Efficient Indexing of Repeated n-Grams

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

Alistair Moffat
Department of Computer
Science and Software
Engineering
The University of Melbourne
Victoria 3010, Australia
ammoffat@unimelb.edu.au

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

ABSTRACT

The identification of repeated n-gram phrases in text has many practical applications, including authorship attribution, text reuse identification, and plagiarism detection. We consider methods for finding the repeated n-grams in text corpora, with emphasis on techniques that can be effectively scaled across a cluster of processors to handle very large amounts of text. We compare our proposed method to existing techniques using the 1.5 TB TREC ClueWeb-B text collection, using both single-processor and multi-processor approaches. The experiments show that our method offers an important tradeoff between speed and temporary storage space, and provides an alternative to previous approaches that scales almost linearly in the length of the sequence, is largely independent of n, and provides a uniform workload balance across the set of available processors.

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

1. INTRODUCTION

Many important tasks in information retrieval require the identification and indexing of n-word phrases or n-grams. For example, similarity search in large collections uses n-grams as one of the important features in computing a document's score (see Metzler and Croft [2005] and Croft et al. [2009]). Another example is the identification of reused, duplicated, or plagiarized text in documents. Techniques developed for this task rely heavily on n-gram indexing to improve efficiency for reuse discovery [Manber, 1994, Bernstein and Zobel, 2006, Seo and Croft, 2008].

The question as to how to identify the repeated n-word phrases in a corpus of text is, at face value, simple. All that is required is that the text be parsed, that each of the n-grams be indexed using some kind of dictionary data structure, and then the ones that occur more than once (or in general, more than m times) be identified and reported.

Thwarting this simple approach is the reality of dealing with large volumes of data. Consider, for example, a (mere) gigabyte of text. Allowing for a modest amount of markup, it probably contains around 100 million words over a vocabulary of maybe 1 million distinct words, and hence contains 100 million 5-grams, each of which requires (say) 15 bytes to represent, presuming that words are represented as 3-byte integers. All of these 5-grams are potentially different, so any dynamic data structure for managing them must be capable of storing 100 million objects, each of which requires a total of 18 bytes of storage to keep the n-gram itself as well as a 3 byte location. Including the pointer overheads associated with a dynamic search structure, as much as 3 GB of random access storage might be required. This might be just feasible on current commodity platforms, but when $n = 10$ and the collection contains a terabyte rather than a gigabyte, the problem is challenging, even on high-end machines. Nor does this simple approach offer a parallel implementation, since partitioning the data across a set of processing nodes and having each report on their duplicates in a localized sense fails to identify inter-node repetitions.

More complex methods are thus required. This paper describes a variety of techniques that have evolved, and then introduces a new approach that has the particular benefit of being amenable to a distributed implementation. Experiments with 1.5 TB of text show that the proposed approach does indeed provide an important balance between execution speed, execution-time random-access memory, and temporary disk space requirements.

2. FINDING REPEATED N-GRAMS

We commence by examining some well-known "textbook" approaches to the problem of finding repeated n-grams. It is assumed throughout that the source document has been parsed into words, and that those words have been converted (via a corresponding vocabulary) into a stream of integer identifiers. We thus assume, as an abstraction, that the sequence S that is to be processed consists of $N + n - 1$ integer word identifiers, each in the range 1 to $|V|$. That is, sequence S is a representation of text that contains N n-grams.

2.1 The simple algorithm

The in-memory dictionary-based algorithm sketched above is

Algorithm MEMORY-BASED ONE-PASS

```
for $i \leftarrow 1$ to $N$ do
    $s \leftarrow S[i \cdots (i+n-1)]$
    $D[s] \leftarrow D[s] \cup \{i\}$
end for
for $s \in \text{domain}(D)$ do
    if $|D[s]| \geq m$ then
        output $(s, D[s])$
    end if
end for
```

Algorithm DISK-BASED ONE-PASS

```
 1: for $i \leftarrow 1$ to $N$ do
 2:     $s \leftarrow S[i \cdots (i+n-1)]$
 3:     append $(s, i)$ to the output file $F$
 4: end for
 5: sort $F$, coalescing paired entries $(s, \ell_1)$ and $(s, \ell_2)$ to
    $(s, \ell_1 \cup \ell_2)$ as soon as they are identified
 6: for $(s, \ell) \in F$ do
 7:     if $|\ell| \geq m$ then
 8:         output $(s, \ell)$
 9:     end if
10: end for
```

Algorithm HASH-BASED TWO-PASS

```
 1: for $i \leftarrow 1$ to $N$ do
 2:     $s \leftarrow S[i \cdots (i+n-1)]$
 3:     $h \leftarrow hash_{b_1}(s)$
 4:     append $(h, 1)$ to the output file $F_1$
 5: end for
 6: sort $F_1$, coalescing paired entries $(h, f_1)$ and $(h, f_2)$ to
    $(h, f_1 + f_2)$ as soon as they are identified
 7: for $(h, f) \in F_1$ do
 8:     if $f \geq m$ then
 9:         append the $b_2$ low-order bits of $h$ to the output file $F_2$
10:     end if
11: end for
12: sort $F_2$, discarding any duplicates
13: $H \leftarrow \text{read}(F_2)$
14: for $i \leftarrow 1$ to $N$ do
15:     $s \leftarrow S[i \cdots (i+n-1)]$
16:     $h \leftarrow hash_{b_2}(s)$
17:     if $h \in H$ then
18:         append $(s, i)$ to the output file $F_3$
19:     end if
20: end for
21: apply steps 5 to 10 of DISK-BASED ONE-PASS to the file $F_3$
```

codified in MEMORY-BASED ONE-PASS. In this procedure, D is a dictionary data structure that is indexed by n-grams, with $D[s]$ storing the set of offsets in S at which the n-gram s appears. In practice, D is a hash table or a search tree of some sort, each element of which is a linked list or variably-sized array. In total, D will require impossibly large amounts of main memory, typically something like n times as much as is required by the sequence being processed. So when N (where N is the number of n-grams) is a million, algorithm MEMORY-BASED ONE-PASS is tractable; but when N is a billion, it probably is not.

2.2 A file-based approach

The first (and widely-known) improvement to the simple mechanism is to use a post-processing sorting phase to bring together the occurrences of each n-gram, rather than require a dynamic data structure. In this process, described as algorithm DISK-BASED ONE-PASS, every possible n-gram in the sequence S is added to an output set F, stored on disk as a sequential file. That file is then sorted into n-gram order, and like n-grams collected together to form the index. Sorting is a well understood problem, even when only sequential-access storage is available, and in this approach the per-item cost of sorting approximately corresponds to the cost of searching D in Algorithm MEMORY-BASED ONE-PASS, depending on which exactly data structure is used to represent D.

The drawback of this DISK-BASED ONE-PASS approach is that F contains one record for each of the N n-grams present in the sequence S, meaning that the peak disk space required is as much as $n + 1$ times as big as S, assuming that an address component takes (at least) the space of one integer. Moreover, this amount of space is required regardless of the degree (or otherwise) of repetition. But no random-access data structures are required – the space required by the dictionary D of the MEMORY-BASED ONE-PASS algorithm is moved to disk, and processed sequentially via the sorting algorithm, rather than as an in-memory search structure.

