq t$ for every $k = 0, 1, \ldots, t$. Here $D_k(\mathcal{F})$ is the maximum cardinality of a subset $\mathcal{H}$ of $\mathcal{F}$ with $|\cap\mathcal{H}| \geq k$.

Let $X = I$ and $\mathcal{F}$= $\{\{ab_{i_1}, b_{i_1}b_{i_2}, \ldots, b_{i_n}c\}|(i_1, i_2, \ldots, 1_n)$ is permutation of $\{1, 2, \ldots, n\}\}$. We claim $t \in \mathcal{F}$ is a minterm of f. To show this,

(1) Suppose all variables in t true, let $s = ab_{i_1}b_{i_2} \ldots b_{i_n}c \in R$, $X_s = \{ab_{i_1}, b_{i_1}b_{i_2}, \ldots, b_{i_n}c\}$. It is clear that $\bigwedge_{x \in X_s} x$ is true and thus f is true.

(2) To set a proper subset of t to true and the rest of variable $x \in I$ to be false makes f false. We note that the set of variables in I represents a direct graph on the vertices $\{a, b_1, \ldots, b_n, c\}$ with a complete graph on $\{b_1, \ldots, b_n\}$ and a source a with edges to all b_i for i from 1 to n and a sink c with edges from all b_i for i from 1 to n to c. Setting any variable in I true corresponds to the presence of a directed edge in the graph. By setting a subset of $\{ab_{i_1}, b_{i_1}b_{i_2}, \ldots, b_{i_n}c\}$ to true and the rest of I false, we have disconnect a from c in the graph. However, $\forall s \in R$, X_s corresponds to a directed path from a to c. Hence $\bigwedge_{x \in X_s} x = false$ for all $s \in R$. Thus f is false.

Next we show $D_k(\mathcal{F}) = (n - k)!$ and thus $D_k(\mathcal{F})/D_{k+1}(\mathcal{F}) = (n - k)$. To see this, let $\mathcal{H}_k = \{t|\ t \in \mathcal{F}$ and $b_i = i$ for $i \in [1, k]\}$. $\mathcal{H}_k = (n - k)!$ which is the number of permutations of the rest of b_i for $i \in [k + 1, n]$. $|\cap\mathcal{H}_k| = k$ and it is easy to verify $\mathcal{H}_k$ is the maximum cardinality set for k. Let $t = \lfloor n/2 \rfloor$, we get $D_k(\mathcal{F})/D_{k+1}(\mathcal{F}) \geq t$ for every $k = 0, 1, \ldots, t$. Therefore the lower bound of f is $exp(\Omega(n/logn))$ which is superpolynomial. (END)

4.3 The Greedy Solution of Free

The previous section shows that it is computationally expensive to compute the minimal queries. Consequently, for now we consider greedy solutions to efficiently find a query Q that subsumes $S*$ and is as small as possible.

We first describe Free's solution, then extend it to develop our solution. Given a regex R, such as the one in Figure 4.1.a, Free builds a parse tree T, as shown in Figure 4.1.b.

Next, for each leaf node of T, which is a string from the alphabet, Free generates all substrings then keep only those that appear in I_{kgram}. Consider acd, a leaf node of T. Suppose that ac is in the index, then we replace this leaf node with node $L(\mathsf{ac})$ (see Figure 4.1.c). (If both ac and cd are in the index, then we would replace the node acd of T with the set $\{L(\mathsf{ac}), L(\mathsf{cd})\}$.) Similarly, suppose e is in the index, then we replace node e with $L(\mathsf{e})$ (see Figure 4.1.c). If a node is such that no substring appears in the index, then we replace it with NULL. For example, suppose b is not in the index, then we replace it with NULL as shown in Figure 4.1.c.

While executing the above step, we do not consider the leaf nodes of any subtree U whose root node is *. Since U may not even appear in a string that matches R, we cannot really "count" on U, and so will replace U with a single node NULL. Continuing with the above example, we replace the lone * subtree of T in Figure 4.1.b with NULL. The final resulting tree is shown in Figure 4.1.c.

Free then analyzes this tree to create the composite query Q (that it will execute). Specifically, starting with the leaf nodes, Free leaves each non-NULL singleton leaf node alone, converts each NULL leaf node into the set of all documents (IDs) in the corpus, and convert each leaf node that is a set into a query that unions the atomic queries in the set (e.g., converting $\{L(\mathsf{ac}), L(\mathsf{cd})\}$ into $(L(\mathsf{ac}) \cup L(\mathsf{cd}))$). Next, Free converts OR into $\cup$ and AND into $\cap$. This yields a composite query that Free simplifies further. The final query Q in our example is shown in Figure 4.1.d. Free will execute Q to obtain a set of document IDs, then retrieves the documents and evaluates R over them.

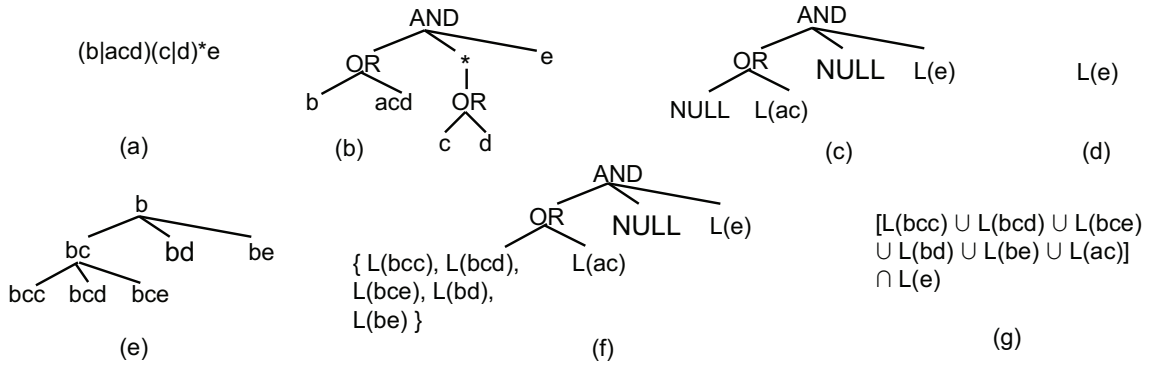


Figure 4.1: Examples of generating index queries for Free (a-d) and Trex (e-g).

4.4 The Trex Solution

The Free solution is limited in two ways. First, it considers only the *substrings* of the leaves as potential k-grams. And second, it does not even consider all leaves (e.g., those of * subtrees are discarded). We address both limitations.

Our basic idea is to do a deeper analysis for each leaf node for which Free fails to find any "good" k-gram. To illustrate, consider b, the leftmost leaf of T. We now use tree T to enumerate all k-grams that (1) start with b, and (2) appear (as a substring) in a string that matches the regex R. Figure 4.1.e shows the enumeration process. Clearly, there is only one 1-gram: b. We then extend b to generate three 2-grams: bc, bd, be, and so on (in a breadth-first fashion). We stop extending a gram s (thus terminating a path on the enumeration tree) as soon as (a) any substring of s is in I_{kgram} (e.g., node bd in Figure 4.1.e), or (b) the size of s is n (assuming that I_{kgram} indexes only up to n-grams, e.g., nodes bcc, bcd, bce in the figure; here $n = 3$), or (c) we have reached the "end" of the regex R and cannot extend s further (e.g., node be), or (d) we have generated m k-grams, where m is a pre-specified value that we use to put a time limit on the enumeration process.

Figure 4.1.e shows the enumeration tree V for b after we have stopped. There are two cases. (1) There is a leaf of V such that none of its substrings is in the index, then our deeper analysis has failed. We simply return NULL for b (exactly what Free also does). (2) Each leaf of V has one or more substrings that are in the index. It is not hard to show that there exist one "maximal" substring in the above that subsumes all other substrings.

Continuing with the tree in Figure 4.1.e, suppose case (2) holds, and suppose that the maximal strings at the leaves are bcc, bcd, bce, bd, be. Then each such string maps into an atomic query (e.g., $L(\mathsf{bcc})$), and we can replace node b of the original parse tree T (Figure 4.1.b) with the set of all such atomic queries (as shown in Figure 4.1.f).

Note that we do not perform the above deeper analysis for nodes acd and e because they already contain substrings that are in the index. Figure 4.1.f shows the final resulting tree. We can then convert this tree to a composite query, shown in Figure 4.1.g, as like Free does.

Figure 4.2 gives the details of Trex's algorithm to use k-gram indexes. Different from Free, Trex operates on both the parse tree and NFA representation of a regex to maximize the chance of finding out indexed k-grams. Note that our query does take into account * subtrees, during the enumeration process (unlike Free). Further, the process of generating and executing queries in Trex clearly can take longer than in Free. But as we show experimentally later, this overhead is more than offset by a significant reduction in regex evaluation time.

4.5 Using the $I_{minspan}$ and $I_{maxspan}$ Indexes

We now discuss using the span and document transformation indexes. Consider using $I_{minspan}$. To do this, first we consider each pair of characters x and y that appear in R, then analyze the parse tree T to see if we can derive an atomic query of the form $minspan(xy, d) \leq v$. (In short, we consider xy as a d-gram of size 2.) In general, the value v can be computed by computing the minimal distance on the parse tree, between x and y (taking into account quantification nodes). We then take all resulting atomic queries and assemble them into a composite query (that contain only minspan constraints), in a way analogous to assembling atomic k-gram queries. See Figure 4.3 for the pseudo code of this procedure.

We generate a composite query for $I_{maxspan}$ in a similar fashion. Next, we generate a composite query for I_{trans} exactly as we do for I_{kgram}, but with a new regex R', obtained from R by applying the same set of transformation operations (that we apply to the corpus). Trex executes each composite query on the corresponding index (i.e., I_{kgram}, $I_{minspan}$, $I_{maxspan}$, and I_{trans}), intersects the

resulting sets of document IDs, retrieves documents in the intersection, then evaluates R on those documents.

