Selectivity Estimation for Extraction Operators over Text Data

Daisy Zhe Wang
Long Wei
Yunyao Li
Frederick Reiss
Shivakumar Vaithyanathan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-107
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-107.html

July 2, 2010

Selectivity Estimation for Extraction Operators over Text Data

Daisy Zhe Wang*, Long Wei*, Yunyao Li†, Frederick Reiss†, and Shivakumar Vaithyanathan†
*University of California, Berkeley and †IBM Almaden Research Center

ABSTRACT

Recently, there has been increasing interest in extending relational query processing to efficiently support extraction operators, such as dictionaries and regular expressions, over text data. Many text processing queries are sophisticated in that they involve multiple extraction and join operators, resulting in many possible query plans. However, there has been little research on building the selectivity or cost estimation for these extraction operators, which is crucial for an optimizer to pick a good query plan. In this paper, we define the problem of selectivity estimation for dictionaries and regular expressions, and propose to develop document synopses over a text corpus, from which the selectivity can be estimated. We develop three classes of document synopses: n-gram synopsis, bloom filter synopsis and roll-up synopsis. We also develop techniques to decompose a complicated regular expression into subparts to achieve more effective and accurate estimation. We conduct experiments over the Enron email corpus using both real-world and synthetic workloads to compare the accuracy of the selectivity estimation over different classes and variations of synopses. The results show that, the top-k stratified bloom filter synopsis and the roll-up synopsis is the most accurate in dictionary and regular expression selectivity estimation respectively.

1. INTRODUCTION

The field of database management has traditionally focused on structured data, providing little or no help for the significantly larger amounts of the world's data that is unstructured. With the rise of text-based applications on the web and elsewhere, information extraction (IE) techniques and database with SQL extensions are used over text to find patterns and extract objects as part of the query or process.

Most of these SQL queries and IE processes over text involves extraction operators, including dictionaries and regular expressions. Dictionaries are used to look for specific entity memberships (e.g., countries, firstnames), and regular expressions (regex) are used to find specific text patterns (e.g., emails, URLs). Many of those SQL queries and IE processes involves multiple dictionaries and regexes, where their selectivity estimation is crucial for an optimizer. This can be illustrated by the following scenario:

EXAMPLE 1. *Suppose we have a Blogs table, containing the id, text and author of all the blog posts. We also have an Author table, containing name, address information of each author of the blog posts. Query 1 is a SQL query extended with dictionary extraction operator* `contains(text,dictionary)` *and regular expression operator* `text like 'pattern'`. *Query 2 is an algebraic expression of an IE process with $\mathscr{E}_{dict}$ and $\mathscr{E}_{re}$ denote dictionary and regex extraction operator respectively. In addition, $\bowtie_{FollowsTok(0,0)}$ denotes that the left extraction should be immediately followed by the right extraction in text.*

Query 1. Find all blog posts mentioning *"Euphonia"*, a folk ensemble, which also contains sentiment words (as defined by a list of sentiment words) and are written by people living in *Seattle*.

```
SELECT *
FROM Blogs B, Author A
WHERE B.author=A.name and B.text like '%Euphonia%'
      and contains(B.text,sentiment.dict)
      and A.address like '%Seattle%'
```

Query 2. Find all instances of a common first name (as defined by an exhaustive list of first names), followed immediately by a capitalized word.

$$\mathscr{E}_{dict}(first.dict) \bowtie_{FollowsTok(0,0)} \mathscr{E}_{re}([A-Z]\backslash w+)$$

To execute the first SQL query, the optimizer needs to decide in which order to apply the three selection conditions involving extraction operators. In addition, since dictionary and regex are expensive operators and in most cases do not have more efficient access method than sequential scan, the optimizer also needs to decide when to execute the join operator to reduce the extraction. These decisions all depend on the selectivity estimation of the dictionary, regex and join operators.

Similarly, to execute the IE process in Query 2, the optimizer needs selectivity estimation of the dictionary and regular expression operators to form a query plan to execute the more selective extraction operator first. This plan maximally reduces the extraction, because the two extraction operators joins on the same document. □

The goal of this paper is to study the selectivity estimation algorithms and techniques for dictionaries and regular expressions operators over text. As illustrated by the previous example, the selectivity estimation for extraction operators is crucial in optimizing (1) SQL queries over text data in a DBMS, and (2) information extraction processes in declarative-IE systems like System T [14].

There is little previous literature studying the selectivity estimation of dictionary and regex operators, as oppose to the vast number

VLDB '10, September 13-17, 2010, Singapore

covering the selectivity estimation of SPJ queries [4, 8, 15]. Studies over selectivity estimation for string predicates [3, 9, 10, 11, 12] focus on database entries with string values of limited length and thus are not directly applicable for dictionary selectivity estimation, where the underlying text can be of arbitrary length. The same body of literature and those for XML path expressions [1, 18] consider only simple patterns in the form of "wild cards" or path expressions and therefore are not applicable for regex selectivity estimation, which needs to support arbitrarily complex regexes.

Our main approach is to develop synopsis over a text corpus, from which the selectivity of dictionary and regex operators can be estimated. This is analogous to develop histograms over a column, from which the selectivity of selection conditions can be estimated.

We develop three classes of document synopses. Firstly, we develop variations of n-gram synopsis, which is a subset of {n-gram, count} pairs computed from the text corpus. The selectivity of both dictionary and regex operators can be estimated from this class of synopsis, among which the top-k n-gram synopsis is the most simple and accurate one. Secondly, we develop variations of Bloom filter synopsis, which stores n-grams in bit-maps by applying hash functions. The Bloom filter synopsis can only support dictionary selectivity estimation. Experiment results show that, given the same size budget, the best Bloom filter synopsis, top-k stratified Bloom filter synopsis, halves the error rate of the top-k n-gram synopsis.

Thirdly, we develop a novel technique to summarize a set of n-grams by rolling-up characters into character classes, resulting in roll-up synopsis, which is only used for regex selectivity estimation. Because it summarizes rather than drops n-grams to reduce the size of the synopsis, it achieves better accuracy compare to the top-k n-gram synopsis of the same size. We also describe the procedure to decompose a complex regex into a set of simpler sub-regexes, whose selectivities can used to more accurately estimate the selectivity of the original regex.

Our key contributions can be summarized as follows:

- we formalize the problem of selectivity estimation for dictionary and regex extraction operators over text data in database;
- we developed variations of three classes of document synopses – n-grams synopsis, bloom filter synopsis, and roll-up synopsis for the selectivity estimation of dictionary and regex operators;
- we developed more effective and accurate techniques to estimate the selectivity of a complex regular expression by decomposing it into sub-regexes;
- we conduct experiments over the Enron email corpus [5] using both real-world and synthetic workloads, comparing the accuracy of dictionary and regex selectivity estimation over different classes and variations of synopsis.