If merge-based sorting is used, and a cascading coalescing step undertaken every time an in-memory output buffer of moderate size is filled with grams, the peak disk space requirement might be reduced, because frequently-occurring grams will be stored only once. But even so, the space cost must be proportional to N, since every gram address is stored in the sorted output file prior to the final post-processing step that identifies the ones that occur more than m times.

2.3 Hashing, and two passes

The next method, HASH-BASED TWO-PASS, builds on two independent observations – first, that hash-derived surrogates can be stored instead of the n-grams, saving space; and second, that the actual locations of the n-grams need only be collected once the values of the repeated n-grams have been identified. The key idea is that the probabilistic hash filter H certainly contains the hash-surrogate of every n-gram that *does* occur more than m times in S, and *might* contain the hash-surrogate of some n-grams that do not.

The first for loop at steps 1 to 5 processes the sequence and generates the n-grams. A relatively wide b_1-bit hash value for each is computed via the function $hash_b(s)$, and recorded to an output file F_1, together with an initial frequency count of one. That is, each n-gram s is condensed into a non-unique b_1-bit string on a many-to-one basis, and that hash string is used as a fixed-width *hash-surrogate*. When b_1 is large – for example, 40-bits or more – the number of collisions will be small. File F_1 is then processed to generate a set of hash values that correspond to n-grams that might occur more than m times in S (steps 6 to 11), and that set of hash values used (step 17) to (perhaps greatly) reduce the number of full n-grams that get processed by step 21. In generating the file F_2, only a b_2-bit suffix of the hash bits, where $b_2 < b_1$, are used, in a delicate balance between collision probability and space required by the table H.

In terms of disk space used, file F_1 still contains as many as N records, even if the sorting and coalescing process indicated at step 6 is carried out in a cascading block-interleaved manner. But now each record is a b_1-bit integer plus a frequency count (the latter requiring only $\lceil \log_2(m+1) \rceil$ bits), and for $b_1 = 40$ (say, the choice of b is discussed below) and $m \leq 255$ (say), the requirement for file F_1 is at most $6N$ bytes. File F_2 is never larger than F_1, and so does not affect the peak disk space requirement; as was already noted, the reason for taking only $b_2 < b_1$ bits of hash into the file F_2 is to control the amount of memory required for the table H.

The third disk file, F_3, is potentially very large – for pathological sequences, as large as the approximately $(n+1)N$ *words* required by algorithm DISK-BASED ONE-PASS. But note that it only contains information about n-grams that either (a) do indeed appear in S more than m times; or (b) by chance, hash to a value that corresponds to some other n-gram(s) that collectively appear more than m times in S. That is, provided that the number of false positives is controlled, the size of F_3 is primarily determined by the total number of occurrences in S of n-grams that appear more than m times, and (disregarding any compression that might be applied to the final index) is of the same magnitude as the n-gram index that is being generated.

In an implementation, the set H can be stored as a bitvector of 2^{b_2} bits for $O(1)$-time random access (which is, of course, impractical when b_2 is larger than around 32 or so); or can be stored as a sorted array of b_2-bit integers with a logarithmic access cost; or can be stored using a compressed queryable structure that requires less space than either of these two alternatives, while still providing logarithmic-time lookup. Sanders and Transier [2007] describe one such hybrid mechanism.

For small files it may be possible to take $b_1 = b_2$. For larger files, having $b_1 > b_2$ ensures that the first-pass filter can be highly discriminating (giving rise to only a small number of collisions); while the smaller value b_2 is chosen so that the second-pass filter H can be accommodated in main memory. This "two b" strategy still gives rise to collisions between repeats and non-repeats during the second pass, but at least prevents the great majority of the first-pass false positives that would arise from non-repeats colliding with non-repeats. (Note that there are many more non-repeats than there are repeats.)

To gauge the extent to which these various costs become bottlenecks, consider a scenario in which a sequence of $N = 10^9$ words is being processed (1 billion, corresponding to around 10–20 GB of HTML data), and that $n = 9$. Further, suppose that on average one n-gram in five in the file occurs more than m times, and that the average number of repetitions for n-grams that do reoccur is 4. If these estimates are appropriate, then there are 5×10^7 distinct repeated n-grams. Under these assumptions, and with $b_1 = b_2 = 40$, file F_1 might be as large as 6 GB; file F_2 contains a little more than 5×10^7 entries (the extras arising because of the probabilistic nature of the filter H) and occupies around 250 MB; and file F_3 contains a little over 2×10^8 entries, each occupying 10 words (40 bytes), for a total of 8 GB. As well, during the second pass, file F_2 must be present in memory as an array-based lookup structure H, meaning that of the order of 250 MB of main memory is required through steps 14 to 20 of HASH-BASED TWO-PASS. In terms of disk traffic, several gigabyte-sized input files must be sequentially processed, including being sorted.

The final n-gram index must then include 5×10^7 9-gram descriptions, plus, for each of them, an average of four addresses in the sequence, for a total uncompressed requirement of approximately 2.6 GB. Non-trivial savings arise once the list of n-grams is compressed, since they can be stored as incremental differences relative to each other [Witten et al., 1999].

The scenario portrayed in the two previous paragraphs is clearly within the limits of what might be achieved with a single "standard" computer; and the data listed in Table 1(a), derived from Newswire data, confirms the appropriateness of the estimates. For less well curated data, the situation is not so tolerable, and parts (b) and (c) of Table 1 show what happens when the dataset contains repeated documents. Significant repetitions of long sequences result, and in excess of one third of the n-grams in the sequence appear more than once.

n	$N = 250 \times 10^6$			$N = 500 \times 10^6$		
	single	multi	repeat	single	multi	repeat
1	0.1	0.2	99.7	0.1	0.1	99.7
2	5.5	3.7	90.7	4.6	3.1	92.3
3	27.3	9.2	63.4	24.5	8.7	66.8
4	52.9	10.0	37.1	50.6	10.2	39.2
5	68.4	8.4	23.2	67.8	8.7	23.6
6	75.8	7.0	17.2	76.2	7.3	16.6
7	79.3	6.3	14.4	80.2	6.4	13.4
8	81.4	5.8	12.8	82.3	5.9	11.7
9	82.8	5.5	11.7	83.7	5.6	10.7
10	83.9	5.2	10.9	84.8	5.4	9.8
20	89.2	3.7	7.1	89.5	4.0	6.4
30	91.4	3.0	5.5	91.4	3.4	5.2

(a) TREC Newswire data (Disks 1-5 of TREC)

n	$N = 250 \times 10^6$			$N = 1{,}000 \times 10^6$		
	single	multi	repeat	single	multi	repeat
1	0.3	0.3	99.4	0.2	0.2	99.6
2	6.5	4.3	89.2	4.0	3.0	92.9
3	26.3	9.6	64.0	19.0	8.6	72.4
4	46.7	10.7	42.6	37.4	11.3	51.3
5	58.8	9.8	31.4	49.9	11.4	38.7
6	65.2	8.9	25.9	56.9	10.8	32.3
7	68.7	8.2	23.1	60.9	10.3	28.8
8	71.0	7.8	21.2	63.4	10.0	26.7
9	72.7	7.4	19.9	65.2	9.7	25.1
10	74.1	7.1	18.8	66.7	9.4	23.9
20	81.1	5.5	13.4	74.4	7.9	17.7
30	84.4	4.7	10.9	78.1	7.1	14.9

