A Fast Regular Expression Indexing Engine

Junghoo Cho
University of California, Los Angeles
cho@cs.ucla.edu

Sridhar Rajagopalan
IBM Almaden Research
sridhar@almaden.ibm.com

Abstract

In this paper, we describe the design, architecture, and the lessons learned from the implementation of a fast regular expression indexing engine FREE. FREE uses a prebuilt index to identify the text data units which may contain a matching string and only examines these further. In this way, FREE shows orders of magnitude performance improvement in certain cases over standard regular expression matching systems, such as `lex`*,* `awk` *and* `grep` *[18, 4].*

1 Introduction

The amount of data in text databases and the world wide web continues to grow at a prodigious rate.[1] This unprecedented increase in the size of modern datasets has made otherwise simple and well understood tasks challenging. A good instance of these challenges is the matching of regular patterns (i.e. find all substrings which satisfy a given regular expression) in a corpus. For reasonably large text databases, say, 100G bytes, matching a regular expression takes several days. This paper addresses scale and performance issues when we match regular expressions (a *regex*) against a large corpus.

We propose and consider one of the simplest, perhaps even obvious, methods to speed up a regex matching engine – using a carefully designed index to restrict the amount of data which needs to be examined. Even in this elementary context, there are a fairly large set of interesting choices and design decisions to make. We first illustrate how one can use an index to speed up a regex matching.

Example 1.1 The following regex describes all the URLs which point to MP3 files on the web.[2]

```
<a href=("|')?.*\.mp3("|')?>
```

From the regex, it is apparent that any web page containing such a URL should contain the substring **`.mp3`** as well. Thus, by looking up the string **`.mp3`** in an index, and only examining pages which contain the string, we could significantly reduce both the processing and the I/O workload. □

The example leaves behind several unanswered questions. For example, should the system also look up **`<a href=`** from the index? For what strings should the system create index entries? Should it create an entry for **`<a href=`**?

A simple answer, commonly known as a k-gram index, is to create an index entry for every sequence of k characters in the corpus. For example, setting $k = 2$ for the text **`ABCDEF`**, there would be an index entry for each of **`AB`**, **`BC`**, **`CD`**, **`DE`**, and **`EF`**. Then by creating k-gram indexes for reasonable k values, say $k = 2, 3, \ldots, 10$, the system looks up any string within a given regex to identify what part of the text may contain a matching string. However, this approach would be prohibitively expensive – both in the time it would take to build such an index and the amount of storage needed in maintaining it.

In this paper, we study how we can identify a set of useful keys to be indexed, which maximize run-time performance, while minimizing the size of the index. In particular, we propose an index structure, called a *multigram index*. A multigram is a consecutive sequence of characters of arbitrary length. For instance, **`<a hre`** and **`.mp3`** are all multigrams of lengths 6 and 4 respectively. Our multigram index automatically identifies "good" multigrams to be indexed and provides a tradeoff between run-time performance and index size. In summary, this paper makes the following contributions:

1. **Index design:** An algorithm to identify a set of good multigram entries to be indexed and provides theoretical and practical rationale validating our index design.
2. **Index construction:** An efficient index construction algorithm. As we will see, this algorithm is related to the frequent-item-set mining problem, which has been extensively studied in the database community.
3. **Query compilation:** A framework to develop an efficient execution plan given a query.
4. **Query execution** In the extended version of this pa-

[1]An Inktomi and NEC joint study [12] announced in early 2000 that the web had exceeded 1,000,000,000 pages.

[2]In Table 1 we summarize the basic syntax of a regular expression and our shorthand.

per [10], we propose a new technique, called *anchoring*, that significantly speeds up in-memory regular expression match.

Motives Text and hypertext corpora have been a subject of interest in the recent database literature. There are two popular approaches to dealing with large text databases.

1. **Search:** The user types in a list of keywords related to a topic he is interested in, and the system returns pages related to the keywords. Note that the system only directs the user to potentially interesting pages. It is still the responsibility of the user to read and understand the returned pages and to extract the information the user is looking for.
2. **Data extraction or mining:** The system extracts relational data (or other structured views) from the raw text and inserts the extracted data to a traditional database system. The user now can use the rich functionality of the database system to summarize and extract the information that she is looking for.

In both of these approaches, we believe having a fast regex matching engine can be very helpful. First, consider the following example to see how a regex engine can help in the search context.

Example 1.2 (Improved search) How does one find the middle name of Thomas Edison? Using a keyword-search interface, the user may type in the keywords "Thomas Edison" or even "Thomas NEAR Edison" and read through the returned pages one by one, to identify the middle name.

Instead, assume a system which can match a regex quickly and return matching strings in the order of their occurrence frequencies. Then by issuing a regular expression, **`Thomas \a+ Edison`** to the system and looking at the results, the user may immediately find a set of interesting possibilities. In fact, when we issued this regex to FREE, the most frequent matching string was Thomas *Alva* Edison, which contains the correct middle name of the inventor, Thomas Edison. □

While the above example might seem like a toy example, it clearly shows that regexes provide a rich syntax in which the user can express her "information need" more concretely. Based on the user's concrete description, the system could summarize the result more compactly and save the user from going through a very large dataset.

Second, regexes are very useful in extracting structured information from text. For instance, Brin [8] proposed an algorithm which can extract relational data from the web, in which he used regex matching as a basic building block. Other systems [15, 1] also use the regex syntax to extract structured/semi-structured data from web pages. Therefore, it seems clear that fast regex matching is a fundamental primitive which will help advance the state of the art in both the search and mining of large text databases.

1.1 Prior work

Suffix trees Baeza-Yates and Gonnet [5] studied how to match a regular expression using a prebuilt index. In their work, they represented *all* suffixes of a given text corpus as a *trie* (called a suffix tree [21, 27, 19]) and performed regex matches directly on the trie. This approach has a clear advantage over ours when the corpus size is relatively small and the entire trie can be loaded into main memory: The regex match can be performed directly on the trie and does not need to refer back to the original corpus for final confirmation. However, the size of the trie is several times as large as the original corpus, so it is not a good option for a large corpus.

