Filtered Dynamic Indexing of Text Streams

Samuel Huston
Center for Intelligent Information Retrieval
University of Massachusetts Amherst
140 Governors Drive
Amherst, MA, 01002
sjh@cs.umass.edu

W. Bruce Croft
Center for Intelligent Information Retrieval
University of Massachusetts Amherst
140 Governors Drive
Amherst, MA, 01002
croft@cs.umass.edu

Andrew McGregor
Department of Computer Science
University of Massachusetts Amherst
140 Governors Drive
Amherst, MA, 01002
mcgregor@cs.umass.edu

ABSTRACT

The identification and indexing of textual features that occur frequently in text streams is vital for many real-world information retrieval applications. Previous research has shown that frequent indexes of text features can be constructed efficiently for static collections. We extend this research to allow the insertion of new documents to the index. The insertion of new documents introduces a problem for these static methods, in that newly-frequent n-grams cannot be distinguished from infrequent n-grams. In this paper we propose and test a novel system for the construction and maintenance of dynamic indexes for frequent text features in text streams. Our system is designed for implementation within a parallel computing environment. We investigate the CountMin sketch and propose a new LinearCountMin sketch for the purpose of tracking the entire text feature vocabulary. We demonstrate that these data structures allow us to efficiently maintain a dynamic index of frequent text features. Furthermore, we show that this system is able to process very high volume text streams using only modest resources.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and software—*performance evaluation*.; H.3.1 [**Content Analysis and Indexing**]: Indexing methods; H.3.3 [**Information Search and Retrieval**]: Information filtering

General Terms

Algorithms, experimentation, performance

Keywords

Repeated phrase, n-gram, hash filter, text reuse, scalability, dynamic, online

1. INTRODUCTION

The identification and indexing of features that occur frequently in text streams is vital for many real-world information retrieval applications. A prime example is the discovery and tracking of trending topics in Twitter feeds. However, this task can also be motivated in the context of personal data search, breaking news search, and general web search. Previous research has shown that, in these applications, frequent n-grams, phrases, and key concepts are important features for computing document scores (e.g., [18, 2, 9]). All of these features share the common property of having a very large possible total vocabulary size with a significant proportion of the vocabulary occurring very few times. In this paper, we look at the problem of identifying and indexing frequent text features in a stream of documents. In these streams, new data is constantly being added, and the newly arrived documents, along with the text feature statistics, must be made available for retrieval as soon as possible.

Huston et al. [14] provides an in-depth analysis of several methods of indexing frequent n-grams within a static collection, where "frequent" n-grams are defined as those occurring within the collection more often than a threshold frequency. By focusing on this subset of the vocabulary, the final indexes are significantly smaller in size and faster to search than a full n-gram index. They conclude that a two-pass method using a hash-based filter provides an efficient balance between space and time usage when constructing a frequent n-gram index.

However, this two-pass method is limited to a static setting for two key reasons. First, it needs to process the data twice. Data streams constantly provide new data, so the stream cannot be read multiple times. Second, if we allow the insertion of new documents into this type of index, it is inevitable that some n-grams will be promoted from infrequent to frequent. These n-grams should then be inserted into the index. The methods considered in [14] discard all information about these infrequent n-grams, so it is impossible to determine when an n-gram becomes frequent. To use these algorithms, the entire collection would need to be re-indexed for each new batch of data.

The obvious solution is to use a dynamic index. Data structures and algorithms for dynamic indexes have been extensively studied in information retrieval. Lester et al. [16] compare several baseline approaches to the geometric indexing approach. This technique allows the insertion and removal of documents from an index or a set of indexes efficiently, while still providing efficient retrieval throughout. Unfortunately, using this technique for the task of indexing frequent text features is not feasible. This is because it is impossible to directly determine which text features may be discarded as they will never occur frequently. Without the ability to estimate frequency, the system must maintain an index over the entire vocabulary to provide 100% recall over the frequent text features. As we will show, the space requirement for a full text feature index is infeasible, even for modest sized collections.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

We define a text feature as a series of two or more words extracted from a document. Phrases, key concepts, n-grams, and unordered windows are all examples of text features. In this paper, we focus on n-gram text features. n-Grams are often used as proxies for more complex text features [2]. Additionally, they provide us with explicit control over the size of the text features.

As mentioned, Huston et al. [14] defined a frequent text feature to be an n-gram that occurs at least h times. We further restrict this definition to a text feature that occurs at least h times within a preceeding time window of length w, where the time window is a stepping window of fixed length over the input text stream S. Note that the size of the time window will influence the total number of frequent features within a given collection. The use of this window focuses the system on common and trending text features. The size of the time window should be determined for each application. When indexing Twitter feeds or breaking news data, an appropriate time window could be between one day and several hours of data. When indexing web data, an appropriate window size could be several days or weeks. For this paper, we will measure the length of a time window by the number of text features the time window spans.

In order to distinguish between frequent and infrequent text features, a filter data structure that tracks the frequency of the entire vocabulary is required. We can make some specific requirements for this structure. The space used by the structure should be independent of the size of the textual features. For example, a structure that tracks 10-grams should be the same size as a structure that tracks 5-grams. The data structure should never underestimate the frequency of text features. This requirement ensures that all frequent text features will be identified and indexed. The size of the data structure should scale linearly as the size of the time window is increased. Finally, the data structure should be able to be effectively distributed across a parallel computing platform.

Naïvely, this filter data structure could consist of a mapping from each textual feature to a current frequency count. In order to satisfy speed and throughput requirements, this structure should be held in memory.

To demonstrate that this simple structure will not scale, we will look at an example based on our calculations of n-gram statistics (see Section 2.) Let the time window in our example span a window of 1 GB of English text. Ignoring mark-up, a reasonable estimate would be that this window contains around 100 million words, and a vocabulary of around 1 million distinct words. If each word and each count is represented as a 3 byte integer, when indexing these words, the data structure requires 6 bytes per entry. Including the pointers required for a dynamic search structure, 1 million entries would require around 10 MB. Now consider 5-grams. Again using 3 byte integers for words and counts, the data structure now requires 18 bytes per entry. It is plausible that almost all 5-grams in this window are unique, thus the data structure needs to store one entry for each 5-gram in the window. Accounting for the pointers required for a dynamic search structure, 100 million entries would require around 2.5 GB to store. While this may be a feasible requirement for current commodity machines, when $n = 10$ and the window is a terabyte of text, the requirement quickly becomes infeasible even for high-end machines.

In this paper we present the CountMin Sketch [8] and introduce the novel LinearCountMin Sketch as viable filter data structures for this task. The CountMin Sketch was originally developed by the stream processing community in order to track frequent items in network traffic data.