(b) TREC GOV2 data

n	$N = 250 \times 10^6$			$N = 1{,}000 \times 10^6$		
	single	multi	repeat	single	multi	repeat
1	0.5	0.4	99.0	0.4	0.3	99.3
2	8.3	4.9	86.8	6.1	3.8	90.1
3	26.2	9.5	64.3	21.9	8.5	69.6
4	40.9	10.4	48.7	37.1	10.2	52.7
5	48.4	9.9	41.6	46.1	10.0	43.9
6	51.9	9.5	38.6	50.4	9.6	40.0
7	53.7	9.2	37.1	52.7	9.3	38.1
8	54.9	9.1	36.0	54.1	9.1	36.8
9	55.9	8.9	35.2	55.2	8.9	35.9
10	56.7	8.8	34.5	56.1	8.8	35.1
20	61.8	8.0	30.2	62.0	8.0	30.0
30	65.0	7.6	27.4	65.9	7.4	26.7

(c) TREC ClueWeb-B data

Table 1: Percentages of n-grams in three categories: "single", distinct n-grams appearing exactly once; "multi", distinct n-grams appearing more than once; and "repeat", the total of the second and subsequent appearance counts of the n-grams that appear more than once. In total, each row in each section of the table adds up to 100%, since every one of the N n-grams in each file is assigned to exactly one of the three categories.

If the sequence is enlarged by a factor of 10 – to process 100–200 GB, of source text, say – the memory space required pushes towards the limits of plausibility. If the factor is instead 100, and 1 TB of HTML data is to be processed, the memory requirement for array H – now 25 GB – makes HASH-BASED TWO-PASS in-

Algorithm SPEX MULTI-PASS

```
 1: compute $H_1$, a list of the 1-grams in the sequence $S$, together
    with their occurrence frequencies
 2: allocate an empty hash filter $H_2$
 3: for $k \leftarrow 2$ to $n$ do
 4:   for $i \leftarrow 1$ to $N$ do
 5:     $h_1 \leftarrow hash_b(S[i \cdots (i+k-2)])$
 6:     $h_2 \leftarrow hash_b(S[(i+1) \cdots (i+k-1)])$
 7:     if $H_{k-1}[h_1] \geq m$ and $H_{k-1}[h_2] \geq m$ then
 8:       $s \leftarrow S[i \cdots (i+k-1)]$
 9:       $h \leftarrow hash_b(s)$;
10:       if $H_k[h] < m$ then
11:         $H_k[h] \leftarrow H_k[h] + 1$
12:       end if
13:     end if
14:   end for
15:   free $H_{k-1}$, and reallocate a new hash filter $H_{k+1}$
16: end for
17: apply steps 13 to 21 of HASH-BASED TWO-PASS using the
    filter $H_n$
```

tractable in terms of memory space required.

2.4 Hash-filter size

The memory required by HASH-BASED TWO-PASS is determined by the tension between b_2, the number of bits in each eventual hash value, and the number of such values that must be stored, which by definition must be less than 2^{b_2}. But choosing a value of b_2 that is too small renders the filter ineffective. In the limit, if (for example) $b_2 < \log_2 N$, then it is likely that the great majority of the 2^{b_2} possible b_2-bit values will be generated as the N n-grams are processed. The behavior of HASH-BASED TWO-PASS will then degenerate to that of DISK-BASED ONE-PASS, but with a futile first pass over the sequence S that serves no purpose. In this case, file F_3 is likely to approach $(n+1)N$ words, and represent a significant resource expectation.

At the other extreme, when b_2 is large, the file F_3 that is generated by HASH-BASED TWO-PASS will be only fractionally bigger than the n-gram index that is being constructed, because the false positive rate during the filtering step will be low.

Between these two extremes, when $b_2 \approx \log_2 N + k$ for some relatively small value of k like 3 or 5, the hash filter reduces the number of false positives to around $N/8$ or $N/32$, and compared to the $N/5$ true positives (working from the data in Table 1), meaning that the temporary file F_3 is less than twice the size of the final n-gram index, once the like entries have been coalesced.

When b_2 is chosen in this way, representing H as a bit-map of $2^{b_2}/8$ bytes is also reasonably efficient – it implies that H occupies somewhere between N and $4N$ bytes, confirming the expectation established in the previous section that for sequences of more than around a billion symbols, the HASH-BASED TWO-PASS approach should not be used in single-processor environments. That is, once terabyte-scale problems are reached, parallel implementations must be employed. But first, before considering the extent to which these disk-based and hash-based algorithms can be parallelized, we consider one further approach to determining repeated n-grams.

2.5 SPEX

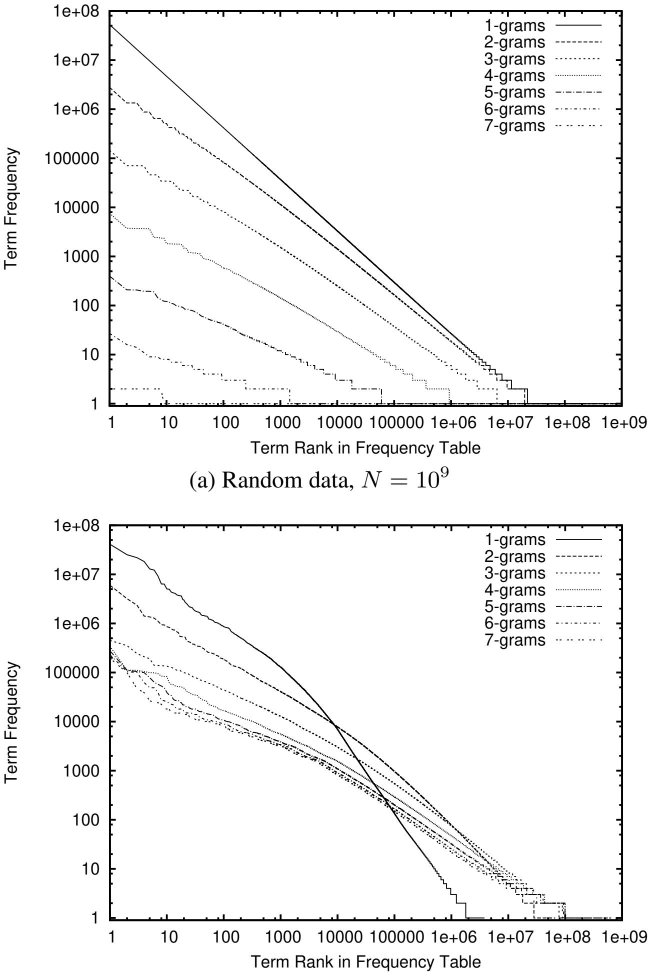
The SPEX MULTI-PASS approach to finding repeated n-grams was developed by Bernstein and Zobel [2006]. It makes n passes through the sequence S to construct a set of n probabilistic filters, building on the key observation that any n-gram consists of two $(n-1)$-grams; and that if either of those two $(n-1)$-grams occurs fewer than m times, then the n-gram in question must also occur fewer than n times. Hence, starting with a plain vocabulary of 1-grams, successive passes over the source sequence generate increasingly longer sets of n-grams. Two generations of the hash filters are required to be active at any given time.