Recently, a disk-based index for strings was proposed in reference [11]. We may build such an index on the text corpus and perform regex match to the index. However, this approach can potentially be very slow, because it will involve a large number of unpredictable random seeks when we traverse the index for regex matching. In our approach, the index lookup is very fast, because all the keys in our index can be cached in main memory due to its small size. However, we need a final confirmation step to find the actual matching strings. It will be interesting to compare the performance of these two approaches.

Finite automata There is a large body of literature studying how to match a regex to a string (see the textbook [17] and [22, 9, 25, 6] for instance). The approach is to first convert a regex into an equivalent deterministic finite automaton (DFA), and then use the DFA to match the regex. To expedite the matching, most systems allow the user to save the constructed DFA, so that the user can reuse it when she wants to match the same regex against different text.

With large text databases, the situation is somewhat different. In many cases, the user often needs to match *different regexes* against the *same text database*. In this context, due to the large size of text databases involved, performance and scaling issues become paramount.

Inverted keyword indexes *Inverted indexes* are the most common index structure for a large text database. Several software vendors and many web search portals use inverted indexes as base technology for their product offerings. However, the key entries in an inverted index are *English words* or other *linguistic constructs* [24, 14, 13, 26, 20]. Therefore, inverted indexes are not helpful in processing regexes, which may contain substrings that are not linguistically meaningful.

Symbol	*Meaning*	*Example*
`.`	any character character `.` itself is represented as `\.`	`a.c` matches `abc`
`*`	zero or more repetition of the previous character	`a*` matches `aaa`
`+`	one or more repetition of the previous character	`a+ = aa*`
`?`	zero or one repetition of the previous character	`a?` matches `a` or null character
`|`	or connective	`a|b` matches `a` or `b`
`[]`	any character within the bracket	`[abc] = a|b|c`
`\a`	any alphabetic characters	`\a = [abc...z]`
`\d`	any numeric characters	`\d = [012...9]`
`[^abc]`	any character other than `a`, `b` or `c`	`[^abc]` matches `e`

Table 1. Brief summary of regular expression symbols and our shorthands

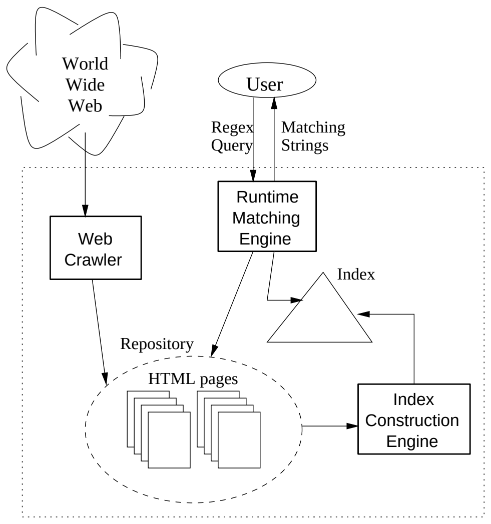


Figure 1. The architecture of FREE

Data mining To build our multigram index, we use some of the data mining techniques. In data mining community, many techniques have been developed to find frequent item sets for a large dataset in order to find association rules within data [2, 3, 23, 16]. While the goal of our project is different, these techniques are clearly helpful to minimize the index construction time for FREE.

2 Architecture overview

In this section, we describe the general architecture of FREE and explain the basic features of the system. FREE can handle general textual data from any source (e.g. Web pages, Net news articles, e-mail messages, etc.). Currently, FREE hosts a large database of HTML pages gathered from the web.

Figure 1 shows the high level architecture of FREE. The FREE system has three main components: (1) a web crawler, (2) an index construction engine and (3) a runtime

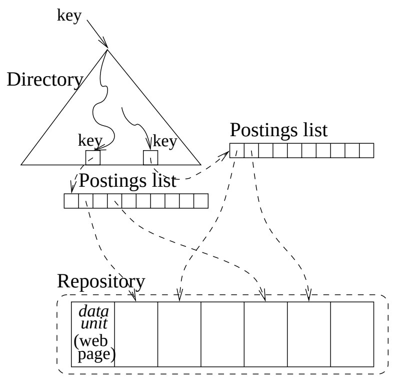


Figure 2. High level structure of an index

matching engine. The user interacts only with the runtime matching engine, through which she submits a regex query and obtains matching strings. We briefly go over some of these components now.

2.1 Index Construction Engine

Once pages are collected by the web crawler, the index construction engine constructs indexes on top of them. FREE uses an index structure called a *multigram index*.

The multigram index has the general structure shown in Figure 2. An index consists of a directory of keys and postings lists. The postings list for a particular key points to the data units where the keys appear. In a multigram index, keys are k-grams for a range of k values. As we have described earlier, not all k-grams are keys. We select very carefully what grams to be indexed as keys. We will describe the algorithm for key selection in Section 3.

2.2 Runtime Matching Engine

The runtime matching engine reads a regex query from the user and returns matching strings. Figure 3 shows the subsystems of the runtime matching engine.