The system we propose does not index all text features. It focuses on indexing currently frequent or trending textual features. This enables larger systems efficient access to statistics for important text features. We believe that collecting this data is sufficient for many practical purposes. However, if there is a need for a *full* text feature index, it is possible to use frequent text feature indexes to improve the efficiency of information retrieval over the entire set of textual features. A full positional term index can be used to re-construct the infrequent text features omitted from a frequent text feature index. Thus combination of indexes avoids the high processing costs involved in merging large amounts of positional data for frequent text features and ensures access to the entire text feature vocabulary.

In this paper, we investigate the vocabularies of n-grams and frequent n-grams in Section 2. In Section 3, we present and analyze the CountMin sketch [8] and the novel LinearCountMin sketch as candidates for the filter data structure. We then show that both structures satisfy all of the requirements we placed on this structure. In Section 4, we propose a system design for the dynamic indexing of frequent text features that can be distributed within a parallel computing environment. We present the details of a parallel implementation in Section 5. We empirically test the performance of several aspects of a dynamic filtered indexing system in Sections 6 and 7. We present related work in Section 8 and make concluding remarks in Section 9.

	WSJ	GOV2	ClueWeb-B
Collection Length, N	$7.8 \cdot 10^7$	$2.2 \cdot 10^{10}$	$3.2 \cdot 10^{10}$
1-grams, $\lvert V_1 \rvert$	$2.4 \cdot 10^5$	$3.9 \cdot 10^7$	$8.9 \cdot 10^7$
2-grams, $\lvert V_2 \rvert$	$9.3 \cdot 10^6$	$5.3 \cdot 10^8$	$1.3 \cdot 10^9$
3-grams, $\lvert V_3 \rvert$	$3.4 \cdot 10^7$	$2.4 \cdot 10^9$	$5.5 \cdot 10^9$
4-grams, $\lvert V_4 \rvert$	$5.6 \cdot 10^7$	$4.9 \cdot 10^9$	$1.1 \cdot 10^{10}$
5-grams, $\lvert V_5 \rvert$	$6.7 \cdot 10^7$	$6.7 \cdot 10^9$	$1.5 \cdot 10^{10}$
6-grams, $\lvert V_6 \rvert$	$7.1 \cdot 10^7$	$7.8 \cdot 10^9$	$1.7 \cdot 10^{10}$

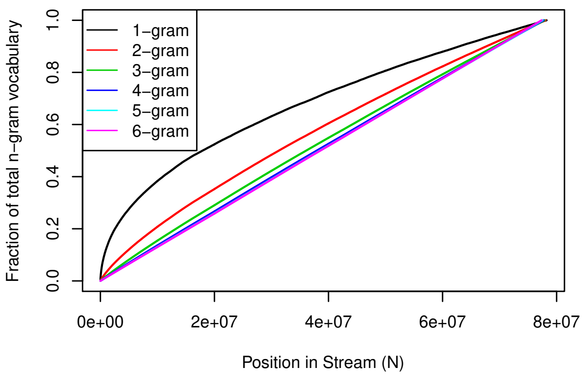
Table 1: Collection length and n-gram vocabulary sizes for three English collections. Measured as the number of word tokens.

2. VOCABULARY SIZES

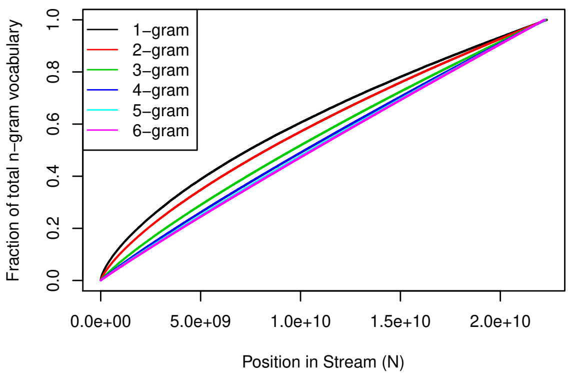
We begin by investigating the statistics of the text streams that we are indexing, and in particular understanding the size of the vocabulary that may need to be monitored by the filter data structure. It is also important to investigate the effect of the relationship between the threshold value and the window size on the frequent text feature vocabulary. These statistics show us the amount of data we are discarding in the selection of a threshold or a window size. As discussed earlier, we only investigate n-gram text features throughout this paper.

The statistics shown in this section were extracted from 3 English TREC collections of increasing sizes – Wall Street Journal (WSJ) (~0.6 GB), GOV2 (~426 GB) and ClueWeb Category B (Clueweb-B) (~1.2 TB). The order of documents is an important consideration for each of these collections. The document order interacts with the time window size to influence the vocabulary size of frequent textual features. Each of these collections is ordered in a different manner; WSJ is in date order, GOV2 is in random order, and Clueweb-B is in crawl order. Wikipedia documents are omitted from the Clueweb collection, as they could not be placed in the correct ordering.

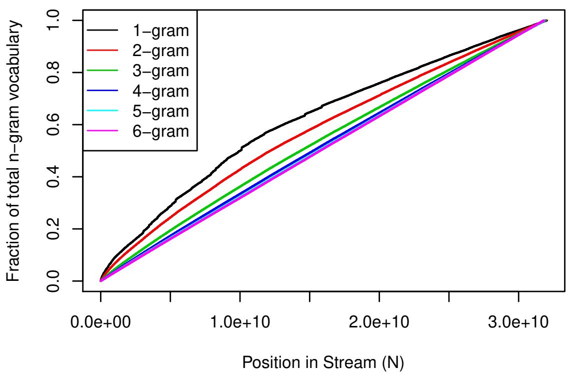
Table 1 shows the collection length and total vocabulary sizes for 1 to 6 grams for each of these collections. Observe that for relatively small values of n, the vocabulary size is a significant fraction of the collection. As discussed in the introduction, it is the massive size of the vocabulary of text features that makes constructing a full



(a) TREC WSJ Data, $N \approx 7.8 \cdot 10^7$

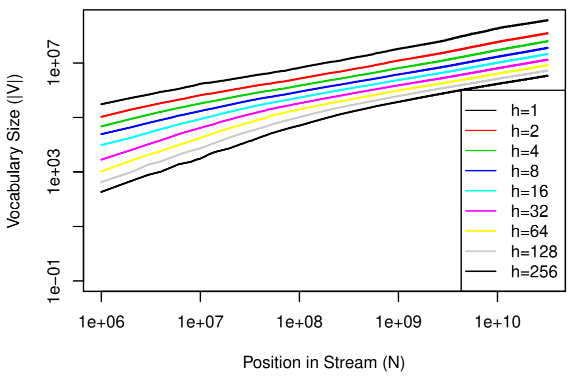


(c) TREC GOV2 Data, $N \approx 2.2 \cdot 10^{10}$

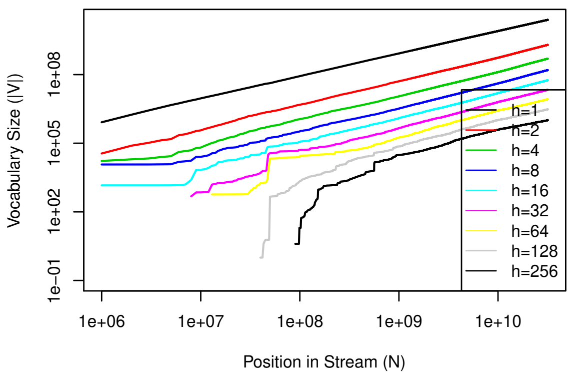


(d) TREC ClueWeb-B Data, $N \approx 3.2 \cdot 10^{10}$

Figure 1: Normalized vocabulary growth rate for various English language text collections. Data points are normalized by the corresponding final n-gram vocabulary.