2. RELATED WORK

Selectivity estimation is a well-known technique for cost-based query optimization. The selectivity estimation techniques have been well developed for queries involving numerical attributes [4, 8, 15]. There are more recent studies over selectivity estimation for string predicates [3, 9, 10, 11, 12] with a focus on substring or fuzzy string matching. On the surface, selectivity estimation for dictionary evaluation may be thought of as disjunction of multiple string predicates. However, unlike matching for string predicates, matching for a dictionary entry often requires ignoring the differences in white space or cases. In addition, the underlying data for dictionary matching can be text documents with arbitrary length, while previous techniques focus on database entries with string value of limited length. Therefore, previous selectivity estimation techniques for string predicates are not directly applicable for selectivity estimation for dictionaries. Shen *et al* describe an inverted-index based approach for selectivity estimation for dictionaries [16] as part of its optimization solution for information extraction. However, no formal study is presented on the accuracy of this technique.

These studies on selectivity estimation techniques for string predicates often supports simple patterns in the form of "Wildcard" (e.g. *M*ovie.name like %p%) as well. However, the regular expressions supported by our work are typically in much more complex form. Similarly, patterns considered by selectivity estimation techniques for XML path expressions [1, 18] are much simpler than those considered in our work.

Language models [13] are widely used in communities of natural language processing and information retrieval. Language models try to capture the properties of a typical document in a collection, which compensate for the unseen words and phrases. On the contrary, document synopsis we developed only try to summarize the document corpus as succinctly as possible for accurate selectivity estimation. Different purposes leads to different techniques, but we based our n-gram synopsis on the n-gram language model. Recent work [17] demonstrates how to use bloom filter [2], to replace the n-gram language models. Based on this, we developed the stratified bloom filter synopsis which drastically outperforms the data structure proposed in [17].

3. SETUP

In this section, we will first introduce dictionary and regular expression operator as the most common span extraction operators over text. Then we will setup the problem statement of this paper. Finally, we will briefly go over some basic concepts and techniques that we base our techniques on in later sections.

3.1 Span Extraction Operators

A span extraction operator identifies segments of text that match a particular input pattern and produces spans corresponding to each such text segment. There are two types of extraction operators: *Dictionary* and *Regular Expression*.

DEFINITION 1 (DICTIONARY OPERATOR). *A dictionary operator $\mathscr{E}_{dict}$ evaluates a dictionary of words and phrases. For each document, $\mathscr{E}_{dict}$ returns spans that correspond to matches of the dictionary words or phrases in the document.* □

DEFINITION 2 (REGULAR EXPRESSION OPERATOR). *A regular expression operator $\mathscr{E}_{re}$ evaluates a character-based regular expression. For each document, $\mathscr{E}_{re}$ returns spans that correspond to matches of the regular expression in the document.* □

3.2 Problem Definition

We focus on the following two problems in this paper: (1) building effective document synopsis, and (2) estimating selectivity of $\mathscr{E}_{dict}$ and $\mathscr{E}_{re}$ over the document synopsis. We now describe each problem in detail.

Document Synopsis: The first problem is how to construct a document synopsis for selectivity estimations of extraction operators.

DEFINITION 3 (DOCUMENT SYNOPSIS). *A document synopsis ϕ is a summary of a document corpus* D*, where the size of the synopsis is usually much smaller than the size of the document corpus.* □

A document synopsis ϕ that is built for selectivity estimation of dictionary $\mathscr{E}_{dict}$ and regular expression $\mathscr{E}_{re}$, best summarizes the document corpus within the space constraint, to be able to estimate the number of matches for such extraction operators.

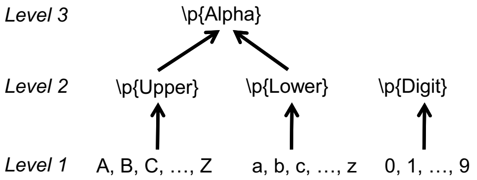

Figure 1: Sample Lattice for characters and character classes.

Selectivity Estimation: The second problem is given a particular type of synopsis ϕ, how to accurately estimate the selectivity of extraction operators $\mathscr{E}_{dict}$ and $\mathscr{E}_{re}$. The *selectivity* of an extraction operator is the expected number of matches of the operator over the document corpus.

DEFINITION 4 (SELECTIVITY). *Let $\mathscr{E}$ be an annotation rule that only involves one extraction operator. The* selectivity *of $\mathscr{E}$ is denoted as $sel(\mathscr{E})$, where*

$$sel(\mathscr{E}) \equiv \mathbb{E}[\,match(\mathscr{E}, d \in \mathsf{D})]$$

with $match(\mathscr{E}, d)$ being the number of matches of $\mathscr{E}$ over a document d. □

3.3 Bloom Filter

The Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. Its results might contain false positives, but not false negatives.

Each Bloom filter contains a bit array of m bits, and k different hash functions, each of which maps or hashes an element to one of the m array positions with a uniform random distribution. To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.

To query for an element (test whether it is in the set), we feed it to each of the k hash functions to get k array positions. If any of the bits at these positions are 0, the element is not in the set, because if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements. The false positive rate of a Bloom filter is $(1-e^{-kn/m})^k$, where n is the number of elements inserted. Solving for the value of k that minimizes this false positive rate yields an overall false positive rate of approximately $(0.6185)^{\frac{m}{n}}$ [6]. For compactness, we use this approximate value in our equations throughout the paper.

3.4 Character Class Lattice

We introduce the term *regex-ngram*, which will be used in Section 6. As oppose to n-gram and regular expressions, regex-ngram is a special subset of regexes that serves as a summary of a set of n-grams. The regex-ngrams are what the roll-up synopsis (Section 6.3) consists of. The regex-ngrams are generated by *rolling-up* the *lattice of character classes*.

Figure 1 shows a sample lattice of character classes. The lattice contains three levels. The first level contains the uppercase characters, lowercase characters and digits. The second level contains character classes `\p{Upper}`, `\p{Lower}`, and `\p{Digit}`, and the containment relationship between the characters in level one and the character classes in level two is shown in the figure. Level three contains one character class `\p{Alpha}`, which contains `\p{Upper}` and `\p{Lower}`.

A roll-up operation replaces a lower level character or character class with a higher level character class. For example, replacing an "a" with a `\p{Lower}` is a roll-up operation.

4. N-GRAM SYNOPSIS

In this and the following two sections, we describe three different kinds of document synopses ϕ over a document corpus D, including *n-gram synopsis*, *bloom filter synopsis*, and *roll-up synopsis*.

Assume a document corpus D has M documents, and each document d consists of a sequence of $|d|$ tokens: $\{g_1, ..., g_{|d|}\}$. In order to keep a synopsis ϕ compact to fit in memory, we look at the problem of building document synopsis ϕ, which do not contain more than k bytes.