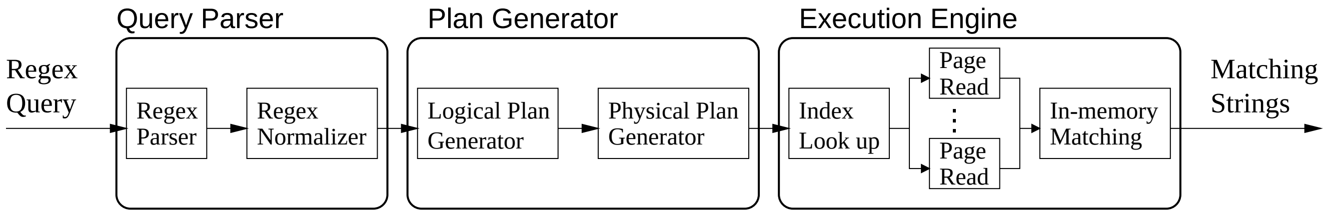


Figure 3. The architecture of Runtime Matching Engine

The runtime process consists of three important phases: (1) query parsing, (2) plan generation and (3) execution phases. Once a regex query is issued by the user, the query is parsed by the query parser and normalized into a standard form. This normalized regular expression then enters the plan generation phase in which the system determines what indexes to look up in what order. Based on this plan, the system retrieves the postings lists and performs the final match in the final execution phase.

The runtime system of FREE is interesting for two reasons:

1. *Similarity to an RDB engine:* The basic architectural blocks of the runtime resemble those in a typical relational database system.
2. *Performance implication:* The design of the index, the entries to look up and the index look-up order (execution plan) have a direct and very significant impact on performance.

We illustrate the second point using Example 2.1.

Example 2.1 Continuing on example 1.1, recall that **`<a href=("|')?.*\B.mp3("|')?>`** describes pointers to MP3 files on the Internet. Moreover, any page containing such a URL should contain the substring **`.mp3`**. Thus, by looking up **`.mp3`** in an index, one could significantly reduce the number of pages to read and examine.

Also, we notice that matches can occur only in pages containing the string **`<a href=`**. However, because most web pages contain at least one instance of the **`<a href=`** tag, using this information may not aid the runtime performance at all. In fact, it may even slow down the process, because of the additional overhead of looking through a large postings list. □

The quandary posed in the example above is typical of one in a relational database query optimization. To execute an SQL query, there exist multiple possibilities, and the system should carefully select a good execution plan to maximize performance. We explain the Runtime Matching Engine in more detail in Section 4.

3 Index structure

This section deals with the details of the index design and construction. We provide algorithms for index construction and mathematical reasoning behind our proposed algorithms.

3.1 Multigram indexes

A k-gram is a string $x = x_1 x_2 x_3 \cdots x_k$, of length k where each $x_i : 1 \leq i \leq k$ is a character. Formally, if Σ is the set of characters, then the set of k-grams is Σ^k. The term *multigram* (or simply gram) will denote any k-gram, i.e. where k is arbitrary, i.e. $\mathcal{M} = \cup_{k=1}^{\infty} \Sigma^k$ is the set of all multigrams. By a *data unit*, we mean the unit in which the raw data is partitioned. This can be a web page (in the case of a web search engine), a paragraph or a page (in the case of a document corpus). Given any multigram $x \in \mathcal{M}$, we define the *selectivity* of x as the fraction of data units which contain at least one occurrence of the gram.

Definition 3.1 (Gram and expression selectivity) Let $x \in \mathcal{M}$ be an arbitrary gram. Let there be N data units in our database, and let $M(x)$ denote the number of data units which contain x. Then the selectivity of the gram x, denoted $\text{sel}(x)$, is

$$\text{sel}(x) = M(x)/N$$

Analogously, if r is any regex, we denote by $M(r)$ the number of data units which contain a string matching r, and use $\text{sel}(r)$ to denote $M(r)/N$. □

Definition 3.2 (Gram and expression filter factor) The *filter factor* of a gram $x \in \mathcal{M}$ is given by $\text{ff}(x)$ is $1 - \text{sel}(x)$. and likewise for regexes. □

Example 3.3 Continuing our earlier example, let $r =$ **`<a href=("|')?.*\.mp3("|')?>`**. Let us further assume that $\text{ff}($**`.mp3`**$) = 0.99$. Thus, $\text{sel}($**`.mp3`**$) = 0.01$. Only 1% of data units contain the gram **`.mp3`**, so potentially we can "filter-out" 99% of data units by looking up **`.mp3`** from a gram index. We need to scan only the remaining 1% of the pages for r. □

Thus, if the selectivity of a gram is small, then it is a useful gram for our purposes. Conversely, when the selectivity of a gram x is large, the gram is not very useful for candidate data unit selection. For example, one would expect $\mathrm{sel}($**`<a href=`**$) \approx 1$, making the gram not very useful for us. This motivates our definition of *useful* and *useless* grams.

Definition 3.4 (Useful and useless grams) For any $0 \leq c \leq 1$, a gram x is c-useful if $\mathrm{sel}(x) \leq c$. A gram which is not c-useful is c-useless. When the value of c is clear from context, we will call grams *useful* or *useless*. □

In our applications, c will be chosen based on several system parameters, particularly processor speed and I/O subsystem performance. For instance, if a random access to data units on disk is 10 times slower than sequential access, then 0.1 would be a good candidate for the value of c. We treat c as a tunable parameter which is adjusted to match individual system performance.

Example 3.5 Does keeping useless grams in the index result in any meaningful benefit? The answer is not entirely clear. Consider for instance the regular expression $r =$ **`bb.*cc.*dd.+zz`**. While each of the 2-grams in r, **`bb`**, **`cc`**, **`dd`** and **`zz`** is likely to be useless, $\mathrm{sel}(r)$ is probably small. Thus, a regex with small selectivity need not contain any useful sub-grams. Therefore, if an index contains useful grams only, the system may not show any performance increase at all, compared to the case when the system scans the entire corpus sequentially. □

While the user may indeed issue the types of queries described above, we feel that such instances are rare and pathological. Therefore, we chose to keep only useful grams in our multigram index. (As our later experiment shows, we can still get quite significant speed-up for many interesting regular expressions, even if we keep useful grams only.) This choice results in two potential benefits: First, the number of indexed grams is decreased. Second, the eliminated grams are exactly those with large postings lists, thus the size of the index would decrease even more dramatically.

