Scalable Phrase Mining for Ad-hoc Text Analytics

Srikanta Bedathur
Klaus Berberich
Jens Dittrich
Nikos Mamoulis
Gerhard Weikum

MPI–I–2009–5-006 April 2009

Authors' Addresses

Srikanta Bedathur
Max-Planck Institute für Informatik
Campus E1 4, Saarbrücken, Germany

Klaus Berberich
Max-Planck Institute für Informatik
Campus E1 4, Saarbrücken, Germany

Jens Dittrich
Saarland University
Saarbrücken, Germany

Nikos Mamoulis
Max-Planck Institute für Informatik
Campus E1 4, Saarbrücken, Germany

Gerhard Weikum
Max-Planck Institute für Informatik
Campus E1 4, Saarbrücken, Germany

Abstract

Large text corpora with news, customer mail and reports, or Web 2.0 contributions offer a great potential for enhancing business-intelligence applications. We propose a framework for performing text analytics on such data in a versatile, efficient, and scalable manner. While much of the prior literature has emphasized mining keywords or tags in blogs or social-tagging communities, we emphasize the analysis of interesting phrases. These include named entities, important quotations, market slogans, and other multi-word phrases that are prominent in a dynamically derived ad-hoc subset of the corpus, e.g., being frequent in the subset but relatively infrequent in the overall corpus. The ad-hoc subset may be derived by means of a keyword query against the corpus, or by focusing on a particular time period. We investigate alternative definitions of phrase interestingness, based on the probability of phrase occurrences. We develop preprocessing and indexing methods for phrases, paired with new search techniques for the top-k most interesting phrases on ad-hoc subsets of the corpus. Our framework is evaluated using a large-scale real-world corpus of New York Times news articles.

Keywords

Text Analytics, Archives

Contents

1 Introduction

We are witnessing a dramatic growth of business-relevant information by the wealth of and convenient access to customer- or product-related events in the form of user-interaction logs (web clicks etc.), news, blogs, and Web 2.0 community data. This enables advanced Business-Intelligence (BI) applications such as customer relationship management, market predictions, campaign planning, and social network analysis. Many of the underlying data sources have a high fraction of potentially valuable text data; this is obvious for news and blogs but also holds for click logs when clicks are enriched by the corresponding web pages (or summary snippets or associated queries in the log). Consequently, text analytics is getting a key role in modern data mining and BI for decision support.

Analysts are often interested in examining a set of specifically compiled documents, to identify their characteristic words or phrases or discriminate it from a second set. Examples are news articles from different newspapers, blog postings in different countries, or Web pages over different time periods. Tag clouds and evolving taglines are prominent examples of this kind of analyses [2, 9, 21]. While there is ample work on this topic for word or tag granularities, there is very little prior research on mining variable-length phrases. Such interesting phrases include names of people, organizations, or products, but also news headlines, marketing slogans, song lyrics, quotations of politicians or actors, and more. The state-of-the-art work on such tasks is the ground-breaking paper on the MCX system [27].

In this paper we are focusing on the analysis of *interesting phrases* in document collections that are dynamically derived for ad-hoc analytics, for example, by a keyword query or metadata-based search from a large document corpus. A particular relevant case is the comparative analysis of two ad-hoc collections against each other. For instance, consider the results of two keyword queries “Steve Jobs” versus “Bill Gates” on a news archive that covers several decades. The analyst would like to identify the most interesting phrases of each of the two results sets when viewing the sets side by side. The result may include phrases such as *"apple chief executive", "mac os x", "the computer maker"* for Steve Jobs and *"bill and melinda gates foundation", "the chairman of microsoft", "the world's wealthiest man"* for Bill Gates. Interestingness here can be defined with the help of statistical measures. For example, a phrase could be interesting if it appears very frequently in one query result, or if its local frequency in one result compared to the other result is high. Alternatively, we could use the local frequency relative to the global frequency in the entire archive, or an information-theoretic measure like point-wise mutual information (PMI) rather than frequency ratios.

For most practical definitions of interestingness, finding the interesting phrases requires computing the frequencies of all phrases in one or two ad-hoc subsets $\mathcal{D}'$ and $\mathcal{D}''$ of the overall corpus $\mathcal{D}$. A possible way to do this, is to scan all documents in $\mathcal{D}'$ (and $\mathcal{D}''$) using a sliding window, and for each phrase we encounter, look it up in a hash table and increase a counter maintained for it. This would be very inefficient as it requires scanning a possibly large number of documents. In [27] a *phrase inverted index* is developed to solve this problem. For each phrase, identifiers of documents that contain the phrase are collected into an index list, built in an IR-style inverted-file fashion [33]. In order to compute the frequencies of the phrases in $\mathcal{D}'$, the inverted lists are accessed and intersected with $\mathcal{D}'$. An approximate counting technique that intersects only a sampled subset of each list with $\mathcal{D}'$ is proposed; still, a very large number of lists has to be accessed – potentially as large as the number of phrases, regardless of how small or large the size of $\mathcal{D}'$ is.

Although [27] has gone a good way towards making these kinds of text-analytics tasks efficient, their approach has limitations in terms of scalability. As news, blogs, and web-usage corpora become rapidly larger, the phrase-inverted-index method becomes practically infeasible for interactive analytics. In fact, the experiments in [27] only reported results on a corpus of 30,000 scientific publications in the field of databases. The main difficulty with respect to scalability is the ad-hoc nature of the set $\mathcal{D}'$ (or sets $\mathcal{D}'$ and $\mathcal{D}''$): unless one wants to impose a-priori restrictions on the analyses that the BI expert can do, the dynamic computation of $\mathcal{D}'$ (and $\mathcal{D}''$) prevents any technique that solely relies on precomputed phrase frequencies. This setting poses a major algorithmic and systems-engineering challenge.

In this paper, we develop an efficient alternative to [27] with much better scalability. We pre-process the documents in the entire corpus $\mathcal{D}$ and extract all phrases (above some minimum-support threshold). We then encode and index the phrases contained in each document. This approach is a major departure from the inverted-files paradigm of traditional IR and text analytics. Our approach resembles the notion of "forward indexes" in the IR literature [6, 22], but it has not received much attention there either. In the following, we refer to our novel notion of per-document compressed phrase lists as *forward index lists.*

Given a subset $\mathcal{D}' \subset \mathcal{D}$ (or two sets $\mathcal{D}'$ and $\mathcal{D}''$), in order to determine the frequencies and compute the interestingness of the phrases there, we scan and merge the forward index lists of the documents in $\mathcal{D}'$ (and $\mathcal{D}''$). We propose several variants of these novel indexing and processing methods, based on different ways of ordering and compressing the phrases in the lists. These variants in turn lead to alternative algorithms for the phrase mining, with different capabilities for pruning the search space. As the number of phrases that are contained in $\mathcal{D}'$ can be very large, we focus on finding the top-k interesting phrases, but we scale to very large corpora $\mathcal{D}$.

In a nutshell, the paper makes the following contributions:

1. a new indexing method for ad-hoc text analytics based on forward index lists,
2. a suite of scalable techniques for computing the top-k interesting phrases for different analysis tasks, and
3. a large-scale experimental study on close to two million articles from the long-term archive of the New York Times.

The rest of the paper is organized as follows. Chapter 2 defines a representative interestingness measure for phrases in an ad-hoc subset of a corpus. In Chapter 3, we present alternative methods for indexing the document corpus and searching for interesting phrases, including the framework that we propose in this paper. Chapter 4 discusses implementation details and issues for our proposed scheme. Alternative definitions for interestingness and how they are incorporated into our framework are the subjects of Chapter 5. We experimentally demonstrate the efficiency and scalability of our approaches in Chapter 6. Chapter 7 reviews related work and Chapter 8 concludes the report.

2 Problem Definition

This paper deals with mining interesting phrases in a document collection derived by a query. A phrase is a sequence of terms that appear contiguously in the text. The frequency of a phrase p in a collection of documents $\mathcal{T}$ is denoted by $freq(p, \mathcal{T})$ and defined as the number of documents in $\mathcal{T}$, which contain p.[1] Formally, $freq(p, \mathcal{T}) = count\{d : d \in \mathcal{T} \wedge p \in d\}$. Let $\mathcal{D}$ be a document corpus and $\mathcal{D}'$ be an ad-hoc subset of it containing all documents that satisfy a query. A straightforward measure for interestingness of a phrase p with respect to $\mathcal{D}'$ is simply the *local* phrase frequency $freq(p, \mathcal{D}')$. However, this favors common phrases that contain stopwords and have a high frequency in the whole corpus. Definition 1, which normalizes the occurrences of a phrase in $\mathcal{D}'$ by its *global* frequency in $\mathcal{D}$, is a more appropriate measure for interestingness.

Definition 1 *Let $\mathcal{D}'$ be an ad-hoc subcollection of a document corpus $\mathcal{D}$. Let p be a phrase. The interestingness $I_D(p, \mathcal{D}')$ of p w.r.t. $\mathcal{D}'$ is defined by:*

$$I_D(p, \mathcal{D}') = \frac{freq(p, \mathcal{D}')}{freq(p, \mathcal{D})} \tag{2.1}$$

It may not be realistic to compute it for all phrases that appear in any document of $\mathcal{D}'$. So, we define our mining problem as finding the k phrases with the highest interestingness. In the next section, we show how the corpus $\mathcal{D}$ can be pre-processed and indexed, in order to solve this problem. Our methods are not essentially restricted to Definition 1; in Section 5, we discuss alternative ways of defining interestingness and show how our indexing framework can be used for them.

[1] In accordance to previous work [27], we do not count multiple occurrences of p in the same document, because this would favor phrases that repeatedly occur in the same document over those that are spread over $\mathcal{T}$; the latter are more characteristic to the collection itself. Nevertheless, it is possible to extend our proposal for a frequency definition that takes multiple occurrences in the same document into account. The details are out of the scope of this work.

3 Indexing and Mining Techniques

In this section, we investigate methods for mining interesting phrases. We first outline a baseline method that does not rely on indexing the corpus $\mathcal{D}$. Then, we review the inverted indexing approach of [27] and discuss how it can be applied to solve our problem. Finally, we present in detail our *forward indexing* proposal, investigating alternative ways to order the contents of the lists, which either facilitate compressibility or allow for early termination of top-k interesting-phrase search in $\mathcal{D}'$.