(a) TREC Clueweb-B Data, $n = 1$, $N \approx 3.2 \cdot 10^{10}$



(b) TREC Clueweb-B Data, $n = 5$, $N \approx 3.2 \cdot 10^{10}$

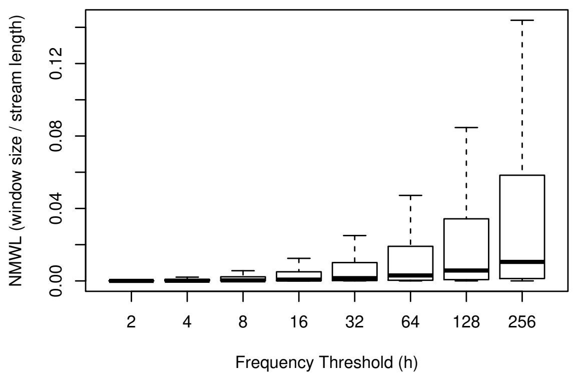
Figure 2: Threshold vocabulary growth rates for Clueweb for 1 and 5-grams. Graphs for the other two collections were produced, these show similar trends.

index of text features infeasible.

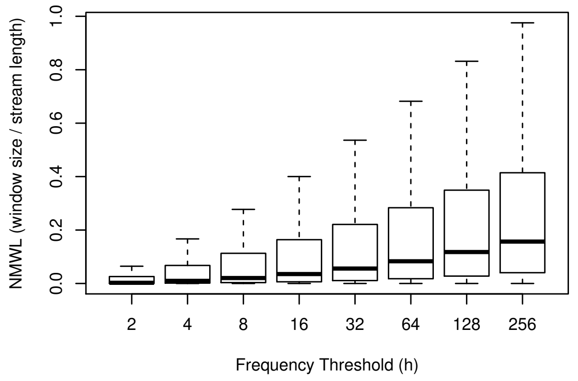
In a dynamic setting we are also interested in the growth rates of the text feature vocabularies. Figure 1 shows the vocabulary growth rates for 1 to 6 grams. The data in this graph is normalized by the total vocabulary size; a diagonal line from the bottom left to the top right would indicate that the vocabulary grows at a constant rate. Sharp rises in these graphs indicate a sudden influx of new n-grams, while horizontal plateaus indicate that few, if any, new n-grams are arriving. Each of these graphs show that the vocabulary of n-grams grows continuously at a slowly decaying rate. Thus, the filter data structure must be able to cope with a constantly changing vocabulary as the time window steps accross the data.

Figure 2 shows the growth rates for frequent n-gram vocabularies given a range of threshold values. We show graphs for data extracted from Clueweb-B, and have produced graphs for the other collections and verified that they show similar trends. Note that both axes for this graph are logarithmic. The anomalous data at the lower left side of the graph is produced by the scarcity of 5-grams that occur more than 32 times. As the amount of data in consideration grows, the growth rate of these vocabularies stabilizes. We can see from these graphs that the threshold and the frequent vocabulary sizes are linked in an interesting way. As the threshold is increased exponentially (using a factor of 2), the size of the frequent vocabulary decreases logarithmically (using base 10). It is unclear if this observation holds for English generally, or if it is a feature of these three collections.

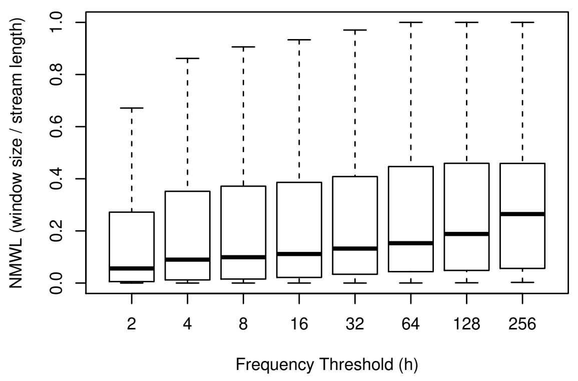
We now investigate the effect of the choice of time window size. Figure 3 shows the distribution of normalized minimal window sizes for a range of values of n and h for the WSJ collection. The WSJ collection is used for this experiment, as it is the only collection that can be sorted into temporal order. A minimal window size is defined as the smallest window size that would allow each n-gram in the data stream to be be defined as frequent. We normalize the size of the minimal windows using the size of the observed



(a) TREC WSJ Data, $n = 1$, $N \approx 7.8 \cdot 10^7$



(b) TREC WSJ Data, $n = 2$, $N \approx 7.8 \cdot 10^7$



(c) TREC WSJ Data, $n = 5$, $N \approx 7.8 \cdot 10^7$

Figure 3: Box and whisker plots for window lengths for a variety of values of n and h. NMWL stands for Normalized Minimal Window Length. This value is the smallest possible fraction of the stream that would allow a particular feature to be identified as frequent.

stream. This adjusts for a possible sampling bias produced by measuring this data from collections of finite size. This graph shows how the window size affects the vocabulary size of a frequent index. Using this data we can estimate the number of potentially frequent features that we will not be indexed due to the selection of the window size. For example, if we set the window size to 0.2 of the length of the WSJ collection and the threshold to 32, we will index around 75% of potentially frequent 2-grams.

3. FILTER DATA STRUCTURES

We outlined a series of requirements that a filter structure must meet in the introduction. We will address some of these requirements in this section. First, text features, such as phrases, key concepts or n-grams, can be arbitrarily large and we require that the size of the filter structure does not depend on the size of the text feature. Second, it is possible that the vocabulary of text features is as large as the time window, so the data structure must be able to track a vocabulary that is the size of the time window. Given these requirements, hash function-based data structures are an appropriate choice for tracking the vocabularies of textual features.

We have chosen to base our filter data structures on the Count-Min Sketch described by Cormode and Muthukrishnan [8]. The authors define a 'Sketch' to be a compact summary of a large amount of data. This structure was originally developed as a solution to the 'heavy hitter' problem for network traffic data. In our case we require the CountMin Sketch to summarize the frequencies of text features. This data structure provides an excellent method of controlling the trade-off between space requirements and probabilistic error guarantees. As we will show, it also provides better performance than a simple hash indexed array of count values.