Even if we index only useful grams, we still have a large number of useful grams: Any extension of a useful gram is useful by definition. For example, if the gram **`DELLPC`** is useful within a text "**`DELLPC is great`**", then any extension of **`DELLPC`** (e.g., **`DELLPC i`**, **`DELLPC is gre`**, etc.) will be a useful gram. For this reason, we propose to maintain only the *minimal* useful grams in the index. In the above example, only the gram **`DELLPC`** will be indexed, because it is minimal (or the shortest one).

When we index only the minimal useful grams, we can prove that the size of the index does not exceed the size of the original text corpus. To explain this fact, we introduce the notion of a prefix-free set.

Definition 3.6 (Prefix free sets) A set of grams $X \subseteq \mathcal{M}$ is prefix free if no $x \in X$ is a prefix of any other $x' \in X$. □

Example 3.7 The set $X_1 = \{$**`ab`**$,$ **`ac`**$,$ **`abc`**$\}$ is not prefix free because **`ab`** is a prefix of **`abc`**. □

It is easy to see that the set of minimal useful grams is prefix free: Otherwise, we would have two grams x, x' such that x' is a prefix of x, contradicting the minimality of x. If the keys for an index are minimally useful (thus prefix free), we can show that the total size of the postings lists of the index does not exceed the original corpus size.

Observation 3.8 *Let $D = \{T_1, T_2, \cdots T_N\}$ be a finite collection of finite data units. Let $|T_i|$ denote the length of T_i (in characters). Moreover, let $|D| = \sum_i |T_i|$. $|D|$ represents the total size of the corpus. Let X be any prefix-free set of grams extracted from D. Then,*

$$\sum_{x \in X} M(x) \leq |D|$$

Proof Assume the contrary. Then (by the pigeon hole principle) there is a data unit T_i such that more than $|T_i|$ grams in X occur in it. Therefore, there must be two different grams x and x', both members of X such that they occur at the same position in T_i. The shorter of x, x' must be a prefix of the longer. This contradicts our assumption that X is prefix free. ■

From the observation, we know that we can construct a space efficient gram index by indexing only the minimal useful grams: The total size of the postings lists (and therefore, the size of the index) can never exceed the total size of the dataset (in characters). In practice, we find that the actual index size that we build is much smaller than the corpus size, as we report in Section 5.

Finding the minimal useful grams from a large text corpus is somewhat akin (both in spirit and formally) to data mining. In data mining, one wishes to compute only the *maximal* frequent sets, and here we want to find the *minimal* useful grams. The algorithm in Figure 4 makes this connection explicit, which finds the minimal useful grams for a dataset. The algorithm is derived from the *a priori* frequent set mining. Essentially, it attempts to find *maximal* useless grams by following the standard *a priori* recipe, generating grams in increasing order of length in an iterative loop. Within each iteration (say iteration k), the algorithm extends all useless grams of length k to grams of length $k + 1$. It then evaluates which of the new set of grams are useful and which are useless. The useless ones are expanded further. The useful ones are minimal useful grams because their prefixes are all useless. These observations are formalized in theorem 3.9 below.

```
Algorithm 3.1   Multigram index
Input:    database
Output:   index: multigram index
Procedure
  [1] $k$ = 1, expand = $\{\cdot\}$   // $\cdot$ is a zero-length string
  [2] While (expand is not empty)
  [3]     k-grams := all $k$-grams in database
                  whose ($k$-1)-prefix $\in$ expand
  [4]     expand := $\{\}$
  [5]     For each gram $x$ in k-grams
  [6]         If sel($x$) $\leq c$ Then   // check selectivity
  [7]             insert($x$, index)   // the gram is useful
  [8]         Else
  [9]             expand := expand $\cup$ $\{x\}$
  [10]    $k$ := $k$ + 1
```

Figure 4. Construction of a multigram index

Theorem 3.9 *Let X be the set of grams indexed by algorithm 3.1.*

1. *If $x \in X$, then x is useful.*
2. *Conversely, if x is useful, then either $x \in X$ or there is a unique prefix x' of x such that $x' \in X$.*
3. *X is a prefix free set.* □

Proof (1) follows trivially from the algorithm, all the grams indexed in step [7] are useful as per step [6].

(2) (sketch) By induction on k. Let the statement hold for all grams of length less than k. Now consider the k-gram x. Let x'' be the $k-1$-prefix of x. If x'' is useful, then the claim follows from the inductive hypothesis. Otherwise, x'' is in expand, and thus, x should be picked in step [6] in next iteration.

(3) Assume that both x and x' are in X and that x' is a prefix of x. Clearly, x' must have been picked up in step [7]. Then x' cannot be included in the set expand in step [9], so any grams that has x' as its prefix cannot be included in X. ■

The algorithm in Figure 4 is intended to describe the key concepts, not the most efficient implementation of index construction. Several optimizations are possible in implementation. For example, in the first iteration of the algorithm, we may find useless grams for both $k = 1$ and 2, not just for $k = 1$. That is, we keep counters for all grams of length 1 and 2 in the first iteration and estimate their selectivity in one pass. Similarly, we can apply other optimizations for frequent-set mining to our context.

3.2 Extension: shortest common suffix rule and presuf shell

Consider any useful gram $x \in \mathcal{M}$. Any gram which can be derived by adding something to the front of x would also be considered useful. Thus, the gram ax would be useful for any $a \in \Sigma$. We illustrate this point using the following example:

Example 3.10 A multigram index could contain all the keys **`<a href="k`**, **`a href="k`**, **`href="k`**, …, **`="k`** if the gram **`="k`** is useful. However, the grams **`<a href="k`**, **`a href="k`**, …(except the last **`="k`**) are not very useful *in practice*, because the string **`<a href="`** is very common on the web and does not have any discriminating power. The discriminating power essentially comes from the last character **`k`**. Therefore, instead of keeping all of the above grams, if we keep only the last one **`="k`**, we may reduce the size of the index significantly while performance remains about the same. □

We generalize this idea by introducing the notion of a *suffix-free set*.