4.6 Using the I_{trans} Indexes

We now discuss using the document transformation indexes. We generate a composite query for I_{trans} exactly as we do for I_{kgram}, but with a new regex R', obtained from R by applying the same set of transformation operations (that we apply to the corpus). Trex executes each composite query on the corresponding index (i.e., I_{kgram} and I_{trans}), intersects the resulting sets of document IDs, retrieves documents in the intersection, then evaluates R on those documents.

4.7 Empirical Evaluation

We now compare the performance of Trex and Free in terms of regex evaluation time. Two sets of regular expressions used in two real world applications are used as our benchmarks. The list of regexes can be found in the appendix of this thesis. We first show the total regex evaluation time comparisons. Next we perform careful analysis on what components of Trex contribute to the performance gain. After that we study how the performances of Trex and Free depend on the key indexing parameters.

4.7.1 Regex Evaluation Time

The two tables in Figure 4.4 shows the regex evaluation time for Enron and DBLife(Web), respectively. Evaluation time of a regex R is the sum of the time to analyze R (analysis time), the time to probe the indexes (index probe time, or query evaluation time), and the time to match R against retrieved documents.

In the Enron table, "total time" means the total evaluation time of the 106 regexes mentioned earlier. The table shows that Free 5 is 2% slower compared to Free 10 (see column "speedup"), and that Trex 0.05 and Trex 0 significantly outperformed Free 10, by 29.2-30.3%. (We did not

	Enron		Web	
	Time (s)	**% change**	**Time (s)**	**% change**
Trex 0	5633	-	151	-
-kgram	10216	+ 81.2%	472	+ 312.6%
-zerodoc	6572	+ 16.7%	175	+ 15.9%
-span	5662	+ 0.5%	150	- 0.7%
-transform	6337	+ 12.5%	149	- 1.1%
-regex analysis	6592	+ 17.0%	163	+ 7.9%

Table 4.1: Effects of removing each Trex component on evaluation time.

experiment with Trex 0.1 and 0.15 because Trex 0.05 appears superior; it is comparable to them in index construction time, and yet contains slightly more k-grams.)

The next three columns of the figure show the breakdown of the total time. The results suggest that Trex incurs higher analysis and index probing times (70+) compared to Free (10+), but locates far fewer documents (78-79K vs. 113-115K, see column "avg. # doc. matched"). As a result, Trex spends less time matching documents (5500+ vs. 7300+). So overall Trex substantially reduces regex evaluation time, as observed earlier.

The Web table shows similar results, with Trex reducing Free 10's time by 27.6-41.1%.

Figure 4.5 provides another perspective on time comparison. Here we sorted the regexes in increasing evaluation time for Free 10 (the thick line), then added evaluation time for Trex 0 (the thin line) for comparison purposes. The figures clearly show that for many regexes, Trex 0 significantly outperformed Free 10, by as much as 94%.

4.7.2 Contributions of Trex's Individual Components

Table 4.1 shows the regex evaluation time of Trex 0 as we "turned off" each major component. The first row shows the times of the complete system. Each subsequent row, such as "-kgram", means that Trex does not index and use k-grams.

The table shows that each component makes a meaningful contribution, albeit at different amounts. As expected, k-grams make the most contribution: turning them off increases time by 81.2-312.6%. Our new (α, β) index also contributes to the overall performance gain. Without

it, the overall times degrade by 17.8-29.1% even with the help of all other features. "Zero document" is also clearly a significant problem (at 15.9-16.7%). Document transformation can make a significant contribution too (at 12.5% for Enron; it was slightly worse for Web, however, at -1.1%).

Finally, the last row shows that if Trex turns off the new regex analysis method (Section 4) and uses the old method from Free, time increases by 7.9-17%, suggesting the utility of the new regex analysis method.

4.7.3 Sensitivity and Scalability Analysis

Now we consider the effect of varying major parameters. We have considered the effect of varying β earlier. We now consider varying α. The two figures of Figure 4.6 are quite interesting. They show the total regex evaluation time for Free 10 and Trex 0, as we vary α from 0.1 to 0.5. First, Trex 0 outperformed Free 10 everywhere. Second, as α increases, Trex 0's time decreases. This is as expected, since higher α means we can store more k-grams in the index, which helps at evaluation time. But of course we pay with more index space; so a specific application will have to balance these two factors. It does appear that the α values that we selected (0.2 and 0.23 for Enron and Web, respectively, using the method in [22]) offer a reasonable compromise. They perform only somewhat worse than at $\alpha = 0.5$, and yet incur less index space and construction time.

But the most interesting result is about Free 10. As α increases, evaluation time of Free 10 decreases, then increases again. This can clearly be seen on the Web data set; on Enron the results for $\alpha > 0.5$ (not shown on the figure) show the same phenomenon. This is counter-intuitive, as increasing α ought to admit more k-grams into the index, thereby reducing evaluation time. It turned out that this is not the case with Free. Because Free keeps only prefix- and suffix-free k-grams (Section 2), increasing α often reduces the index size, because each newly admitted k-gram s may actually "prune away" many other k-grams from the index (i.e., those that contain s). This supports our argument earlier (Section 2) that Free fundamentally cannot make use of more index space, even if it wants to.

4.8 Summary

In this chapter, we have developed the theory and algorithms on how to effectively query the various indexes to locate documents on which R is to be evaluated. Detailed experimental results show the significant improvements of Trex over the best available solution currently found in literature: Free. In particular, we have made the following main contributions:

1. We formally define the notion of "minimal query" over k-gram indexes. Next we prove that deriving such "minimal query" requires superpolynomial time and thus is computationally infeasible. Our proof is based on the classic lower bound on monotone boolean circuits which have only AND or OR gates.

2. We develop a novel greedy query generation method. Compared with that of Free, our algorithm performs a much deeper analysis of regex structures like Kleene stars and repetitions. Our algorithm uses the set of indexed k-grams to dynamically generate index query. Note that Free does not use index information during its query generation and thus often misses pruning opportunities.

3. We present detailed empirical results that compare our index querying methods to that of Free over a wide range of regular expressions.

Input: a regex R
Output: the query Q of R on the k-gram index

1. Convert R into a tree P and an NFA N

2. Foreach leaf node a in P

2.1* Let $e_a :< v_1, s_a, v_2 >$ be the edge of a in N and $G = \{\}$

2.3 If s_a contains an indexed k-gram g, let $G = \{g\}$

2.4 Else start a search from the NFA state v_2

Init a queue $Q = \{< v_2, s_a >\}$, an empty hash H of visited states

While Q is not empty

If $|Q| > MAX\_STATES$, $G = \{\}$ and exit while loop

Dequeue $S :< v, s >$, the head of Q

Foreach edge $< v, s', v' >$ in N

Let s'' be $s \circ s'$**

If v' is an end state and $s \circ s'$ has no indexed k-gram,

let $G = \{\}$ and exit the while loop

If $s \circ s'$ contains an indexed k-gram g', add g' to G

If the search state $< v_1, s'' >$ is not in H, add

$< v_1, s'' >$ to Q and H

2.5 Let $Q_a = g_1 \cup g_2 \ldots \cup g_n$ for each $g_i \in G$

3. Return the final query Q of R from the leaf queries using

AND and OR operator with the method of Free

* $< v_1, s_a, v_2 >$ is a NFA edge from state v_1 to v_2 with string s_a.
** $s \circ s'$ means appending s' to s

Figure 4.2: Generating the query for I_{kgram}

Input: a regex R
Output: the minspan query Q of R

1. Parse R into tree P

2. Convert P to P' by converting the strings on leaf nodes of P
   into an AND node of characters.

3. Foreach leaf node a in P' as a non-descendant of Repeat node
   3.1 Derive tree P_a by removing all nodes in P' which
   precedes a or cannot occur together with a
   3.2 Delete from P_a all Repeat nodes with 0 min repeat time
   with its descendants
   3.3 Convert remaining Repeat nodes in P_a to AND nodes
   3.4 Foreach leafnode b of P_a, compute the minspan t of a
   and b in the original tree P'
   3.5 Replace each leafnode in P_a with its atomic minspan query
   derived in 3.4 to get Q_a of node a

4. Replace each leaf node a in P' with its query Q_a to get Q

*If b is a descendant of a Repeat node in P, convert the Repeat node to AND node during span calculation

Figure 4.3: Generating the query for $I_{minspan}$

Enron α = 0.2, 106 regexs

Method	Total time	Speedup	Analysis time	Index probing time	Matching time	Avg. time	Avg. # doc. matched
Free 10	7337	-	0.1	12.2	7325	69.2	113673
Free 5	7497	-2%	0.1	11.8	7485	70.7	115523
Trex 0.05	5677	29.2%	2.3	70.1	5604	53.6	79720
Trex 0	5633	30.3%	2.7	72.0	5558	53.1	78530

Web α = 0.23, 29 regexs

Method	Total time	Speedup	Analysis time	Index probing time	Matching time	Avg. time	Avg. # doc. matched
Free 10	213	-	0.03	0.7	212	7.3	2849
Free 5	215	-0.9%	0.04	0.6	214	8.2	2850
Trex 0.05	167	27.6%	2.1	5.4	160	5.8	2086
Trex 0	151	41.1%	2.4	5.9	142	5.2	2049

Figure 4.4: Comparison of total evaluation times (in second)