Phrases that occur in very few documents of the corpus $\mathcal{D}$ are expected to give little insight to the analyst. Therefore, we restrict our search to only phrases that exist in a minimum number of documents τ (e.g., $\tau = 5$ or 10). We denote this set of candidate phrases by $\mathcal{C}$. We also limit ourselves to phrases with meaningful length limits, between configurable bounds *minlength* and *maxlength*. We typically consider *minlength*=2, thus disregarding all individual words – single *terms* in information-retrieval jargon, but capturing person names and composite nouns. For *maxlength*, 5 would be a canonical choice, as this still considers short catchphrases (e.g., in quotations or slogans) but excludes entire sentences. $\mathcal{C}$ can be computed at a pre-processing phase, by scanning each document $d \in \mathcal{D}$ using a sliding window to extract all phrases of lengths *minlength* to *maxlength*. Duplicate phrases in d are removed (by sorting or hashing) and then a counter is increased for each phrase in d. Finally, the phrases whose counters are at least τ are inserted to $\mathcal{C}$. As we use a small value of *maxlength*, considering phrases of various lengths simultaneously is more efficient than using a sequence mining approach (e.g., [32]) that would scan the documents multiple times.

3.1 A Baseline Method

A baseline approach for computing the local frequencies in $\mathcal{D}'$ of all phrases in $\mathcal{C}$ is to perform a window-scan over each document $d \in \mathcal{D}'$ and extract all phrases. By keeping a hash map for the phrases, we can count the documents that contain them (while we avoid double-counting phrases that appear multiple times in the same document using a hash set). In the end, assuming that we have pre-processed and indexed the global frequencies of all phrases (i.e., set $\mathcal{C}$), for each phrase in the hash map, we look up its global frequency and compute $I_D(p, \mathcal{D}')$ according to Definition 1.

This method is expected to be expensive because of the scanning cost which can be high, even for compressed text documents, since it involves tokenization (e.g., identifying sentence boundaries). The cost can be reduced if we replace the original document representations by term-ID arrays, after having defined a mapping between tokens and term-IDs at a preprocessing phase on the complete corpus. This way, the token processing cost can be saved. In addition, term IDs can be compressed to save storage. Still, even after this improvement, there is a burden of window-scanning the term-ID arrays and formulating the phrases. If a phrase appears multiple times in the same document, we still need to care for not double-counting it (by hash-set lookups). In addition, document-counters for each phrase have to be maintained and updated during the process. Overall, there is a need for more effective approaches than this baseline method.

3.2 Phrase-Inverted Indexing

As a module of their multidimensional context exploration framework, Simitsis et al. proposed in [27] a method for finding interesting phrases in an ad-hoc subset of a document corpus. Their definition for interestingness (termed *relevance* in the paper) is a normalized version of our Definition 1. Their approach to find the most interesting phrases efficiently is to create a *phrase inverted index* for the corpus as follows. During a preprocessing phase, for each phrase p, an inverted list that contains the IDs of documents that include p is constructed. Now, the local frequency of a phrase p in $\mathcal{D}'$ can be found by intersecting the inverted list of p with $\mathcal{D}'$. The global frequencies are already stored in the heads of the lists (they are equal to the list lengths); therefore, by iterating through *all* inverted lists we can find the interestingness of all phrases and determine the top-k response set. Intersecting $\mathcal{D}'$ with long lists can be accelerated by randomizing the contents of the lists and (i) skipping uniformly over the randomized lists, (ii) stopping

phrase	**contents**	I_D
$p_1(4)$	$\{d_2, d_3, d_9, d_{13}\}$	1/4
$p_2(4)$	$\{d_4, d_5, d_{12}, d_{18}\}$	4/4
$p_3(4)$	$\{d_5, d_8, d_{16}, d_{17}\}$	2/4
$p_4(4)$	$\{d_9, d_{12}, d_{16}, d_{19}\}$	2/4
$p_5(5)$	$\{d_4, d_5, d_{12}, d_{16}, d_{19}\}$	3/5
$p_6(6)$	$\{d_3, d_4, d_5, d_9, d_{12}, d_{18}\}$	5/6
$p_7(8)$	$\{d_1, d_2, d_4, d_9, d_{12}, d_{15}, d_{16}, d_{17}\}$	5/8
$p_8(9)$	$\{d_3, d_4, d_8, d_9, d_{12}, d_{14}, d_{17}, d_{18}, d_{20}\}$	6/9
$p_9(10)$	$\{d_1, d_3, d_4, d_5, d_9, d_{10}, d_{12}, d_{17}, d_{18}, d_{19}\}$	7/10
$p_{10}(10)$	$\{d_1, d_2, d_3, d_4, d_5, d_8, d_{10}, d_{12}, d_{17}, d_{20}\}$	6/10
$p_{11}(11)$	$\{d_2, d_4, d_5, d_6, d_9, d_{12}, d_{13}, d_{15}, d_{17}, d_{18}, d_{20}\}$	7/11
$p_{12}(12)$	$\{d_1, d_3, d_4, d_5, d_7, d_9, d_{10}, d_{12}, d_{13}, d_{17}, d_{18}, d_{20}\}$	8/12

Table 3.1: Example of a phrase-inverted index

after a maximum number of comparisons or when a large enough intersection is already found. This way, an accurate approximation of the local frequency can be obtained (see [27] for more details).

As an example, consider a corpus $\mathcal{D}$ of 20 documents and a set $\mathcal{C} = \{p_1, p_2, \ldots, p_{12}\}$ of phrases, such that $\forall p_i \in \mathcal{C}$, $freq(p_i, \mathcal{D}) \geq \tau$, $\tau = 4$. Table 3.1 (left) shows the inverted lists for the phrases in $\mathcal{C}$. The length of each list p_i is shown in brackets and the list contents are the document IDs that contain the phrase. Consider a subset of $\mathcal{D}$ containing documents $\mathcal{D}' = \{d_1, d_4, d_5, d_9, d_{12}, d_{17}, d_{18}, d_{20}\}$. By intersecting $\mathcal{D}'$ with each inverted list we can compute the interestingness of the corresponding phrase, as shown in Table 3.1 (right). For $k = 2$, the top-k phrases with the highest interestingness are p_2 and p_6, with $I_D(p_2, \mathcal{D}') = 4/4$ and $I_D(p_6, \mathcal{D}') = 5/6$, respectively.

Approximate top-k interesting-phrase mining. The main drawback of this approach is that the whole set of inverted lists must be scanned to obtain the result. In a typical corpus consisting of few thousand documents millions of phrases can be found. In order to avoid scanning all inverted lists, [27] suggest computing the K phrases ($K > k$) with the highest local frequency first, and then post-processing them using the interestingness formula to find the k most interesting phrases in them. To find these K phrases efficiently, they store and access the phrase-inverted lists in *decreasing* global frequency order, and terminate as soon as the K-th smallest local frequency found is greater than the global frequencies of the remaining phrases. However, there is no guarantee that the real k most interesting phrases will be included in the K locally most frequent ones. Consider again the example of Table 3.1 and assume that we find the $K = 6$ phrases with the largest local frequency first and then the top-k most interesting ones among these for $k = 2$. Finding the K locally most frequent phrases requires scanning all inverted lists of Table 3.1 in reverse order until p_5. By then, there are already $K = 6$ phrases of local frequency 5 or more (these are p_6 to p_{12}). However, the top-2 interesting phrases p_{12} to p_5 do not include p_2, which is among the two actual most interesting phrases in $\mathcal{D}'$. Therefore, the method of [27] has to be regarded as an approximate method.

Exact top-k interesting-phrase mining. Although such an approach is not investigated in [27], we observe that the phrase-inverted index can be used to find the actual k most interesting phrases, if we access the lists in *increasing* order of global frequency. (In Section 3.4 we propose an analogous method on our indexing scheme.) In a nutshell, we can access the inverted lists and compute the interestingness of the corresponding phrases, while updating the set R with the top-k interesting phrases found so far. Bound θ is defined as the k-th lowest interestingness in R. Let p_{next} be the next phrase to be accessed in this order. If $|\mathcal{D}'|/freq(p_{next}, \mathcal{D}) \leq \theta$, then p_{next} and all succeeding phrases (which have no lower global frequency than $freq(p_{next}, \mathcal{D})$) may not end up in the top-k result, which means that we can terminate without accessing any more lists. Consider again the example of Table 3.1 and assume that we access the lists in ascending order of the global phrase frequencies, searching for the top-2 interesting phrases. After accessing all lists from p_1 to p_8, the top-2 interestingness values so far are $I_D(p_2, \mathcal{D}') = 4/4$ and $I_D(p_6, \mathcal{D}') = 5/6$, therefore $\theta = 5/6$. Note that we can avoid intersecting p_9's list (and all other lists after it), since the highest possible local frequency of p_9 is $|\mathcal{D}'| = 8$ and its length is 10, so the highest possible interestingness of p_9 is $8/10 < \theta$. Since phrases after p_9 can only have global frequency larger than 10, they can all be safely skipped.

3.3 Forward Indexing

Our *forward indexing* approach creates for each document $d \in \mathcal{D}$ a list F_d, which contains the phrases from $\mathcal{C}$ that are included in d. Thus, we replace the (exact) textual content of a document, by the phrases from $\mathcal{C}$ included in it. For simpler presentation, assume for now that we capture all phrases of $\mathcal{C} \cap d$; in Section 3.5, we will show that it is possible to work with merely a suitably chosen subset of $\mathcal{C}$.

While in [27] the inverted lists of the vast majority of phrases are intersected with the input set of documents $\mathcal{D}'$, in our approach, we intersect the forward lists only of the documents in $\mathcal{D}'$, in order to obtain the frequency of phrases in $\mathcal{D}'$. The advantage is that the number of accessed lists for finding the frequent phrases directly depends on the size of $\mathcal{D}'$, not $\mathcal{D}$.

The basic algorithm on this scheme (Algorithm 3.3) performs a $|\mathcal{D}'|$-way merge join of the lists, by scanning the sorted input lists in parallel. This is easily implemented by a $|\mathcal{D}'|$-length min-heap of pointers to the current read positions of the merged lists. During the merge, for the current phrase p, the local frequency $freq(p, \mathcal{D}')$ can be counted by the number of inputs where p is seen. The interestingness of p can be determined by accessing its global frequency. Global frequencies can either be explicitly stored in the forward lists (e.g., embedded in phrase IDs) or kept in a separate phrase-dictionary in memory. Thus, we can update the set of top-k interesting phrases after every output of the merge operator and the whole process can terminate with one synchronized scan over the forward lists of the documents in $\mathcal{D}'$. Depending on how the phrases in the forward lists are ordered, in the rest of this section, we investigate different alternatives of our indexing scheme, that improve upon this basic algorithm.