Definition 3.11 A *suffix-free set* $X \subseteq \mathcal{M}$ contains no two strings x and x' such that x is a suffix of x'. □

Definition 3.12 A *presuf-free set* is both prefix free and suffix free. We say that Y is a *presuf shell* of a prefix-free set X if

1. for every $x \in X$, either $x \in Y$, or $\exists y \in Y$ such that y is a suffix of x.
2. Y is suffix free.
3. $Y \subseteq X$. □

Observation 3.13 *The presuf shell of any prefix-free set X is unique and can be computed in time $O(|X| \log |X|)$. Here, $|X|$ denotes $\sum_{x \in X} |x|$.* □

Proof The idea is to reverse the strings in X and then sort them in lexicographic order. The presuf shell can be read off this sorted list. We leave the details to the reader. ■

Observation 3.14 *The presuf free shell of the prefix free set X identified by the algorithm in Figure 4 contains at least one substring of* every *useful gram.* □

The above observation indicates that we may significantly reduce the index size without hurting performance, if we compute the presuf shell of the grams (identified by the algorithm of Figure 4) and index only those grams. We will study the impact of this technique in Section 5.5.

4 Runtime

This section describes the runtime system. We focus on the plan generation engine of the runtime system (Figure 3)

```
Algorithm 4.1   Logical index access plan
Input:    a regular expression $r$
Output:   logical index access plan
Procedure
    // Generate logical access plan
    [1] Rewrite regular expression $r$ so that it only
        uses OR (|) and STAR (*) connectives
    [2] Construct a parse tree based on rewritten $r$
    [3] Replace * node with NULL
    [4] Remove NULL nodes using Table 2
```

Figure 5. Algorithm for the generation of a logical access plan

4.1 Plan Generation

Given a regex, we need to determine which index entries to look up. A simple approach would 1) identify all the grams used in the regex 2) look up grams from the index 3) combine the resulting postings lists appropriately and 4) read the identified candidate data units to find matching strings. For instance, consider the following running example for this section:

Example 4.1 Let r = **`(Bill|William).*Clinton`**. The multigrams contained in the regex are **`Bill`**, **`William`**, and **`Clinton`**. The regex r corresponds to the Boolean formula

`(Bill` OR **`William)`** AND **`Clinton`**

However, not all grams (**`Bill`**, **`William`** and **`Clinton`**) in the regex may be in the index. Thus, the access plan has to be adjusted based on the availability of the grams. In the following subsections, we describe how we can obtain the above Boolean formula from a regex and how we can adjust it based on index availability.

Note that the algorithm that we present in this section is just *one* way of obtaining a physical execution plan. Clearly, many optimizations can be done to obtain the most efficient plan given an index. We defer the study of such optimizations to future work.

4.2 Generation of logical access plan

In Figure 5, we show the algorithm that can generate a *logical* index access plan given a regular expression.

First, we note that any regular expression can be rewritten, so that it only uses string characters, OR connectives (|) and star symbols (*) (and possibly parentheses for precedence). For example, the regular expression **`[0-9]`** can be rewritten as **`0|1|...|9`**, and **`C+`** is equivalent to **`CC*`**. In case of our running example, it may be rewritten as **`(Bill|William)(a|b|c)*Clinton`** (To make our discussion simple, we assume the dot corresponds only to **`a`**, **`b`**, and **`c`**. In practice, the dot should be expanded to the set of all characters.) In Step [1] of Figure 5, we perform this rewriting step.

		Right child	
		Regular	NULL
Left child	Regular	–	Left
	NULL	Right	NULL

(a) AND node

		Right child	
		Regular	NULL
Left child	Regular	–	NULL
	NULL	NULL	NULL

(b) OR node

Table 2. Simplification of NULL nodes

In Step [2], we then construct a parse tree, such as the one shown in Figure 6(a), based on the rewritten regular expression. In the parse tree, the leaf nodes correspond to the grams within a regular expression, and the internal nodes correspond to Boolean connectives.

From the parse tree, we identify grams that can be looked up from the index. In particular, we note that when a gram is adorned with *, the string *may* or *may not* appear in the matching string, so we cannot use the adorned gram for index lookup. We indicate this fact by replacing a branch with * with a NULL node (Step [3]). Essentially, NULL means that we cannot use any grams below the node to reduce candidate data units. Any data unit satisfies a NULL node. The parse tree after this step is shown in Figure 6(b).

We now describe how we deal with NULL nodes. Logically, we may consider NULL as Boolean value "TRUE": A NULL node can be satisfied by every data unit. Therefore, if the parent of a NULL node is an AND node, the AND node can be replaced by the other child of the node (x AND TRUE is x). Similarly, if a NULL node is connected by an OR node, the OR node should be replaced by a NULL node (x OR TRUE is TRUE). We summarize this transformation rule in Table 2. The bar (-) in the table means that the AND or — node should remain intact.

This NULL node minimization is performed in Step [4] of Figure 5. We show the parse tree of our running example after this step in Figure 6(c).