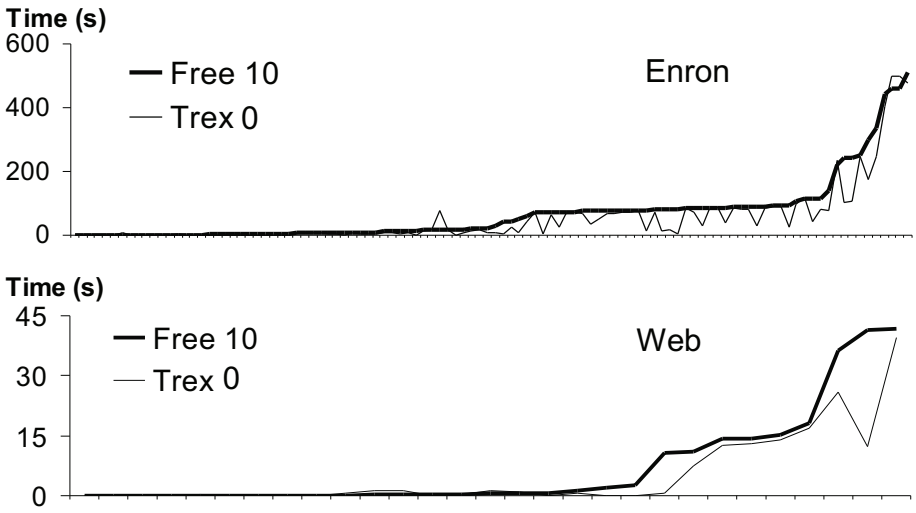


Figure 4.5: Comparison of evaluation times for individual regexes

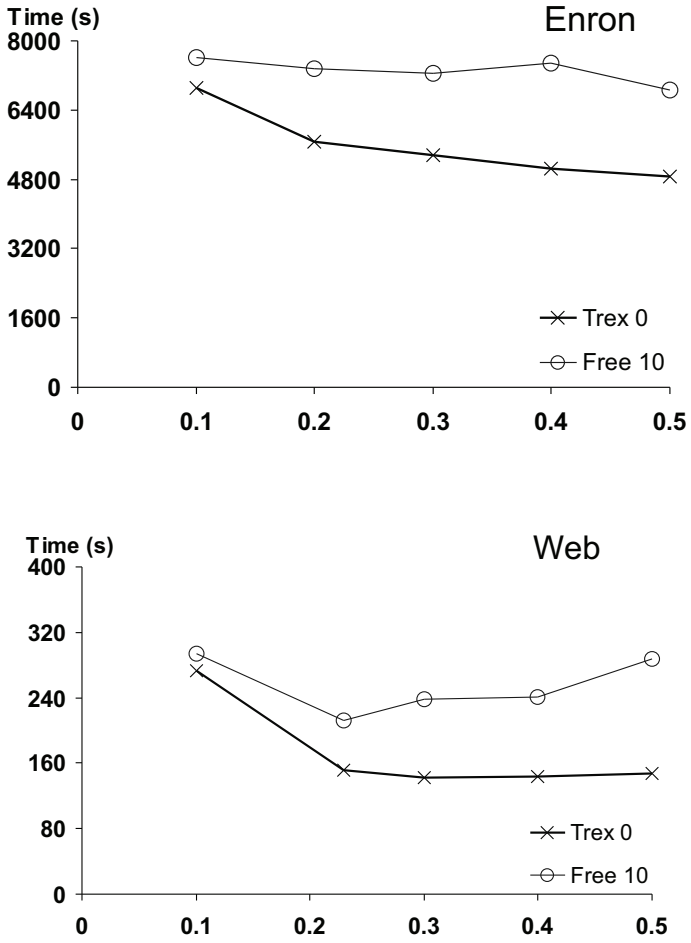


Figure 4.6: Effects of varying α in the run time query performance

Chapter 5

Maintaining the Indexes

So far we have assumed the underlying text corpus is static and thus always build the indexes from scratch. However, many real-world applications regularly add and delete documents from the corpus. Thus, we want to explore how to incrementally maintain the indexes. For now, we will consider only k-gram indexes. Incremental index maintenance poses new challenges to Trex. How can we update the index while keeping it online? How can we minimize the index update cost while still maintaining the properties (e.g., α selectivity and β optimality) of the index? To address these problems, we propose new methods tailored for k-gram indexes to make Trex capable of indexing evolving corpora efficiently.

In this chapter, we first define the k-gram index maintenance problem. Next we present our incremental index update method designed for handling the case of adding documents. Section 5.3 discusses how to handle the case of deleting documents. Finally, Section 5.4 presents the experimental results.

5.1 The K-gram Index Update Problem

To ensure that Trex stays on-line during corpus updates, we accumulate in memory a separate index I' for the recently added documents besides the existing index I on disk. At query time, we query both I and I' and combine the results. When the memory is exhausted, we flush I' to disk and merge it with I. During the merge, we can simply turn the index offline and stop serving user requests. Alternatively, we can initialize a new in-memory index I'' to continue serving user requests and accept new incoming documents. Concurrently, we merge I' with I to produce a new

on-disk index during which both I and I' can still serve user search requests. After the merge, I' is discarded and I'' takes over the role of I'. In either of the above cases, it is imperative to make the merge as efficient as possible: the first case is obvious; in the second case the faster the merge can be done, the sooner the system resources such as I/O bandwidth and CPU cycles used by the merge process can be freed up.

Formally, given a corpus C and its (α, β) optimal k-gram index I on disk and an in-memory k-gram index I' for newly added documents C', we want to construct the (α, β) optimal index of the combined corpus C and C'.

In general, k-gram indexes only keep the posting lists of a subset of all k-grams based on properties of the corpus such as α-selectivity. Updating such indexes poses significant challenges. For example, the addition of new documents may turn a previously unselective k-gram g into a selective one. However, in the current solution of Trex, the existing index does not keep the posting list of the unselective k-gram g. The naive solution requires re-scanning of the entire corpus to build g's posting list, which results in high update costs as we will show in the experiments.

To avoid costly corpus rescan, we propose to keep "extra" corpus information along with the k-gram index so that when necessary we can recover *all* information related to k-grams with lengths up to the max length. Among the many choices for the "extra" corpus information, it is natural to choose the one that is compact and allows fast processing. As we pointed out earlier 3.1, the set S of all max length k-grams with their posting lists contains all necessary information for k-grams with length up to the max length. Because the current k-gram index of Trex has already a subset S' of the max length k-grams with their posting lists, we just need to keep the posting lists of those max length k-grams not in S'.

Consider the example in Figure 5.1. Figure 5.1.b-c show the k-gram index (with $\alpha = 0.7, \beta = 0.1$) for the three documents in Figure 5.1.a. Compared to the index format defined in Section 2, here the index also includes the extra information E_2: posting lists of non-(α, β) optimal max length k-grams such as ab. When two new documents d_4 and d_5 (Figure 5.1.d) are added, an in-memory inverted index (Figure 5.1.e) is constructed which contains all k-grams with length up to 2 and their posting lists. The indexing system can still answer user queries when the new documents

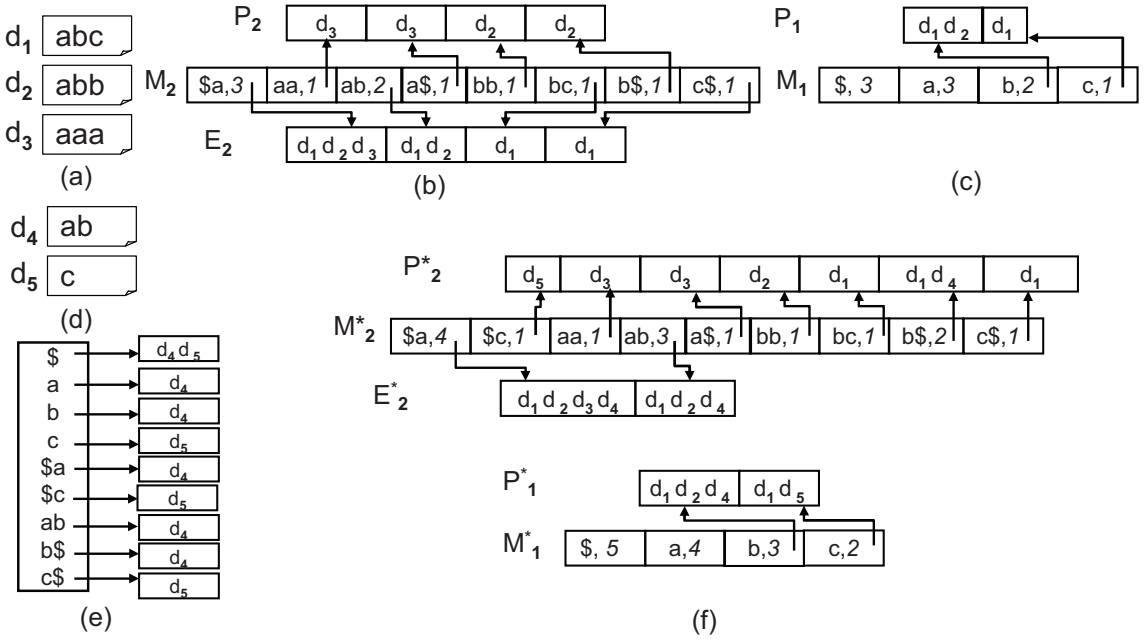


Figure 5.1: An example of incremental index update in Trex: (a) An existing corpus with 3 documents (b) the existing index I: the 2-grams with their posting lists and the extra information file E_2 (c) the existing index I: the 1-grams with their posting lists (d) the new documents (e) the in-memory inverted index I' for the new documents (f) the final updated index.

arrive. Suppose the memory is full after d_5, we merge the in-memory index with the existing one to the new index (Figure 5.1.f). The example illustrates the benefits of keeping the "extra" information. For instance, the k-gram bc becomes (α, β) optimal only after the corpus update. If bc's posting list is not stored in E_2, we have to rescan the corpus to construct its posting list in P_2^*.

Taking into account the new extra corpus information, we revise the k-gram index update problem definition as: given a corpus C, its (α, β)-optimal k-gram index I with the extra information file E on disk and the in-memory k-gram index I' for new added documents C', construct the (α, β)-optimal index I^* and the new extra information E^* of the combined corpus C and C'.