For each synopsis ϕ we describe the following algorithms:

1. ϕ.**BUILD**(D, **k**): the algorithm to construct the synopsis from a document corpus D, where k is the maximum size of the ϕ in bytes;

2. ϕ.**ESTCOUNT**(ngram) / ϕ.**ESTCOUNT**(re): the algorithm to estimate the selectivity of an n-gram ngram or regular expression re over the corpus D from the synopsis ϕ.

The selectivity of a dictionary over D can be estimated by iteratively calling ϕ.ESTCOUNT(ngram) over every entry ngram in the dictionary, and adding up all the counts.

4.1 Top-k N-gram Synopsis

The *top-k n-gram synopsis* of a document corpus D is an n-gram-to-count map, which consists of the set of n-grams in D with highest counts. The construction algorithm for top-k n-gram synopsis ϕ_{topk}.**BUILD**(D, **K**) is as follows:

- Find all unique n-grams and their counts in corpus D;
- Sort n-grams by counts;
- iteratively pick the n-gram with the highest count, and insert the {n-gram,count} pair into ϕ_{topk}, until the total size of the synopsis ϕ_{topk} exceeds k bytes.

For n-grams in the synopsis, ϕ_{topk}.**ESTCOUNT**(ngram) returns the count correspond to ngram in the synopsis ϕ_{topk}. For n-grams that are not included in the synopsis ϕ_{topk}, there are different ways to estimate the count of ngram as described in Appendix 10.1. We experimented with them, and decide to use the "LeftBackOff" estimator. The "LeftBackOff" estimator finds the longest prefix of the n-gram included in synopsis ϕ_{topk}, if the remaining postfix is also included in ϕ_{topk}, then multiply the counts of the prefix and the postfix divide by the total n-gram count. If the remaining postfix is not in ϕ_{topk}, then multiply the count of prefix with "AvgValue", the average count across all dropped n-grams, divide by the total count. If no prefix is found in ϕ_{topk}, the "LeftBackoff" uses "AvgValue".

Given a regular expression re, ϕ_{topk}.**ESTCOUNT**(re) returns the sum of counts of all the n-gram entries in ϕ_{topk} that matches re.

4.2 Other N-gram Synopses

A simple variation of the top-k n-gram synopsis is the *bag-of-words synopsis*, which consists of the set of the one-grams in D, instead of n-grams, with the highest counts. We also experimented with more sophisticated variations, including *probabilistic n-gram synopsis*, where low-count n-grams are included with probabilities proportional to their counts, and *wavelet-like n-gram synopsis*, where counts of longer n-grams are estimated from that of the shorter. However, we found that none of those synopsis outperforms top-k n-gram synopsis.

5. BLOOM FILTER SYNOPSIS

In the synopses described in the previous sections, most of the space in the synopsis goes towards storing the tokens of the n-grams. This pattern of space usage limits the number of n-grams that can be stored within a given number of bytes. In this section, we discuss an alternative class of synopsis that reduces this space pressure by avoiding storing the n-grams themselves, and instead, uses Bloom filters to represent sets of n-grams. This approach allows the synopsis to store information about every n-gram in the corpus, with the accuracy being controlled by the space allocated to the filters.

5.1 Stratified Bloom Filter Synopsis

A Stratified Bloom Filter, or SBF, is a synopsis consisting of an array of simple Bloom filters. The i^{th} Bloom filter of this array represents the set of n-grams whose total counts, if encoded as binary numbers, would have a "1" in the i^{th} bit position.

Inserting an n-gram into an SBF synopsis is a straightforward procedure: First, compute the total count of the n-gram in the document corpus, and convert this count to a binary number. Then, for each bit in the count that is set to "1", insert the n-gram into the corresponding Bloom filter.

The basic algorithm for finding the estimated count for a given n-gram is also straightforward: Start with an estimated count of zero, and look up the n-gram in each of the Bloom filters in the synopsis. If the Bloom filter for bit i contains the n-gram, then add 2^i to the estimated count.

The simplest type of SBF synopsis is what we call the *naive* Stratified Bloom Filter. A naive SBF divides the space budget for the synopsis evenly among k Bloom filters (where k is the highest bit position that is set to one in any n-gram's count), and then applies the basic insertion and estimation algorithms to these Bloom filters.

As it turns out, a naive SBF makes a very poor synopsis for estimating dictionary selectivities, generally producing much greater error rates at a given synopsis size than a top-k n-gram synopsis (refer to results of SBF.naive in Section 7.2). Please refer to Appendix 10.2 to understand the nature of the errors that a SBF introduces when estimating the selectivity of a dictionary.

5.2 Important SBF Optimizations

The naive SBF synopsis suffers from high estimation errors, which stems from Bloom filter's false positives, particularly on the Bloom filters associated with higher-order bits. We have developed three different optimizations to address these issues: (1) compensate for false positives so as to bring the expected estimation error to zero, (2) reallocate the total space budget among the different Bloom filters to reduce the variance of the estimation error by as much as possible, and (3) avoid inserting n-grams with low counts.

We cover the first two methods in this section; the third method effectively creates a hybrid of the SBF and top-k approaches, so we devote a separate section (Section 5.3) to it.

5.2.1 Compensating for false positives

To compensate the one-sided estimation error of the naive SBF, we use a corrective factor based on the false positive probability of a Bloom filter. This correction is not applied at the level of individual lookup operations, since it is not possible to tell the difference between a Bloom filter that contains the n-gram and Bloom filter that has a false positive. Instead, we compensate for false positives across a series of lookups by applying a corrective factor to the number of times each Bloom filter reported that a given n-gram was *not* present.

If one were to probe a Bloom filter with n values that were never inserted into the Bloom filter, then one would expect that the Bloom filter would report that $(1-P[FP])n$ of the values were actually present, where $P[FP]$ is Bloom filter's false positive rate. So dividing the number of times the Bloom filter returned "not present" by $1-P[FP]$ compensates for the false positive rate.

Using this correction factor requires turning the `estCount` procedure into a function that is applied across an entire dictionary file instead of a single entry. Instead of computing a separate estimated count for each dictionary entry, we track the number of times each Bloom filter reported that a given n-gram in the dictionary was *not* present. We then divide each n-gram's count by the correction factor, $1-P[FP]$, and subtract the result from the total number of n-grams to produce an estimate of the number of counts that truly contain a "1" in each bit position. Finally, we multiply these counts by the appropriate powers of two to obtain an overall estimate of the dictionary's selectivity.

This updated procedure removes the bias from the SBF's estimator, but it does not change its variance. Also, since the correction factor relies on each Bloom filter reporting that at least some of the n-grams in the corpus are *not* present, it does not work when the false positive probability is close to 1. Sections 5.2.2 and 5.3 address the issues of reducing variance and correcting very high false positive rates, respectively.

5.2.2 Allocating space across Bloom filters

Compensating for the expected error, as described in the previous section, creates an unbiased estimator, but the variance of this estimator is just as high as before. We reduce this variance by using a more effective method to allocate space across the Bloom filters.

