Searching for Complex Patterns over Large Stored Information Repositories*

Nikhil Deshpande[1], Sharma Chakravarthy[1], and Raman Adaikkalavan[2]

[1] CSE Department, The University of Texas at Arlington
[2] CIS Department, Indiana University South Bend
{sharma,raman}@cs.iusb.edu

Abstract. Although Information Retrieval (IR) systems, including search engines, have been effective in locating documents that contain specified patterns from large repositories, they support only keyword searches and queries/patterns that use Boolean operators. Expressive search for complex text patterns is important in many domains such as patent search, search on incoming news, and web repositories. In this paper, we first present the operators and their semantics for specifying an expressive search. We then investigate the detection of complex patterns – currently not supported by search engines – using a pre-computed index, and the type of information needed as part of the index to efficiently detect such complex patterns. We use an expressive pattern specification language and a pattern detection graph mechanism that allows sharing of common sub-patterns. Algorithms have been developed for all the pattern operators using the index to detect complex patterns efficiently. Experiments have been performed to illustrate the scalability of the proposed approach, and its efficiency as compared to a streaming approach.

Keywords: Information retrieval, Complex patterns, Document search.

1 Introduction

Although current IR systems [1,2,3,4] are convenient for doing keyword searches, in domains such as federal intelligence, fugitive tracking and searching full-text patent information, there is a need to detect (or search[1]) more complex patterns in data sources. Users in these domains may have more precise requirements in terms of what they are searching for. They may be searching for patterns that involve term frequency (e.g., at least 5 occurrences of the phrase "protein clustering"), proximity with sub-patterns (e.g., "peptide" near "saccharide", in any order, within 5 words of each other), sequence of sub-patterns (e.g., "DNA" followed by "modification") and so on. Further, the patterns that need to be detected may be arbitrarily complex; that is, they may need to be specified in terms of other patterns (e.g., ("militant" followed by "bomb") near "Iraq", separated by 5 positions or less). The expressiveness of search/query specification provided by current IR systems, although satisfactory for general searches, is not adequate for the above application domains.

* This work was supported, in part, by the following NSF grants: IIS-0326505, EIA 0216500, and IIS 0534611.

[1] We use the terms "search" and "detect" interchangeably in this paper.

A.A.A. Fernandes et al. (Eds.): BNCOD 2011, LNCS 7051, pp. 68–82, 2011.

Detecting complex patterns over text streams has been studied and shown to be possible in [5], in which a suite of complex operators and their algorithms were developed that detect complex patterns over stream data. Detecting patterns over a dynamic text source (e.g., news feeds, IP packets) essentially entails streaming the data to detect the required patterns. In other words, to detect a pattern, the entire data source must be read (or parsed) every time. This is inefficient, but unavoidable, because of the fast-changing nature of the data source. Also, if freshness of the search results are important to the user, it becomes necessary to read the data source every time while processing a query. However, if the data source is relatively static (e.g., Web repositories), it is unnecessary to read the entire source each time a pattern is to be detected. The inefficiency will be exacerbated as the data source grows larger. A better approach would be to build and leverage an index on the data, as is done by search engines, and the information in the index could be used for answering queries. Since the index would be computed off-line, this approach may result in an occasional out-of-date search result. However, considering that the data source is not frequently updated, we can assume this is acceptable to the user. For such relatively static data sources, the gains in terms of efficiency of retrieval that leveraging an index will bring outweigh the slight disadvantage of an occasional out-of-date result.

The techniques developed for searching complex patterns over streams (as in XML streams, news feed, stock prices) in [5] makes use of the sequential inflow of patterns by reading the entire data source to detect a pattern. However, if the same patterns need to be detected in stored data (as in web repositories) then streaming is very inefficient. It is more efficient to index the repository (or use an already existing index) to detect the patterns. Indexing will lose the sequence of occurrence of patterns in the data. This order of occurrence of patterns is the key to detecting patterns based on proximity, containment, sequence, etc. The main contributions of this paper are to: (i) Identify information that is needed as part of the index to correctly and efficiently detect complex patterns as compared to the streaming approach, (ii) Explore the extent of the complexity of the patterns that can be detected using indexed information, and (iii) Develop efficient algorithms for index-based pattern detection.

The rest of this paper is organized as follows: Section 2 discusses the semantics of the InfoSearch operators and Section 3 explains the algorithms used by the operators. Section 4 explains the design of the InfoSearch System. Section 4.1 explains the implementation aspects of the system. Section 5 shows detailed experimental results. Section 6 reviews the related work, and Section 7 concludes the paper.