5.2 The Index Update Algorithm

Our index update algorithm first sorts the in-memory index I' by its k-grams alphabetically. Next, for each k-gram length from the max length to 1, it performs a merge of the k-gram posting lists in the existing index I , I' and the extra information file E. To explain the idea, let us consider the example in Figure 5.1. We first merge the posting lists of 2-grams in M_2 and the in-memory index (Figure 5.1.e). The posting lists of α-selective k-grams are written into P_2^*. In contrast to the original index construction algorithm, here we also write posting lists of non-α-selective k-grams (e.g., $\$a$) into the extra information file E_2^*. Then we proceed to merge the 1-grams' posting lists. In contrast to the case of 2-grams, only selective 1-grams' posting lists like b and c are output to P_1^*. Finally, similar to the original index construction algorithm, we perform the β optimality pruning to remove posting lists of non-β optimal k-grams except those belonging to the max length k-grams (e.g., ab in M_2^*).

An important optimization of our index update algorithm is that for each posting list we store its last document ID. With this information, when we append the posting list from an in memory index to the posting list in an existing index (both belong to the same k-gram), we do not need to re-encode the document of the later doc ID list.

Figure 5.2 shows the pseudo codes of our index update algorithm. Our method is incremental because it does not re-scan the original corpus to extract k-grams. Instead, it relies on the compact and structured existing index and the extra information file. Similar to the index construction algorithm, our index update algorithm performs only sequential IOs over the index files and thus is very efficient as shown in the empirical results in the next section.

5.3 Handling Deleting Documents

So far we have focused on handling the case of adding documents to the existing corpus. For the case of deleting documents already in the corpus, the existing solutions such as Lucene [67] accumulate in memory a list of deleted document ids without actually deleting the ids from the index. At query time, the above list functions as a filter to remove documents from the returned

Input: an on-disk index I and extra info file E, an in-memory index I'
Output: a new on-disk (α, β) optimal index I^* and extra info file E^*

1. Sort the in-memory index I' by its k-grams alphabetically

2. For each k-gram length L from the max length to 1

   2.1 Merge the k-gram posting lists of k-gram g in the existing index I, I' and E

      2.1.1 If g is α-optimal, output the merged list to the file P_L^*

      2.1.2 Otherwise if L is the max length, output the merged list to the file E_L^*

3. Perform the β optimality pruning to remove posting lists of non-β optimal k-grams

except those belonging to the max length k-grams

Figure 5.2: Merging an in-memory index and an on-disk k-gram index

results. It is only during the index merge phase that the list of deleted document ids will finally be removed from their posting lists.

We notice that the above idea can be incorporated into our index merge algorithm as it is straightforward to remove those deleted ids during the scan of the indexes. Thus, our experimental results focus only on the case of adding new documents.

5.4 Empirical Evaluation

In this section we present experimental results on the performance of our incremental index maintenance algorithm on both the Enron and Web datasets.

To simulate a real index update scenario, we partition the Enron dataset into two parts: the first partition has 200,000 documents and the second has the rest of around 50,000 documents. To start, we build a k-gram index for the first partition. Next we update the existing index by adding to the corpus documents from the second partition, 1000 at a time for 14 times in total. The setting closely resembles an Email search application which keeps undisrupted service (by building an in-memory index for new incoming documents) and performs batch updates during off-peak hours.

As for the Web dataset, the corpus has already been organized by the dates that the documents are crawled from the internet. Thus we start with a k-gram index of the documents crawled in the first day. After that we keep adding to the existing corpus the documents crawled in the following day and update the index. We measure the performance of our incremental update algorithm over a period of 14 days.

Our incremental index update algorithm set aside 1.5G of main memory to hold the in-memory inverted index of the new incoming documents. In our experiments, the main memory size is sufficient to hold the index for the batches of documents added.

5.4.1 Index Update Times

We compare the index update time of merging the in-memory index with the existing on-disk index on two methods: our incremental index update algorithm and the naive method which rebuilds the index from scratch using our original index construction algorithm. As Figure 5.3.a-b shows, our method outperforms the rebuild method by a factor of 8 on the Enron dataset and around 4.5 on the Web datasets. The performance gaps of two methods increase over time during the period of 14 days. Interestingly, the index update times may drop in some cases compared with the previous days. The reason is that our update algorithm will prune the posting lists of non-α selective k-grams and the set of such k-grams vary (instead of increasing or decreasing monotonously) based on document statistics.

The main reason why our method has such an advantage is that our incremental update algorithm saves the costly rescan of the text corpus. It sequentially reads the existing index and extra information files. These files are smaller in size compared with the text corpus. In addition, they already contain structured information such as k-grams with their posting lists and thus save the computational cost of re-extracting the k-grams from the documents. Finally, it does not re-encode the posting lists of the original index.

5.4.2 Extra Information File Sizes Used for Index Update

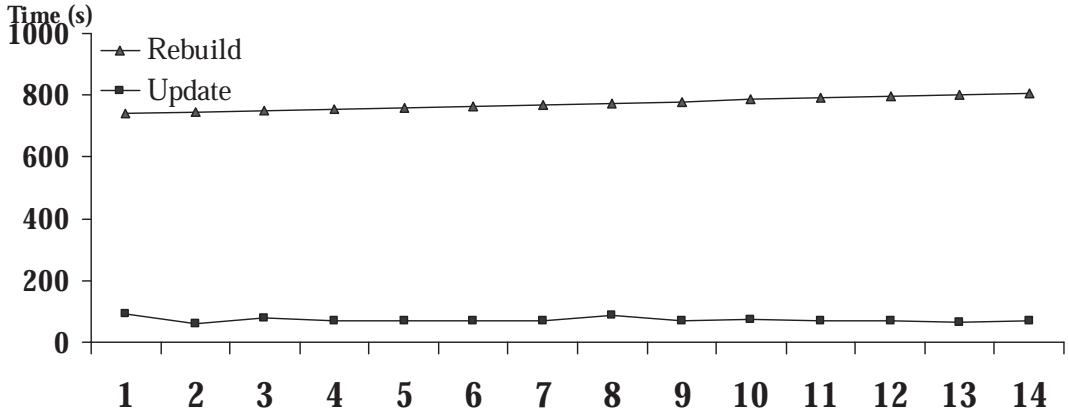
In terms of the size of the extra information file used by our incremental update algorithm, Figure 5.4.c-d shows that when the β values are close to 0 (which are usually the preferred values to use), the extra file sizes are relatively small compared with the corpus size (almost 0 when β is 0). The reason is that the existing index has already contained the posting lists of max length k-grams when β is close to 0. On the other hand, even in the case β is the same as α, the extra file sizes are still much smaller than the corpus size due to the index compression techniques we use.

In summary, by utilizing compact extra information files, our index update algorithm reuses the computation during the construction of the original index and thus achieves great saving during the index maintenance. We note that the index update times still grow with the corpus size: this is a consequence of keeping the strict (α, β) optimality guarantee.

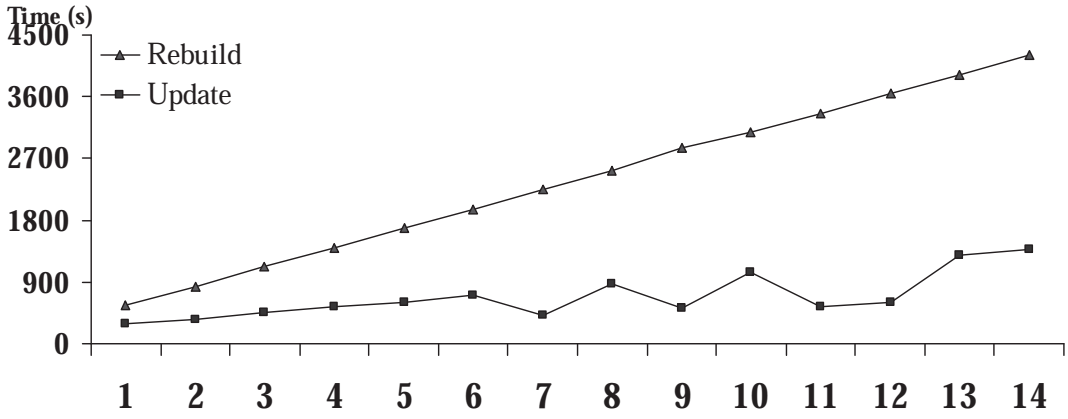
5.5 Summary

In this chapter, we have studied the problem of incremental k-gram index maintenance. Compared with inverted word index maintenance, incremental k-gram index update has its additional challenges: the selectivity changes of k-grams result in inclusion and exclusion of k-gram postings in the existing index. Rescanning the text corpora is very costly and may cause long service disruption.

We propose to keep a corpus "summary" together with the k-gram index: the summary consists of all max length k-grams with their posting lists. By doing so, our index update algorithm reuses the computation during the construction of the original index. The index update algorithm then becomes similar to the final merge phase of the k-gram index construction algorithm. Because the summary file size is much smaller than the original corpus size and contains structured information, we achieve significant speedup in index updates as our experiments on two real-world datasets show.