Bernstein and Zobel argue that the multiple passes made by SPEX MULTI-PASS are justified because the memory space required for the filters is reduced compared to any possible "all in one" approach that seeks to directly generate a hash-filter for n-grams. In fact, compared to using a wide b_1-based hash in the first pass of Algorithm HASH-BASED TWO-PASS, the additional passes serve little purpose except to save the disk space used by the file F_1 (required in algorithm HASH-BASED TWO-PASS), and come at a considerable cost in terms of execution time. In particular, the SPEX MULTI-PASS approach does not result in any saving in terms of memory space, because during the final resolution pass at step 15 when the hash filter H_n is being used, exactly the same amount of memory is required as by algorithm HASH-BASED TWO-PASS for any given level of false positive performance. Nor is the removal of file F_1 a great saving, since file F_1 is smaller than the final file F_3, which is still required if the output of the SPEX MULTI-PASS algorithm is to be deterministic rather than probabilistic.

In their experiments, Bernstein and Zobel process a file of $N = 2^{26} = 64 \times 10^6$ symbols derived from 476 MB of newspaper stories using filters H_{k-1} and H_k each of 64 MB of memory. This requirement matches the estimates made in Section 2.4, falling (when the two filters are combined) in the middle of the N to $4N$ bytes range estimated as being a reasonable memory requirement for algorithm HASH-BASED TWO-PASS. Bernstein and Zobel also report that for a collection of 5 GB (presumably approximately 500 million symbols) of Newswire data that their SPEX MULTI-PASS implementation required a little over four hours on a Pentium III processor with 768 MB of memory. The repeated 8-gram index that was generated occupied a total of 2.5 GB, including the 8-gram vocabulary.

Figure 1 helps explain why the multiple passes of final SPEX MULTI-PASS are no more effective than a single pass that generates the n-grams directly. The top graph shows what happens when randomly generated text is processed into n-grams for varying values of n, plotting gram frequency as a function of gram rank when sorted by decreasing frequency. The cascading pattern of lines demonstrates that as n gets larger, the frequency of commonly-occurring n-grams gets lower, the fraction of the n-grams that repeat gets smaller, and the tail of the distribution gets longer. The bottom graph shows what happens when the same analysis is undertaken on an equivalent volume of English text. The difference is marked – on the English text, the distributions of 6-, 7-, and 8-gram frequencies are almost identical. That is, with high probability, any particular repeated $(k-1)$-gram is extended to form a repeated k-gram. The same effect is demonstrated in Table 1.

The SPEX MULTI-PASS algorithm builds successive hash-filters for $k = 1$, $k = 2$, and so on, through until $k = n - 1$, in the final pass the $n - 1$ filter is applied to the text and the passing n-grams are indexed. In total n passes through the data are made. The number of passes can be reduced to $\log n$ by pausing at fewer values of k, and checking for more subsequences at step 7. For example, if the values of k are drawn from $\{1, 2, 3, 5, 8, 13, \ldots\}$, then a total of four 5-grams are used in a quest to eliminate each 8-gram as H_8 is constructed from H_5. We call this approach SPEX LOG-PASS, and have tested it as an alternative to the SPEX MULTI-PASS approach. It relies on the same assumptions as SPEX MULTI-PASS, namely that the number of repeated n-grams for any given value of



(a) Random data, $N = 10^9$

(b) English text, $N = 10^9$

Figure 1: Frequency of n-grams as a function of n-gram rank, for different values of n, for: (a) randomly generated Zipfian data; and (b) English text from the TREC GOV2 collection. In both cases the source sequence contains approximately $N = 10^9$ symbols.

n is a diminishing fraction of the number of distinct n-grams. As Table 1 and Figure 1 demonstrate, this assumption is not valid for typical text.

3. SCALABLE IMPLEMENTATIONS

Scalability is a desirable property for any algorithm to possess. Informally, it represents the extent to which additional resources are required as problem instances get larger. When assessing scalability, three resources are of interest: the time taken to execute the algorithm; the peak amount of memory space required to execute the algorithm; and the peak amount of disk space required for temporary files while the algorithm is executing. It is assumed that there is sufficient disk space available for the inputs and outputs of the algorithm to be stored.

Two scalability scenarios need to be considered. The first is when a *single* machine, with a fixed amount of main memory, is available. If the algorithm neither requires more and more memory as the problem instance increases, nor requires excessive execution time, then it is clearly scalable. But it is also usual to allow increased memory usage too, even though memory is a fixed resource, and even though a program that cannot execute in the available memory cannot execute at all. That is, in single-processor environments, scalability is determined by an evaluation of the asymptotic growth in execution time of the algorithm – heapsort is a scalable algorithm even though it cannot be used on data sets that are too large for main memory. As a general demarcation, algorithms that require quadratic execution times would not be defended as being scalable, whereas execution times in $O(N \log N)$, where N is a measure of the problem size, would be. Roughly speaking, if an algorithm is scalable in this first sense, then if the problem size grows by a factor of p, the time taken to execute the algorithm will grow by not much more than p.

The second situation is the more complex one. In a parallel computing environment, scalability has a slightly different meaning. Now what is required is that, as the problem size is increased and the number of processors is increased by a matching proportion, a distributed implementation of the algorithm executes in time that remains fixed, or grows only slowly. In addition, while there might be a memory-imposed limit on how much data can be loaded on to any single processor, there must not be any overall memory limit imposed. Communication costs must also be shown to be bounded as a function of the expected running time, before an algorithm can be argued to be scalable. More precisely, if a problem of size N can be executed on a single processor in time t, then a problem of size pN when distributed over p machines, should be solvable in time not significantly greater than t. Moffat and Zobel [2004] discuss issues to do with performance evaluation in distributed environments.

3.1 Parallel implementations

Consider the behavior of the DISK-BASED ONE-PASS, SPEX MULTI-PASS, and HASH-BASED TWO-PASS methods if a cluster of computers is available, and the computational load is to be shared across them so as to reduce the elapsed execution time.

Algorithm DISK-BASED ONE-PASS translates naturally to a distributed processing model. If the input sequence S is split into p equal length parts (with some slight overlap at the fringes so that none of the N n-grams are lost) steps 1–4 can be performed across p processors to make p intermediate files. Those files are then sorted on their respective host machines, before being partitioned into a total of p^2 components that are written to a shared file system so that they can be accessed by all of the machines, and then combined in a set of p independent p-way merge operations, one per machine. This approach to sorting is well understood for parallel computation (see, for example, Tridgell and Brent [1993]) and accounts for steps 5 to 10. Finally, the p separate sorted lists of repeated n-grams are combined into a single file via a sequence of concatenations. Throughout these processing phases all p machines are equally busy, provided that the p^2 subfiles are all of approximately the same size; furthermore, there is no additional work introduced by the partitioning. That is, algorithm DISK-BASED ONE-PASS is scalable in both the first single-processor sense, and also in the second distributed sense. In a parallel implementation it continues to have the drawback of requiring approximately $(n + 1)N$ words of temporary disk space, spread across the p processors.