Algorithm 1 Basic Forward Index Search

```
   FWB($\mathcal{D}'$, $k$)
1: $R = \emptyset$                    ▷ contains $k$ phrases of highest $I_D(p, \mathcal{D}')$
2: perform $|\mathcal{D}'|$-way merge join of the lists $F_d$ for all $d \in \mathcal{D}'$
3: **for** each output phrase $p$ **do**
   compute $freq(p, \mathcal{D}')$, compute $I_D(p, \mathcal{D}')$ and update $R$ to include $p$ if applicable
4: return $R$
```

3.4 Frequency-based Ordering and Early Termination

It is possible to adapt Algorithm 3.3 to avoid scanning the forward lists F_d for each $d \in \mathcal{D}'$ completely, if the order of the phrases p in each list is defined according to its global frequency $freq(p, \mathcal{D})$. That is, less frequent phrases appear first in the forward lists of the documents. While the lists are scanned in parallel by Algorithm 3.3 (and R is being updated), we know that for any phrase p that has not been seen so far, the maximum possible $I_D(p, \mathcal{D}')$ value is

$$max^p_{I_D} = \min\left\{1, \frac{|\mathcal{D}'|}{freq(p, \mathcal{D})}\right\} \tag{3.1}$$

Due to the ordering of phrases in the lists, if phrase p precedes phrase q in this order, $max^q_{I_D} \leq max^p_{I_D}$. Therefore, as soon as for the next unexamined phrase p, $max^p_{I_D} \leq \theta$, where θ is the lowest interestingness of phrases in the current top-k set R, the algorithm can safely terminate reporting R as the result. Algorithm 3.4 shows how Algorithm 3.3 can be extended along these lines.

Sorting the phrases according to their global frequencies in the forward lists allows for an economic embedding of these frequencies into the lists. Since multiple phrases share the same frequency, we can use a representation that stores the phrases that have the same frequency as a group and the frequency once in the heading of the group. This way, the global frequency needs not be accessed at an external hash table. Table 3.2 is an illustration of this storage model. Alternatively, global frequencies can be embedded in the identifiers of the phrases, occupying the most significant bits, to force frequency-based ordering and access to frequencies by means of ID decoding.

Algorithm 2 Forward Index Search with Early Termination

```
   FWE($\mathcal{D}'$, $k$)
1: $R = \emptyset$                          ▷ contains $k$ phrases of highest $I_D(p, \mathcal{D}')$
2: perform $|\mathcal{D}'|$-way merge join of the lists $F_d$ for each $d \in \mathcal{D}'$
3: for each $p$, output by the merge operator do
       compute $freq(p, \mathcal{D}')$
       compute $I_D(p, \mathcal{D}')$
       update $R$ to include $p$ if applicable
       $max^p_{I_D} = \min\{1, \frac{|\mathcal{D}'|}{freq(p,\mathcal{D})}\}$
       $\theta = k$-th interestingness value in $R$
       if $max^p_{I_D} \leq \theta$ terminate merge-join
4: return $R$
```

As an example, let $\mathcal{D}' = \{d_1, d_4, d_5, d_9, d_{12}, d_{17}, d_{18}, d_{20}\}$. Table 3.2 shows the forward lists of the documents in $\mathcal{D}'$, wherein the phrases are grouped and ordered in ascending global frequency order. Assume that we are looking for the top-2 most interesting phrases. The forward lists are synchronously scanned and merged. At the point after $|\mathcal{D}'|$ equals the global frequency of the last accessed phrase p_{last} from the merger (i.e., $p_{last} = p_7$), we can start applying the early termination test. At this stage the 2 phrases with the highest interestingness are $R = \{p_2, p_6\}$ and $\theta = freq(p_6, \mathcal{D}') = 5/6$. Since $\theta < |\mathcal{D}'|/freq(p_{last}, \mathcal{D}')$, we cannot terminate, as it is possible to find a phrase later that can enter the top-2 set R. The algorithm terminates after p_9 is found and $|\mathcal{D}'|/freq(p_9, \mathcal{D}') = 8/10 < \theta$. In practice, as demonstrated by our experiments in Section 6, this method can prune a large percentage of the lists, especially if they contain phrases with very high global frequency in their tails.

list	**contents**
F_{d_1}	$(9)p_8$ $(10)p_9, p_{10}$ $(12)p_{12}$
F_{d_4}	$(4)p_2$ $(5)p_5$ $(6)p_6$ $(8)p_7$ $(9)p_8$ $(10)p_9, p_{10}$ $(11)p_{11}$ $(12)p_{12}$
F_{d_5}	$(4)p_2, p_3$ $(5)p_5$ $(6)p_6$ $(10)p_9, p_{10}$ $(11)p_{11}$ $(12)p_{12}$
F_{d_9}	$(4)p_1, p_4$ $(6)p_6$ $(8)p_7$ $(9)p_8$ $(10)p_9$ $(11)p_{11}$ $(12)p_{12}$
$F_{d_{12}}$	$(4)p_2, p_4$ $(5)p_5$ $(6)p_6$ $(8)p_7$ $(9)p_8$ $(10)p_9, p_{10}$ $(11)p_{11}$ $(12)p_{12}$
$F_{d_{17}}$	$(4)p_3$ $(8)p_7$ $(9)p_8$ $(10)p_9, p_{10}$ $(11)p_{11}$ $(12)p_{12}$
$F_{d_{18}}$	$(4)p_2$ $(6)p_6$ $(9)p_8$ $(10)p_9$ $(11)p_{11}$ $(12)p_{12}$
$F_{d_{20}}$	$(9)p_8$ $(10)p_{10}$ $(11)p_{11}$ $(12)p_{12}$

Table 3.2: Example of a forward index

3.5 Prefix-maximal Phrases and Lexicographic Phrase Ordering

An important issue with the forward indexing scheme is its space requirements. Since all phrases in $\mathcal{C}$ that are contained in a document must be included in the corresponding forward list, the sizes of the lists may grow significantly, even after compressing the phrase representations in them. In this section, we investigate the possibility of reducing the sizes of these lists by avoiding to index all phrases in them.

The main idea comes from the monotonicity of $freq(p, \mathcal{D})$ with respect to the set-containment relationships between phrases. If a phrase p is included in a document d, all sub-phrases $p' \subset p$ should also be present in d. Therefore, if we include p in the forward list F_d of d, it is not necessary to add any $p' \subset p$ there; the contents of the forward lists can be minimized, if we include only the maximal-length phrases (in $\mathcal{C}$) that are present in each document.

Figure 3.1(a) shows the contents of three documents. For ease of presentation, we denote distinct terms by characters and the content of a document as a string (i.e., sequence of terms). Figure 3.1(b) shows the set of distinct phrases of length $minlength = 1$ to $maxlength = 4$ that occur in each document. Assuming that $\mathcal{C}$ contains only phrases up to length 4 and that all such phrases are included in $\mathcal{C}$, Figure 3.1(c) shows the maximal-length phrases in each document. Clearly, if we include only these, the lengths of the forward lists can be greatly reduced.

On the other hand, this compressed representation complicates evaluation. Now, computing $freq(p, \mathcal{D}')$ for an arbitrary phrase p cannot be achieved by simply scanning (and intersecting) the forward lists (i.e., using Algorithm 3.3). For example, assuming that $\mathcal{D}' = \{d_1, d_2, d_3\}$, it is not clear how the local frequency of phrase "bef" can be obtained by intersecting the lists shown in Figure 3.1(c).

A brute-force algorithm that applies on lists with maximal phrases would scan the lists one-by-one and derive counts for subphrases dynamically. That

doc	contents
d_1	abcabef
d_2	defabce
d_3	bdaceda

(a)

doc	phrases
d_1	a,ab,abc,abca,abe,abef,b,bc,bca, bcab,be,bef,c,ca,cab,cabe,e,ef,f
d_2	a,ab,abc,abce,b,bc,bce,c,ce,d,de, def,defa,e,ef,efa,efab,f,fa,fab,fabc
d_3	a,ac,ace,aced,b,bd,bda,bdac,c,ce, ced,ceda,d,da,dac,dace,e,ed,eda

(b)

doc	maximal phrases
d_1	abca, abef, bcab, cabe
d_2	abce, defa, efab, fabc
d_3	aced, bdac, ceda, dace

(c)

doc	prefix-maximal phrases
d_1	abca, abef, bcab, bef, cabe, ef, f
d_2	abce, bce, ce, defa, efab, fabc
d_3	aced, bdac, ceda, dace, eda

(d)

merged stream
abca(1), abce(1), abef(3), aced(1), bcab(1), bce(1), ...

(e)

Figure 3.1: Example of prefix-maximal lists

is, for each document in $\mathcal{D}'$, the corresponding list is scanned and for each phrase encountered, we increase frequencies $freq(p, \mathcal{D}')$ of all its subphrases. In order to avoid double-counting a common subphrase of multiple maximal phrases, we use a hash table to check if a subphrase has been seen again in that document. This method is expected to be computationally expensive, due to the dynamic decomposition of each list posting to subphrases and the hash table look-ups for each of them.

Fortunately, we can still avoid indexing the majority of non-maximal phrases and run Algorithm 3.3, without any additional book-keeping of non-maximal phrase counters in a hash table. Our proposal is a forward index that includes only *prefix-maximal* phrases in the lists:

Definition 2 *A phrase p is prefix-maximal w.r.t. document $d \in \mathcal{D}$, if (i) $p \in \mathcal{C}$ (i.e., p is globally frequent), (ii) $p \in d$, and (iii) there exists no phrase $p' \in \mathcal{C}$, such that p is a prefix of p' and $p' \in d$.*

For example, in Figure 3.1(a), phrase “abc” is not prefix-maximal w.r.t. d_1 because d_1 also contains “abcd” and “abc” is a prefix of “abcd”.

Now, we adapt the forward index to contain for each document d only the prefix-maximal phrases w.r.t. d. In addition, in each forward list, the phrases are ordered lexicographically. Figure 3.1(d) shows the prefix-maximal forward lists for the documents of Figure 3.1(a). Observe that the lists are only slightly larger than those containing only maximal phrases (Figure 3.1(c)), but significantly smaller than the lists containing all phrases (Figure 3.1(b)).