2 Pattern Specification and Detection

The InfoSearch framework discussed in this paper consists of an expressive query language (introduced in [5]) through which the user can specify patterns and a pattern detection engine capable of using the index to retrieve documents. InfoSearch detailed in this paper has been briefly summarized in [6]. InfoSearch adopts the Pattern Specification Language (PSL) and its associated parser and pattern validator used in InfoFilter [5]. The focus of this paper is on the detection of complex patterns over large document repositories.

2.1 Pattern Specification

An occurrence of a pattern ***P*** is the presence of the pattern ***P*** in a given document. There is an offset (multiple ones if the pattern occurs multiple times in the document) at which the pattern occurs in the document. O_s is the start offset, and O_e is the end offset of the pattern, where offset is the position of *words* relative to the beginning of the document.

Simple patterns are the basic building blocks and can be either *System-defined* (i.e., pre-defined in the system), or *User-defined*. Begin_para, Begin_document are examples of system-defined patterns. Examples of simple user-defined patterns are: keywords or phrases.

Complex patterns are composed of simple patterns, complex patterns, and pattern operators (listed below). Any arbitrary complex pattern can be composed using the pattern operators. Current operators supported are summarized below:

OR: Disjunction of two simple or complex patterns P_1 and P_2, denoted by (P_1 *OR* P_2), occurs when either P_1 or P_2 occurs. For example, *"information" OR "filtering"* will be detected when either one of the keywords occurs.

NEAR: Proximity of two simple or complex patterns P_1 and P_2, denoted by *(*P_1 *NEAR [/D]* P_2*)*, occurs when both P_1 and P_2 occur, irrespective of their `order of occurrence`. "*D*" is the maximum distance allowed between the patterns P_1 and P_2. Default value of "*D*" is the scope of the operator (which can be the entire document).

FOLLOWED BY: Sequence of two simple or complex patterns P_1 and P_2, denoted by *(*P_1 *FOLLOWED BY [/D]* P_2*)*, occurs when the occurrence of P_1 is followed by the occurrence of P_2 in a non-overlapping manner. The end offset of P_1 is less than the start offset of P_2; "*D*" is the maximum distance allowed between the two patterns P_1 and P_2. If the value of "*D*" is 1 (minimum value), this indicates that the patterns P_1 and P_2 form a phrase.

WITHIN: Occurrence of a simple or complex pattern P in the range formed by the start offset of the pattern P_S and the end offset of P_E, denoted by *(*P *WITHIN (*P_S, P_E*))*. The pattern is detected each time pattern P occurs in the range defined by patterns P_S and P_E. For example, *"information filtering" WITHIN (BeginPara, EndPara)* will be detected whenever the phrase *"information filtering"* occurs within a paragraph. When an expression is specified without a system-defined pattern, the default structure (e.g., a document) is used as the default. User defined P_S and P_E can be used.

NOT: Non-occurrence of a simple or complex pattern P in the range formed by the start offset of P_S and the end offset of P_E. The general specification is *(NOT [/F](P)(*P_S, P_E*))*, where P, P_S, and P_E can be arbitrary patterns. "*F*" indicates the minimum number of occurrences and its default value is 1. For example, *NOT ("filtering")("information", "retrieval")* will be detected whenever *"information"* is followed by *"retrieval"* without the word *"filtering"* occurring at least once in between them.

FREQUENCY: Multiple occurrences of a simple or complex pattern that exceed or equal to *F*, denoted by *(FREQUENCY /[F] (P))*. A pattern *P* is detected each time *P* occurs at least F times, where "*F*" is the minimum number of occurrences specified by the user. The default value of *F* is 1. All the occurrences that are used for detection should be disjoint (i.e., the end offset of each pattern occurrence should precede the

start offset of the subsequent pattern occurrence). The same set of occurrences will not be used for detecting multiple instances of the same pattern.

SYN: This is an option and is specified along with a single-word pattern (currently), denoted by *(P [SYN])*, to indicate multiple single-word patterns that have the same meaning, in a succinct manner. Specifying a single-word pattern with SYN option is equivalent to specifying *N* simple patterns that carry the same meaning (synonyms) as the original pattern. For example, if you specify the word *"bomb"[SYN]* is equivalent to specifying *"bomb" OR "explosive device" OR "weaponry" OR "arms" OR "implements of war" OR "weapons system" OR "munition"* . If any of these words or phrases appears in the text, the pattern *"bomb"[SYN]* is detected. This option adds simplicity and flexibility to the specification of single-word patterns. The same is true for complex patterns with embedded synonym specification, e.g. *"Bomb"[SYN] NEAR "Ground Zero"*.