Most of the variance in the estimation error of a naive SBF is due to false positives on the higher-order bits. Reducing this variance requires allocating more of the synopsis's space budget to the Bloom filters corresponding to these bits, reducing their false positive rates. Note that one cannot just allocate all the space to the higher-order bits; as they shrink in size, the Bloom filters for the lower-order bits would quickly become just as big a source of variance.

We allocate the space budget among Bloom filters by solving the following optimization problem: *Minimize the variance of the estimator by allocating space to different Bloom filters, subject to the constraint that the total size of all the Bloom filters must add up to the space budget.*

To compute a concrete value of the variance of this estimator, one needs to know the distribution of counts of the n-grams that will be probed against the synopsis. To be conservative, we use a "worst-case" estimate of this variance: the variance of the estimator when all n-grams probed against the synopsis have a true count of zero. Using the random variable V_i to represent the false positive rate for bit i, we can express this variance as

$$\mathrm{Var}\left[\sum_{i=1}^{k} 2^i V_i\right] \tag{1}$$

As we noted in Section 3.3, V_i is 1 with probability $0.6185^{\frac{m_i}{n_i}}$ and 0 otherwise, where m_i is the number of bits in the Bloom filter for bit i and n_i is the number of n-gram counts for which this bit is set to 1. Substituting into the above equation, we obtain the following optimization problem:

Minimize:

$$\sum_{i=1}^{k} 2^{2i} 0.6185^{\frac{2m_i}{n_i}} + 0.6185^{\frac{m_i}{n_i}} - 2^{i_1} 0.6185^{\frac{2m_i}{n_i}} \tag{2}$$

Subject to:

$$\sum_{i=1}^{k} m_i = M,\ \ m_i > 0 \text{ for all } 1 \le i \le k \tag{3}$$

where M is the total space budget.

This optimization problem is difficult to solve in principle, but we found that a simple hill-climbing algorithm that iteratively reallocates space among Bloom filters in one-byte increments works well in practice.

5.3 Top-k Stratified Bloom Filter Synopsis

The Stratified Bloom Filter synopsis as described so far stores information about all n-grams in the corpus. When the number of bits allocated to the synopsis is small relative to the total number of unique n-grams, this policy can lead to two problems. First, the variance of the estimator rises because fewer bits are available to represent the counts of each n-gram. Second, as the space budget decreases, eventually the false positive rate approaches 1.

The *Top-K* Stratified Bloom Filter addresses these two problems by only storing information in the synopsis about the n-grams with the top k counts. This approach effectively creates a hybrid of the SBF and top-k synopsis types.

Removing the low-count n-grams from the synopsis reduces the variance of the estimator, and also reduces the false positive rates for all bits. However, the resulting synopsis will likely return a count of zero for the n-grams that were removed.

Unlike the systematic error we corrected in Section 5.2.1, this *false negative* error does not have a structure that allows for a similar type of correction factor. Instead, we reduce the bias of the estimator by balancing this negative error against the portion of the false positive error that cannot be corrected. Recall that the correction factor described in Section 5.2.1 stops working as the false positive probability approaches 1.

We define the *uncorrectable* false positive error as the expected false positive error due to the Bloom filters false positive probabilities greater than c, where c is a tuneable constant. We then a hill-climbing algorithm to find the value of k (the number of n-grams to retain) such that the total false positive and false negative errors cancel each other out over the entire corpus. The algorithm starts with $k = |\mathsf{D}|$ (that is, retain all n-grams) and gradually decreases the value of k. At each stage of the search, the algorithm uses the technique described in the previous section to allocate the space budget among Bloom filters, and computes expected false negative and uncorrectable false positive errors. The search stops when these errors cancel each other out.

6. REGEX SYNOPSIS

A straightforward way to estimate the selectivity of a regex is to match the regex over a sample document collection. This approach can suffer from the following two problems: first, due to the limited size permitted for the sample collection, and the randomness in picking the samples, this approach can provide little guarantee in accuracy; second, this approach can also be very expensive as the given regex can be arbitrarily complex.

To overcome these issues, we first describe how to estimate the selectivity of a complex regex from only subparts that can be matched over n-grams. This rewrite can potentially reduce the estimation cost as well. Then we propose a new document synopsis that summarizes the {n-gram,count} pairs from the entire corpus to estimate the selectivity for regexes. It obtains higher accuracy than using sample document collection that is just a small subset of the entire corpus, or using top-k n-gram synopsis that drops rather than summarizes n-grams given a size budget. Finally, we show how such a synopsis can be built efficiently.

6.1 Regex and Subregex

The intuition is that each regex can be viewed as being composed of multiple subregexes sequentially in conjunctive form. If we can decompose the regex in such as way that some of its subregexes can be matched against the ngram-based synopsis discussed earlier, then the lowest selectivity of such subregexes provides an upper bound on the selectivity of the original regex. For instance, the regex R in Figure 2 is composed of R_0, R_1, and R_2 sequentially (denoted as $R_0R_1R_2$). Both the subregexes R_0 and R_2 corresponds to two tokens in the original corpus and their selectivity thus can be estimated using the ngram-based synopsis. Since the subregexes are composed together to form the original regex in conjunctive form, the selectivity of the original regex is no more than the lowest selectivity of R_0 and R_2.

R: `\bTo:\s*.1,200\s*\n(>\s*)*\s*(CC):\s*`

R_0: `To:`

R_1: `\s*.1,200\s*\n(>\s*)*`

R_2: `\s*(CC):\s*`

Figure 2: Sample regex and its subregexes.

```
REWRITE(R)
ListOfTBR.reset(); TBR.reset()
for each character $c_i \in R$ do
    if matchesTokStart($c_i$) then TBR.append($c_i$);
    endif
    if matchesTokMiddle($c_i$) then TBR.append($c_i$);
    else
        if matchesTokEnd($C_i$) then
            ListOfTBR.add(TBR); TBR.reset();
        else TBR.reset();
endif endif endfor
return merge(ListOfTBR)
```

Figure 3: Algorithm for rewriting a regex into *TBR*s.

Formally, we denote a regex as R and each of its subregex as R_i $(i \in [0,m])$ where $R = R_0R_1...R_m$. Then we have $\text{sel}(R) \le \min(\text{sel}(R_0), ..., \text{sel}(R_m))$. In order to utilize n-gram synopsis, we only use the subregexes of R that matches at token boundary. For simplicity, from now on we refer to such subregexes as token bounded subregexes, denoted as TBRs.

Figure 3 describes the algorithm to rewrite a regex into a set of TBRs. We start from the left of a regex, and first identify a character that matches the start of a token. Then we keep on looking unless we find a character class that matches the end of a token. If before finding a "end-of-token" character class, we find another a character that is not in the middle of a token, we discard whatever we have found so far and start again. If we find a "end-of-token" character class, then we have successfully found a subregex candidate. The algorithm returns the merged list of all subregex candidates.