Having constructed this index, the challenge is to adapt Algorithm 3.3 to compute the interestingness of all phrases contained in a set of documents. This is not straightforward, as we need to avoid over-counting of subphrases that are not explicitly stored in the forward lists but implied by prefix-maximal phrases which contain them. Algorithm 3 is a pseudocode of this adaptation. The lists are merged, and the prefix-maximal phrases at all inputs are retrieved in lexicographical order. Figure 3.1(e) shows the order by which the phrases are retrieved when merging the lists of Figure 3.1(d) — ignore the numbers in parentheses for the moment. At each access, the algorithm identifies the changes in the prefix of the current phrase p, compared to the previous phrase p_{prev} (Line 10).[1] Specifically, it finds the longest common prefix s between p and p_{prev}. Every prefix of p_{prev} that is longer than s corresponds to a phrase that can never be seen subsequently by the algorithm (because prefix-maximal phrases are sorted lexicographically in the lists). Thus, the local frequencies of all these prefixes in $\mathcal{D}'$ can be immediately determined and output (Lines 13–16).

In order to compute the local frequencies of all prefixes correctly, we keep a list of counters C, one for each possible phrase length. For each new prefix-maximal phrase p encountered, say from document d_j we check its difference to the previous phrase p^j_{prev} seen at document d_j (Line 9). For all positions in s, where p_{prev} and p^j_{prev} differ, we increase the corresponding counter, in order to take into consideration that the corresponding prefixes have been seen at d_j (Line 11–12) and avoiding over-counting for the prefixes where p_{prev} and p^j_{prev} are common (these have been counted when p^j_{prev} was considered). When comparing p with p_{prev}, for all prefixes that are different, we compute the local frequencies of the corresponding prefixes using the counters (Lines 13–16). Then, for the current phrase p, we initialize the counters for the new prefixes to 1. Finally, after the last phrase is accessed from the merged input, we use the existing counters to count and output the frequencies of all its prefixes (Lines 20–23). Summing up, Algorithm 3 correctly computes the local frequencies of all phrases by merging the forward lists containing the prefix-maximal phrases.

p^j_{prev} can be computed if the algorithm (or the merger) keeps track of the previous phrase seen at each input. In our implementation, we avoid its computation by embedding this information in the inverted lists. That is, together with the phrase representation, we also store for each phrase the first position where it differs from the previous phrase in the same list. This information is also used for compressing the lists, as it indicates the common prefix between consecutive phrases; this prefix needs not be repeated in the representation of the next phrase (see Section 4.1). Figure 3.1(e) shows, for each phrase p that is produced by the merger, the first position where p differs with p^j_{prev} in brackets.

[1]Since we merge multiple inputs, p could be identical to p_{prev}.

Algorithm 3 Merging Prefix-Maximal Forward Lists

```
FWP($\mathcal{D}'$, $k$)
                         ▷ outputs the local frequencies of all phrases in $\mathcal{D}'$
1: $C$ = new array of counters                                    ▷ initially 0
2: $p$ = first phrase from merger
3: $j$ = input where $p$ was seen
4: for all positions $i$ of $p$ do
5:     $C[i] = C[i] + 1$                  ▷ count prefixes of $p$ seen at $j$-th input
6: $p_{prev} = p$                                                  ▷ track prev. phrase
7: while $p$ = next phrase from merger do                        ▷ while more phrases
8:     $j$ = input where $p$ was seen
9:     $jpos$ = first position where $p$ and $p^j_{prev}$ differ
10:    $dpos$ = first position where $p$ and $p_{prev}$ differ
11:    for $i$ = $jpos$ to $dpos$-1 do                           ▷ count current input
12:        $C[i] = C[i] + 1$
13:    $outp$ = first $dpos - 1$ terms of $p_{prev}$                  ▷ common prefix
14:    for $i$ = $dpos$ to $len(p_{prev})$ do                    ▷ for each diff. prefix
15:        append $p_{prev}[i]$ to $outp$
16:        $freq(outp, \mathcal{D}') = C[i]$                            ▷ set frequency
17:    for $i$ = $dpos$ to $len(p)$ do                              ▷ reset counters
18:        $C[i] = 1$
19:    $p_{prev} = p$                                              ▷ track prev. phrase
20: $outp = \emptyset$                             ▷ handle all prefixes of last phrase
21: for $i = 1$ to $len(p_{prev})$ do
22:     append $p_{prev}[i]$ to $outp$
23:     $freq(outp, \mathcal{D}') = C[i]$
```

As an example, let us see how the local frequencies of all phrases in $\mathcal{D}' = d_1, d_2, d_3$ can be computed, using the forward index of Figure 3.1(d). First, we initialize an array of 4 counters, assuming that the minimum (maximum) length of a phrase in $\mathcal{C}$ is 1 (4). The first phrase that comes from the merger (see Figure 3.1(e)) is "abca" from input d_1, therefore we set $p_{prev} =$"abca" and add 1 to all 4 counters. The next phrase is "abce" (seen at input $j = d_2$) and the first positions where it differs with p^j_{prev} and p_{prev} are *jpos*=1 and *dpos*=4, respectively (p^j_{prev} is null, as p is the first phrase seen at input $j = d_2$). The algorithm will increase the counters for positions 1 though 3, reflecting that the prefixes "a", "ab", and "abc" have been seen also at input $j = d_2$. In addition, "abca" (i.e., the common prefix "abc" of p_{prev} and p padded with the last term of p_{prev}) will be output, with frequency $C[4]$=1. Next, counter $C[4]$ is reset to 1 and p_{prev} are updated to "abce", before accessing the next phrase $p =$"abef" from the merger. This phrase differs at positions *jpos*=3 and *dpos*=3 from p^j_{prev}="abca" and p_{prev}="abce", respectively. No counters are updated (since *jpos*=*dpos*), and the algorithm outputs "abc" and "abce" with frequencies 2 and 1, respectively, while resetting $C[3]$ and $C[4]$ to 1. Note that at any stage the counters will correctly capture the number of inputs where all prefixes of p_{prev} have been seen.

Algorithm 3 computes only the local frequencies of phrases (at Lines 16 and 23) but not their interestingness. The global frequencies of the non-maximal phrases (required for the computation of interestingness) are not explicitly included in the index, so for the interestingness of each phrase to be computed we need to do a look-up at a hash table that stores the global frequencies of all phrases. This table typically fits in memory and searches are efficient. In the case where the table exceeds the available memory, look-ups can be avoided by storing the global frequencies of all phrases in $\mathcal{C}$ in a file, sorted lexicographically. Since the local frequencies of the phrases in $\mathcal{D}'$ are also computed in lexicographic order by the algorithm, we could immediately feed them to a merger with the global frequency file that would output the interestingness of each phrase.

Summing up, Algorithm 3 operates on a very economical representation of the forward lists. On the other hand, it has to exhaustively access all merged lists, being unable to apply early termination heuristics like Algorithm 3.4, due to the lexicographic ordering of phrases in the lists as required for this method.

3.6 Analytical Comparison

In this subsection, we do a worst-case analysis for the complexities of the different indexes and search methods that have been presented.

Space complexity of the indexes. For each phrase occurrence in each document, the phrase-inverted index (Section 3.2) and the basic version of the forward index (Sections 3.3 and 3.4) include a posting in a phrase-inverted list and a forward list, respectively. Thus the total number of postings in both indexing methods is identical. Let ρ be the maximum length of phrases in $\mathcal{C}$. In the worst-case, there are ρ different phrases per term position in each document, so the overall storage requirements of these indexes (disregarding compression) are $O(\rho \cdot T(\mathcal{D}))$, where $T(\mathcal{D})$ is the total number of term occurrences in the corpus $\mathcal{D}$. This could be much larger than just storing the sequence of term IDs for each document (i.e., the index used by the baseline method of Section 3.1), at a cost of $O(T(\mathcal{D}))$. Indexing prefix-maximal phrases only (as described in Section 3.5) brings down the cost of forward indexing from $O(\rho \cdot T(\mathcal{D}))$ to $O(T(\mathcal{D}))$ (i.e., the worst-case cost of the baseline method); we only have to slide a ρ-length window over each document d, extract all the ρ-length phrases, and add to them the last $\rho - 1$ suffixes of d. Thus, if ρ is much smaller than the average document length, the storage cost is $O(T(\mathcal{D}))$.

Space and time complexities of the algorithms. All algorithms, except for the baseline method (Section 3.1), have low memory requirements as no bookkeeping for temporary counting of phrase frequencies is required by them. The baseline method scans the documents in $\mathcal{D}'$ one-by-one and generates all phrases dynamically, while counting their frequencies in a hash table. The time complexity of this method is high, as $O(\rho \cdot T(\mathcal{D}'))$ phrases are generated and counted. The methods that operate on the phrase-inverted index (Section 3.2) must scan the whole index in the worst case and perform intersections of $\mathcal{D}'$ with all lists. This results in a very high $O(|\mathcal{C}| \cdot |\mathcal{D}'| + \rho \cdot T(\mathcal{D}))$ worst-case cost, as $\mathcal{D}'$ must be read and intersected $|\mathcal{C}|$ times and all postings in the index should be read. In practice, the cost is lower, due to approximate counting and early termination possibilities.

The worst-case cost of Algorithm 3.4 (and the simpler Algorithm 3.3) is that of accessing all $|\mathcal{D}'|$ forward lists, of average length λ. During the merging of the lists, and each time a posting is produced by the merger, the merger itself requires $\log |\mathcal{D}'|$ time to update its priority queue. Thus, the overall cost of Algorithms 3.3 and 3.4 is $O(|\mathcal{D}'| \log |\mathcal{D}'| \cdot \lambda)$. For Algorithm 3, the cost at each iteration is dominated by looping over the positions of the current phrase p and comparing it with p_{prev} to determine $dpos$. Two more loops are required to generate the prefixes to be output and reset the counters.

Thus, the cost per access from the merger is $O(\rho)$. Assuming that the average length of a forward list in the prefix-maximal index of Section 3.5 is $\lambda' < \lambda$, there are $|\mathcal{D}'|$ lists and $|\mathcal{D}'|\lambda'$ accesses in total. Thus, the overall cost is $O(|\mathcal{D}'|\cdot\lambda'(\rho+\log|\mathcal{D}'|))$. Typically $\rho < \log|\mathcal{D}'|$, thus theoretically Algorithm 3 is more efficient than Algorithm 3.4, while also requiring less space.