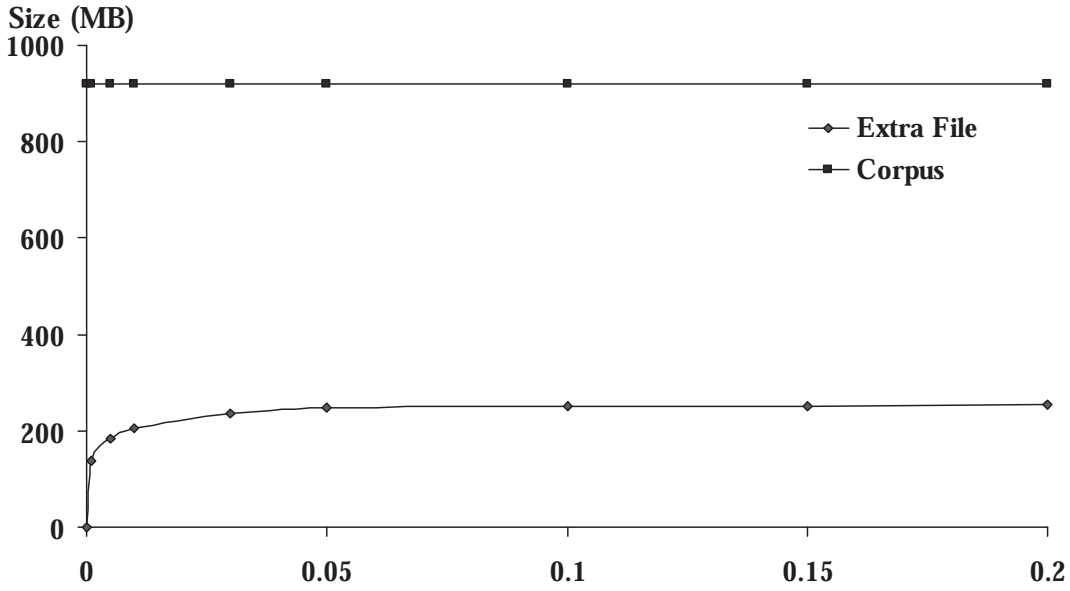


(a) update time vs. days: Enron

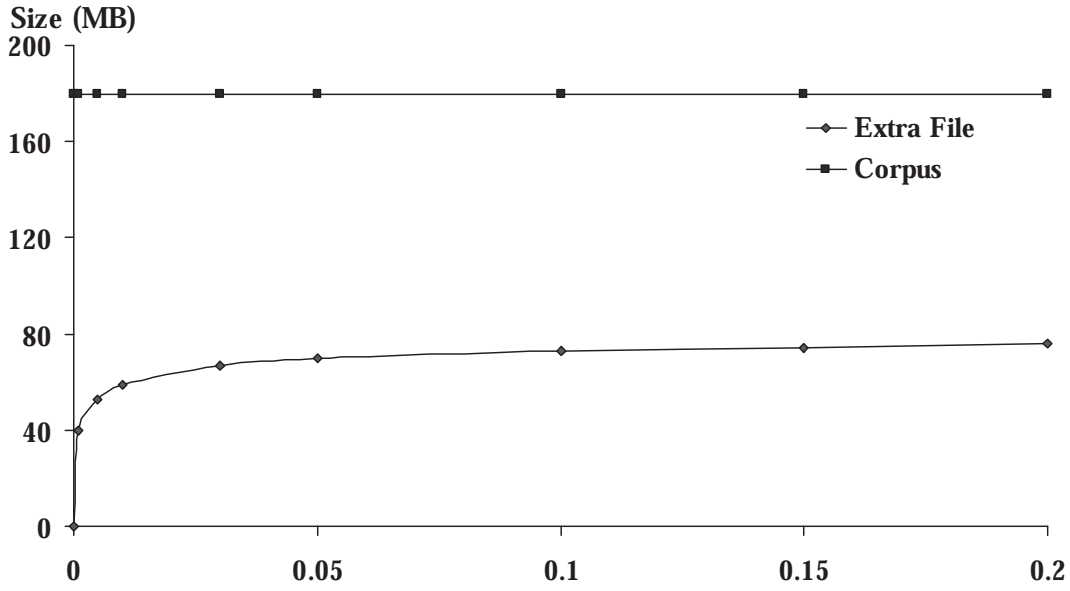


(b) update time vs. days: Web

Figure 5.3: Index update time



(a) extra information file size vs β: Enron



(b) extra information file size vs β: Web

Figure 5.4: Auxiliary file sizes used in index update

Chapter 6

Related Work

Inverted index engines like Trex consist of several major components: the indexer, the index query optimizer and the index updater. Not surprisingly, we utilize many related research results in various fields of computer sciences during the construction of Trex. We will review the related work in this chapter.

We start the chapter by reviewing the work on regular expression. Regular expression is a classic formal language model which has been the focus of many research studies in the past decades. In our work, we apply the classic results which state the equivalence of regular expression to finite state automaton to devise an effective and efficient index query generation algorithm.

Inverted index construction and maintenance for large scale text corpora are the core technology behind many text centric applications such as Web search engines, email search tool kits and so on. This field draws the attention of researchers from various backgrounds, database, information retrieval, distributed systems and data compression. Section 6.2 looks into the related work of inverted index construction in the past decades. Section 6.3 reviews the work on querying inverted indexes. Section 6.4 surveys the work on index maintenance (or online index construction).

This thesis studies the construction of k-gram indexes, which is a special type of inverted indexes. K-gram (or q-gram, n-gram as called in the literature) is a tool frequently used in many linguistic tasks such as language translation and approximate string matching. We will first review the applications of k-gram. Recently, together with the rising needs for searching non-western text data such as oriental language texts, music data and DNA sequences, we have seen a new wave of research on k-gram index construction and querying. We will survey the related work of this area in Section 6.5.

Finally in Section 6.6 and 6.7 we will survey recent developments in applying regular expressions in non-traditional settings such as network packet filtering and digital content publishing and subscription.

6.1 Finding Regex Matches in a Document

Much of the early work on regex focused on efficiently finding regex matches in a string or a document (e.g., [97, 6, 45, 92]), using finite automaton [45], special index structures such as Patricia tree [6], or approximate string matching against the regex (e.g., for the Unix tool agrep [97]). This line of work and ours are clearly complementary. Once we have used an inverted index to locate a small set of documents, we can use the above techniques to speed up evaluating regexes over those documents.

In [54], Kleene shows that a regular expression is equivalent to a finite state automaton. In the classic paper [71] by McNaughton and Yamada, it is shown that a regex can be converted to a non deterministic finite automata in linear time. We utilize this result in our query construction algorithm.

In recent years, there have been renewed interests in efficient regular expression matching in the area of deep network packet filtering [87, 10]. These new research can be categorized into two main directions: new hardware [11, 23] are designed specifically for regex matching; and software based approaches which combine the strengths of time-efficient DFAs and space-efficient NFAs to improve the performance of automata-based regex matching [37, 86].

6.2 Building Inverted and K-gram Indexes

A popular algorithm to build inverted indexes for text documents is Spimi [89], which does not require the vocabulary to fit into memory. In Trex, the step of processing the document corpus to create the runs shares the same idea. Besides regex evaluation as in the current work, k-gram indexes are also used for substring searching and approximate string matching. Two main types of k-gram indexes exist: keeping both doc IDs and offsets, or keeping only doc IDs. The work

[53] builds the first type of index and uses relational normalization technique to reduce the amount of space used by offset information. TinyLex [24] builds the second type. It keeps the number of indexed k-grams manageable by filtering out k-gram whose selectivity is close to the joint selectivity of its substrings. Its index construction method however is inefficient in that it scans the corpus multiple times, like Free, and performs random reads during pruning.

The book by Witten [96] provides an excellent summary of early work on inverted index construction. More details can be found in [74] and [60]. The latter paper is the first publicized method for use sorting in index construction. Storing the list of vocabulary or lexicon is an essential step in both index construction and query. The papers by Heinz et. al. [43], Zobel et. al. [103] provide a thorough comparisons of various techniques. The much celebrated work [9] by Brin and Page describes in detail how to build inverted index for billions of Web pages for Google, now the most popular Web search engine.

Building inverted index in a distributed cluster can utilize more computing resources and thus improve the index construction time almost linearly as demonstrated by the earlier work in [72, 80, 49]. More recently, the widely used Map-Reduce framework [28, 34, 41] has also been applied in inverted index construction as shown in the work by McCreadie et. al. [69].

Index compression plays an essential role in reducing index size, construction time and index query time. The recent survey [31] by Ferragina et. al. provides both theoretical foundation and practical implementation of a vast array of techniques employed in the literature. It also provides downloads of the programs mentioned in the paper. For more-depth coverage, there are many papers on this topic like [82, 3, 94, 4, 105, 2, 104, 100, 44, 98].

In recent years increasing attention has been paid to the problem of indexing documents for regex evaluation, the focus of this thesis (e.g., [22, 46, 25]). But most work have either considered restricted settings or are limited in several ways. The work [46] for example considers the restricted setting of indexing relational tuples to support efficient processing of the LIKE predicate in SQL queries. And the work [25] indexes XML documents to support evaluating path expression queries, a restricted class of regex. The work [22] (i.e., Free) addresses a more general setting, and is the closest to our work. However, it is limited in several ways, as we have discussed in this thesis.

Finally, we note that the ideas employed to reduce the number of k-grams in k-gram indexes are similar to the pruning techniques used in constructing compact synopses for substring selectivity estimation, which is an active research area within the database query optimization community.

Text string data are ubiquitous within the databases used by real-world applications today. To retrieve these data, keyword queries or wild card queries are often used. The database query optimizer that formulates the corresponding query plans needs good estimates on the selectivities of the query keywords. Data structures based on suffix trees [70] such as count-suffix trees [55] and pruned count-suffix trees [47] can be used to get the estimates.

A count-suffix tree stores the suffixes of all the strings. The edges of the tree are character symbols. Thus a path in the tree corresponds to a substring in the text after we concatenate the characters of the edges on the path together. Each node in the tree contains the count (i.e., the number of occurrences) of the substring corresponding to the path that leads to the node. Given a query string, the optimizer can traverse the tree from the root and follow the path corresponding to the string to get the count. The count is then used as the selectivity estimate.

Count-suffix trees are often too big in size to fit in the main memory due to the large number of possible suffixes. Pruned count-suffix trees are then introduced to reduce the tree sizes. There are several pruning strategies such as limiting the maximum length of suffixes in the tree or deleting all suffixes whose counts are less than a threshold. We notice that these pruning methods are very similar to the ideas used to reduce the number of k-grams in k-gram indexes. For example, limiting the suffix length stored is the same as keeping k-grams of length up to a maximum value. Deleting all suffixes with counts less than a threshold corresponds to removing all k-grams whose selectivities are below a threshold.