3.1 CountMin Sketch

The CountMin Sketch [8] consists of a two dimensional table of counters and a set of methods for updating and estimating count values. Each row in the table is a hash-indexed array of counters. Each row must use a different hash function. When a new feature instance is added to the structure, one cell in each row is incremented. To estimate the frequency of a feature, the frequency estimates from each row are extracted and the minimum is returned. We will now describe the guarantees provided by this structure.

The width and depth of the structure are defined using two parameters, (ϵ, δ). The parameter ϵ is the expected error as a fraction of the time window w, and the parameter δ controls the probability of observing error greater than the expected error. The width of the hash table is defined as $v = \lceil \frac{e}{\epsilon} \rceil$ and depth $d = \lceil \ln \frac{1}{\delta} \rceil$. d hash functions that are chosen uniformly at random from a family of pairwise independent hash functions.

When some feature i is to be added or removed from the sketch, one count in each row of the CountMin sketch is updated as determined by the corresponding hash function. Formally:

$$\forall_{j<d} : \mathit{count}[j, h_j(i)] = \mathit{count}[j, h_j(i)] + 1$$

$$\forall_{j<d} : \mathit{count}[j, h_j(i)] = \mathit{count}[j, h_j(i)] - 1$$

The frequency of some feature, i, can be estimated by returning the minimum count in the set.

$$\hat{a}_i = \min_j \ \mathit{count}[j, h_j(x_i)]$$

Conservative update, originally presented by Estan and Varghese [12], is a heuristic method that is often used to improve the frequency estimates produced by the CountMin sketch by avoiding some collisions. It operates by only updating the minimum set of rows in the CountMin sketch. The update function for each row j for the event i is:

$$\mathit{count}[j, h_j(i)] = \max(\mathit{count}[j, h_j(i)], \min_{k<d}(\mathit{count}[k, h_k(i)]+1))$$

Unfortunately this technique can only be applied where sketch updates are only positive. When attempting to remove feature i from the time window, the method cannot determine which rows should be decremented and which should not. We will not consider this heuristic in our experiments.

Cormode and Muthukrishnan [8] has shown that this structure provides several guarantees about the estimated frequencies. First, the estimate $\hat{a}_i$ is always an over-estimate of the true frequency; $a_i \leq \hat{a}_i$. This guarantee means that no frequent text feature will falsely identified as infrequent. Second, with probability $1 - \delta$, $\hat{a}_i \leq a_i + \epsilon\|w\|$. This guarantee provides a probabilistic limit on the number of infrequent features that we falsely identify as frequent.

3.2 LinearCountMin Sketch

We now present the LinearCountMin Sketch. This is a new data structure similar to the CountMin Sketch. The underlying data structure is changed from a table to a one dimensional array of counts of width $v \cdot d$. Where v and d are selected in the same manner as in the CountMin Sketch. Each of the d hash functions should be selected from a family of fully independent hash functions. Each of these hash functions are now used to update this array of counters. We present this structure as it provides simple, distributable implicit and explicit implementations.

We now introduce a new collision reduction heuristic. It is possible that two of the hash functions we defined collide for some particular feature i. We revise the update procedure to only update each cell only once. Let $count'[h_j(i)]$ be the value of the count cell prior to the update.

$$\forall_{j<d} : count[j, h_j(i)] = count'[j, h_j(i)] + 1$$

If we assume all hash functions are independent, we can now show that this data structure has error guarantees that are identical to the original CountMin Sketch. For the purposes of this proof, we view the stream of features for a particular time window as a sequence of updates, w. Where each update $i \in w$ is an element of the vocabulary of the current time window. Each update $i \in w$ has a corresponding a frequency a_i. The pairwise nature of the additions and removals for the structure allows us to guarantee that all count cells are positive.

Theorem 2: *The estimate $\hat{a}_i$ has the following guarantees from the LinearCountMin sketch: $a_i \leq \hat{a}_i$, and with probability at least $1 - \delta$, $\hat{a}_i \leq a_i + \epsilon \cdot \|w\|$, where w is a set of features inserted in the sketch and a_i is the true count of feature i.*

PROOF. Let $count_i$ be a duplicate array of counters, except that this array was not updated with the data from feature i. This array can be interpreted as an array of possible collision values for the feature i.

Let $I_{i,k}$ be an indicator variable for the expression $count_i[k] > \epsilon \cdot \|w\|$. This variable indicates if a particular cell k, is overfilled with respect to the expected error for i. We can estimate an upper bound on the total number of cells with this property as the total amount of data in the counter array divided by the minimum value that allows the property to hold.

$$\sum_k I_{i,k} \leq \frac{d \cdot \|w\|}{\epsilon \cdot \|w\|} = \frac{d}{\epsilon}$$

Given that a hash function j will uniformly at random select a counter cell for the feature i, we can compute an upper bound on probability that the selected cell has the property I. We compute this upper bound as the maximum number of cells with the property I divided by the total number of cells.

$$\Pr[I_{i,h_j(i)}] \leq \frac{d}{\epsilon} \div \frac{d \cdot e}{\epsilon} = \frac{1}{e}$$

Given that all hash functions are fully independent, we can compute an upper bound on the probability that all hash functions select cells with this property.

$$\begin{aligned}\Pr[\forall_j\, I_{i,j}] &= \prod_j \Pr[I_{i,h_j(i)}] \\ &\leq \prod_j \frac{1}{e} \\ &= \delta\end{aligned}$$

Now we can show the final part of the proof:

$$\begin{aligned}\Pr[\hat{a}_i > a_i + \epsilon\|w\||] &= \Pr[\forall_j\, count[h_j(i)] > a_i + \epsilon\|w\||] \\ &= \Pr[\forall_j\, a_i + count_i[h_j(i)] > a_i + \epsilon\|w\||] \\ &= \Pr[\forall_j\, count_i[h_j(i)] > \epsilon\|w\||] \\ &= \Pr[\forall_j\, I_{i,h_j(i)}] \\ &\leq \delta\end{aligned}$$

□

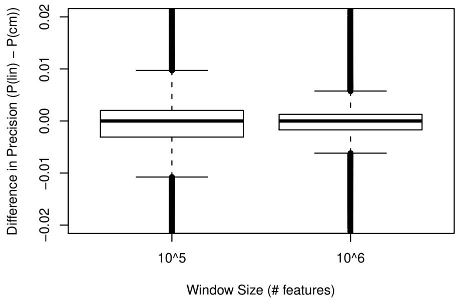
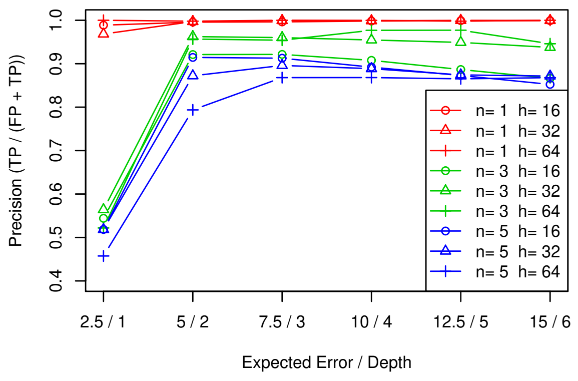


Figure 4: Comparison of CountMin and LinearCountMin Sketch data structures. Shown is the difference in precision for a range of parameters. $n \in \{1, 3, 5\}$, $h \in \{4, 16, 64\}$, $\epsilon \in \{(0.3 \cdot h/w), (0.5 \cdot h/w), (0.9 \cdot h/w)\}$, $\delta \in \{1/e^2, 1/e^4, 1/e^6\}$. All permutations of these parameters were tested. Data used for this experiment are subsets of Clueweb-B.