4 Implementation Details

In this section we discuss different phrase representations used in the baseline method (cf. Section 3.1) and the forward indexing variants proposed in Sections 3.3–3.5. In addition, we discuss the issue of random I/Os, which may become a bottleneck of forward indexing.

4.1 Representations and their Trade-offs

In total, we have implemented five different representations for the contents of a document.

1. TermID. In this baseline implementation, we use integer identifiers to encode the terms of each document. Thus, the document, originally a sequence of terms, is represented as a sequence of term IDs. Compressibility of term IDs is low, i.e., only 7bit encoding is used [30]. This representation only makes sense for the enhanced baseline method as discussed in Section 3.1; at query time we need to perform a window scan over each input in order to compute the phrases.

2. IntArray. All phrases are converted to integer arrays containing the term IDs of its corresponding terms. For instance, a phrase "Steve Jobs" would be translated to [442,312] assuming that *Steve* corresponds to term ID 442, and *Jobs* corresponds to term ID 312. Integer arrays may have different length. Each forward list keeps phrases in lexicographical order. At query time we do a merge based on the lexicographical order. Global phrase frequencies are not encoded in the index, thus we have to look them up at a phrase dictionary. This dictionary stores for each int array that is a valid phrase the frequency of the phrase in the corpus. Compressibility of the index is high, as only changed suffixes have to be stored for consecutive phrases in a list. Individual integers are compressed using 7bit encoding. Algorithm 3.3 can be applied on this index.

3. PrefixMaxIntArray. This scheme is similar to IntArray. However, we only keep the prefix-maximal phrases in each list. For instance, if a document contains phrases [442,312,7,2], [442,312,7], [442,312], we only store [442,312,7,2] in its list. At query time we run Algorithm 3 to perform the merge. Again, global phrase frequencies are not encoded in the index and need to be looked up in a phrase dictionary. Compressibility is high.

4. PhraseID. We globally assign phrase IDs, according to their global frequency. For each phrase p, we set the first 32 bit-block of its phrase ID (of type long) to the global frequency $freq(p, \mathcal{D})$. The second 32 bit-block is used to break ties among phrases having the same global frequency.[1] Each forward list keeps all phrases of the corresponding document in ascending phrase ID order. At query time we do a merge based on this order. This order allows us to apply the early termination Algorithm 3.4. Compressibility is high as adjacent phrase IDs may be delta-encoded.

5. FreqIntArray. We again use integer arrays for individual phrases. However, we keep the global frequency of each phrase explicitly in the list, as explained in Section 3.4 and illustrated in Table 3.2. For each frequency, we keep the group of phrases having that frequency. Thus, in contrast to PhraseID each frequency has to be stored only once (using an integer instead of a long). Inside each group phrases are ordered lexicographically. For instance, assume that a document has phrases [1,3,7,2], [4,3,2], [3,1] with a global frequency of 42 and [7,3,2], [8,3,2,9], [2,8] with a global frequency of 12. Then, the corresponding forward list is $\{12 \mapsto ([2,8],[7,3,2],[8,3,2,9]), 42 \mapsto ([1,3,7,2],[3,1],[4,3,2])\}$. An advantage of this scheme is that frequencies of successive phrases are not repeated and they also do not need to be looked up in any global phrase dictionary. Compressibility is lower than IntArray and PhraseID as the variation among adjacent entries is higher.

Tables 4.1 and 4.2 summarize the features and trade-offs of the individual representations. Note that all methods require a term dictionary mapping from term IDs to actual terms in order to report string phrases. Search engines typically have such a term dictionary available anyway, and, to integrate our method into an existing system, we would leverage that term dictionary. Note that all representations except TermID provide interesting

[1]This technique can be generalized into dividing a 64-bit block into variable-length parts for the frequency and tie-breaker encodings.

orders as output. Therefore, we may easily merge outputs from different queries as required for the alternative interestingness measures to be discussed in Section 5.

representation	**phrase dict**		**term**
	global freq	**terms**	**dict**
TermID	yes	no	yes
IntArray	yes	no	yes
PrefixMaxIntArray	yes	no	yes
PhraseID	no	yes	yes
FreqIntArray	no	no	yes

Table 4.1: Dictionaries required at query time for different phrase representations

representation	**index order**	**compressibility**
TermID	as in doc	low
IntArray	lexicographic	high
PrefixMaxIntArray	lexicographic	high
PhraseID	phrase ID	high
FreqIntArray	(freq, lexicographic)	medium

Table 4.2: Index sort orders and compressibility for different phrase representations

4.2 Handling Random I/O

As noted above, our methods scale linearly in the size of $|\mathcal{D}'|$. This is because we have to fetch the forward lists of all documents in $\mathcal{D}'$ and then perform a merge operation for them. In case the forward lists are stored on a hard disk and using a single hard disk, we expect to have a minimal query response time of $|\mathcal{D}'| \cdot T_{\text{latency}}$ assuming an average disk latency of $T_{\text{latency}} = 5ms$. Thus, a disk-based system would suffer from this random I/O cost. There are several approaches to overcome this problem.

First, $\mathcal{D}'$ might be reduced to a sample $S(\mathcal{D}') \subset \mathcal{D}'$ where $S(\mathcal{D}')$ contains the query results with the highest rank. Then only $S(\mathcal{D}')$ will be considered in our algorithms. As we discuss in the next section, the interestingness measure can be weighed to take into account the relevances of the documents to the query, favoring this type of biased sampling. Alternatively, random sampling may be used to derive $S(\mathcal{D}')$. In the future, we are planning to explore these ideas.

Second, we could compute the result progressively by partitioning $\mathcal{D}'$ into a sequence of m disjoint subsets $\mathcal{D}'_i \subset \mathcal{D}'$, such that $\mathcal{D}'_i \neq \mathcal{D}'_j \; \forall 1 \leq i < j \leq m$; $\bigcup_{i=1}^{m} \mathcal{D}'_i = \mathcal{D}'$. We would then apply our algorithms to $\mathcal{D}'_1$ first and report the result to the user. While the user is exploring the result, we would apply our approach successively to all $\mathcal{D}'_i, i > 1$, each time updating the result. Note that we would have to keep all intermediate results and merge them with $\mathcal{D}'_n$ in the end. This approach is similar to online aggregation in OLAP [16].

Third, we are currently witnessing a migration of computer system towards using flash-based storage [5]. Solid state-disks have a $T_{\text{latency}} = 0.1ms$. Upcoming PCI-based flash memory [13] has a latency of $T_{\text{latency}} = 0.05ms$. Furthermore, upcoming flash-devices will combine multiple chips to a RAID thus enabling concurrent random read and write operations. So in contrast to the observations in in [5]^2 these devices will support parallel I/O supporting up to 100,000 random I/O read operations per second [13]. As our approach does not depend on the order in which documents are processed, they are able to make full use of this performance.

Fourth, for real search engines, it is highly questionable whether they suffer much from hard disk-based random I/O. For instance, big search engines today are already main memory based, e.g. Google [8] and SAP TRex [4]. In these systems T_{latency} is in the order of nanoseconds. Furthermore, systems like Google already use a horizontal partitioning on document IDs [8]. Thus our approach would perfectly fit into such a system.

^{2}The authors of [5] could not test fusion-io as it was not made available on time. An extension of their study to include this device is planned.

5 Other Definitions of Interestingness

In this section, we discuss alternatives to Definition 1 for assessing the interestingness of phrases and explain how our forward indexing approaches can be used for each of these measures.

PMI-based surprising frequency. Point-wise mutual information (PMI) is a measure from information theory for quantifying surprise in the co-occurrence of terms, based on their joint (multivariate) probability versus independent (univariate) probabilities [31]. Based on this, Definition 3 measures interestingness of a phrase by dividing its frequency in $\mathcal{D}'$ by the product of the individual terms' frequencies in $\mathcal{D}$.[1]

Definition 3 *Let $\mathcal{D}'$ be an ad-hoc subcollection of a document corpus $\mathcal{D}$. Let p be a phrase. The interestingness $I_D(p, \mathcal{D}')$ of p w.r.t. $\mathcal{D}'$ is defined by:*

$$I_D(p, \mathcal{D}') = log \frac{freq(p, \mathcal{D}')}{\Pi_{t \in p} prob(t, \mathcal{D})} \tag{5.1}$$

where $prob(t, \mathcal{D})$ denotes the probability of term t to occur in a random document from $\mathcal{D}$.

We can compute the interestingness of p without having to know its global frequency $freq(p, \mathcal{D})$, but only the global frequencies of the terms, which are much fewer than the phrases that appear in the corpus. All methods discussed in Section 3 can be implemented to apply Definition 3, if we replace the global frequencies of the phrases (wherever they are stored in the indexes) by the corresponding normalized term probability products. These products

[1] In the classic PMI definition, the frequencies in the numerator and denominator refer to the same set (i.e., either $\mathcal{D}$ or $\mathcal{D}'$). Definition 3 complies with our interestingness concept, where the frequency of a phrase in $\mathcal{D}'$ is divided by the expected frequencies of its terms in the whole corpus.

can either be pre-processed and incorporated in the index or computed on-the-fly by look-ups against a term dictionary.