6.3 Querying Inverted Indexes

Querying inverted word index for keyword query has been studied in [95, 12, 33]. The issue of random assessing an inverted index is important for efficient index query because most queries look up several terms in the indexes. In [66], the authors demonstrated that page-oriented access

can reduce the IO costs during index query phase. There is relatively few papers on querying k-gram indexes. Ogawa et. al. [78] looks into the query formulation for keyword queries on k-gram indexes and presents a dynamic programming solution to reduce the index query time.

6.4 Maintaining Inverted Indexes

In many applications like search engines and email search, the underlying text corpora change over time with the arrival of new documents. There are several approaches to update the indexes. One can re-build the entire index from scratch and this approach is used by modern search engines like Google [9].

Another approach is to perform incremental index update as pioneered by the work [13] of Brown et. al.: newly added documents are first indexed in memory; when the memory is full, the in-memory index is merged with the existing on-disk index. [62] compares two main index merge methods: The in-place method, as shown in [93] and [85], leaves gap between posting lists and appends doc IDs of new documents to the end of each existing posting list. Statistical methods can be applied to estimate the amount of gap space required for accommodating updates as shown in [84]. On the other hand, the re-merge method, which is studied in [26], reads existing posting lists and writes out new combined posting lists to a new disk location. The re-merge method is shown to be better than the in-place method in most practical cases except when the new added document set is tiny or the memory cache is small. The geometric partition method [62] extends the re-merge method: it keeps several sub-indexes whose sizes are arranged in a geometric sequence. The new index is always re-merged with the smallest existing index to save index merge time. The drawback of such a method is that an index query processor has to query and combine the results of several indexes. Lucene [67] is an open source inverted index engine which uses the geometric partition update approach. All the work above focus on inverted *word* index maintenance. There is no study on maintaining inverted k-gram indexes.

More work on inverted index maintenance can be found in [61, 15] on how to maintain the good balance of index fragments and query aggregation time. [16, 63] studied how to minimize index space during on-line maintenance. More recent work on the area of index maintenance cover issues

such as utilization of multi-core CPUs as in [40], using balanced tree data structures and landmarks as in [39, 65], minimizing the operating cost as in [68] and providing real-time query support as in [59].

Finally, we notice the close resemblance between incremental inverted index update and view update for database and data warehousing systems [8, 102, 57, 101]. The latter is a long standing problem in the database community which has been studied in the past two decades. Many techniques such as computing source data changes (or delta) and delayed batch update are shared by the two lines of research.

6.5 K-Grams and Their Applications

K-gram or N-gram model [14, 27] has long been studied in the information retrieval and statistical natural language processing literature because it requires little knowledge (e.g. parsing structure) on the underlying language. N-gram also finds its usage in three important areas of text applications:

1. Substring search over text collection where there is no available text parsing tool or tokenizers [19, 18, 73]. N-gram index system has the distinct advantage that there is no need to query rewriting and term stemming.

2. Approximate string search [36, 64, 5, 91]: In many practical situations like inputs with error or exploratory search, we only looks for approximate matching to an input string. N-gram index can be used because it breaks a string into n-grams and look for matchings in the n-grams. Two similar strings have overlapping n-grams and the number of such identical n-grams can help to detect the similarity of two strings.

3. Search oriental text [58, 75, 56, 77, 76] and multimedia text [42] (e.g. music notes) which have not word delimiters as in the English text.

6.6 Indexing Regexes

A related recent line of work focuses on the following problem: how to index a large set $\mathcal{R}$ of regexes such that given a document d, we can quickly locate a relatively small set of regexes that potentially match d. This problem arises in applications such as information extraction, data stream monitoring and filtering, publish/subscribe systems, and others (e.g., [83, 29, 20, 52]). This work is orthogonal to ours. Though it would be interesting to study how to combine the two lines of work to address cases of matching many regexes over a large number of documents.

6.7 Domain-Specific Regex Evaluation

Finally, we note that many specialized techniques have also been developed to evaluate regexes efficiently in various specific application domains. Examples include information extraction [35, 1, 9, 17, 83, 30, 81, 48, 88], network packet scanning and filtering [99], information dissemination[29] and document search tools like grep[38] and agrep[97]. In information extraction, for example, the work [21, 32] develop specialized techniques to scale up d-entity recognition. Given a document corpus and a set of token-based regexes the work [32] incrementally builds a token-centric index to speed up regex evaluation. As other examples, [36] used positional q-gram inverted index for approximate string matching in RDBMSs. The work [7] developed techniques that use sample query workload to reduce k-gram inverted index size. Finally, the work [90] proposed frequent subsequence mining technique to prune k-gram indexes, as used in chemical molecule text searching.

Chapter 7

Conclusions

Fast regex evaluation is a fundamental operation in many text-centric applications, such as information extraction, text integration, business intelligence, exploratory analysis, Web search, and personal information management. As the number of such applications increases, so does the critical need for supporting this operation.

In response, we have analyzed the limitations of Free, a leading method for indexing text documents to support fast regex evaluation. We then proposed Trex, which significantly advances the state of the art. Trex builds k-gram indexes, like Free, but uses a radically different and more efficient solution. Trex indexes novel features, such as transformed documents. In addition, Trex developed a more efficient novel regex analysis method. Finally, we have developed an incremental index update method to maintain the indexes over time, as documents are added or deleted from the corpus. Extensive experiments on two real-world data sets show the significant utility of Trex.

Bibliography

[1] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *Proc of the 5th ACM Intl Conf on Digital Libraries*, 2000.

[2] V. N. Anh and A. Moffat. Universal code word sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

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

[4] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. on Knowl. and Data Eng.*, 18(6):857–861, June 2006.

[5] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06. VLDB Endowment, 2006.

[6] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[7] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE 09*, 2009.

[8] J. Blakeley, P.-A. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, 1986.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, 1998.

[10] Bro intrusion detection system. http://www.bro-ids.org/.

[11] B. Brodie, D. Taylor, and R. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 191–202, 2006.

[12] E. Brown. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '95, pages 30–38, 1995.

[13] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, 1994.

[14] F. Brown, P. deSouza, R. Mercer, V. Pietra, and J. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, December 1992.

[15] S. Büttcher and C. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, 2005.

[16] S. Büttcher, C. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *In SIGIR 06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 356–363, 2006.

[17] R. Caruana and P. Hodor. A high-precision workbench for extracting information from the protein data bank (pdb). In *KDD Workshop on Text and Information Extraction*, 2000.

[18] W. Cavnar. N-gram-based text filtering for trec-2. In *Proceedings of TREC 2*, 1993.

[19] W. Cavnar. Using an n-gram-based document representation with a vector processing retrieval model. In *Proceedings of TREC 3*, 1994.

[20] C.-Y. Chan, M. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003.

[21] A. Chandel, P. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE 06*, 2006.

[22] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *ICDE02*, 2002.

[23] C. Clark. and D. Schimmel. Scalable pattern matching for high speed networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257, 2004.

[24] D. Coetzee. Tinylex: Static n-gram index pruning with perfect recall. In *CIKM 08*, 2008.

[25] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB 01*, 2001.

[26] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '90, 1990.

[27] F. Damerau. *Markov Models and Linguistic Theory*. 1971.

[28] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, 2004.

[29] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *ICDE 02*, 2002.

[30] O. Etzioni, M. Cafarella, D. Downey, A. Popescu, T. Shaked, S. Soderland, D.Weld, and A. Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 2005.

[31] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *J. Exp. Algorithmics*, 13, February 2009.

[32] S. B. G. Ramakrishnan and S. Joshi. Entity annotation using inverse index operations. In *Empirical Methods in Natural Language Processing*, 2006.

[33] E. Garfield. The permuterm subject index: An autobiographical review. *Journal of the American Society for Information Science*, 27(5):288–291, 1976.

[34] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[35] G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.

[36] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a dbms for approximate string processing. *IEEE Data Eng. Bull.*, 24(4):28–34, 2001.

[37] T. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, December 2004.

[38] grep. http://www.gnu.org/software/grep/.

[39] R. Guo, X. Cheng, H. Xu, and B. Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, 2007.

[40] H. H. Yamada and M. Toyama. Scalable online index construction with multi-core cpus. In *Proceedings of the Twenty-First Australasian Conference on Database Technologies*, 2010.

[41] Apache hadoop. http://hadoop.apache.org/.

[42] M. H. Hamza, editor. *Internet and Multimedia Systems and Applications, EuroIMSA 2005, Grindelwald, Switzerland, February 21-23, 2005*. IASTED/ACTA Press, 2005.

[43] S. Heinz, J. Zobel, and H. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, April 2002.

[44] W. Hon and R. Shah. Compression, indexing, and retrieval for massive string data. In *Proceedings of the 21st annual conference on Combinatorial pattern matching*, CPM'10, pages 260–274, 2010.

[45] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. 2001.

[46] B. Hore, H. Hacigumus, B. Iyer, and S. Mehrotra. Indexing text data under space constraints. In *CIKM 04*, 2004.

[47] H. V. Jagadish, O. Kapitskaia, R. Ng, and D. Srivastava. One-dimensional and multi-dimensional substring selectivity estimation. *The VLDB Journal*, 9(3):214–230, 2000.

[48] T. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Engineering Bulletin*, 2006.

[49] B. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, February 1995.

[50] S. Jukna. A criterion for monotone circuit complexity. *Unpublished script*, 1991.

[51] S. Jukna. Finite limits and monotone computations: the lower bounds criterion. In *Proc. 12-th Ann. IEEE Conf. on Comput. Complexity 1997*, 1997.

[52] R. Kandhan, N. Teletia, and J. M. Patel. Sigmatch: Fast and scalable multi-pattern matching. In *VLDB 10*, 2010.

[53] M. Kim, K. Whang, J.-G. Lee, and M. Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB 05*, 2005.

[54] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, 1956.

[55] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD 96*, SIGMOD '96, 1996.

[56] K. Kwok. Comparing representations in chinese information retrieval. *SIGIR Forum*, 31(SI):34–41, July 1997.