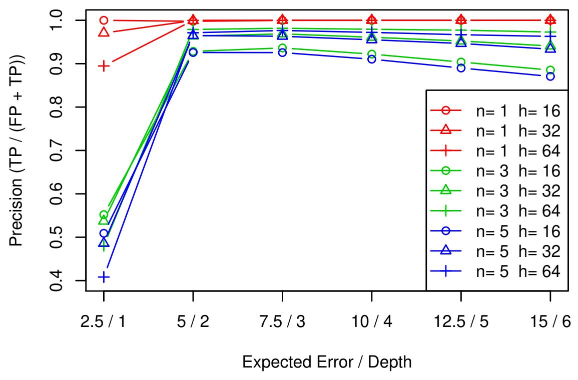
3.3 Experiments

In this section we analyze the performance of the CountMin and LinearCountMin sketch for the task of identifying frequent n-grams extracted from sequences of English text. Each of these experiments measures the precision of the identification of frequent n-grams. Precision is calculated as the true number of frequent n-grams divided by the number of n-grams identified by the structure as frequent. This is an appropriate measure of performance as we are guaranteed that neither of these structures will omit any frequent n-grams. We are guaranteed that $\hat{a}_i \geq a_i$ for both of these structures.

First, we investigate the difference between the performance of the CountMin and LinearCountMin structures. Figure 4 shows the distribution of differences in precision between the two data structures for our task. Data points shown in the graph have been produced using a range of possible parameter values. This graph shows that the two methods produce very similar performance, with the mean difference between the performances very close to 0. The actual differences in the mean values for each window size was calculated to be 0.0005 for a window of size 10^5 features and -0.0003 for the windows of size 10^6 features. Both of these observed differences was found to be statistically significant using the paired-t-test, ($p < 0.05$). These results show that the measured precision, with respect to this application, of each of these structures is approximately equal.



(a) Subset of TREC ClueWeb-B Data, $S \approx 1 \cdot 10^5$



(b) Subset of TREC ClueWeb-B Data, $S \approx 1 \cdot 10^6$

Figure 5: Graphs show the trade off between width and depth in LinearCountMin Sketches. The total number of cells in the data structure is constant for all data points in each graph. Expected Error is computed as $\epsilon \cdot w$. Each data point is averaged over 5 repeated trials.

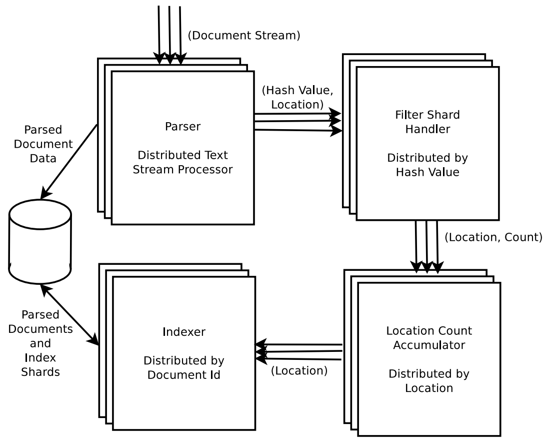
Next, we find the optimal balance between ϵ and δ in the context of tracking the frequency of text features. We test this by using a static number of counters and adjust the ratio between the width and the depth of the CountMin and LinearCountMin structures. We measure the precision of the LinearCountMin sketch for the identification of frequent n-grams, for a range of n, h, and w. Figure 5 shows that the optimal ratio between δ and ϵ for this data set occurs where the depth is 2.

We choose to use the LinearCountMin Sketch in our system because it is it provides identical precision for this task as the CountMin Sketch, and it provides a more efficient distributed implementation for a cluster computing platform.

4. SYSTEM DESIGN

We now detail the system design that we will be investigating. Using this system, we can analyze the performance of a filter data structure for our task. This system is designed to naturally provide a scalable implementation within a distributed computing environment. Our system is composed of four components; parser, filter shard handler, location count accumulator, and indexer.

The first component reads data from the document stream. Each document is parsed and tokenized. The tokenized document is copied to a persistent storage location. Textual features are then extracted from the tokenized document and d hash values are computed using the hash functions associated with the filter data structure. Each of the hash values and the location of the text feature are sent to the filter shard handler component. By sending only hash values and locations, the size of the communication between the parser and the filter shard handler does not depend on the size of the text feature.



(Document Stream)
Parser
Distributed Text Stream Processor
(Hash Value, Location)
Filter Shard Handler
Distributed by Hash Value
Parsed Document Data
(Location, Count)
Parsed Documents and Index Shards
Indexer
Distributed by Document Id
(Location)
Location Count Accumulator
Distributed by Location

Figure 6: Filtered dynamic index system design. Each part may be distributed across a shared-nothing parallel computing environment.

The second component maintains the filter data structure. For each received hash value and location, the filter data structure is updated. The updated frequency estimate for each hash value is extracted and sent to the frequency estimator component. This component also maintains the time window. It does this using a circular buffer. As hash values are added to the buffer, the oldest hash values are removed and the filter data structure is updated.

The third component classifies locations as frequent or infrequent. It collects d counts for each location. Once all counts have been collected, the minimum count in this set is checked against the threshold. Frequent locations are passed on to the indexer component. In a single threaded environment this component can be implemented as part of the previous component. In this case no additional memory is required.

When the indexer receives confirmation that a location is frequent, the textual feature is extracted from the stored document data and added to the dynamic index structure. The details of this index structure are beyond the scope of this paper. We use the geometric indexing approach presented by Lester et al. [15].

5. PARALLEL SYSTEM DESIGN

We make the claim that this system design can be distributed within a parallel computing environment in an efficient and scalable manner. An overview of this system is shown in Figure 6. There are two key considerations for a parallel implementation. First, we need to distribute the work load evenly across the nodes for each component. Second, we need to ensure that the inter-component communication does not become a system bottleneck.