Depending on the subregexes generated and how they are composed together to form the original regex, the accuracy of the selectivity estimated based on TBRs varies. By measuring the goodness of the estimation using error rate of the estimation, denoted as $e(\text{sel}(R))$, we have the following regex classification algorithm:

$$e(\text{sel}(R)) = \begin{cases} 0 & \text{if } m = 0, R_0 = R. & (G1) \\ (0,\delta], \delta \ll 1 & \text{if } m = 0 \text{ and } R_0 \subset R. & (G2) \\ (\delta, c], c \ll \infty & \text{if } m \ge 1, \forall R_i \text{ matches} & (G3) \\ & \text{strings of limited length.} & \\ (c, \infty) & \text{otherwise.} & (G4) \end{cases}$$

As shown in Section 7.3, the TBRs generated from most regexes from a real Named Entity Resolution workload fall into the first three conditions in the above equation.

6.2 Roll-Up Synopsis

Given the TBRs generated from a regex, one way to estimate the selectivity is to match all TBRs with every n-gram in the top-k n-gram synopsis. However, the estimation error can be significant, as the regex could match many less frequent n-grams, which are dropped by the top-k n-gram synopsis. Moreover, the bloom filter synopsis cannot be used, because it hashes n-grams into bit-maps without storing them literally. But for regex selectivity estimation, we need the original n-grams to perform the match. To address these issues, we propose the roll-up synopsis.

The intuition of the roll-up synopsis is that rather than dropping low-count n-grams as in top-k n-gram synopsis, we summarize n-grams into regex-ngrams (refer to Section 3.4) to provide more accurate selectivity estimation. In most cases, the summaries over sets of n-grams are sufficient for regex selectivity estimation, as regexes themselves match sets of n-grams. The basic technique to summarize n-grams is to merge n-grams by rolling-up the character class lattice described in Section 3.4. Multiple n-grams can be rolled-up into one regex-ngram. In order to decide, which *roll-up* operations to perform for each n-gram, we rely on a *utility function*.

The utility function of a set *roll-up* operations over a set of n n-grams that result in the same regex-ngram consists of two parts: (1) benefit: the number of unique n-gram reduced $(n-1)$; and (2) cost: the error induced by the summarization. Since the *roll-up* operation induce error over each instance of a unique n-gram, the error can be estimated by $d \times \Sigma_{i=1}^{n} n_i \times c_i$, where n_i is the number of *roll-up* operations, c_i is the count of the ith n-gram, and d is the weight of the cost relative to benefit. Putting cost and benefit together, the utility function is as follows:

$$\begin{aligned} f \quad = \quad & n-1-d\times\Sigma_{i=1}^{n} n_i \times c_i \qquad (4) \\ & (n_i >= 0, c_i >= 1, n >= 1, d >= 0) \end{aligned}$$

According to Equation (4), the maximization of the utility function will guide us to merge more n-grams (i.e., higher n) and merge n-grams that have fewer roll-ups and low counts (e.g., lower d_i, c_i's) to minimize the estimation error.

The goal of the construction algorithm for the roll-up synopsis is to find the set of roll-up operations performed on each n-gram to achieve the *global maximum utility value*. The global maximum utility value which is the sum of the utility value of all the resulting regex-ngrams. The search space of this optimization algorithm is exponential to the number of unique n-grams, thus a brute force algorithm is computationally infeasible.

We use a greedy algorithm to construct the roll-up synopsis by picking, at each step, one regex-ngram with the maximum utility value to include in the roll-up synopsis, and rolling up all n-grams that match this regex-ngram. The algorithm ϕ_{rollup}.BUILD(D, k) to construct the roll-up synopsis from a set of n-grams generated from a corpus D contains following steps:

1. For each unique n-gram $a \in \alpha$ from D, generate all possible regex-ngrams that can be derived by one or more of the roll-up operations from a;
2. For each unique candidate regex-ngram $b \in \beta$ generated from step one, compile the set of n-grams α_b that can be merged into b, and compute the utility value $f(b) = n - d \times \Sigma(c_i \times n_i)$, where n is the number of unique n-grams in α_b, and $d \times \Sigma(c_i \times n_i)$ is the weighted sum of their counts;
3. Pick the candidate regex-ngram $b' \in \beta$ with the highest utility value to be included in the roll-up synopsis and perform the roll-up operations for all n-grams in $\alpha_{b'}$;
4. Remove the n-grams in $\alpha_{b'}$ from the other α_b, where $b \in \beta \backslash \{b'\}$, and recompute the utility values for each remaining candidate regex-ngrams in $\beta = \beta \backslash \{b'\}$;
5. Repeat step 3 and 4, until no roll-up operation can increase the global utility value, or the synopsis reaches a predefined budget size k.

EXAMPLE 2. *Suppose we have a set of three n-gram and count pairs {"the" 100, "Abc" 1, "All" 10}. In step one, we generate all possible regex-ngrams, where some regex-ngrams can be generated from multiple n-grams. For example, "the" can generate* 3^3 *candidate regex-ngrams, one of which* `\p{Alpha}{3}` *can be generated from all three n-grams.*

In step two, the utility values can be computed for each regex-ngram. For example, if we set $d = 0.01$*, the utility value for* `\p{Alpha}{3}` *is* $f(\texttt{\textbackslash p\{Alpha\}\{3\}}) = 2 - 0.01 \times (6*100+6*1+6*10) = -4.66$*. The benefit is* 2 *because all three n-grams can be rolled-up into it. The number of roll-up operations for each n-gram is* 6*, because they all have three characters, and each characters roll-up two levels in the character class lattice.*

In step three, we pick the regex-ngram with the highest utility value: `A\p{Lower}{2}`*, where* $f(\texttt{A\textbackslash p\{Lower\}\{2\}}) = 1 - 0.01 \times (2*1+2*10) = 0.78$*, and perform the roll-ups on "Abc" and "All". In step 4, we do the book keeping to remove "Abc" and "All" from the n-gram list of other regex-ngrams.*

In this toy example, no more roll-up can be performed, and the final set of regex-ngrams in the roll-up synopsis is: $\gamma =$ *{ "the" 100, "*`A\p{Lower}{2}`*" 11}. In general if the synopsis has not been reduced to the expected size budget, we iteratively call step three and four to perform more roll-ups to reduce the size of the synopsis.* □

6.3 Pruning the Search Space

Although the greedy algorithm described in the previous section reduce the computation from exponential to the number of n-grams of the brute force algorithm to linear, the computation needs to be done for each n-gram is still expensive. This computation involves generating all candidate regex-ngrams β from each n-gram, and picking out the one with the highest utility. This a search problem where the search space is β with size as large as $O(N \cdot 2^{maxlen})$, where N is the number of n-grams and *maxlen* is the maximum length of the n-gram and 2 signifies the boolean decision of whether to roll-up a character or not. For long n-grams, this search space can be prohibitively large.