Differential frequency to another query result. Instead of examining interestingness with respect to the whole corpus $\mathcal{D}$, an analyst may wish to identify phrases that occur surprisingly frequently in $\mathcal{D}'$ compared to another ad-hoc subset $\mathcal{D}''$ of $\mathcal{D}$, which is derived by another query. $\mathcal{D}'$ and $\mathcal{D}''$ could have any set-relationship (e.g., disjoint, overlap, containment). For instance, assume that we are interested in phrases that appear surprisingly frequently in documents $\mathcal{D}'$ relevant to "Steve Jobs" compared to documents $\mathcal{D}''$ relevant to "Bill Gates". For this purpose we can use the following definition:

Definition 4 *Let $\mathcal{D}'$ and $\mathcal{D}''$ be two ad-hoc subcollections of a document corpus $\mathcal{D}$. Let p be a phrase. The interestingness $I_D(p, \mathcal{D}')$ of p w.r.t. $\mathcal{D}''$ is defined by:*

$$I_{D''}(p, \mathcal{D}') = \frac{1 + freq(p, \mathcal{D}')}{1 + freq(p, \mathcal{D}'')} \tag{5.2}$$

We add 1 to both frequencies in order to prevent division by zero if $freq(p, \mathcal{D}'')$ is 0. Finding the most interesting phrases based on this definition using the suggested techniques is still possible, however, the early termination heuristics may not be applicable. The reason is that the phrases in the forward lists cannot be pre-processed and ordered with respect to any ad-hoc subset $\mathcal{D}''$ of the corpus. Therefore, we need to first find the local frequencies of *all* phrases in $\mathcal{D}'$ and $\mathcal{D}''$, intersect them, and pick the top-k interesting phrases according to Definition 4. Note that all methods described in Section 3 can be used to compute the local frequencies of all phrases, by disregarding global frequencies and any pruning/termination condition. More importantly, our forward indexing approaches facilitate evaluation in an operator-based fashion. If the same algorithm is applied twice as two operators that compute the local phrase frequencies in $\mathcal{D}'$ and $\mathcal{D}''$ in parallel, the results from the two operators can be pipelined to a merger, which computes the interestingness of each phrase on-the-fly (as phrases are examined in the same order) and updates the top-k set.

OLAP-style frequency analysis. Finally, our measures (e.g., Definition 1) can be applied at different partitions of a document set, facilitating an OLAP-style analysis of interestingness, as shown below.

Definition 5 *Let $\mathcal{D}'$ be a subcollection of a document corpus $\mathcal{D}$ comprising the documents that satisfy a keyword query q. Let $\mathcal{G}_i, i = 1, \ldots, m$ be a partitioning of $\mathcal{D}'$ into m groups, according to a combination of dimensional axes* *(e.g., time). Let p be a phrase. We can define the interestingness $I_{D'}(p, \mathcal{G}_i)$ of p w.r.t. group $\mathcal{G}_i$, as follows:*

$$I_{D'}(p, \mathcal{G}_i) = \frac{freq(p, \mathcal{G}_i)}{freq(p, \mathcal{D}')} \tag{5.3}$$

Similarly to Definition 4, we can directly apply the forward list merging approaches to find the most interesting phrases in each group. Again, early termination is not possible, since the denominator refers to an ad-hoc document subset that is not known at index time. While the forward lists of all documents in $\mathcal{D}'$ are merged, we compute for each phrase one *group* counter per $\mathcal{G}_i$, plus a local frequency counter for the whole $\mathcal{D}'$. For each phrase that is encountered, we can immediately compute its interestingness by applying Definition 5 for each group. At the same time, we maintain for each group the set of k most interesting phrases seen so far.

Weighted frequencies. All definitions above assume that the documents in $\mathcal{D}'$ (or $\mathcal{D}''$) have the same influence in phrase interestingness. Nonetheless one could argue that if these documents are obtained together with a relevance score (e.g., from a keyword query), then we should weigh their contribution to the interestingness of phrases, accordingly. This can be realized by altering the definition of $freq(p, \mathcal{D}')$ from $freq(p, \mathcal{D}') = count\{d : d \in \mathcal{D}' \wedge p \in d\}$ to $freq(p, \mathcal{D}') = \sum_{d \in \mathcal{D}' \wedge p \in d} rel(d)$, where $rel(d)$ is the relevance of d to the query. Thus, instead of counting the documents that contain p, we add their relevances to the query that determines $\mathcal{D}'$. This change does not affect the functionality of the mining algorithms. On the other hand, it may allow for techniques that examine only a subset of $\mathcal{D}'$ that is most relevant to the query, find the top-k interesting phrases in that subset only and obtain confidence bounds for their interestingness with respect to the complete $\mathcal{D}'$. Exploring this direction is left for future work.

6 Experimental Evaluation

6.1 Dataset and Setup

We used the recently released annotated New York Times corpus [25] as a dataset for our experimental evaluation. This dataset comprises a total of more than 1.8 million New York Times articles published between 1987 and 2007. The raw size of the textual content amounts to about 8 GB. In addition to mere article contents, the dataset includes rich metadata (e.g., an article's publication date, topical classification, and author biography), as well as, annotations (e.g., persons, organizations, and places mentioned in an article).

All methods described in this paper were implemented in Java (using SUN JDK 1.6). Experiments were run on a SUN server-class machine having 16 GB of main memory, four AMD Opteron single-core CPUs, a large network-attached RAID-5 disk array, and running Windows Server 2003.

For processing keyword queries, we employ conjunctive semantics, i.e., all query keywords are mandatory for a document to be reported as a result. We use Okapi BM25 [24, 26] (using the established parameter setting $k_1 = 1.2$ and $b = 0.75$), as a state-of-the-art relevance model to rank result documents. Further, we index the dataset such that metadata and annotations provided with it can easily be leveraged at query time. In detail, for a document containing a key/value pair representing a piece of metadata or an annotation (e.g., that document's publication date is March 11th, 1999), we add the artificial term publication_date$March_11th,_1999. This allows us to include metadata and annotation aspects, when querying the dataset. Notice, though, that these artificial terms are hidden from the relevance model and thus do not affect the relative ranking of results.

Table 6.1 shows the different indexing methods and their representations that we compare against each other.

method	**description**
TermID	baseline method, Section 3.1
IntArray	Section 3.3
PrefixMaxIntArray	
FreqIntArray	
PhraseID	
EarlyTerminationFreqIntArray	Section 3.4
EarlyTerminationPhraseID	
MCX	[27] (reviewed in Section 3.2)

Table 6.1: Query methods evaluated

6.2 Queries

We defined a set of benchmark queries, which we consider to reflect the envisioned application. Our queries, as shown in Figure 6.1 with their respective result sizes, are subdivided into three categories, namely, (i) person-related queries, (ii) news-related queries, and (iii) location-related queries. The first two categories are standard keyword queries, whose choice was inspired by Google's Zeitgeist [14] service. For the third category, we make use of the topical classification provided with the dataset. Here, we focus on geographical locations as they are mentioned in travel-related and news-related articles, respectively.

6.3 Index Sizes

Our first experiment compares the different methods with regard to the size of their corresponding indexes. To this end, for each of the methods, we generated the corresponding indexes for different values of the minimum support threshold (τ). That is, in each case, we only index phrases whose global frequency in the corpus $\mathcal{D}$ is at least τ. In all cases, we index only phrases of lengths between 2 and 5.

Figure 6.2 shows how the sizes of the corresponding indexes vary with respect to this parameter. The TermID representation is the most economical (consuming about 1.8 GB of space), as only the terms are included in each list. The PrefixMaxIntArray and PhraseID representations follow. Even though PrefixMaxIntArray indexes only prefix-maximal phrases, which are roughly equal to the number of terms, because these are represented as integer arrays that occupy a few bytes, the representation takes more space than TermID. PhraseID indexes all phrases, but keeping only a phrase identifier

(i) Person-Related Keyword Queries
jennifer lopez (1,130), osama bin laden (5,951), eminem (796), steve jobs (2,982), kobe bryant (1,385), paris hilton (1,119), harry potter (2,053), martha stewart (3,044), camilla parker bowles (173), prince charles (4,019), barack obama (867), hillary clinton (11,073), rudy giuliani (2,485), martin luther king (6,533), george harrison (2,494)

(ii) News-Related Keyword Queries
world trade center (25,562), american airlines (14,187), world cup (17,974), winter olympics (4,357), korea (22,474), hurricane katrina (3,515), american idol (3,009), tsunami (1,452), bankruptcy (21,188), iphone (82), presidential election (26,492), aol time warner merger (588), sars (1,153), space shuttle columbia (1,322), tour de france (3,328)

(iii) Location-Related Metadata Queries
/travel/guides/destinations/europe (94,789)
/travel/guides/destinations/middle_east (58,457)
/travel/guides/destinations/north_america (325,300)
/travel/guides/destinations/ asia (62,290)
/travel/guides/destinations/africa (22,592)
/travel/guides/destinations/europe/france (12,874)
/travel/guides/destinations/europe/france/rhone_valley/lyon (40)
/travel/guides/destinations/europe/germany (11,165)
/travel/guides/destinations/europe/italy (6,722)
/travel/guides/destinations/central_and_south_america (20,466)
/news/world/africa (14,961)
/news/world/asia_pacific (43,886)
/news/world/americas (26,710)
/news/world/europe (54,230)
/news/world/middle_east (46,020)

Figure 6.1: Keyword & metadata queries (number of query results given in parentheses)

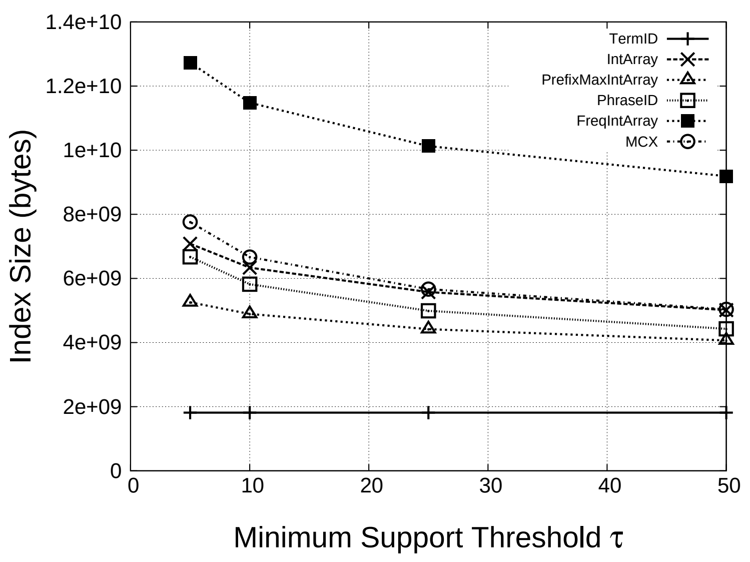

Figure 6.2: Index sizes (bytes) for different values of τ

for each of them, so its overhead over PrefixMaxIntArray is not high. The phrase-inverted index employed by MCX and the IntArray representation yield indexes that consume about the same amount of space. Finally, the FreqIntArray representation produces indexes that, for all values of τ, are considerably larger than the indexes based on other representations. The index for $\tau = 5$, as a concrete figure, consumes about 13 GB of space. This is not surprising, given that, in comparison to IntArray, the method keeps additional information and suffers from lower compressibility as discussed in Section 4.