We will first consider how to distribute data to each component. The aim is to ensure that each instance of each component will process approximately the same amount of data. The parser is distributed by assigning documents from the document stream to instances using a round robin algorithm. This will ensure each node

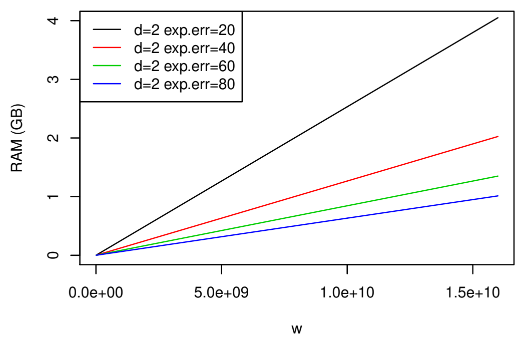


Figure 7: Implicit filter RAM requirements as a function of time window size for a range of structure depths and expected error rates. We chose to show the expected error rate instead of ϵ, $\epsilon = exp.err/w$.

will process roughly the same amount of data.

The filter shard handler component is distributed by hash value ranges. Each node maintains a fraction of the CountMin sketch corresponding to a range of hash values. This method distributes the space requirements of this part equally across all instances of the component. However, a side effect of this decision is that the distribution of input data may be skewed.

Previous research has observed that the distribution of English text using terms as hash keys will produce a skewed work load, see Moffat et al. [19]. However, in this system we are using extracted text features as hash keys. As shown in Section 2, extracted text features tend to have much a larger vocabulary than terms. The larger vocabulary drastically reduces the imbalance in the work load distribution. We empirically test this claim in Section 7.2.

The location count accumulator is distributed using locations as hash keys. Locations contain both the source document identifier and the offset information for the text feature. Unlike the previous stage, there is no skew in amount of data associated with each location; there are d counts associated with each location. Assuming an appropriate hash function is used to distribute locations it is fair to expect an equal load distribution.

Finally, the instances of the indexer component are distributed by randomly assigning document identifiers. Previous research indicates that this method of data distribution leads to an approximately equal distribution of retrieval load, see Moffat et al. [19].

Inter-component communication can be a bottleneck in parallel systems. We have made several design decisions to avoid this problem. First text features themselves will never be communicated between components. Thus we can ensure that the communication is not dependent on the size of the text feature. Second, the inter-component communication using these distribution schemes will be non-duplicated and non-redundant. Finally, all communication may be buffered and processed asynchronously in batches. This measure reduces the network packet overhead costs.

6. SYSTEM IMPLEMENTATION

We consider two implementations of the LinearCountMin Sketch, through implicit and explicit representations of the structure. An explicit representation must store the structure directly. An implicit representation consists of a sorted set of hash values and their associated non-zero count values. There are some significant differences in space usage and processing throughput between these two representations.

Updating the explicit structure depends on the number of hash functions being used by the Sketch, it requires $O(d)$ time. Batch processing sets of features will not improve throughput.

The RAM requirements for the explicit filter structure can be determined specifically. The width of the CountMin sketch is defined to be $\frac{e}{\epsilon}$, where $\epsilon < \frac{h}{w}$. It is clear that RAM usage is linearly related to the size of time window. Some actual RAM requirements are shown in Figure 7. Note that the largest window size considered in this graph is approximately the same size as the GOV2 collection.

The implicit LinearCountMin Sketch consists of a sorted set of hash values. This structure performs best using batch updates. Each batch update requires a linear pass over the set of hash values, this requires $O(d \cdot |w|)$ time. This cost is amortized over the size of the batch. The key advantage of an implicit structure is that it can be stored on disk instead of memory and still maintain reasonable throughput. This allows the use of large hash tables and therefore permits the system to use very large window sizes with low threshold values.

Unfortunately, batch processing adds a delay between the observation of a frequent feature and indexing. For some applications, this delay could be a problem. For example, Twitter data should not be buffered longer than a few minutes. For this type of data we would expect the buffer size to be a very small fraction of the time window. We will investigate how this ratio affects throughput in the next section.

7. SYSTEM TESTING

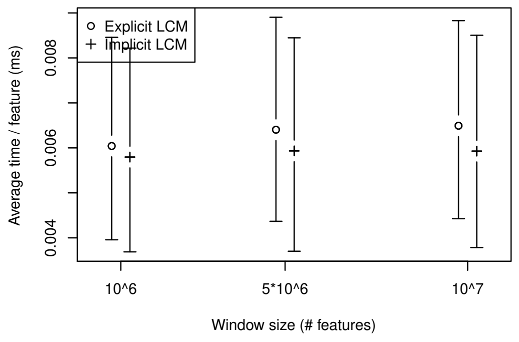
7.1 Single CPU Implementation

We will now perform experiments on the processing speed of both the implicit and explicit versions of the LinearCountMin structure for various sized time windows, n-grams, and thresholds. Figure 8 shows the results from these experiments. All noticeable differences between the mean values displayed in these graphs are significantly different using the two sample t-test.

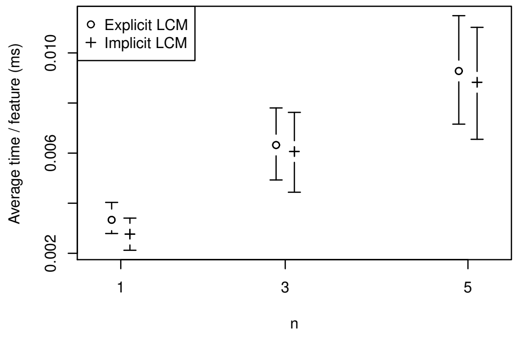
In each of the graphs in Figure 8, we investigate how parameters of our system influence the throughput. For each of these experiments we test all permutations of a range of parameter values: $n \in \{1, 3, 5\}$, $h \in \{4, 16, 64\}$, $\epsilon \in \{(0.3 \cdot h/w), (0.5 \cdot h/w), (0.9 \cdot h/w)\}$, $\delta \in \{1/e^2, 1/e^3, 1/e^4\}$. For each set of parameters, we compute the average time to perform 10^6 filter updates, where a filter update consists of one inserted and one removed feature instance. We replicate these experiment $2,000$ times. In all of these experiments the implicit filter structure is stored in memory instead of on disk.

Figure 8 (a) shows the impact of window size on the throughput of each system. We can see that the implicit filter system is not dramatically affected by the size of the time window. However, the explicit filter system shows a slow increase in the average processing time of a text feature as the time window grows. The ratio between the buffer size and the time window was held static at $1/10$ for all of these experiments. This is an important result, it shows that both systems will scale to large time windows relatively effectively.