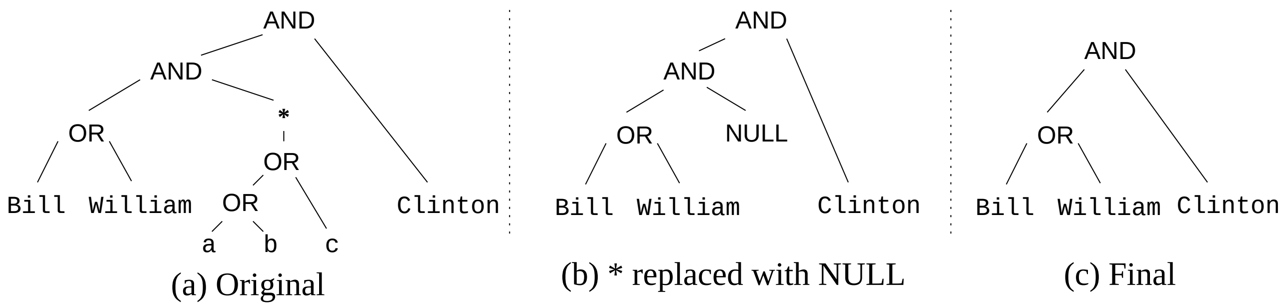


Figure 6. Parse tree for `(Bill|William).*Clinton`

4.3 Generation of physical access plan

Given the logical index access plan shown in Figure 6(c), we need to figure out exactly what index entries are available and how we should access them. Entries could be unavailable in the index for one of two reasons:

1. The corresponding gram is a useless gram.
2. The gram is useful, but not minimally useful, so it was pruned away when we construct a presuf shell of grams.

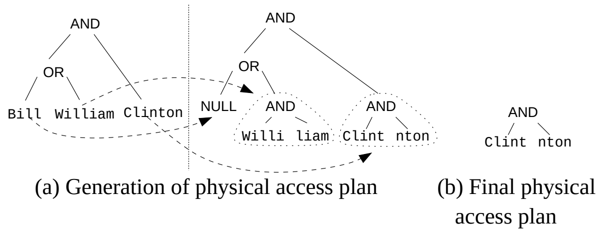
In the first case, no substring of the gram will be available in the index. In the second case, at least one (perhaps more) substring of the gram will be available in the index (see observation 3.14).

We replace nodes of the first type by NULLs. We replace nodes of the second type by logical AND of all its substrings available in the index. After this replacement, we process NULL nodes, again, using Table 2.

To return to our running example, for the node **`William`** in Figure 6(c), let us assume that the grams **`Willi`** and **`liam`** are available in the index. (We can check gram availability by looking up the directory of the index. As we will see in Section 5, the directory of a multigram index is often very small and can be loaded into main memory.) In this case, we replace the node **`William`** with logical AND of **`Willi`** and **`liam`** as is shown in Figure 7(a). Similarly, we replace the node **`Clinton`** with **`Clint`** and **`nton`** assuming the two grams are available. Finally, we replace the node **`Bill`** with NULL, assuming that **`Bill`** is a useless gram. In Figure 7(b) we show the final plan after we eliminate NULL nodes using Table 2.

5 Experiments

In this section, we report performance studies on the various ideas presented in this paper. We start by describing the experimental setup.



AND
OR
Bill
William
Clinton
AND
OR
NULL
AND
Willi
liam
AND
Clint
nton
(a) Generation of physical access plan
AND
Clint
nton
(b) Final physical access plan

Figure 7. Final physical index access plan for "`(Bill|William).*Clinton`"

5.1 Experimental setup

For our experiment, we used 700,000 random Web pages downloaded in 1999. The total size of our dataset was around 4.5GB. While our system can handle larger corpora, we restricted ourselves to a small subset to make our experiments manageable. We ran most of our experiments on a standard Linux machine, with a Pentium III processor (450MHz), 256 MB RAM and Ultra-Wide SCSI Bus (Red Hat 6.2).

Benchmark queries For our experiments, it is important to select a "fair" set of regular expressions, because the results will heavily depend the selection. With the lack of a standard benchmark, it is clearly difficult to make an objective judgment on the performance improvement from our techniques. Therefore, the results in this section should be interpreted as the *potential* of our techniques not as an absolute improvement.

We compiled our benchmark regular expressions from the researchers at IBM Almaden: We asked them to provide a list of regular expressions that they wish to run on a large Web dataset. We note that the researchers were not aware of any of the techniques we have proposed in this paper. Since we asked the researchers independently, many of the regular expressions they submitted were very similar or almost identical. Therefore, we manually selected only

1. MP3 URLs (mp3):
   `<a\s+href\s*=\s*("|')?[^>]*\.mp3("|')?>`
2. US city name, state and ZIP code (zip):
   `(\a+\s+)*\a+\s*,\s*\a\a\s+\d\d\d\d\d(-\d\d\d\d)?`
3. Invalid HTMLs (html): All html pages with starting tag "⟨", but with another "⟩" before the end tab "⟨"
   `<[^>]*<`
4. Middle name of President Clinton (clinton):
   `william\s+[a-z]+\s+clinton`
5. Motorola PowerPC chip numbers (powerpc): Motorola PowerPC chip part numbers starts with XPC or MPC followed by digits.
   `motorola.*(xpc|mpc)[0-9]+[0-9a-z]*`
6. HTML scripts on web pages (script):
   `<script>.*</script>`
7. US phone numbers (phone):
   `((\d\d\d)|\d\d\d-)\d\d\d-\d\d\d\d`
8. SIGMOD papers and their locations (sigmod): The URLs ending with .ps or .pdf and with the word SIGMOD within 200 characters from the URL.
   `<a\s+href\s*=\s*("|')?[^>]*(\.ps|\.pdf)("|')?>.{0,200}sigmod`
9. Stanford email addresses (stanford):
   `(\a|\d|_|-|\.)+((\a|\d)+\.)*stanford.edu`
10. Pages pointing to deep links of eBay (ebay):
    `<a\s+href\s*=\s*("|')?http://(www\.)?ebay\.com/[^/]+/[^>]*("|')?>`

Figure 8. Ten regular expressions used for our experiments

10 regular expressions from the compiled list, which are 1) popular and 2) semantically interesting. We show the complete list of our selection in Figure 8.

Measurements In this section, we report the following set of results.

1. *Index size and construction time:* We describe the benefits of a multigram index and a presuf-shell-based index on size and construction time. We report significant reductions in both index size and construction time. For example, the size of presuf-shell-based index is only 5% of a full n-gram index, and it took less than 10% of time to construct.
2. *Overall matching time:* We report on runtime performance on our benchmark queries. We report significant performance improvements. In some cases, where the regex is fairly rare, we got performance improvement of more than a factor of 100.