We used two rules to prune this search space while still find the regex-ngram with the highest utility value. Both rules explore the characteristics of the utility function in Equation (4). The first rule states that any roll-up operation that does not merge additional n-grams has a decreasing utility value. The second rule states that the utility value of a set of roll-up operations $f(b2)$ is larger than the utility value of a other set $f(b1)$ if and only if:

$$\begin{aligned} n2 - n1 \quad & > \quad d \times \Sigma_{i=1}^{n2} c_i \times n_i - \Sigma_{j=1}^{n1} c_j \times n_j \\ & > \quad d \times \Sigma_{i=1}^{n2} c_i \times n_i \qquad (5) \end{aligned}$$

Based on these two rules, the algorithm to find the regex-ngram with the highest utility value can be found in Appendix 10.3.

6.4 Matching Function

The selectivity estimation function of a regex over the roll-up synopsis ϕ_{rollup}.ESTCOUNT(re) involves: (1) Rewrite the regex into subregexes according to the rewrite algorithm in Section 6.1; (2) If the subregexes cannot give an accurate estimate of the selectivity of the regex according to the classification algorithm in Section 6.1, fall back to match the regex over a sample document collection to estimate the selectivity; (3) Otherwise, we match each subregex over the regex-ngrams in the roll-up synopsis and take the lowest estimated selectivity of the matched regex-ngrams as the estimation.

The matching algorithm of a subregex over a regex-ngram is similar to the Java regex matching engine [7], except that the matching

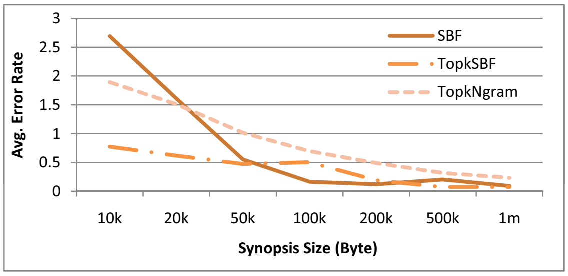

Figure 4: Average error for all dictionaries: Top-k N-gram vs. Stratified Bloom-Filter Synopsis.

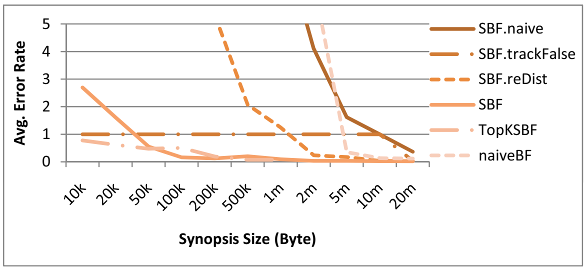

Figure 5: Average error for all dictionaries: different Bloom-Filter based Synopsis.

also considers the containment relationship between characters and character classes is described in Section 3.4.

7. EVALUATION

Having described three types of document synopsis and the selectivity estimation functions for dictionaries and regexes over them, we now present the experiment results comparing the accuracy of selectivity estimation over different synopsis.

7.1 Setup

We implemented the n-gram, bloom filter and roll-up synopsis as described in Section 4, 5, 6 in Java 1.6. We conducted the experiments reported here on a 2.5GHz duo core Intel Pentium 4 Windows 7 system with 4GB RAM.

Dataset: The dataset we used in all the experiments is from the collection of $256,000$ emails in the Enron corpus [5].

Baseline Synopsis: *Random sample synopsis* is constructed by randomly picking a sample document collection, whose size is bounded by a budget of N tokens.

Workload I: NamedEntity Rules: A set of rules taken from a named entity annotator (NEA) developed in System T for Lotus Notes 8.01 and several other IBM products.

- *NERegexes* A set of 32 regular expressions chosen based on the profiling result of the NEA over Email, including the five most expensive regular expressions and the five least expensive ones, as well as 22 others from the middle of the distribution.
- *NEDicts* A set of 137 dictionaries chosen from the NEA, as well as from the annotators described in [14]. The dictionaries ranged in size from one entry to $90,000$ entries.

Workload II: Synthetic Queries: For increased coverage of the query space, we also ran experiments with synthetic workloads.

- *GeNgrams* We randomly pick 100 n-grams from the corpus D.
- *GeRegexes* We generate a set of level two regex-ngrams from all the n-grams in a corpus D, by rolling up all lowercases and uppercases into `\p{Lower}` and `\p{Upper}`, and digits into `\p{Digit}`. Then randomly pick 100 from this set.

7.2 Dictionary Selectivity Estimation: N-gram vs. Bloom-Filter Synopsis

In this experiment, we first compare the accuracy of dictionary selectivity estimation over top-k n-gram synopsis (TopkNgram), stratified bloom filter (SBF) (see Section 5.1), and top-k stratified bloom filter (TopkSBF) synopsis (see Section 5.3). We measure the average estimation error rate (y-axis) over all dictionaries in *NEDicts* for the three synopsis given different space budget (x-axis), varying from 10 thousand to 1 million bytes.

In Figure 4, the results show that at small synopsis sizes ($< 20k$), SBF has higher error rate compares to the TopkNgram. This is because SBF contains all n-grams, which is overflowing the bloom filters. At larger synopsis sizes ($> 50k$), SBF achieves lower average error, because it has enough space to store all the n-grams with a low false positive rate.

The TopkSBF combines the merits of top-k n-gram synopsis at small synopsis sizes, and that of the stratified bloom filter at larger sizes. As we can see, the error rates of TopkSBF are smaller than that of SBF, and half that of TopkNgram for most synopsis sizes.

In addition, we compare the accuracy of different variations of bloom filter synopsis. Figure 5 shows the accuracy of selectivity estimation of six variations of bloom filter synopsis. SBF.naive is the naive stratified bloom filter described in Section 5.1; SBF.trackFalse is SBF.naive with the optimization to compensate for the false positives; SBF.reDist is SBF.naive with the optimization to allocate space unequally among bloom filters; SBF is SBF.naive with two optimizations; TopkSBF is the top-k stratified bloom filter; and finally naiveBF inserts each n-gram into a single big bloom filter.

As we can see, naiveBF, SBF.naive, and SBF.trackFalse perform the worst—the error rate does not drop under 1 until the synopsis size increases beyond 5 million bytes. SBF.reDist performs better, where the error rate drops to below 1 after 1 million bytes. SBF and TopkSBF perform better by far—the error rate drops below 1 at size $50k$ for SBF and at as low as $10k$ for TopkSBF.

7.3 Regex Selectivity Estimation: Regex Rewriting and Classification Algorithms

In this experiment, we present the effectiveness and accuracy of the regular expression rewriting and classification algorithms described in Section 6.1.