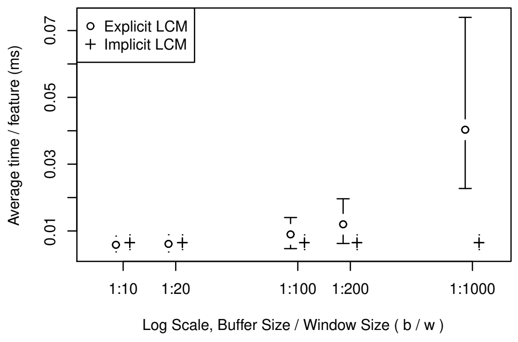
Figure 8 (b) investigates the effect of the buffer size on the throughput of the implicit system. This figure shows how throughput is affected by as the ratio between the buffer and the time window changes. We observe a clear correlation between the buffer ratio and the throughput. This graph shows that an implicit filter implementation is not suitable for high volume text streams, with a frequent update schedule.



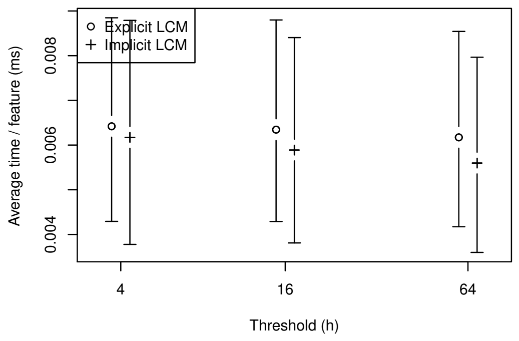
(a) Subsets of Clueweb-B, $w \in \{10^6, 5 \cdot 10^6, 10^7\}$



(c) Subsets of Clueweb-B, $w \in \{10^6, 5 \cdot 10^6, 10^7\}$



(b) Subsets of Clueweb-B, $w = 10^7$



(d) Subsets of Clueweb-B, $w \in \{10^6, 5 \cdot 10^6, 10^7\}$

Subsets of TREC ClueWeb-B Data

Figure 8: Comparison of implicit and explicit LinearCountMin sketch implementations for different window sizes. Shown is the 1st quartile, mean, and 3rd quartile of the measured times to update a LinearCountMin sketch. Parameter settings used to generate data: $n \in \{1, 3, 5\}$, $h \in \{4, 16, 64\}$, $\epsilon \in \{(0.3 \cdot h/w), (0.5 \cdot h/w), (0.9 \cdot h/w)\}$, $\delta \in \{1/e^2, 1/e^3, 1/e^4\}$. In all plots except (a), the buffer size for the implicit data structure is 10% of the size of the time window w.

Figure 8 (c) investigates the effect of the size of the vocabulary on the throughput of each system. Recall that the vocabulary size can be controlled using the size of n-grams. We infer from this data that the throughput of each implementation drops as the vocabulary size increases.

Finally, Figure 8 (d) investigates the effect of the threshold parameter on throughput. We can see that this parameter does not dramatically alter the throughput of the implicit system. As the threshold is increased, the throughput of the explicit system increases. This is likely to be caused by the impact of this parameter on the width of the data structure. Note that we have set ϵ as a fraction of the threshold value in each of these experiments.

7.2 Parallel Implementation

As discussed previously load balancing is a critical factor in any distributed system. We perform two experiments to test the load balance and scalibility of a parallel implementation of our system.

First, we investigate the load imbalance produced by the distribution of hash values in the filter shard handler component. We measure the imbalance of a component instance to be the ratio between the observed number of has values and the expected number of hash values. The imbalance present in the system is measured

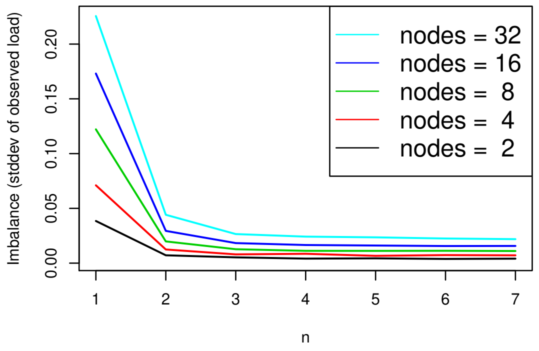


Figure 9: Load imbalance in parallel implementation. Data used for this experiment consists of subsets of Clueweb-B. All runs used window size: $w = 10^6$.

as the standard deviation of the imbalance in each node. For this experiment we use a window of size 10^6, threshold $h = 16$, depth

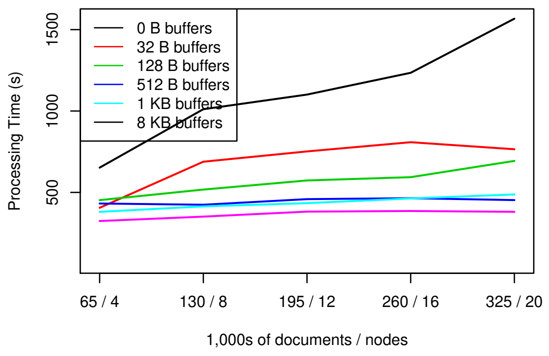


Figure 10: System processing times for parallel implementation with a range of network communication buffer sizes. Reported times are the average of 20 timed runs. Ratio between the number of nodes and the number of documents, $16,250 : 1$, is constant for all data points in this graph. Data used for this experiment consists of subsets of Clueweb-B. All runs used window size: $w = 10^6$.

of the CountMin Sketch $d = 2$, and error rate $err = 8$.

Figure 9 shows the load imbalance for various values of n and various distribution factors. We make two observations, the imbalance is reduced when indexing modest sized text features, and the imbalance increases with the number of component instances. From this data we can conclude that an average component instance will have $\pm 5\%$ deviation in work load.

The second experiment tests the scalability of the system in a parallel environment. For this experiment we measure the time required to index the text features. The number of documents to be indexed is increased as the number of nodes available is increased. A scalable system should demonstrate only a minor decrease in processing time as the number of nodes available and the number of documents to be indexed is increased. This means that if the number of documents is doubled, then the number of processing nodes is also doubled and the processing time should remain constant. Network communication overhead in this system is controlled by buffering data packets into specific sizes. The size of the buffer is investigated in this experiment.

This experiment was performed on a non-dedicated cluster of 32 dual core machines, each with $4GB$ of RAM. Network attached storage was used to store the parsed documents. System parameters for this experiment are: $w = 10^6$, $d = 2$, $h = 16$, $err = 8$.

Figure 10 shows processing times for the parallel implementations. Reported times are the average of 20 identical runs. The ratio between the number of documents processed and the number of computing nodes was kept constant at $16,250 : 1$. This graph shows that the system is scalable where network communication uses blocks of size 128 bytes or larger.

8. RELATED WORK

There have been several previous papers on the topic of indexing of text features. Williams et al. [20] presents the nextword index. This index allows n-grams or phrases to be accessed using a pairwise strategy. This index has been used to significantly improve phrase retrieval performance [1]. We assert that providing an index over larger n-grams should result in further efficiency improvements for phrase queries, For some phrase of length k, $k - 1$ disk accesses would be required within the nextword index, whereas $k - (n - 1)$ disk accesses would be required for the n-gram index.