6.4 Performance Comparison

In this set of experiments we examine the different methods with regard to their query-processing performance. In the first experiment, we investigate how the methods behave as the cardinality of D' is altered. To this end, we vary the cardinality of the input D' between 100 and 10,000. We do this by using only queries with larger results and truncating them to their highest ranked 100 to 10,000 documents according to the per-query relevance ranking. We fix the result size for the interesting phrases as $k = 100$ and the minimum support threshold as $\tau = 10$. We compare our methods to

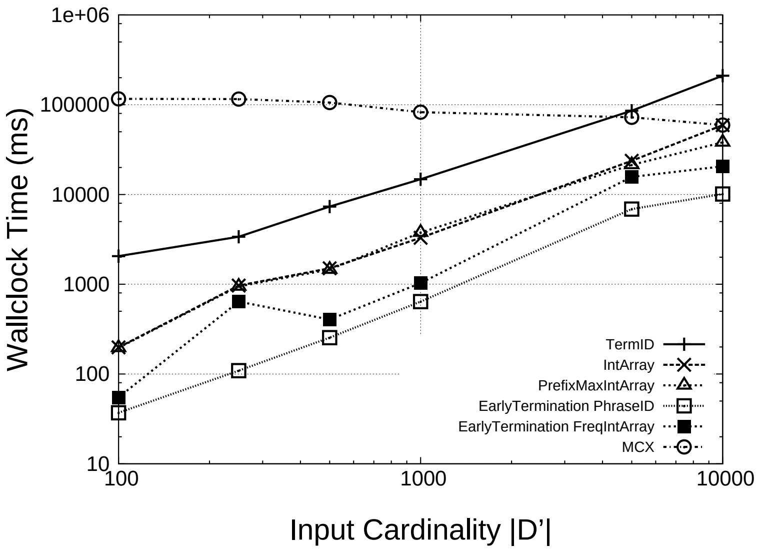

Figure 6.3: Wallclock times (ms) for different values of $|D'|$

an implementation of MCX, in accordance to [27], where we first find the K=1000 phrases with the highest local frequency using the index and then post-process them to find the k most interesting ones. The maximum number of comparisons per approximate list intersection is set to $M = 64$.

Figures 6.3 and 6.4 report the mean wallclock time per query (measured with warm caches) and the mean amount of data read per query for the different approaches. MCX is not affected by the input cardinality $|D'|$. It requires about 100 seconds for a single query. In contrast, all other methods benefit from smaller input cardinalities. For $|D'| = 100$ our best method EarlyTerminationPhraseID outperforms MCX by three orders of magnitude. For $|D'|$ =1,000 the improvement over MCX is still more than a factor of 128. For larger $|D'|$ MCX becomes competitive to some of our methods like IntArray but is still outperformed by EarlyTerminationFreqIntArray by a factor of 10. In summary, these results show that MCX will only be competitive for very large input cardinalities, but for such queries the phrase analysis cannot be done in real-time, so its applicability is limited. On the other hand, for our test corpus and the practical, medium-sized query results of our benchmark, the forward-index-based methods are always the method of choice. A full-fledged system should perhaps implement both methods and let a query optimizer decide which method to use.

Figure 6.4 shows the amount of data read at query time for the same

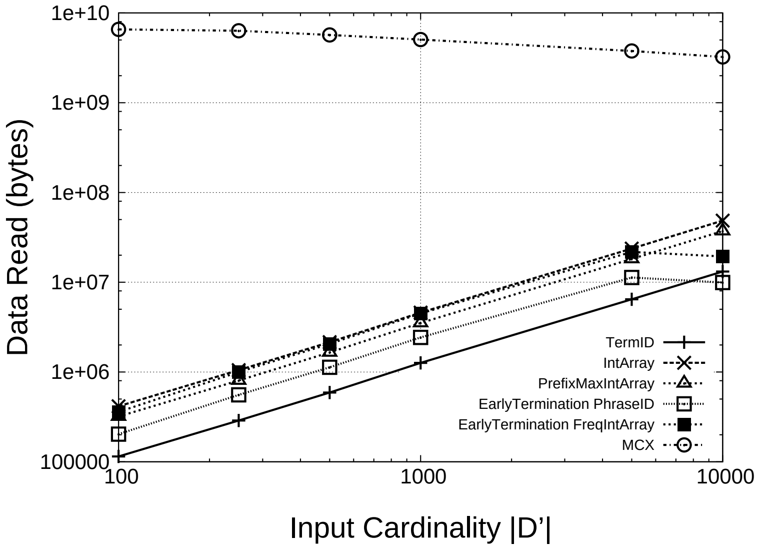

Figure 6.4: Number of bytes read for different values of $|D'|$

experiment. We observe a linear increase for all forward indexing methods starting at 200 KB for $|D'| = 100$ up to 6 MB for $|D'|$ =10,000. In contrast, MCX needs to read at least 3 GB in all cases. For MCX we may also observe a slight decrease when increasing $|D'|$. This is due to the fact that for larger $|D'|$ this method may stop earlier, as the top-K locally frequent phrases can be found faster.

In a second experiment, we fix the result size as $|\mathcal{D}'| = 500$, by taking only the first 500 results of queries that exceed this selectivity, and vary the minimum support threshold τ. In all cases, we determine the $k = 100$ most interesting phrases. Figures 6.5 and 6.6 report the mean wallclock time per query (measured with warm caches) and the mean amount of data read per query for the different approaches, respectively. We skipped MCX from this comparison because its cost exceeded 100 seconds at all times.

The results in Figure 6.5 show that the baseline method TermID requires about 4 seconds to compute a query. In contrast, the different forward indexing methods require at most 1.3 seconds for all parameters. Among our methods, EarlyTerminationPhraseID performs best requiring only 250 ms. This is by a factor of 8 better than the baseline method. In addition, this method also outperforms the other forward indexes like PrefixMaxIntArray which takes about 1,000 to 750 ms. Although our worst-case theoretical analysis (Section 3.6) predicts that PrefixMaxIntArray would perform bet-

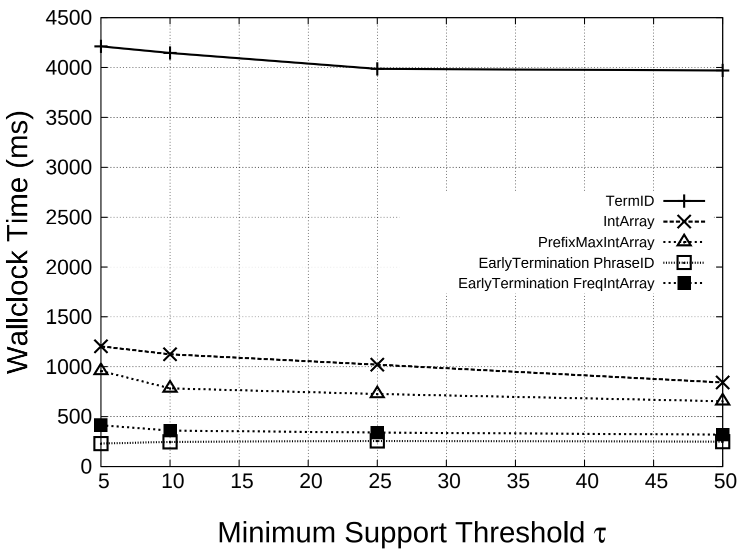

Figure 6.5: Wallclock times (ms) for different values of τ

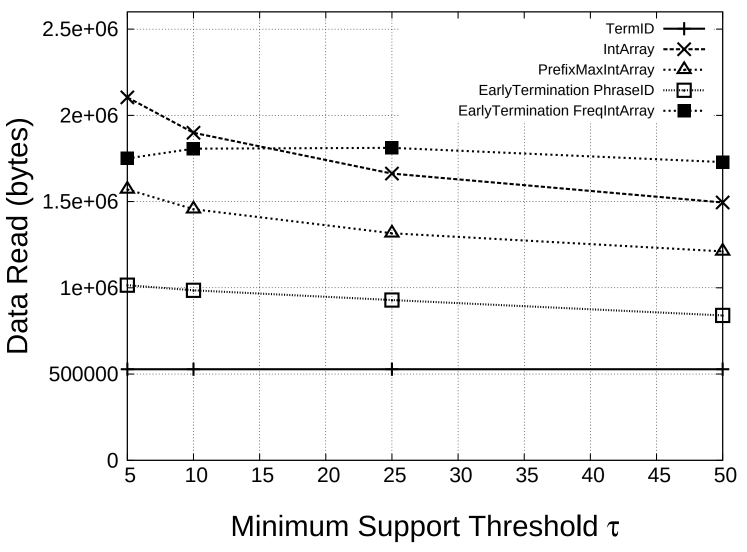

Figure 6.6: Number of bytes read for different values of τ

ter, EarlyTerminationPhraseID is the winner because of (i) the ultra-fast merging of phrase-IDs, (ii) its higher compressibility, and (iii) the early termination heuristic.

Figure 6.6 shows the amount of data read during query processing for the same experiment. The baseline TermID method performs best here, which is not surprising as it only has to read the termID representation of the input documents. However, the cost of computing phrases at query time slows down this method considerably (see Figure 6.5). From the other methods EarlyTerminationPhraseID performs best, due to its high compressibility when compared to integer array phrases and the effect of early termination. For EarlyTerminationFreqIntArray we see a slight increase from $\tau = 5$ to $\tau = 25$. For very low values of τ phrases might qualify as interesting if their global frequency is very low. Thus their interestingness score is very high, which is favorable for the early termination. Since these globally infrequent phrases tend to be long, EarlyTermination achieves higher savings in terms of the amount of data read, when applied on the FreqIntArray representation that keeps phrases explicitly as integer arrays. With increasing τ these globally infrequent phrases are ruled out from both representations.

In general, the influence of parameter τ is relatively mild. This is because after the initial pruning of extremely rare phrases (the long tail of phrases that occur only once or twice), removing more infrequent phrases does not affect the lengths of the forward-index lists too much. Because of these phrases being infrequent, their probability of occurring in the query-result documents is much lower than the occurrence probability of a frequent phrase.

6.5 Effectiveness of Early Termination

Our last experiment studies the effectiveness of the early-termination method introduced in Section 3. For this, we fix the minimum support threshold as $\tau = 10$ and the cardinality of the input as $|D'| = 500$ and vary the number k of interesting phrases to be determined. Figures 6.8 and 6.7 report the mean amount of data read and the mean wallclock times for the following methods: PhraseID, FreqIntArray, EarlyTermination-PhraseID, and EarlyTermination-FreqIntArray.