Sample Query: Using the above operators, users can specify complex and meaningful patterns. A complex pattern *("bomb" occurring prior to "ground zero" occurring twice, with a single occurrence of "automotive" or its synonyms)*, can be specified as:

Pattern P_1 = "bomb" FOLLOWED BY "groundzero"
Pattern P_2 = FREQUENCY/2 (P_1)
Pattern P_3 = P_2 NEAR "automotive"[SYN]

2.2 Pattern Detection

Pattern detection semantics are needed for detecting meaningful patterns, since in an unrestricted semantics (where none of the pattern occurrences are discarded after participating in pattern detection) not all the detected patterns are meaningful for an application. Detection semantics essentially delimit the patterns detected and accommodate a wide range of application requirements.

We want to emphasize that we have chosen to define *proximal-unique* semantics in this paper based on the intuition of proximity and disjoint pattern detection. It is certainly possible to define other meaningful constraints leading to other useful semantics. However, the framework remains the same and the algorithms change depending upon the semantics used. It is indeed possible to include semantics of detection as an additional parameter when several of them are defined and supported.

Consider a document containing occurrences of words as shown in Figure 1. Suppose we want to find occurrences of *"cell" FOLLOWED BY "nucleic"* within this document. As shown in the figure, there are two occurrences of *"cell"*, one occurring at position 10, say $cell^1$ and the other at position 15, say $cell^2$. The occurrences of nucleic are at position 28 and 41, say $nucleic^1$ and $nucleic^2$ respectively. We could combine either

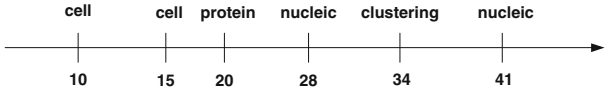

Fig. 1. Pattern Occurrences (Example)

$cell^1$ with $nucleic^1$, or $cell^2$ with $nucleic^1$, or $cell^1$ and $cell^2$ both with $nucleic^1$ as occurrences of the combined pattern *"cell" FOLLOWED BY "nucleic"*.

However, it makes more intuitive sense to combine only the closest occurrences, because closely occurring patterns are more likely to be of interest for a search as the correlation here is measured in terms of *proximity*. Hence, we discard the occurrence of $cell^1$ and combine $cell^2$ with $nucleic^1$. In the above example, $cell^2$ is called the *initiator* because it initiates the pattern detection, and $nucleic^1$ is called the *terminator*, because its occurrence results in the pattern being detected.

Second, sub-patterns once used are not used for detecting another instance of the same pattern, i.e., patterns need to be *unique*. For example, it does not make intuitive sense to combine $cell^2$ with $nucleic^2$, because $cell^2$ has already been used in a combination. Combining it again with $nucleic^2$ will result in the detection of another instance of the same pattern using a previously used sub-pattern. Of course, the above takes the distance into consideration where specified.

The Proximal-Unique semantics has been defined to take this intuitive sense of proximity and uniqueness into consideration when detecting a pattern by applying restrictions on the usage of sub-patterns. It is less complicated to detect patterns with unrestricted semantics although a large number of them are likely to be generated.

Non-overlap or disjoint aspect is also assumed. For example, suppose we want to find the occurrence of *("cell" FOLLOWED BY 'nucleic") NEAR ("protein" FOLLOWED BY "clustering")*. According to the semantics discussed above, *"cell" FOLLOWED BY 'nucleic"* occurs in the interval (15, 28) and *"protein" FOLLOWED BY "clustering"* occurs in the interval (20, 34). The sub-patterns satisfy the condition of being proximal, and of being the most recent un-combined occurrence of their type. However, they overlap and are not disjoint. Under the disjoint constraint, the combined pattern is not detected. It is also possible to relax the disjoint constraint in which case the NEAR operator will detect the above pattern.