Chang and Poon [6] modify nextword indexes to index arbitrary length phrases. The vocabulary of their index is restricted to a set of *common* phrases. This type of index is similar to frequent indexes of text features. Their paper does not detail the costs of indexing phrases, nor the process of identifying *common* phrases to index. We believe that their method relies on a static list of phrases to index, thus in a dynamic setting their method would not discover important new phrases.

There have been two previous papers on the topic of *frequent* indexes. We have already discussed the indexing techniques presented by Huston et al. [14]. Bernstein and Zobel [3] present SPEX, an n-pass approach to indexing repeated n-grams. Their approach relies on the construction of a series of in-memory filters using hash tables. The k^{th} filter in their system identifies k-grams that occur at least twice, and the filter is constructed using the $(k-1)^{th}$ filter. Their approach is not suitable for the problem of dynamically indexing frequent text features for the same reasons that the *frequent* indexing approaches investigated by Huston et al. [14] are unsuitable. SPEX requires several passes over the document collection and it is not possible to detect and index newly frequent n-grams.

There are a variety of dynamic indexing approaches that are relevant to this work. Lester et al. [16] compare baseline techniques to their geometric partitioning dynamic indexing technique. This technique is shown to be an effective method of balancing the tasks of inserting new documents and merging inverted index shards while maintaining efficient retrieval speed. Büttcher et al. [5] present a hybrid dynamic indexing strategy that separates shorter posting lists from longer posting lists. This approach avoids repeated copying of the longer lists. Yamada and Toyama [21] present a method of performing parallel dynamic index construction over a multi-core platform. This work shows the first steps towards a full parallel dynamic index system on a shared-nothing cluster. Guo et al. [13] present a balanced tree based indexing strategy. This method is designed for applications where-in document deletions are commonplace. Each of these dynamic indexing algorithms may be used in conjunction with our filtered dynamic indexing system.

There are a variety of stream processing algorithms designed to find 'heavy hitters' or frequent items from a stream of data. Cormode and Hadjieleftheriou [7] provide a good summary of some of the recent work being conducted on this generalized problem. The first section of the paper focused on *counter* techniques (see [17, 11]). The second section looked at the more general problem of returning items within a given quartile. The third section focused on sketch based techniques including the CountMin Sketch that we investigate in this paper.

As specified in the introduction, we have placed several restrictions on possible filter data structures. This makes many of the methods discussed in this paper unsuitable as filter data structures. Each of the techniques from the first and second sections tracks a subset of the vocabulary. Each of these data structures stores text features verbatim, thus their space requirements are dependent on the size of the textual features being indexed. So, we can eliminate each of these data structures as they violate one of our key requirements.

If there is insufficient system resources to buffer the text stream data from the time window on disk, we could consider algorithms that simulate the effect of the sliding window on the data structure. Datar et al. [10] present a method of reducing counts based upon a set of timestamp buckets. Braverman and Ostrovsky [4] present a second approach based on smooth histograms. This second approach uses a similar bucket-based technique, but views the first bucket as an approximation of the tail buckets. We expect that these approaches would be useful when the disk space requirement could not be satisfied.

9. CONCLUSIONS

In this paper, we have proposed a novel system for the dynamic indexing of frequent text features. We have shown that our system provides several important guarantees.

First, the proposed system is lossless; all frequent text features appear within the index. Second, the space requirements of the system are not determined by the size of the text feature. The RAM requirements of our system scales linearly with respect to the size of the time window. Finally, we have shown that the system scales efficiently in a parallel computing environment.

We have proven that the LinearCountMin sketch provides identical guarantees as the CountMin sketch. While there is almost no difference in the accuracy of each structure, the LinearCountMin sketch provides a more efficient distributed implementation.

We have tested the maximum throughput in a single threaded environment and showed that the throughput of the filter structure is independent of the size of time window and the threshold value. We observe that the throughput is inversely correlated with the size of the vocabulary. It should be noted that the vocabulary is strictly limited by the time window size. In our single machine tests, we are able to process between 100 and 300 features per millisecond, depending on the parameters and used. This means we can process one thousand documents, each consisting of $\sim$ 500 terms, every two to five seconds on a single machine.

Finally, we have tested this system within a parallel computing environment. We have seen that there is a load imbalance in the filter component. But this imbalance does not dramatically hinder the system's performance. Finally, we have observed that the throughput of the system scales effectively.

Acknowledgment. This work was supported in part by the Center for Intelligent Information Retrieval and in part by NSF grant #IIS-0934322. Any opinions, findings and conclusions or recommendations expressed in this material are the author(s) and do not necessarily reflect those of the sponsor.

References

[1] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Efficient phrase querying with an auxiliary index. In *Proceedings of the 25th Int. ACM SIGIR Conf.*, pages 215–221, 2002.

[2] Michael. Bendersky, Don. Metzler, and W. Bruce Croft. Learning concept importance using a weighted dependence model. In *Proc. 3rd ACM Int. Conf. Web Search and Data Mining*, pages 31–40, 2010.

[3] Y. Bernstein and J. Zobel. Accurate discovery of co-derivative documents via duplicate text detection. *Information Systems*, 31:595–609, 2006.

[4] Vladimir Braverman and Rafail Ostrovsky. Smooth histograms for sliding windows. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 283–293, 2007.

[5] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 356–363, 2006.

[6] Matthew Chang and Chung Keung Poon. Efficient phrase querying with common phrase index. *Inf. Process. Manage.*, 44:756–769, March 2008.

[7] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19:3–20, 2010.

[8] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[9] W. Bruce Croft, Don Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley, USA, 2009. ISBN 9780136072249.

[10] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31:1794–1813, 2002.

[11] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *In Proceedings of the 10th Annual European Symposium on Algorithms*, pages 348–360. Springer-Verlag, 2002.

[12] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[13] Ruijie Guo, Xueqi Cheng, Hongbo Xu, and Bin Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 751–760, 2007.

[14] Samuel Huston, Alistair Moffat, and W. Bruce Croft. Efficient indexing of repeated n-grams. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 127–136, 2011.

[15] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast online index construction by geometric partitioning. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, 2005.

[16] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33:19:1–19:33, September 2008.

[17] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[18] Don Metzler and W. Bruce Croft. A Markov random field model for term dependencies. In *Proc. 28th Ann. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 472–479, Salvador, Brazil, 2005. ACM Press, NY.

[19] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, 2006.

[20] Hugh E. Williams, Justin Zobel, and Phil Anderson. What's next? - index structures for efficient phrase querying. In *Proc. Australasian Database Conference*, pages 141–152, 1999.

[21] Hiroyuki Yamada and Motomichi Toyama. Scalable online index construction with multi-core cpus. In *Proceedings of the Twenty-First Australasian Conference on Database Technologies - Volume 104*, pages 29–36, 2010.