	Complete	*Multigram*	*Suffix*
Construction time	63 h	8 h 23 min	6 h 10 min
Number of gram-keys	103,151,302	988,627	64,656
Number of postings	18,193,048,399	1,744,677,072	820,396,717

Table 3. The size of various gram indexes

3. *Response time for first 10 answers:* We look at how long it takes to output the first 10 matches. This measurement shows the potential of our techniques in an interactive environment and it also measures the overhead involved in the plan-generation phase. The results in this case largely mirror the earlier case.
4. *Effect of the shortest suffix rule:* Does restricting the index to a presuf shell result in reduced performance? We report experiments showing very little impact in most cases.

5.2 Index construction

To measure the space and the performance implications of our proposed techniques, we constructed the following three indexes:

1. **Complete:** Nine n-gram indexes for $n = 2, 3, \ldots, 10$
2. **Multigram:** a plain multigram index (without the shortest suffix rule). We cut off useful grams at length 10. That is, we only indexed useful grams when their lengths are 10 or shorter.
3. **Suffix:** a multigram index based on a presuf shell. We also cut off useful grams at length 10.

In case of Multigram and Suffix indexes, we need to pick the value for *usefulness threshold* c, which was 0.1 in our experiments. In this paper, we do not attempt to optimize this threshold value. We defer the study of this issue to future work.

Note that the result from the first index (complete) can be considered as an "optimal" case, because we can look up any substring in a regex using the index. Also, as a worst case scenario we report run time results when we scanned the entire dataset for regex matching. By comparing our techniques to these two cases, we can see how well our techniques perform for the benchmark queries.

In Table 3, we report the construction time and the size of the indexes. To build the multigram (or suffix) index, the entire data was scanned 5 times. During the first four scans, the system identified the gram-keys that should be indexed in the multigram (or suffix) index. This gram-key identification could be done in less than 10 scans, because we identified useful grams of multiple lengths in one scan, as we described in Section 3.1. After this identification, the system generated postings lists in the final scan, using the identified keys. Roughly, the first four scans took around 4 hours and the remaining time was spent to 1) *generate* postings lists 2) *sort* the gram keys and postings list, and 3) actually construct the index.

Note that the total construction time of the suffix index was smaller than that of the multigram index. This is because the suffix index had smaller postings lists, so it took much less time for sorting and actual index construction. Also note that it takes much longer to build the complete index than the multigram (or suffix) index. While we scan the entire data only once to build the complete index, the sorting and the actual construction took much longer due to its large size.

In the third row, we show the number of unique gram keys that each index has entry for, and in the fourth row we show the total number of postings that are associated with the keys. For example, the plain multigram index (Multigram, 3rd column) has 988,627 unique keys and 1,744,677,072 postings associated with them. Note that the number of postings are significantly larger than the number of gram keys. For any index, the number of postings is more than 100 times larger than that of gram keys: The index size is primarily determined by the number of postings not by the number of gram keys.

From the table, we can clearly see that the multigram index reduces index size very significantly. For example, the posting size of the plain multigram index (Multigram, 3rd column) is 10 times smaller than that of the complete n-gram indexes. This result is mainly because we built 9 n-gram indexes ($n = 2, 3, \ldots, 10$) for the complete case. Depending on how many n-gram indexes we build, the savings in the index size can be much larger. Also note that a presuf-shell-based index reduces size even further. The number of postings decreases by a factor of 2 compared to the plain multigram index.

The reduction in the number of gram keys is even more dramatic. Compared to the complete gram index, the multigram index has less than 1% gram keys and the presuf-shell-based index has less than 0.06% gram keys. While this smaller gram key size does not reduce the index size much (because the index size is dominated by the number of postings), it may have a significant impact on the run time performance: Since the multigram index has a small number of gram keys, the entire gram keys can be loaded into the main memory, and we need to perform disk IOs only when we read postings lists.

5.3 Total execution time

In Figure 9 we report the total execution time for regex matching. In the graph, "Scan" means that we scanned the entire dataset to match regexes, and "Multigram" and "Complete" means that we used the multigram and the complete gram indexes, respectively. For the multigram index, we used a plain multigram index without the shortest suffix rule. (We study the performance implication of the shortest suffix rule in Section 5.5.)

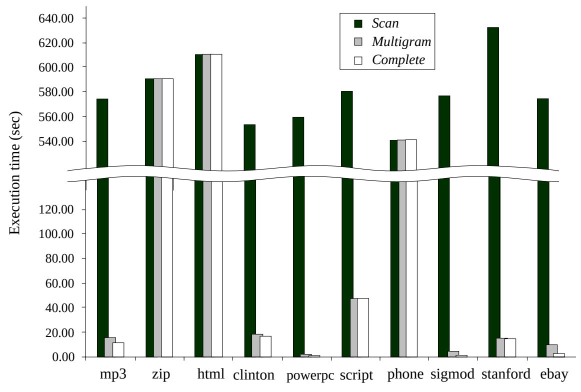


Figure 9. Total execution time

From the graph, we can clearly see that indexing techniques reduce execution time very significantly. For most regular expressions, the execution time of "Multigram" and "Complete" is shorter than "Scan" by orders of magnitude. Only for 3 regular expressions (zip, phone, html), "Scan" shows comparable performance to others: For these three regular expressions, there is no gram key entry to look up from the index. However, even for these regular expressions, we emphasize that indexing techniques do not degrade performance. The results of "Multigram" and "Complete" are almost identical to those of "Scan," because index lookup takes negligible time compared to the entire scanning. On average, the multigram index reduces the total matching time by a factor of 16 compared to raw scanning.