[57] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, 2000.

[58] J. Lee and J. Ahn. Using n-grams for korean text retrieval. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '96, 1996.

[59] R. Lempel, Y. Mass, S. Ofek-Koifman, D. Sheinwald, Y. Petruschka, and R. Sivan. Just in time indexing for up to the second search. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, 2007.

[60] M. Lesk. Grab - inverted indexes with low storage overhead. *Computing Systems*, 1(3):207–220, 1988.

[61] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, 2005.

[62] N. Lester, J. Zobel, and H. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *IN PROCEEDINGS OF THE 27TH CONFERENCE ON AUSTRALASIAN COMPUTER SCIENCE*, pages 15–23, 2004.

[63] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, July 2006.

[64] C. Li, B. Wang, and X. Yang. Vgram: improving performance of approximate queries on string collections using variable-length grams. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 303–314. VLDB Endowment, 2007.

[65] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, 2003.

[66] X. Liu and Z. Peng. An efficient random access inverted index for information retrieval. In *Proceedings of the 19th international conference on World wide web*, WWW '10.

[67] Lucene. http://lucene.apache.org/.

[68] G. Margaritis and S. Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, 2009.

[69] R. McCreadie, C. Macdonald, and I. Ounis. On single-pass indexing with mapreduce. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, 2009.

[70] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[71] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[72] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, July 2001.

[73] E. Miller, D. Shen, J. Liu, and C. Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *JOURNAL OF DIGITAL INFORMATION*, 1, 2000.

[74] A. Moffat and T. Bell. In situ generation of compressed inverted files. *J. Am. Soc. Inf. Sci.*, 46(7):537–550, August 1995.

[75] S. Mustafa and Q. Al-Radaideh. Using n-grams for arabic text searching. *J. Am. Soc. Inf. Sci. Technol.*, 55(11):1002–1007, September 2004.

[76] J.-Y. Nie, J. Gao, J. Zhang, and M. Zhou. On the use of words and n-grams for chinese information retrieval. In *Proceedings of the fifth international workshop on on Information retrieval with Asian languages*, IRAL '00, 2000.

[77] Y. Ogawa and T. Matsuda. Overlapping statistical word indexing: a new indexing method for japanese text. In *Proceedings of the 20th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '97, pages 226–234, New York, NY, USA, 1997. ACM.

[78] Y. Ogawa and T. Matsuda. Optimizing query evaluation in n-gram indexing. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, 1998.

[79] A. Razborov. Lower bounds for the monotone complexity of some boolean functions. *Soviet Math. Doklady*, 31:354–357, 1985.

[80] B. Ribeiro-Neto, E. Moura, M. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, 1999.

[81] B. Rosenfeld and R. Feldman. Ures: an unsupervised web relation extraction system. In *Proc of the COLING/ACL*, 2006.

[82] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '02, 2002.

[83] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB 07*, 2007.

[84] W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. *Inf. Process. Manage.*, 41(2):275–288, March 2005.

[85] K. Shoens, A. Tomasic, and H. García-Molina. Synthetic workload performance analysis of incremental updates. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, 1994.

[86] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, 2008.

[87] Snort. http://www.snort.org.

[88] S. Soderland. Learning information extraction rules. *Machine Learning*, 1999.

[89] H. Steffen and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.

[90] B. Sun, P. Mitra, and C. Giles. Mining, indexing, and searching for textual chemical molecule information on the web. In *WWW 08*, 2008.

[91] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, CPM '96, pages 50–63, London, UK, 1996. Springer-Verlag.

[92] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[93] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994.

[94] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, January 2003.

[95] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, November 1995.

[96] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. 1999.

[97] S. Wu and U. Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[98] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, WWW '09, 2009.

[99] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc of ACM / IEEE Symposium on Architectures for Networking and Communications Systems 2006*, 2006.

[100] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08.

[101] J. Zhou, P.-A. Larson, and H. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, 2007.

[102] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, 1995.

[103] J. Zobel, S. Heinz, and H. Williams. In-memory hash tables for accumulating text vocabularies. *Inf. Process. Lett.*, 80(6):271–277, December 2001.

[104] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.

[105] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, 2006.

APPENDIX
Regular Expressions Used in the Experiments

A.1 Regular Expressions In Enron Dataset

1. [a-zA-Z0-9][_a-zA-Z0-9-]*(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+
(\.[a-zA-Z0-9-]+)*\.(([0-9]{1,3})|([a-zA-Z]{2,3})|(aero|coop|info|museum|name))

2. (([A-Z][a-z](.){0,10}))?(([A-Z]\.?){1,2}\s+)?([A-Z][a-z](.){0,20}(['-].(.)
{1,8})?[a-zA-Z\d_]{0,2}\s*[/]\s*)(([A-Z](.){1,20}\s*){1,2}[/]\s*){1,2}
([A-Z]){2,20}(@(.){1,20})?

3. \+?(1[\-\.\s]\s*)?(\(\s*)?[1-9]\d{2}(\s*\))?[\-\.]\d{3}[\-\.]\d\d\d\d

4. (X|x)[\.\-]?\d{4,5}

5. (\+\s*)?\(\s*[1-9]\d{2}\s*\)\s*[\-\.]?\d{3}[\-\.\s*]?\d{4}

6. \d[.\d]+\s*(miles|mins|min|mi|kms|km|hours|hrs|ft)\s

7. (\+?1[\-\.\s]\s*)?8\d{2}\s*[\s\-\.][A-Z\d]{3}[\.\-]?[A-Z][A-Z][A-Z][A-Z]

8. ([A-Z][a-z][a-zA-Z\d_]{1,20}\s*/\s*)(([A-Z].{1,20}\s*){1,2}/\s*){1,2}([A-Z])
{2,20}(@(.){2,20})?

9. (\+\s*)?[1-9]\d{2}\s*[\s\-\.]\d{3}[\-\.\s]\s*\d{4}

10. ((0)|(1)|(2)|(3))?[\d]( )*[\-]( )*((0)|(1)|(2)|(3))?[\d]

11. \d\d\d[-.)]*\d\d\d\d

12. (((call|dial)-?in)|(toll)|(p/c))\s*((#|number)|(.{0,10}:))\s*.{0,10}
\s*\(?\s*\d{3,}(-|\d|\.| |\)){0,10}

13. \d\d{1,2}\s*[\(\s\-\.]((\d{2}\s*[\)\s\-\.]\d{8})|(\d{5}\s*[\)\s\-\.]
\d{5})|(\d{2}\s*[\)\s\-\.]\d{4}\s*[\s\-\.]\d{4}))

14. (January|Jan|February|Feb|March|Mar|April|Apr|May|June|Jun|July|Jul|
August|Aug|september|Sep|October|Oct|November|Nov|December|Dec)\s+\d{1,2}

15. (\d{1,2}|\d{4})[\-/]\d{1,2}[\-/]\d{1,4}

16. \s[A-Z][A-Z&-]+\s+

17. (follow\s+(([A-Za-z]+\s*-?\s*\d+)|(\d+\s*-?\s*[A-Za-z]+))\s+
(to\s+the\s+end|for))

18. Subject\s*\n

19. (keep\s+going\s+straight)

20. (make\s+a?\s*(right|left)|(directions?\s+(from|to)\s+[a-zA-Z\d_]+

```
\s+(from|to)[a-zA-Z\d_]+)|(suggested\s+(route|directions))|(take\s+exit)
|(u(\s*-\s*)?turn\s+(on|at))|(at\s+the\s+[a-zA-Z\d_]+\s+light)|
(will\s+be\s+(on|to)\s+(the|your))\s+(right|left))([ A-Za-z0-9\t])*
```

21. `let.{0,15}meet\s+at`

22. `I\s+am\s+`

23. `(exit|merge)\s+(onto|at|on)?\s*(([A-Za-z]+\s*-?\s*\d+)|`
    `(\d+\s*-?\s*[A-Za-z]+))`

24. `((turn|bear)\s+(left|right)\s+(onto|on|to|at))`

25. `(take\s+(the\s+)?(([A-Za-z]+\s*-?\s*\d+)|(\d+\s*-?\s*[A-Za-z]+)))`
    `([ A-Za-z0-9\t,])*`

26. `call\s*me.?\s*(\b\w+\b\s*){0,3}\s+(at\s*)?`

27. `(From\s+[A-Z][a-z]+)([ A-Za-z0-9\t,])*`

28. `(Give|give)(\s)*me(\s)*a(\s)*call(\s)*((\b\w+\b)\s*){0,3}`

29. `((ht|f)tps?://[a-zA-Z\d_]+[a-zA-Z\d_\-:&=?/~.<>@:]+[a-zA-Z\d_\-:&=?/~]{2,100})`

30. `my\s+name\s+is\s+`

31. `(pass|conference)\s?code\s*.{0,2}\s*\d{3,}(-|\d|\.){0,10}`

32. schedule\s+to\s+meet.{0,15}at

33. ((corner|intersection)\s+of\s+[A-Z][a-z]+)([A-Za-z0-9\t,])*

34. (head.{0,20}(north|south|east|west)([A-Za-z0-9\t,])*)

35. (take\s+(([A-Za-z]+\s*-?\s*\d+)|(\d+\s*-?\s*[A-Za-z]+))(\s+exit)\s*to)

36. (From:\s*.{1,40}\s*\[mailto:.{5,50}\])|(From:\s*.{1,50}\s*\n\s*To:)

38. (\w+)\s+talks with\s+(\w+)

39. exit.{0,20}(north|south|east|west)

40. (Start(ing)?\s+at)([A-Za-z0-9\t,])*

41. (block.{0,20}(left|right))([A-Za-z0-9\t,])*

42. reserved.{0,15}conference\s+room

43. fax to (\d){3,10}

44. (972[\-\.]\d{1,2}[\-\.]\d{7})

45. (directions\s+from\s+(([A-Za-z]+\s*-?\s*\d+)|(\d+\s*-?\s*[A-Za-z]+)))