On the other hand, Algorithms SPEX MULTI-PASS and SPEX LOG-PASS do not readily parallelize. The problem is the amount of memory required by the hash filters H_{k-1} and H_k, shown in use at step 7 and step 10 of Algorithm SPEX MULTI-PASS. These two arrays must be sized in proportion to the total number of distinct n-grams that are anticipated to occur in the sequence, and as is shown

Algorithm Sequence-Blocked Two-Pass

```
1: apply steps 1 to 6 of Disk-Based One-Pass to generate file $F_2$
2: for $i \leftarrow 1$ to $N$ do
3:   $s \leftarrow S[i \cdots (i+n-1)]$
4:   $h \leftarrow hash_b(s)$
5:   append $(h, s, i)$ to the in-memory buffer $B$
6:   if $B$ has reached the memory limit then
7:     sort $B$
8:     for $(h, s, i) \in B$ where $h \in F_2$ do
9:       append $(s, i)$ to the output file $F_3$
10:    end for
11:    $B \leftarrow \{\}$
12:  end if
13: end for
14: apply steps 5 to 10 of Disk-Based One-Pass to the file $F_3$
```

in Table 1, even for curated text such as the Newswire collection, the number of repeated n-grams is a non-decreasing fraction of the sequence length. Worse, the hash-filters are required in full at every processing node, and, by their very randomized nature, cannot be partitioned into segments that correspond to the partitioning of the sequence. In terms of single-processor scalability, the two SPEX Multi-Pass-based methods can be said to be scalable through until the memory limit is reached, but even so, note that the execution time grows as a linear function of both N and n.

Bernstein and Zobel [2006] comment in their paper that “early experiments on the 426 GB GOV2 collection have also indicated that DECO [and hence SPEX] is able to scale to collections of this size on a standard production server”, but in a subsequent document note that “indexing of the 426 GB GOV2 collection has thus far proved impossible ... due to ... the extremely high level of duplication within GOV2” [Bernstein, 2006, p. 107].

3.2 Sequential processing of the filter

We now turn to the question of whether a scalable parallel implementation of Algorithm Hash-Based Two-Pass is possible. It also uses a hash filter that (step 16) is presumed to be searchable, and thus available in random-access memory at every processing node. We have explored two different ways of avoiding that requirement while still retaining the underlying nature of the process.

The first is shown as Algorithm Sequence-Blocked Two-Pass. In this approach, subsegments of S are extracted, joined with their hash key, and added to a buffer B whose size is determined by the amount of available main memory. Once the buffer is full, it is sorted by hash key, and then that sorted buffer merge-intersected with the sorted on-disk file F_2, which contains all of the hash keys of interest. The ones that appear in both B and F_2 are identified, and condensed to form the n-gram index in the second pass. In a parallel implementation, each of the p processors is assumed to have local memory, and is able to work with a local buffer, it’s own full copy of the file F_2, and its own slightly overlapping subsequence of S. Once the n-grams of interest have been identified, they are sorted back into n-gram order and the index built in the usual manner, at step 14.

The Sequence-Blocked Two-Pass approach avoids the need for an unknown amount of memory for H, and replaces that need by a buffer B of pre-determined size. However, there is a cost – the smaller the size of B, the more often the file F_2 needs to be merge-intersected against it. So halving the size of B relative to N doubles the overall cost of executing steps 3 to 12. Or, put another way, for any given fixed B, and assuming that the number of n-grams is a fixed fraction of N (as is shown in Table 1), then the running time of the merge-intersect phase grows as N^2.

Algorithm Location-Based Two-Pass

```
1: for $i \leftarrow 1$ to $N$ do
2:   $s \leftarrow S[i \cdots (i+n-1)]$
3:   $h \leftarrow hash_b(s)$
4:   append $(h, i)$ to the output file $F_1$
5: end for
6: sort $F_1$, coalescing paired entries $(h, \ell_1)$ and $(h, \ell_2)$ to
   $(h, \ell_1 \cup \ell_2)$ as soon as they are identified
7: for $(h, \ell) \in F_1$ do
8:   if $|\ell| \geq m$ then
9:     append the whole of $\ell$ to the output file $F_2$
10:  end if
11: end for
12: sort $F_2$
13: for $i \in F_2$ do
14:   $s \leftarrow S[i \cdots (i+n-1)]$
15:   append $(s, i)$ to the output file $F_3$
16: end for
17: apply steps 5 to 10 of Disk-Based One-Pass to the file $F_3$
```

3.3 Sorting by location

The second alternative is to store more information into the intermediate file F_2. The repeated merge-intersect can be eliminated if F_2 is stored in sequence address order rather than hash-value order. To get F_2 into sequence order requires that disk locations be included in it, in addition to the hash values that are the sort key used at first.

Algorithm Location-Based Two-Pass describes the resultant process. Like Algorithm Disk-Based One-Pass, there is very little memory used at each processing node. File F_1 is a list of hash and location pairs, with the hash values taking up more space than the n-gram frequencies that are maintained in Algorithm Hash-Based Two-Pass, but less space than the n-grams themselves, as stored in Algorithm Disk-Based One-Pass. After the second sorting phase at step 12, file F_2 is a sorted list of “locations of interest” in S. So if there are very few repeated n-grams, file F_2 will be relatively short, and the second “pass” at steps 13 to 16 will only process a minority of the n-grams in S, rather than all of them. That is, Location-Based Two-Pass can be expected to have an advantage over Hash-Based Two-Pass in terms of execution speed, with the advantage greater when the repeated n-grams are sparser.

To parallelize this method across p processors, the same approach is taken as when parallelizing Disk-Based One-Pass. The source sequence is partitioned into p slightly overlapping subsequences, and each of the processors generates an F_1 file for its subsequence. Those p files are then globally sorted, by creating and exchanging p^2 smaller segments, and carrying out p independent p-way merges. Following that first sort, each processor then carries out steps 7 to 11 on the sorted F_1 segment that it holds, before again slicing it into p subsegments and swapping across the p processors, in preparation for the second sort at step 12. Each processor then checks (steps 13 to 16) a subset of the locations of interest against the same subsequence of S that it originally worked with in the first pass. Finally, at step 17, the n-gram index is built from the file F_3 via a third sorting stage.

Name	Size	Documents	Words
TREC Newswire	5.20 GB	1.6×10^6	0.75×10^9
TREC Gov2	426 GB	25×10^6	22.5×10^9
TREC ClueWeb-B	1.46 TB	50×10^6	40.7×10^9

Table 2: The three datasets used in the experiments.

4. EXPERIMENTAL EVALUATION

We have undertaken an exhaustive evaluation of these various techniques. This section documents the results of that evaluation. The data sets used and experimental environment are discussed first. The second subsection then describes a range of experiments on a single processor in which first N is varied with n fixed; and then N is fixed and n is varied. Section 4.3 then repeats those experiments, but using a cluster of processing nodes, and shows the extent to which the methods can genuinely be regarded as being scalable in the parallelized sense. Section 4.4 reports the peak disk space required during processing; and then Section 4.5 summarizes the performance of the LOCATION-BASED TWO-PASS method on the full 1.5 TB ClueWeb collection.

4.1 Dataset and test environment

We use the ClueWeb09-TREC-B collection to test the time and space efficiency of each of these indexing techniques. This collection contains fifty million English web documents. Compressed using `gzip`, it requires around 226 GB; uncompressed, nearly 1.5 TB. Table 2 lists the characteristics of this and two other smaller data collections, including the length N of the sequence of words contained in the collection when the document boundaries are ignored.

To provide a straightforward basis for the experimental evaluation, the data was pre-processed to form a sequence of word tokens, stored as 32-bit integers. All WARC header data and document segmentation information were discarded, as was embedded markup and other non-text content such as executable scripts; and then the resultant data was treated as a continuous stream of words. In a practical system, n-grams would not be permitted to span document boundaries, but the difference is small, and our experiments are realistic. Where smaller test sequences were required, prefixes of this long whole-collection sequence were used. We note that there may be some bias resulting from this method of collection sampling, however, this bias is constant within each experiment, so the comparison of the various techniques remains valid. In all of the experiments reported below $m = 2$ was used, and the output was thus a list of all of the n-integer patterns that appeared twice or more in the input sequence, together with their locations as ordinal offsets from the start of the sequence.