The results show that for $k = 10$ the EarlyTerminationPhraseID reads about 0.7 MB in 184 ms. For $k = 500$ it reads 1.1 MB in 263 ms. This is an increase of about 57% and 42%, respectively, for a 50-fold increase of k, quite a remarkably good result. Comparing the early-termination methods with their merge-based counterparts, one can observe significant reductions both in terms of data read and wallclock time even for large values of k. As

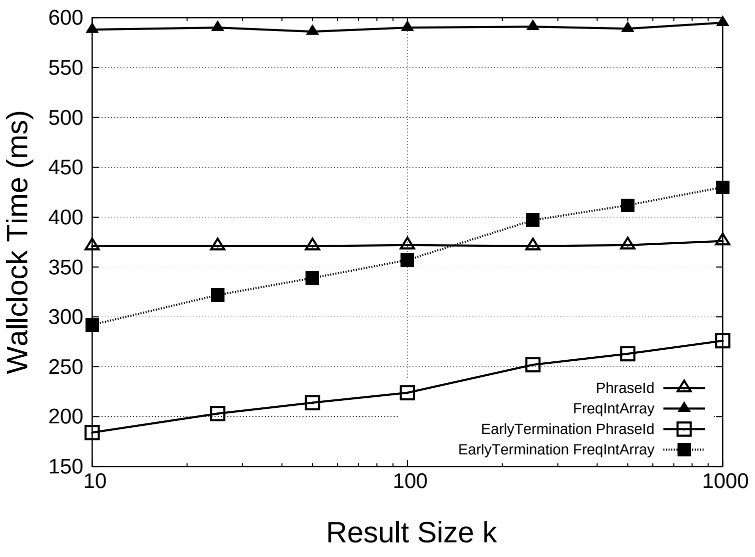

Figure 6.7: Wallclock times (ms) for different values of k

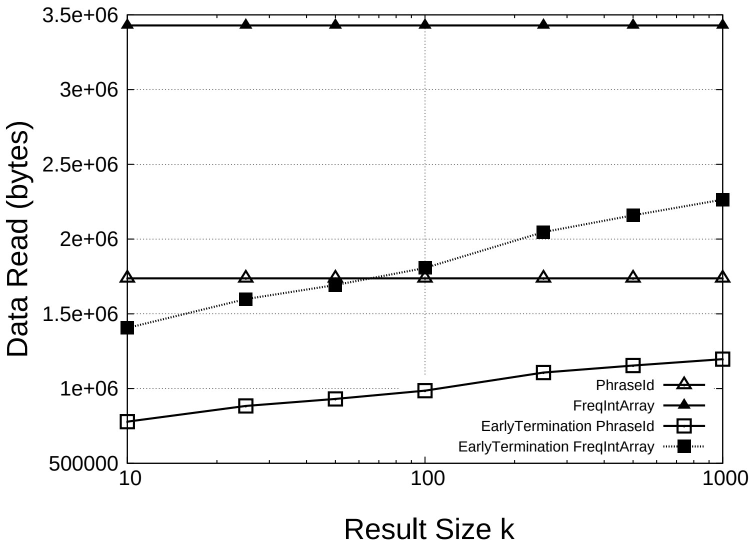

Figure 6.8: Number of bytes read for different values of k

a concrete figure, for $k = 1,000$, the early-termination methods accomplish reductions of about 30% in both figures.

7 Related Work

Text analytics is increasingly gaining attention due to its value in gathering business intelligence. Previous efforts in this regard have primarily focused on term-level analytics, identifying terms that are specific to a group of documents [9, 19, 28]. Some [10, 11, 17, 18, 23] have taken a multi-dimensional view of the text collection and proposed OLAP-style models for performing drill-down/roll-up on text databases, but remain at the level of terms.

While term-level analytics is important, it is not sufficient for identifying important entity names or marketing slogans which have tremendous value for business intelligence. Early approaches [1, 20] fall short in scaling to large-scale text collections or consider only the collection as a whole but no ad-hoc document sets. Only recently, the MCX system [27] was proposed as an important step in the direction of scalable phrase-level analytics. As we explained earlier, their framework identifies only the most frequent phrases in a given set of documents, and then re-ranks these phrases based on their interestingness. In addition, due to the approximate counting used in this method, it becomes difficult to keep track of documents that actually contain an identified interesting phrase – an important aspect if analysts should also be guided to the containing documents. Further, as our results clearly demonstrate their choice of using inverted indexes does not scale very well for large-scale data collections.

Also other communities have explored the idea of extracting phrases dynamically. In [29] the use of phrases as a means to guide the user through search results is studied. Since their focus is on at most the top-30 most relevant documents, scalability issues, as we address them, do not play a role in their application. In [31] point-wise mutual information (PMI), as a measure of a phrase's statistical surprisingness, is used to identify key phrases in a document. Thus identified phrases from a document are used as a signature of the document. The signature is then used for querying the collection for similar or related documents. Our framework can be easily adapted for incorporating the PMI-based interestingness measure, as discussed in Section 5.

Faceted search [15], as pioneered in the Flamenco project [12], is akin to our work in dynamically aggregating information for ad-hoc sets of documents. Although recent extensions [3, 7] move beyond mere count-based aggregation of facets, the key difference to our approach remains that facets are pieces of meta data about the documents. Our approach, in contrast, operates directly on the document contents.

In addition to scalable phrase-level analytics, good visualizations are important for comprehending the results in an intuitive manner. MemeTracker [21], TagLines [9], and BlogScope [2] have explored this for visualizing bursty phrases, terms, or social annotations over time. Similar interfaces could easily be built on top of our framework, and would then enable exploration of interesting phrases for ad-hoc document sets in a scalable and interactive manner, which is not possible in any of the aforementioned systems.

8 Conclusion

This paper has presented scalable techniques for interesting phrase mining from large text corpora. In contrast to the state-of-the-art method MCX [27], which solves the problem only approximately and is inefficient for large corpora, our forward indexing method scales very well with the corpus size. This is the decisive property to enable users to perform interesting phrase analyses on large real world datasets. We have provided several variants of forward indexing and discussed the different trade-offs w.r.t. index size and query response times. Our performance study used a large-scale real-world corpus consisting of 1.8 million articles from New York Times. We compared the different variants of our method against MCX. Our results confirm that our forward indexing scheme outperforms MCX by orders of magnitude when the cardinality of the documents to be analyzed is not impractically large.

In terms of future work, we are planning to explore other more high-level query types enabled by our method including: timeline interesting phrase analysis, phrase groupings, as well as, an interesting phrase CUBE operator.

Bibliography

[1] H. Ahonen. Knowledge Discovery in Documents by Extracting Frequent Word Sequences. *Library Trends*, 48(1), 1999.

[2] N. Bansal and N. Koudas. BlogScope: A System for Online Analysis of High Volume Text Streams. In *VLDB*, 2007.

[3] O. Ben-Yitzhak, N. Golbandi, N. Har'El, R. Lempel, A. Neumann, S. Ofek-Koifman, D. Sheinwald, E. Shekita, B. Sznajder, and S. Yogev. Beyond Basic Faceted Search. In *WSDM '08*, 2008.

[4] C. Binnig, S. Hildenbrand, and F. Faerber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*, 2009. to appear.

[5] L. Bouganim, B. Þ. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30:107–117, 1998.

[7] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G. Lohman. Dynamic Faceted Search for Discovery-driven Analysis. In *CIKM*, 2008.

[8] J. Dean. Keynote: Challenges in Building Large-Scale Information Retrieval Systems. In *WSDM*, 2009.

[9] M. Dubinko, R. Kumar, J. Magnani, J. Novak, P. Raghavan, and A. Tomkins. Visualizing Tags over Time. *ACM Trans. Web*, 1(2):7, 2007.

[10] R. Fagin, R. Guha, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Multi-Structural Databases. In *PODS*, 2005.

[11] R. Fagin, P. Kolaitis, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Efficient Implementation of Large-scale Multi-structural Databases. In *VLDB*, 2005.

[12] Flamenco Project
`http://flamenco.berkeley.edu`.

[13] Fusion-io
`http://www.fusionio.com`.

[14] Google Zeitgeist
`http://www.google.com/press/zeitgeist.html`.

[15] M. A. Hearst. Clustering versus Faceted Categories for Information Exploration. *Commun. ACM*, 49(4):59–61, 2006.

[16] J. M. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *SIGMOD*, 1997.

[17] A. Inokuchi and K. Takeda. A Method for Online Analytical Processing of Text Data. In *CIKM*, 2007.

[18] S. Keith, O. Kaser, and D. Lemire. Analyzing Large Collections of Electronic Text Using OLAP. *CoRR*, abs/cs/0605127, 2006.

[19] J. Kleinberg. Bursty and Hierarchical Structure in Streams. In *KDD*, 2002.

[20] B. Lent, R. Agrawal, and R. Srikant. Discovering Trends in Text Databases. In *KDD*, 1997.

[21] J. Leskovec, L. Backstrom, and J. Kleinberg. Memetracker. `http://memetracker.org`.

[22] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Agarwal. Dynamic Maintenance of Web Indexes Using Landmarks. In *WWW*, 2003.

[23] C. X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao. Text Cube: Computing IR Measures for Multidimensional Text Database Analysis. In *ICDM*, 2008.

[24] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008.

[25] New York Times Annotated Corpus
`http://corpus.nytimes.com`.

[26] S. E. Robertson and S. Walker. Okapi/Keenbow at TREC-8. In *Text Retrieval Conference*, 1999.

[27] A. Simitsis, A. Baid, Y. Sismanis, and B. Reinwald. Multidimensional Content eXploration. *PVLDB*, 1(1):660–671, 2008.

[28] Y. Sismanis, B. Reinwald, and H. Pirahesh. Document-Centric OLAP in the Schema-Chaos World. In *BIRTE*, 2006.

[29] R. W. White, J. M. Jose, and I. Ruthven. Using Top-Ranking Sentences to Facilitate Effective Information Access. *JASIST*, 56(10):1113–1125, 2005.

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[31] Y. Yang, N. Bansal, W. Dakka, P. Ipeirotis, N. Koudas, and D. Papadias. Query by Document. In *WSDM*, 2009.

[32] M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42(1/2):31–60, 2001.

[33] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Comput. Surv.*, 38(2):6, 2006.