The regular expression rewriting algorithm decomposes a regular expression into a set of TBRs (token bounded subregexes), where the selectivity of the regex is estimated by the lowest selectivity of the TBRs. We use the rewriting algorithm to rewrite the 32 regexes in *NERegexes* workload, then use the classification algorithm to classify the regexes based on how its TBRs are composed together and finally compute the error rate of estimating the selectivity of a regex from its TBRs. The results are listed in Table 6. Table 6 also shows for each group an example regex and the subregexes they are written into. As can be seen, out of the 32 regexes, 16 are classified as G1, with their estimates exactly the correct selectivity; 7 are identified as G2, with their estimates within 0.2 error rate; 5 of them are classified as G3, with the error rate of their estimates falling between 0.2 and 1; and only 4 are G4 regexes, with error rate larger than 1.

The results shows that the selectivity of most regular expressions in *NERegexes* workload can be accurately estimated by the selectivity of their subregexes. The results also demonstrates that our classification algorithm can effectively classified regexes of different expected error rate based on how their subregexes are composed together. It thus can be used to pick out the one whose subregexes cannot provide an accurate estimate, for which we use the baseline random sample synopsis.

<table>
<tr><th></th><th># of Regexes</th><th>Error Range</th><th>Example regex</th><th>Example TBRs</th></tr>
<tr><td>G1</td><td>16</td><td>0</td><td>\d{4}</td><td>\d{4}</td></tr>
<tr><td>G2</td><td>7</td><td>(0,0.2]</td><td>[0-9\:\.]+</td><td>([0-9]+)|([\:\.])</td></tr>
<tr><td>G3</td><td>5</td><td>(0.2,1]</td><td>To:\s*.{1,200}\s*\n(>\s*)*\s*(CC|cc|Cc):\s*(\n)?</td><td>To:\s*
\s*(CC|cc|Cc):\s*</td></tr>
<tr><td>G4</td><td>4</td><td>(1,∞)</td><td>.+[/].+[/].+(\@.+)?\s*\n(>\s*)*\d{2,4}[/.]
\d{2,4}[/.]\d{2,4}\s\d{2}\:\d{2}(\s+(PM|AM))?</td><td>[/]
[/]
\d{2,4}[/.]\d{2,4}[/.]\d{2,4}\s\d{2}\:\d{2}</td></tr>
</table>

Figure 6: Result of regex rewriting algorithm over *NERegexes* workload.

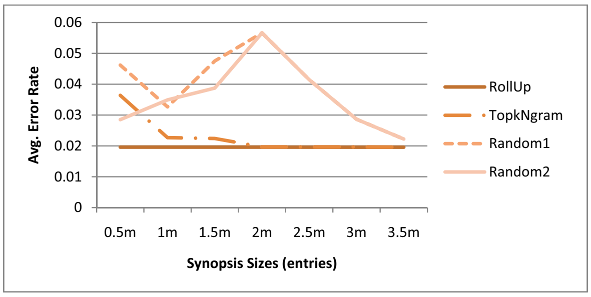

Figure 7: Average error rate for all Group 1 to 3 regexes in *NERegex*: roll-up vs. top-k n-gram vs. random sample synopsis.

7.4 Regex Selectivity Estimation: N-gram vs. Roll-Up Synopsis

In this experiment, we compare the accuracy of regex selectivity estimation over top-k n-gram synopsis (TopkNgram), roll-up synopsis (RollUp) (see Section 6.3), and random sample synopsis (Random). We measure the average estimation error rate (y-axis) over all regexes in G1 to G3 (described in Section 6.1) of *NERegexes* for the three synopsis given different space budget (x-axis), varying from 0.5 million entries to 3.5 million entries. Since the entries include unigrams, bigrams and trigrams in RollUp and TopkNgram, we allow the upper bound of the space budget—three times as many tokens—for Random synopsis to be conservative.

In Figure 7, the results show that the error rates of two runs of the Random synopsis, Random1 and Random2, are quite different from each other with smaller synopsis sizes ($< 2m$). Moreover, the error rates do not decrease monotonically with the increase of the synopsis sizes. This is because the randomness in the set of documents included in Random synopsis. In general, the error rate of Random is much higher than both TopkNgram and RollUp. Comparing TopkNgram and RollUp, we can see that the error rate of TopkNgram is two times that of RollUp at size $0.5m$, and gradually decreases to be the same as RollUp. The error rate of RollUp keeps the same at 0.02 from $0.5m$ to $3.5m$, which corresponds to the error rate of the regex rewriting algorithm.

We also conducted the experiment to compare the accuracy of the TopkNgram and RollUp synopsis with synthetic workloads *GeNgrams* and *GeRegexes*. We measure the average error rate where the size budget is $0.5m$. Using *GeNgrams*, the average error rate is 0.3 for RollUp, 0.1 for TopkNgram; using *GeRegexes* on the other hand, the average error rate is 0.005 for RollUp, 0.32 for TopkNgram. This result shows that the RollUp synopsis is less accurate in estimating selectivity for regexes containing n-grams, because a lot of them are rolled-up into regex-ngrams. But, RollUp greatly outperforms TopkNgram in accurately estimating the selectivity for regexes containing no ngram.

7.5 Summary

The results showed that accurate selectivity estimation, of error rate less than 0.1, can be achieved by supporting document synopsis as small as $100k$ bytes, less than $1/1000$ size of the text corpus. The experiments reported in Section 7.2 demonstrated that the top-k stratified bloom filter halves the error rate of dictionary selectivity estimation from the top-k n-gram synopsis. The results in Section 6 and 7.4 showed that the roll-up synopsis is the most accurate in regular expression selectivity estimation, compared to the top-k n-gram and random document synopsis. Using these accurate selectivity estimations, optimizer can be extended to pick good plans for queries involving extraction operators over text.

8. CONCLUSION

The need for query processing to efficiently support extraction operators over text data, including dictionaries and regular expressions, is becoming increasingly acute. A lot of text processing queries involve multiple extraction and join operators, resulting in many possible query plans. However, there has been little research on building the selectivity estimation for these extraction operators. In this paper, we proposed a document synopsis-based approach for selectivity estimation. We developed three classes of document synopsis: n-gram synopsis, bloom filter synopsis and roll-up synopsis. Our experimental results show that these synopsis are compact and enable accurate selectivity estimations. As future work, we intend to look at cost estimation of extraction operators and extend an query optimizer to use these statistics.

9. REFERENCES

[1] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB*, 2001.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[3] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE*, 2004.

[4] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.

[5] Enron Dataset. http://www-2.cs.cmu.edu/ enron/.

[6] Li Fan, Pei Cao, and Jussara Almeida. Summary cache: A scalable wide-area web cache sharing protocol. In *SIGCOMM*, 1998.

[7] Mehran Habibi. In *Java Regular Expressions: Taming the java.util.regex Engine*. Apress, 2003.

[8] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[9] H. V. Jagadish, Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. One-dimensional and multi-dimensional substring selectivity estimation. *VLDB Journal*, 9(3):214–230, 2000.

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

[11] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, 2007.

[12] Liang Li and Chen Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, 2005.

[13] C. D. Manning, P. Raghavan, and H. Schtze. Language models for information retrieval. In *Introduction to Information Retrieval*, 2008.

[14] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, 2008.