The pre-processing stage significantly reduced the amount of data to be stored during the actual experiments, by a ratio of around 10:1. That is, each 1 GB ClueWeb file produced around 100 MB of numeric data, containing approximately 25 million integers representing parsed words. Pre-processing also reduces the I/O costs and parsing costs associated with handling text (rather than binary) input data. The complete conversion took approximately one day using a single CPU and over 20 GB of RAM to store the vocabulary of the collection, implemented as a hash table.

The experimental hardware consisted of a cluster of 32 dual-core 64-bit Intel processors with a 3.2 GHz clock speed, and 4 GB of RAM each. The experiments were all configured so that only one core and only 2 GB of memory were used on each processor, with the other core on each machine forced to be idle. Inter-process communication was via a shared network attached file system, with all processors able to write files to the shared disk, and to read the files written by other processors.

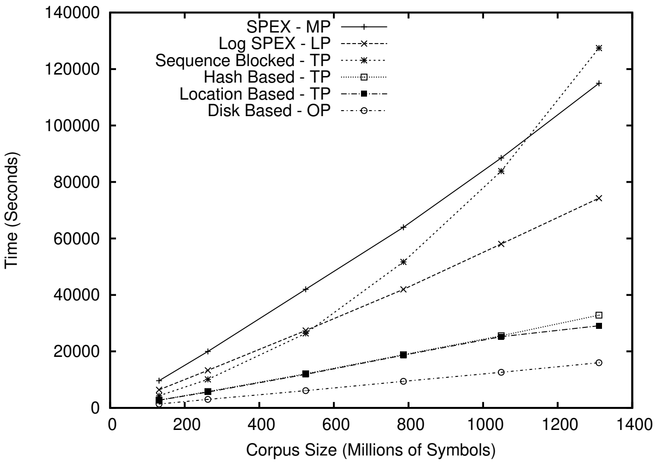


Figure 2: Execution time as a function of N, when computing repeated 8-grams using a single processor. The SEQUENCE-BLOCKED TWO-PASS method requires time that grows super-linearly; over this range of N the other methods are all essentially linear in the volume of data being processed. All data points in this graph represent an average of 10 timed runs.

Two values of b were used, dependent on whether the hash-filter was required to be memory resident or not. In the HASH-BASED TWO-PASS, SPEX MULTI-PASS, and SPEX LOG-PASS methods, $b_2 = 32$ was used for the size of the in-memory hash table, and it was stored as a direct-access. With this value the hash-filter contains four billion entries, and at two bits per entry, requires 1 GB of main memory, the largest amount that could be usefully managed on the experimental hardware (and note that SPEX MULTI-PASS and SPEX LOG-PASS require two such filters to be concurrently available). A filter of this size provides useful discrimination for sequences of up to approximately $N = 10^9$ in length for HASH-BASED TWO-PASS, and four to five times larger for SPEX MULTI-PASS and SPEX LOG-PASS, the difference arising because of the iterative and more selective nature of the insertion policy in the two SPEX MULTI-PASS variants. For the hash-filter methods where the filter is stored only on disk – SEQUENCE-BLOCKED TWO-PASS and LOCATION-BASED TWO-PASS – a hash function of $b_1 = 56$ bits was used in all experiments, and provided a high degree of discrimination and a low false match rate.

4.2 Processing on a single machine

Figure 2 shows evaluation time as a function of N, for a single fixed value of $n = 8$, and single-CPU execution. Five of the methods have execution times that grow linearly as a function of N, with the DISK-BASED ONE-PASS the fastest of the six methods tested. The SEQUENCE-BLOCKED TWO-PASS algorithm has performance that grows super-linearly, a consequence of the fixed memory allocation, and the increasing number of blocks B that must be processed against the hash-filter stored in file F_2. The two SPEX MULTI-PASS approaches are not competitive, and while they can physically process data files of more than 10^9 symbols, are hindered by the time taken to perform their multiple passes. False matches do not affect the execution time in any significant way, but do result in large temporary disk files F_3 being created. From that point of view, for large N, it becomes preferable to abandon the SPEX MULTI-PASS approach entirely, and simply use the DISK-

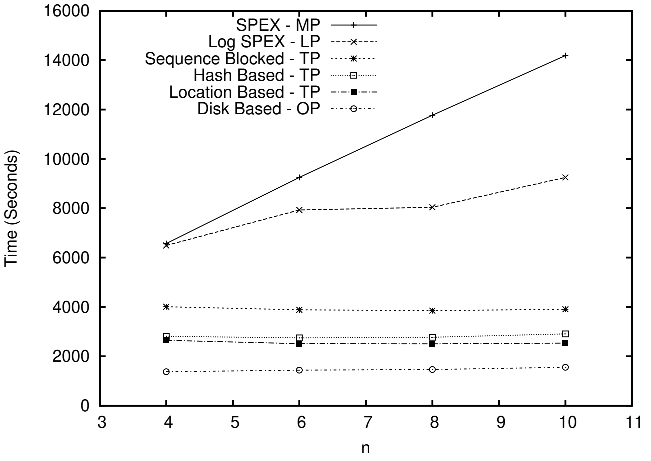


Figure 3: Execution time as a function of n, processing a total of $N = 125 \times 10^6$ symbols representing English words. Each pass that is made adds considerably to the time taken to identify the repeated n-grams. All data points in this graph represent an average of 10 timed runs.

Based One-Pass method – it uses the same disk space, and is n times faster.

Figure 3 then fixes N, and varies n, again working on a single CPU. The effect of the multiple passes undertaken by the two SPEX Multi-Pass variants is apparent, and even the second pass through the sequence associated with the hash-filter based variants adds to the running time, meaning that the Disk-Based One-Pass method is the fastest. It, and also the three hash-filter based methods (Hash-Based Two-Pass, Sequence-Blocked Two-Pass, and Location-Based Two-Pass) are relatively insensitive to n, and the cost of evaluating the md5 hash function is not a significant factor in the overall running time, at least over this spectrum of n values.

Measurement of the reading and hashing loop in isolation (steps 1 to 3 of Hash-Based Two-Pass, and also used in several of the other approaches) on a sequence of $N = 500 \times 10^6$ symbols showed that there was some small increase in execution time, from around 2,800 seconds when $n = 2$ to around 3,120 seconds when $n = 10$, indicating that the cost of the hash evaluation to generate a $b = 32$-bit value, while varying as n, has only a small effect on overall execution time. Without the call to the hashing function, the same loop required 2,100 seconds at $n = 2$ and 2,400 seconds when $n = 10$, confirming that the md5 hash routine is a non-trivial, but also non-dominant part of the first processing loop, and is relatively unaffected by the length of the n-gram it is acting on.

It is also possible to compute the hash function over a sliding window of n symbols in such a way that each new evaluation after a unit shift of the window requires only $O(1)$ time, but we do not use such an approach in our experiments, preferring the demonstrated reliability of the md5 mechanism.