46. (follow\s+the\s+signs?\s+(to|for))/

47. (>>)+

48. Quoting\s*.{1,120}?:\s*(\n)+\s*(>\s*)

49. (directions?\s+to\s+my\s+house)

50. \n\s*directions\s*(\n|(\W+(:|From|To|Via))([A-Za-z0-9\t,]))*

51. \-{5}\s*Message\sfrom\s*.{5,30}\s.{6,40}\s*on\s*[A-Z][a-z]
    {2,8},\s*\d{1,2}\s*[A-Z][a-z]{2,8}\s*\d{4}(.|\n){10,18}\s*\-{5}

52. (Participant|participant|Host|host|Moderator|moderator)\s*(pin)?
    \s*(access)?\s*code\s*.{0,10}\s*\d{3,}(-|\d|\.){0,10}

53. \[(Attachment|attachment)\s*deleted\s*by\s*.{1,60}?\]

54. ((Tie|(T/L)|TL|PC|Intl|International|Domestic|number|Toll|
    Call-in\s+(information|info)|dial|info|password))
    (.{0,10}\s*(-|\d|\.| |\(|\)){4,15})*

55. directions?\s+to\s+(my\s+(house|home)|the\s+party)

56. (An Der|An der|Am|am|die|Die|der|Der|das|Das)?[\t]*
     (Brcke|Park|Postfach|Hauptbahnhof)\b\s*,?\s*([[1-9].\-\,])*
      \d([\d./\-\,])*

57. ((Driving|driving)[\t]+(Directions|directions).{0,30}
     (:|From|To|Via))([A-Za-z0-9\t,])*

58. ([A-Z]\.\s*){1,5}

59. \s*\-{5}\s+Original\s*Message\s+\-{5}\s*(\n{0,3})?
\s*From:.{1,50}?

60. (My|my)(\s)*((\b\w+\b)\s*){0,2}((#|number)|(phone|cell
([-(\s)*]*phone)?)(\s)*(#|number)?)(\s)*((\b\w+\b)\s*){0,3}

61. total\W{0,2}\w{0,15}\W{0,2}((time)|(distance))([A-Za-z0-9\t,])*

62. ([A-Za-z]{4,10}.{0,5}becomes.{0,5}[A-Za-z]{4,10})([A-Za-z0-9\t,])*

63. (Subject:)|(Subject\s*\n)

65. At\s+\d{2}:\d{2}\s+.{1,9}\s+.{1,6},\s+you wrote:

66. conference\s+call.{0,10}\s*\(?\s*\d{3,}(-|\d|\.| |\)){0,10}

67. Subject:

68. [a-zA-Z\d_]+[a-zA-Z\d_\-:&=?/~.<>@:]+(\.com/|\.edu/|\.org/)
[a-zA-Z\d_&?~.<>@:][a-zA-Z\d_\-:&=?/~.<>@:]+[a-zA-Z\d_\-:&=?/~]{2,100}

69. (Ext|ext)\s*[\.\-\:]?\s*\d{3,5}

70. \d{2,4}(/|\.)\d{2,4}(/|\.)\d{2,4}\s\d{2}\:\d{2}(\s+(PM|AM))?

71. \d+(\.\d+)?\%

72. (\d{1,2}:\d{2}(:\d{2})?)\s*

73. To:\s*[^:]{1,200}\s*\n(>\s*)*\s*(CC|cc|Cc):\s*(\n)?

74. (Subject:)|(Date:.{20,80}\s*\n)

75. ([\.\?!]+\s)|(\n\s*\n)

76. (buy|sell).{0,25}[A-Z]{2,4}

77. [A-Z]+[0-9]+[a-zA-Z]*

78. ([A-Z][a-z](.){0,20})?(Allee|allee|Berg|berg|Chaussee|chaussee
|Damm|damm|Gasse|gasse|Gaerten|gaerten|Halde|halde|Hof|hof|Hoefe|hoefe
|Landstrasse|landstrasse|Markt|markt|Maerkte|maerkte|Pfad|pdad|Platz
|platz|Ring|ring|Steig|steig|Str\.|str\.|Strasse|strasse|Ufer|ufer|Weg
|weg|Zeile|zeile)\s*,?\s*([[1-9]\.\-\,])*\d([\d./\-\,])*

79. ((20)|(19)|(18)|(17)|(16))\d\d

80. ((([a-zA-Z\d_]+:)//)|(w[a-zA-Z\d_]+\.))(([a-zA-Z\d_]|%[a-fA-f\d]{2,2})
+(:([a-zA-Z\d_]|%[a-fA-f\d]{2,2})+)?@)?([a-zA-Z\d_][-a-zA-Z\d_]{0,253}
[a-zA-Z\d_]\.)+[a-zA-Z\d_]{2,4}(:[\d]+)?(/([-+_~.\d\w]|%[a-fA-f\d]{2,2})*)*
(\?(&?([-+~.a-zA-Z\d_]|%[a-fA-f\d]{2,2})=?)*)?(#([-+~.a-zA-Z\d_]|%
[a-fA-f\d]{2,2})*)?

81. (\+\s*){0,2}[\d(]([\d ()\-./]){9,20}\d

82. [a-zA-Z\d_]+[a-zA-Z\d_\-:&=?/~.<>@:]+\.(com|edu|org)/
[a-zA-Z\d_&?~.<>@:][a-zA-Z\d_\-:&=?/~.<>@:]+[a-zA-Z\d_\-:&=?/~]{2,100}

83. (jan(uary)?\.?|feb(ruary)?)\.?\s\d\d?(st|nd|rd|th)?

84. .{1,30}/.{1,25}/.{1,20}(\@.{1,20})?\s*\n(>\s*)*

85. On\s*(([A-Z][a-z]{2,10},?\s*\d{1,2},?\s*[A-Z][a-z]{2,10}\s*\d{2,4},)
|(\d{1,2}/\d{1,2}/\d{1,2},))\s*.{1,100}?\s*wrote\:

86. To\s*\n(>\s*)*.{5,1000}?\s*\n(>\s*)*\s*cc\s*\n

87. \d[\dA-Za-z_\-\,]*(( )*\d[\dA-Za-z_]*)?

88. [A-Z][a-z](.){0,20}

89. (D-)?[0-9]{4,5}

90. [A-Za-z\&\.-/]{1,20}

91. I\s*can\s*be(\s)*((\b\w+\b)\s*)at\s*.?\s*

92. (([1-9A-Za-z_][\dA-Za-z_\-\,]*)?\d[\dA-Za-z_\-\,]*)\b

93. \s*\sme\s*((\b\w+\b)\s+){0,2}

94. [\d(].+

95. \d{1,5}-?[A-Z]?

96. [0-9]{5}(\-[0-9]{4})?

97. \d{1,2}([:.][0-5]\d)?\s*(-\s*\d{1,2}([:.][0-5]\d)?)?

98. [1-9]\d{0,2}(\s*(N|S|E|W))?(\s*(N|S|E|W))?

99. \(?\d{2,4}[\.\-]?\)?(\s*\(0\)\s*)?(\s*[\-\.]\s*)?(\s*[\s\.\-]
\(?\d{1,2}(\s*\d{2}\s*){1,2}\)?)?(\s*[\s\.\-]?\d{2,4})(\s*[\s\.-]
?\d{1,4}){1,3}

100. ([A-Z]\.?\s*){1,5}

101. \d{5}

102. [A-Z](.){0,10}(['-][[A-Z]])?(.){1,10}

103. ([A-Z]){1,2}((\s*&\s*)?([A-Z])+|([a-z])+([A-Z])([a-z])+)

104. [a-zA-Z][\w\&\-]+

105. (Thanks|Regards).*\n+

106. [A-Z]([a-z]|[\&\.'\-\,])+\b

A.2 Regular Expressions In DBLife (Web) Dataset

1. UW(\s+(at|in|,|-|--))?\s+Madison

2. Rice\s+(University|Univ.|Univ)

3. Indian\s+(Institute|Inst.|Inst)\s+of\s+Technology
   (\s+(at|in|,|-|--))?\s+Bombay

4. RWTH\s+(\s+(at|in|,|-|--))?\s+\s+Aachen\s+(University|Univ.|Univ)

5. National\s+((University|Univ.|Univ)\s+of|U|U.)\s+Singapore

6. Upsala\s+Universitet

7. Yale

8. Columbia\s+(University|Univ.|Univ)(\s+(at|in|,|-|--))?\s+
the\s+City\s+of\s+New\s+York

9. INRIA

10. Prof(\.?|essor)\s+Places

11. Dr\.?\s+(DeWitt|Dewitt)

12. (David|Dave)\s+(DeWitt|Dewitt)

13. A\.?\s*M\.?\s+Tjoa

14. A\.?\s+Al\-[Kk]hudair

15. Wierman\,\s+A\.?

16. Anagnostopoulos\,\s+Achilleas

17. Zohreh\s+Talebi

18. Z\.?\s*A\.?\s+Talebi

19. Z\.?\s+Li

20. Kovacs\,\s+Z\.?

21. Lacroix\,\s+Z\.?

22. [A-Z][A-Z0-9*\-=]+

23. [\s*\-=]+[A-Z]

24. [A-Z][A-Za-z0-9',\-\(\)]+?[:]?[*\-=\s]*

25. \s*(Organiz|Award|Program|Industrial|Conference|Session|PC|Programme|
Workshop|Tutorial|Local Arrangements|General|Demonstration|Proceeding|Registration
|Web|Deputy|Area)\s+(chair|members|committee)

26. <[hH]\d|strong>.*?<\/[hH]\d|strong>

27. [A-Z][A-Z]+\s*['-]\s*(20\d\d|\d\d)

28. \([A-Z]\w+[\W\s]*(\d\d+)?\)

29. sigmod\/pods|vldb|sigmod|pods|icde|icml|kdd|edbt|cidr|cikm|icdt|webdb|icdm|aaai|sigkdd|pkdd|ecml