[15] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[16] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[17] David Talbot and Miles Osborne. Smoothed bloom filter language models: Tera-scale lms on the cheap. In *EMNLP*, pages 468–476, 2007.

[18] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom histogram: path selectivity estimation for xml data with updates. In *VLDB*, 2004.

10. APPENDIX

Section 10.1 describes four different estimator for an n-gram not in the top-k n-gram synopsis. Section 10.2 explains in detail the nature of the errors introduced by a naive SBF when estimating the selectivity of a dictionary. Section 10.3 provides more details on the algorithm that efficiently prune the search space to find the regex-ngram that has the highest utility value.

10.1 Estimators for Top-k Ngram Synopsis

The following are four different estimators $\phi_{\mathsf{topk}}.\mathrm{ESTCOUNT}(\mathsf{ngram})$ of the count of a ngram not included in ϕ_{topk}:

- "MaxLikelihood": returns a count of zero
- "AddOne": returns a count of one, also adds 1 to the counts of every n-gram in the synopsis
- "AvgValue": returns the average count across all n-grams that were excluded from the synopsis
- "LeftBackoff": finds the longest prefix of the n-gram included in synopsis ϕ_{topk}, if the remaining postfix is also included in ϕ_{topk}, then multiply the counts of the prefix and the postfix divide by the total n-gram count. If the remaining prefix is not in ϕ_{topk}, then multiply the count of prefix with "AvgValue" divide by total count. If no prefix is found in ϕ_{topk}, the "LeftBackoff" technique falls back on "AvgValue".

10.2 Error Analysis for Naive SBF

The estimation error of the Stratified Bloom Filter synopsis depends on the counts of the n-grams that are inserted into the synopsis. This relationship is best explained with an example. Consider an SBF synopsis with 10 Bloom filters. Since there is one Bloom filter per bit, this synopsis can encode any n-gram count between 0 and 2047. Imagine that the counts for every n-gram in the corpus have been inserted into this synopsis, using the basic insertion procedure outlined above.

Recall that the error of a Bloom filter is one-sided: If an item has been inserted into a Bloom filter, lookup operations on the Bloom filter will always indicate the presence of the item inside the Bloom filter. If, on the other hand, an item has *not* been inserted into the Bloom filter, the lookup operation may report that a given element is present even though it is not.

Now consider what happens if one probes our example synopsis with an n-gram whose true count (the number of times the n-gram appears in the corpus) is 2047. The binary representation of 2047 is 1111111111 (ten 1's), so this example n-gram must have been inserted into every Bloom filter. Any subsequent lookup operation will find this n-gram to be present in every Bloom filter, so the estimation algorithm will always report an estimated count of 2047 for the n-gram. Hence, the estimation error for this hypothetical n-gram will always be zero.

Now consider what happens if the synopsis is used to estimate the count of an n-gram whose true count is zero. Since the n-gram's count is zero, the n-gram should not be present in any of the Bloom filters. However, a lookup operation a given Bloom filter may generate a false positive. If we denote the false positive probability of the ith Bloom filter by the random variable V_i, then the overall estimation error is the sum of the (independent) errors across the different bit positions:

$$2^0V_1 + 2^1V_2 + \ldots + 2^10V_{10} \tag{6}$$

For n-grams with true counts between 0 and the maximum count, the estimation error will fall between these two extremes. In general, for an SBF with k Bloom filters, the error is given by

$$\sum_{i=1}^{b} 2^i z_i V_i \tag{7}$$

where $z_i = \begin{cases} 1 \text{ if bit } i \text{ is } \textbf{not} \text{ } 1 \\ 0 \text{ otherwise} \end{cases}$ and V_i is a random variable as in Equation 6.

In most text corpora, the distributions of n-gram counts tend to be quite uneven, with a few very large counts and a large number of small counts. This type of distribution creates a problem for a naive SBF synopsis, as the errors from the Bloom filters associated with higher-order bits quickly add up. For example, in the Enron corpus used in our experiments, the most common n-gram occurs about 14 million times (meaning that 23 bits are required to store these counts), while the averge n-gram count is only about 26.3. Imagine if, in an SBF synopsis over this corpus, the Bloom filter for bit 23 had a false probability of 0.001 percent. Then, for any n-gram whose true count was less than 2^{23}, the error from this bit alone would have an expected value of $2^{23} \times 0.00001$, or about 83.9. This expected error is more than three times the average n-gram count for the corpus.

Worse, the variance of this error is also approximately 83.9. Since the false positive probabilities of a Bloom filter for individual n-grams are nearly uncorrelated, this variance adds up very quickly when one is trying to estimate the selectivity of a large dictionary.

10.3 Details on Pruning the Search Space

The algorithm to find the regex-ngram, generated by rolling-up level one characters to level two character classes, that has the highest utility value in one step of the greedy algorithm described in Section 6.3 is as follows:

1. For each unique ngram we generate a *level two regex-ngram* by rolling-up each character or digit into level two character classes;
2. For each unique level-two regex-ngram b, if it matches only one ngram, the original ngram has the highest utility of all the regex-ngrams generated from it based on rule 1); if it mathes more than one ngram, compile the list of matched ngrams α_b;
3. Enumerate all the combinations of the ngrams in α_b that can generate a different regex-ngram from b, which has the complexity of $O(len)$ where len is the length of ngrams merge into b;
4. For each combination, if it excludes more than $d \times \Sigma_{i\in\alpha_b} c_i \times n_i$ number of ngrams in α_b, then its utility function is less than b, otherwise computes its utility value;
5. Pick the highest value regex-ngram to include in the roll-up synopsis from the ones we computed the utility value, and perform book-keeping.

Similar algorithm is used to find the regex-ngram, generated by rolling-up lavel two character classes to level three character classes, that has the highest utility value.

EXAMPLE 3. *Suppose we have the set of ngram and count pairs $\alpha_b = \{$"Jun" 10, "Jul" 20, "Aug" 100$\}$. At step one, we generate all the level two regexes, in this case only one $b =$* `\p{Upper}\p{Lower}{2}`. *At step two we generate the set of ngrams $\alpha_b = \{$"Jun", "Jul", "Aug"$\}$ matches to it. At step three, the combinations of this set of ngrams can be enumerated by looking at each position: at position 1, the combinations are:*

$\{$"Jun", "Jul"$\}$; at position 2, the combinations are $\{$"Jun", "Jul", "Aug"$\}$; at position 3, the combinations are $\emptyset$.

In step four, we check if each combination contains more than $(3-d\times(10\cdot 3+20\cdot 3+100\cdot 3))$ ngrams. When $d=0.01$, only the second combination, that generates a regex n-gram `\p{Upper}u\p{Lower}`*, satisfies the condition. Thus in step four, we compute its utility value, which is higher than the utility value of* `\p{Upper}\p{Lower}{2}`*. Thus, we pick* `\p{Upper}u\p{Lower}` *to be included in the roll-up synopsis in step five.* □