2.3 Inverted Index

The inverted index (also called an inverted list) is the most common mechanism used in Search Engines [2,7] to maintain a mapping from a keyword to the documents that contain the keyword. Given a collection of documents, IDs are assigned to each document. A document ID uniquely identifies a document. The basic information stored in the inverted index is just a keyword - document ID mapping. For example, a sample set of documents is shown in Table 1 and the corresponding inverted index is shown in Table 2. This information is sufficient to answer simple keyword queries and queries involving Boolean operators. In other words, given a keyword, we can return document IDs of documents that contain at least one occurrence of that keyword. For example, in the given example, if the user is searching for *"information" AND "retrieval"*, the intersection of the document IDs corresponding to the keywords *"information"* and *"retrieval"* gives us the desired result (documents 1 and 3 in this example).

However, to answer queries involving proximity, sequences, frequency and containment, this information is *not* sufficient. First, the above scheme does not store information about *every* occurrence of a keyword. It only provides information about the presence or absence of a term within a document. Second, to answer such complex

Table 1. A sample set of documents

Document ID	Document contents
1	information retrieval
2	Specifying complex queries
3	information on information retrieval

Table 2. Inverted index on documents in Table 1

Keyword	Documents
information	1,3
retrieval	1,3
Specifying	2
complex	2
queries	2
on	3

Table 3. Inverted index with position information

Keyword	Documents with position
information	1<1>, 3<1,3>
retrieval	1<2>, 3<4>
Specifying	2<1>
complex	2<2>
queries	2<3>

queries, we need to compute the distance between two given patterns, and also the relative order of occurrence of these patterns. For example, a query such as *"information" NEAR/2 "retrieval"* cannot be answered using information from such an index, because the distance between occurrences of *"information"* and *"retrieval"* within a given document needs to be computed. This distance cannot be computed given just the document which the patterns belong to. The position of *every* occurrence of the keyword within a document must also be provided by the index [8]. Table 3 shows an inverted index generated on the documents in Table 1 with the position information stored.

Hence, InfoSearch needs at least the document ID and the position of a given keyword from the index with which it is integrated, in order to detect complex patterns. One of the main goals of this work was to assess whether this information is sufficient to enable complex pattern detection over an index, if the same patterns can be detected by reading the data source in sequence.

2.4 Pattern Detection Graphs

Patterns are detected using a data structure called Pattern Detection Graph (PDG). A query submitted to InfoSearch is converted into a PDG. Leaf nodes of the PDG correspond to simple patterns such as keywords, phrases or system defined patterns. Internal nodes correspond to complex patterns and encapsulate the logic of the corresponding operator. For example, the PDG corresponding to the pattern *"Protein" FOLLOWED BY "clustering"* is shown in Figure 2. The input to a leaf node is a set corresponding to the index lookup for the term or phrase represented by the leaf node. This set consists of *<docID, start offset, end offset>* **tuples**. As shown in the figure, "protein" occurs once at offset 10 in document 1 and "clustering" occurs once at offset 12 in document 1. As another example, the set of tuples for the keyword *"information"* from the index shown in Table 3 is: 1<1,1>, 3<1,1>and 3<3,3>.

Every node in a PDG has one or more parent nodes (also called as subscriber nodes), except the root node. Leaf nodes propagate their input sets to their parent nodes. A parent node, which corresponds to one of the operators such as OR or NEAR, thus gets one or more sets of tuples as its input. The operator merges its input sets according to the Proximal-Unique semantics for that operator to create an output set. After the merged set is created, it is propagated to the parent node of the operator. This process of propagating merged sets continues all the way up to the root. The merged output of the root operator corresponds to the result set for the query. For example, in figure 2, the input sets from leaf nodes are propagated to the "followed by" node, where the complex pattern is detected over the interval $< 10, 12 >$.

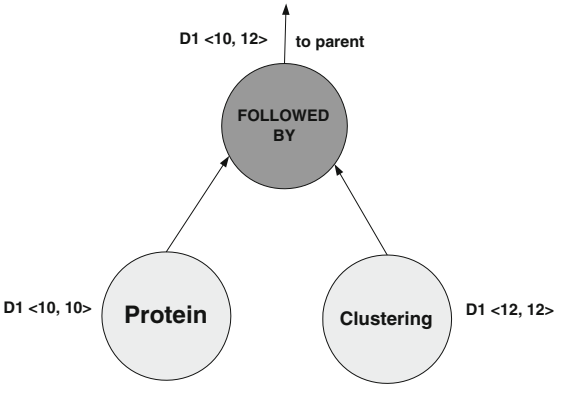

Fig. 2. PDG corresponding to *"Protein" FOLLOWED BY "Clustering"*

3 Pattern Operator Processing

InfoSearch computations are different from that of the algorithms used in a streaming system. In a streaming system [5], the operators work by reading the data source sequentially, and passing simple pattern occurrences to the respective PDG nodes as and when they occur while the data is being read. In other words, the input to a leaf node will be a tuple and not a set of tuples. Because the data is read sequentially, simple patterns are detected in their `order of occurrence` in the data source. As a result, at any operator, the initiator is always available when the terminator arrives. The occurrences can then be combined and propagated, or discarded, as per the semantics of the operator.