4.3 Processing on a cluster

A critical claim in this paper is that the Location-Based Two-Pass approach is scalable, and can readily be implemented across a cluster of computers in order to deal with very large sequences. Figure 4 demonstrates the validity of that claim. To generate this graph different length prefixes of the ClueWeb sequence were taken, and split over 5, 10, 15, 20, and then 25 processing nodes, with the prefix lengths varied in proportion to the number of processing nodes involved. In Figure 4, each processing node hosts approximately 270×10^6 sequence values. With this experimental design, elapsed time for a scalable algorithm should be either constant, or slightly growing if there is a logarithmic overhead on a per-processor basis. (Details of the computational cost of handling the entire ClueWeb-TREC-B sequence are presented in Section 4.5 below.)

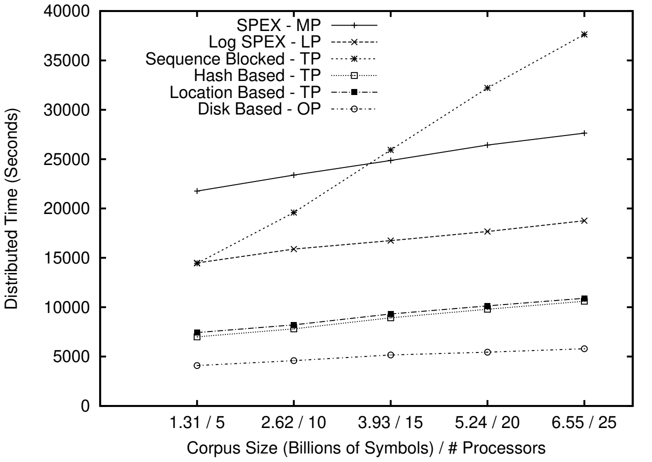


Figure 4: Elapsed time required by parallel implementations. All data points in this graph represent an average of 5 timed runs.

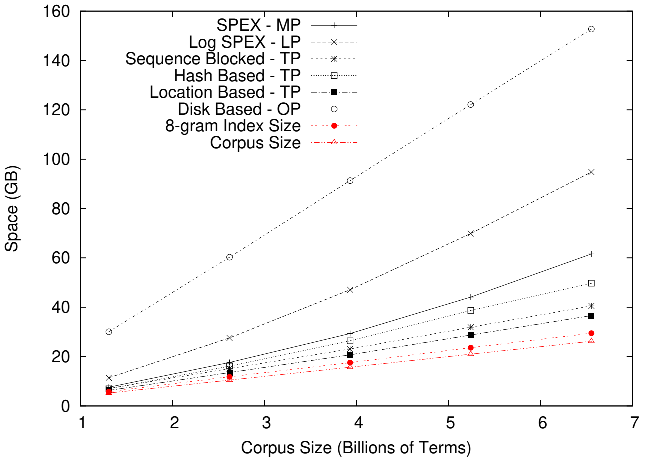


Figure 5: Peak temporary disk space required as a function of N, when computing repeated 8-grams.

Five of the methods show the required trend, while the sixth, for the Sequence-Blocked Two-Pass method, does not – as was anticipated above, it cannot be regarded as being scalable. The Hash-Based Two-Pass and the two SPEX Multi-Pass variants do scale in a "CPU cost only" sense as the sequence length increases, and the breakdown in the performance of the hash-filter as N increases is not dramatic in terms of execution time. However, as is discussed in the next section, they are not as well behaved in terms of disk space requirements.

4.4 Disk space required

Figure 5 plots the peak temporary disk space required by each method, as a function of N when $n = 8$, for sequences in the range approximately 1×10^9 symbols to 6×10^9 symbols. The Disk-Based One-Pass method is very expensive, a consequence of the fact that around $(n+1)N$ words are required in the temporary file F. It takes more than twice the peak disk space of the hash-filter methods.

Working down the graph, the next most expensive method is the Hash-Based Two-Pass approach. The issue raised by the $b_2 = 32$ bit hash size is now clearly apparent; hashing a billion or more objects into a $b_2 = 32$ table yields a significant number of false matches, and all of these create entries in the F_3 file. This problem is exacerbated as more and more symbols are indexed. For smaller values of N (not shown in the graph), Hash-Based Two-Pass is more competitive, but like the SPEX Multi-Pass variants, when N is large it is simpler and more efficient to revert to the Disk-Based One-Pass approach.

The next two curves are for the SPEX Log-Pass and SPEX Multi-Pass algorithms. These methods do not write a hash-filter to disk at all, but both generate a large F_3 file even for small N, primarily because the hash-filter that is constructed by the incremental process is not completely precise – it generates many false matches. This stems from the use of the $b = 32$ bit hash size throughout these algorithms. It is clear that for a fixed, maximal hash size, as the collection size increases the effectiveness of the final filter is slowly diminished.

The Location-Based Two-Pass method performs well, despite requiring more space for the F_1 file than does the Hash-Based Two-Pass approach (recall that the difference is that a five-byte sequence location offset is stored with each $b_1 = 56$ bit hash value, rather than a 2-bit counter). Storing the first temporary file F_1 is the dominant space cost for this amount of data and this degree of reuse. That is, compared to Disk-Based One-Pass, Algorithm Location-Based Two-Pass requires significantly less disk space, and executes less than two times more slowly.

The two (red) lines at the bottom of Figure 5 show, respectively, the cost of storing the eventual 8-gram index, computed assuming that each 8-gram requires eight words to store the gram, plus one address per gram occurrence; and the actual sequence size, stored one value per word. The peak temporary disk space requirement for all of the methods still exceeds the final index size, and also exceeds the corpus size. It may be that this gap represents room for further improvement, but it seems likely that a fundamentally different paradigm will be required before that possibility can be realized. Note also that at least some of the apparent gap is a consequence of the data being distributed across nodes – the "coalescing of like entries during sorting" steps of the various algorithms become less effective as the data is partitioned across more machines.

4.5 Very large data sets

Table 3 documents the cost of applying the Location-Based Two-Pass technique to the full 1.46 TB TREC ClueWeb-B data set on a cluster of Xeon 5355 computers, each with eight cores operating at 2.66 GHz and sharing 16 GB of memory (that is, 2 GB per core). Of these, 50 cores were used for the half-sized data set, and 101 for the full-sized dataset. (There was insufficient spare disk space available on this cluster to execute the Disk-Based One-Pass method over the full 1.46 TB dataset.)

As the table shows, the Location-Based Two-Pass algorithm is capable of identifying the repeated 8-gram symbols in this collection in a scalable and distributed implementation, and while the Location-Based Two-Pass approach requires more execution time that the well-known Disk-Based One-Pass approach, it also requires significantly less temporary disk space.

	Disk-Based	Location TP	
Corpus size (TB)	0.715	0.715	1.46
Grams required, n, m	8, 2	8, 2	8, 2
Sequence length, N ($\times 10^9$)	20.15	20.15	40.7
Numerical Data Size, GB	75.1	75.1	151.7
Processors, p	50	50	101
N/p ($\times 10^6$)	402.9	402.9	402.9
Elapsed time (sec $\times 10^3$)	**8.50**	**12.38**	**13.23**
Peak disk Usage (GB)	**417.4**	**159.9**	**386.3**
Final 8-gram Index Size (GB)	103.0	103.0	202.8
Fraction single (% of N)	49.9	49.9	40.5
Fraction multi (% of N)	9.8	9.8	10.4
Fraction repeat (% of N)	40.3	40.3	49.1

Table 3: Measured performance of two algorithms, Disk-Based One-Pass and Location-Based Two-Pass over two large collections. The last three rows follow on from the $n = 8$ rows in Table 1(c). The first two columns use the first half of the data from the ClueWeb-TREC-B collection. The third column uses all of the data from the ClueWeb-TREC-B collection. Resource limitations meant that Disk-Based One-Pass and Location-Based Two-Pass were the only two mechanisms that could be executed on this volume of data.