Clearly, the improvement from indexing techniques depends on result size. When there are many matching strings, we get relatively small improvement because we need to read many pages to confirm final matching strings. In Figure 10, we show performance gains over result size. The vertical axis shows the improvement of the multigram index over raw scanning (Execution time of Multigram/Execution time of Scan) and the horizontal axis shows result size (number of matching strings). We can clearly see that we get more improvement as result size gets smaller. In the best case (powerpc), we get about 300 times increase in performance.

Finally, note that the multigram index shows comparable performance to that of the complete gram index. In most of Figure 9, the multigram index shows similar execution time to the complete index. On average, the complete index takes only 32% less time for regex matching than the multigram index. Given this result, we believe that a multigram index provides good space-performance tradeoff for regex matching: It reduces index size by a factor of 10, while performance is degraded by only 32%.

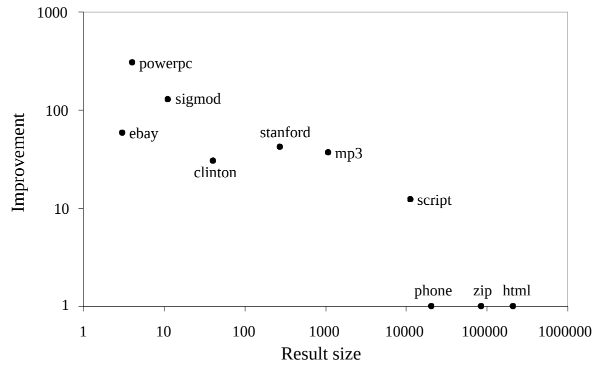


Figure 10. Result size versus improvement

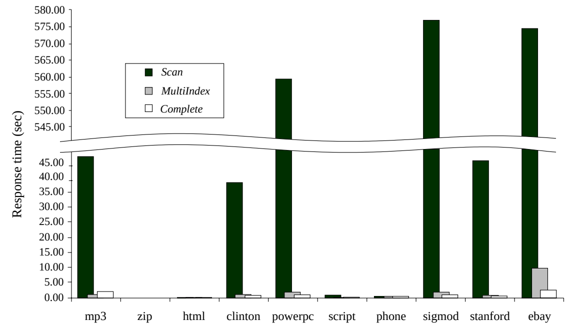


Figure 11. Response time for first 10 results

5.4 Response time

In an interactive environment, it is more important to minimize the response time for the first k results, not for the entire set of results. In Figure 11, we report the time to match the first 10 results. We can see that multigram and complete indexes significantly improve response time. Raw scanning shows heavy fluctuations in response time: Depending on result size, it sometimes takes more than 500 seconds to produce the first 10 answers. The worst cases (sigmod, ebay) happen when the number of matching strings is small. This is because raw scanning has to examine a lot of pages to find matches. In contrast, the complete and multigram indexes consistently take less than 10 seconds. On average, the multigram index shows 20 folds reduction in response time compared to raw scanning. Compared to the complete index, it shows less than 22% performance degradation.

5.5 Shortest suffix rule

So far we have compared the performance of a plain multigram index with that of raw scanning and a complete gram index. In this section, we study how the shortest suffix rule affects run time performance. In Figure 12 we report the total execution time of the plain multigram index (*Plain*) and the presuf-shell index (*Suffix*). From the graph,

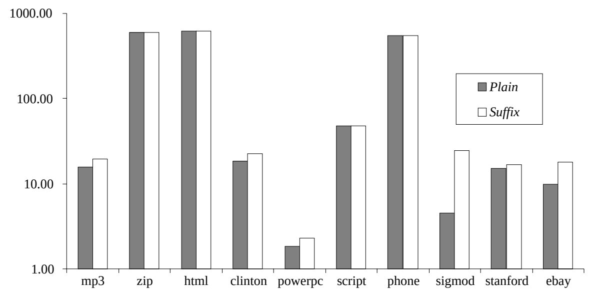


Figure 12. The effect of shortest suffix rule

we can see that the suffix rule shows comparable results in most cases. Only for one regular expression, sigmod, the suffix rule shows relatively large degradation. Given that the shortest suffix rule reduces index size by half and shows comparable performance, we believe that it is a good option for index construction.

6 Conclusion

In this paper, we proposed to use pre-built indexes to speed up regular-expression matching on a large text database. We argued that the issues in this context are very similar to the ones in classical RDB systems. We showed that a careful and savvy use of indexes can result in speedups of two orders of magnitude in some instances. In particular, our multigram index reduces index size by an order of magnitude in certain cases without much degradation in performance.

References

[1] Brad Adelberg. Nodose: A tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1998.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, May 1993.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[4] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, USA, 1988.

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

[6] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[7] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[8] Sergey Brin. Extracting patterns and relations from the world wide web. In *WebDB Workshop at 6th International Conference on Extending Database Technology, EDBT'98*, 1998. Available at http://www-db.stanford.edu/ sergey/extract.ps.

[9] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[10] Junghoo Cho and Sridhar Rajagopalan. A fast regular expression indexing engine. Technical report, UCLA Computer Science Department, 2001.

[11] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.

[12] Inktomi Corp. The web exceeds 1 billion documents. Jan 18., 2000.

[13] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the International Conference on Information Retrieval (SIGIR)*, pages 405–411, January 1990.

[14] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):50–74, March 1985.

[15] Joachim Hammer, Hector Garcia-Molina, Junghoo Cho, Arturo Crespo, and Rohan Aranha. Extracting semistructured information from the web. In *Proceedings of Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[16] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 1–12, 2000.

[17] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[18] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, Sebastopol, 2 edition, 1992.

[19] Udi Manber and Gene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[20] Udi Manber and Sun Wu. Glimpse: a tool to search through entire file systems. In *Proceedings of the USENIX Winter Conference*, pages 23–32, January 1994.

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

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

[23] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 175–186, May 1995.

[24] Gerard Salton. *Automatic Text Processing*. Addison-Wesley, 1989.

[25] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[26] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 8–17, January 1993.

[27] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.