However, in InfoSearch the entire result set corresponding to a pattern is propagated at once because of the stored text. This means that the relative order of occurrence of the operands is lost, because each operand is a *set* containing all occurrences of the pattern corresponding to that operand in the document collection. Hence, to generate correct results, the InfoSearch operators need to restore the order of occurrence of patterns as in the original document. This is crucial in order to determine which operand is the initiator and which one is the terminator. Only when the relative order of occurrence

and position of sub-patterns is known, can a decision be made whether they can be combined or not.

The inputs to the operators are sets of tuples containing the document ID, start offset, and end offset of the corresponding pattern. Each tuple represents a single occurrence of the corresponding pattern in the document collection. It is assumed that these sets of tuples are sorted in ascending order of document ID and by start offset within each document ID. The operators have to process the input sets tuple by tuple. First, they have to ensure that the tuples to be merged have the same document ID. Second, they have to determine which tuple is the initiator and which one is the terminator. Tuples satisfying the criteria of the operator are combined and added to an output set. After the operator is done processing the input sets, the output set is propagated to its parent.

Due to space limitations, we discuss only the NEAR operator. Please refer to [9] for other operator and algorithm discussions.

3.1 The NEAR Operator

When the NEAR operator is processing two tuples from the input sets, it has to make a decision whether the tuples are eligible for combination, and if not, decide which one to keep and which one to discard. As mentioned earlier, the input tuples may either be point tuples or interval tuples. To keep the forthcoming discussion generalized, we assume that the input tuples have both start and end offsets. We now discuss the different cases possible when we consider two input tuples, and the corresponding actions taken in each case.

Merging strategy in NEAR: The inputs to the NEAR operator are two sets of tuples corresponding to the left child and the right child, and an optional distance. Let the left set be denoted by *L* and the right set by *R*. Let the distance be denoted by *d*. We arbitrarily assign the first tuple from *L* as $initiator$, and the first tuple from *R* as $terminator$. Let i_s and i_e denote the start offset and end offset of the $initiator$, and t_s and t_e denote the start offset and end offset of the $terminator$. Let $i+1$ be the next tuple from the set which $initiator$ belongs to, and $t+1$ be the next tuple from the set which $terminator$ belongs to.

If $initiator$ and $terminator$ do not belong to the same docID, we advance the pointer which is pointing to a smaller docID. Since the sets are sorted by docID, this is similar to a sort-merge operation. When $initiator$ and $terminator$ point to tuples belonging to the same docID, three cases are possible.

Case 1 ($i_e < t_e$): This means that the assumed $initiator$ ends before the assumed $terminator$. The different possibilities are shown in Figure 3. We perform the following sequence of actions:

if $initiator$ and $terminator$ overlap **then**
lookahead[2] to determine new $initiator$ and $terminator$
go to the beginning of this operation and re-process the new $initiator$ and $terminator$

[2] The Lookahead algorithm is explained below.

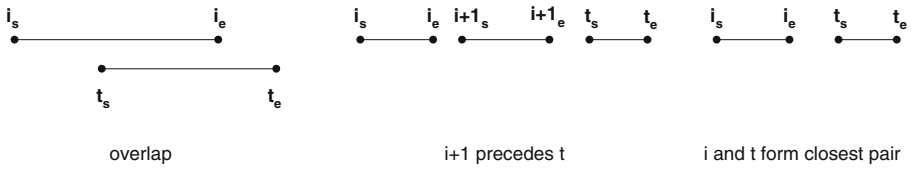

Fig. 3. Some possibilities when $i_e < t_e$

if $i + 1_e \leq t_e$ **then**
make $i + 1$ the new $initiator$, and re-process the new $initiator$ and $terminator$
else
this means $initiator$ completely precedes $terminator$, without any overlap, and there is no other tuple from the $initiator$ set occurring before $terminator$. Now, check if the distance criterion is satisfied.
if (t_s - i_e) $\leq d$ **then**
combine $initiator$ and $terminator$
advance $initiator$ and $terminator$
else
does not satisfy distance
lookahead to determine new $initiator$ and $terminator$, re-process them

Case 2 ($i_e == t_e$): This means $initiator$ and $terminator$ overlap (they have the same end offset). Perform a lookahead, and re-process.
Case 3 ($i_e > t_e$): This means our assumption of $initiator$ and $terminator$ is wrong. The terminator precedes the initiator, either in an overlapping fashion, or a non-overlapping fashion as shown in Figure 4. In this case, we swap the $initiator$ and $terminator$ pointers, and re-process.