Further experiments are currently underway to demonstrate the extent to which the various algorithmic relativities are affected by choice of m. Worth noting is that $m = 2$ is the most resource-intensive choice, and as m gets larger, the computational resources decrease, since fewer candidate repeated strings need to be represented in the hash filter.

5. RELATED WORK

As already noted, Bernstein and Zobel [2006] considered the problem of computing repeated n-grams for large text sequences. They motivate their SPEX Multi-Pass algorithm in the context of deco, a tool for determining co-occurring text.

Fingerprinting techniques have been used for some time to determine the similarity of documents or parts of documents. These techniques are often designed such that there is no explosion of intermediate data. Current techniques focus on keeping only a small sample of repeated n-grams. This makes these techniques inappropriate for many n-gram applications, such as NLP tasks and Search.

In the first paper to suggest fingerprinting technique, Manber [1994] controlled the number of n-grams produced using a $\mod p$ method. If the hash value of the n-gram equals zero modulus p, then the n-gram is used as a fingerprint. Thus reducing the size of the data by approximately a factor of p. There have been a variety of newer fingerprinting techniques that try to reduce the size of the intermediate data further without any significant decline in performance. Seo and Croft [2008] provide a good summary of these techniques. Seo and Croft also present a novel technique to match similar fingerprints based upon the discrete cosine transformation (DCT) algorithm. Their technique enables their system to match similar n-grams.

Locality sensitive hashing (LSH), originally presented by Gionis et al. [1999], was developed for the similar problem of finding similar documents in large collections. Andoni and Indyk [2008] presents a recent summary of both exact and approximate LSH based approaches to a high dimensional nearest neighbor problem.

Depending on the purpose, it may be useful to use this family of hash functions within our n-gram indexing algorithms. The algorithm would need to be modified such that each n-gram is annotated with its hash value. This would enable us to collect matching

n-grams. The direct consequence would be an increase in the required intermediate space. Another option would be to produce an index of the hash values. We postulate that an index of similar n-grams may have many important uses within corpora with excessive spelling or OCR errors. However, it should be recognized that many linguistic and natural language problems require that similar n-grams be considered distinct, in which case these techniques would be inappropriate.

There has been a recent spate of brute-force style map-reduce based approaches to similar problems such as document level similarity computations [Elsayed et al., 2008, Lin, 2009]. A reused n-gram index produced by one of the techniques presented in this paper might be used for these types of problems by considering the set of n-grams in a document to be a set of fingerprints. This would avoid some of the problems of intermediate space, without any loss of precision.

Zhang et al. [2010] present a method of using a shingle index to find reused text sequences. They show that sequence reuse detection using a lossy sequence fingerprinting technique provides similar performance as when using 4-grams. They use a traditional one pass indexing approach to produce the sequence fingerprint indexes. Within their paper; sentence boundaries were used to significantly reduce the total number of 4-grams produced. Our approach allows for the indexing of larger values of n, as well as providing the ability to index *all* repeated n-grams in the collection. A combined approach may produce additional instances of interesting sequence reuse.

Use of a simple map-reduce approach to the problem of identifying repeated n-grams is also, at face value at least, a possibility. A single front-end processor would process the input sequence and hash the grams identified, distributing fixed ranges of hash values to different machines (the “map” step) to be sorted and counted. The “reduce” phase would then consist of sort-merging the resultant lists of duplicate hash values, and checking subsequences of S against that set of hash values. However such an approach has no parallelism during the two gram-processing steps, and hence the mechanism as a whole is completely throttled by the speed – and and also the memory capacity for the hash filter – of the coordinating machine. In contrast the method described in Section 3.3 has the distinct advantage that the gram-processing steps are also parallelized. That is, the volume of data being processed, and the need for inter-processor exchange of information, means that a a genuinely distributed approach is, for repeated n-gram identification, superior to a mechanism based on map-reduce.

6. CONCLUSION

We have explored the problem of constructing an index of repeated n-grams for text corpora in the multi-gigabyte range. Our new LOCATION-BASED TWO-PASS algorithm provides a useful blend of attributes, in that it requires less than half the amount of temporary disk space of the traditional DISK-BASED ONE-PASS approach, while requiring less than twice as much processing time. This represents a significant benefit in terms of practical usefulness.

We have also demonstrated that our approach can readily be adapted for use across a cluster of computers, and is scalable in this distributed sense, a virtue that more than compensates for its slightly slower execution speed. Our experiments included the processing of 1.5 TB of ClueWeb data using $m = 2$, and as a result we have been able to demonstrate the ability to identify repeated n-grams on a much larger scale than previous work.

Acknowledgment. This work was supported in part by National Science Foundation grant IIS-0534383 and in part by the Australian Research Council grant DP0880065. Any opinions, findings and conclusions or recommendations expressed in this material are the authors, and do not necessarily reflect those of the sponsors.

References

A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Comm. ACM*, 51(1):117–122, 2008.

Y. Bernstein. *Detection and Management of Redundancy for Information Retrieval*. PhD thesis, School of Information Technology and Computer Science, RMIT University, Australia, 2006.

Y. Bernstein and J. Zobel. Accurate discovery of co-derivative documents via duplicate text detection. *Information Systems*, 31:595–609, 2006. doi: 10.1016/j.is.2005.11.006.

W. B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley, USA, 2009. ISBN 9780136072249.

T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with MapReduce. In *Proc. 46th Ann. Meeting of the ACL on Human Language Technologies*, pages 265–268, Columbus, OH, June 2008. Association for Computational Linguistics.

A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th Int. Conf. on Very Large Data Bases*, pages 518–529, Edinburgh, Scotland, September 1999. Morgan Kaufmann.

J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In *Proc. 32nd Ann. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 155–162, Boston, MA, 2009. ACM Press, NY.

U. Manber. Finding similar files in a large file system. In *Proc. USENIX Winter 1994 Technical Conf.*, pages 1–10, San Francisco, January 1994. Usenix Association, CA.

D. Metzler and W. B. 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. doi: http://doi.acm.org/10.1145/1076034.1076115.

A. Moffat and J. Zobel. What does it mean to “measure performance”? In *Proc. 5th Int. Conf. on Web Informations Systems*, pages 1–12, Brisbane, Australia, November 2004. LNCS 3306, Springer.

P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 10th ALENEX Workshop on Algorithm Engineering and Experiments*, pages 71–83, New Orleans, LA, January 2007. SIAM.

J. Seo and W. B. Croft. Local text reuse detection. In *Proc. 31st Ann. Int. ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 571–578, Singapore, 2008. ACM Press, NY.

A. Tridgell and R. P. Brent. An implementation of a general-purpose parallel sorting algorithm. Technical Report TR-CS-93-01, Australian National University, 1993. URL `http://gan.anu.edu.au/~brent/pd/rpb140tr.pdf`.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999. ISBN 1-55860-570-3.

Q. Zhang, Y. Zhang, H. Yu, and X. Huang. Efficient partial-duplicate detection based on sequence matching. In *Proc. 33rd Ann. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 675–682, Geneva, Switzerland, July 2010. ACM Press, NY.