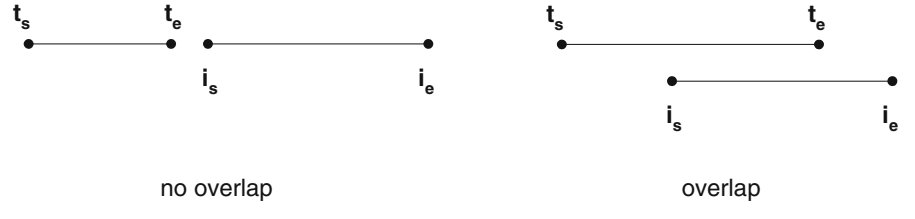

Fig. 4. Some possibilities when $t_e < i_e$

Lookahead algorithm: A lookahead is done when the current $initiator$ and $terminator$ cannot be combined due to an overlap, or because the distance criterion is not satisfied. At this point, we cannot determine which one from $initiator$ and $terminator$ to keep, and which one to discard. We look ahead one tuple from both sets, and assign the one that occurs first as the new terminator. The older tuple from the

opposite set becomes the new $initiator$. Three possibilities exist when we consider the lookahead tuples:

Case I ($i + 1_e < t + 1_e$): This means the next tuple in the *initiator* set occurs before the next tuple in the *terminator* set. (We assume they belong to the same docID).

Make old *terminator* the new *initiator*
Make $i + 1$ the new *terminator*

Case II ($i + 1_e$ == $t + 1_e$): This means the next tuples have the same end offset. In this case, we look at the older pair, and keep the one that occurs later as the new $initiator$.

if $i_e < t_e$ **then**
Make old $terminator$ the new *initiator*
Make $i + 1$ the new *terminator*
else
This means $initiator$ and $terminator$ have the same end offset
Keep the $initiator$
Make $t + 1$ the new $terminator$

Case III ($i + 1_e > t + 1_e$):
Keep the $initiator$
Make $t + 1$ the new $terminator$

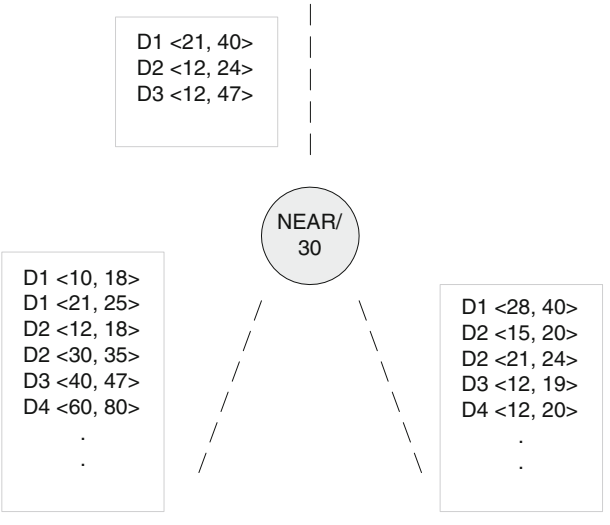

Fig. 5. Example of the NEAR operator algorithm

Figure 5 shows an example of the working of the NEAR operator. To begin, *initiator* points to D1 <10, 18> in the left set, and *terminator* points to D1 <28, 40> in the right set. Since the next tuple in the *initiator* set lies completely before *terminator*, it is assigned as the new *initiator* (*initiator* is advanced). Now, *initiator* and *terminator* point to a proximal pair of tuples, and hence they are merged and added to the output set as the tuple D1 <21, 40>. When *initiator* and *terminator* point to D2 <12, 18> and D2 <15,

20> respectively, an overlap is detected, and hence a lookahead is done in both sets. The lookahead determines that the next tuple from the right set (D2 <21, 24>) ends before the next tuple from the left set (D2 <30, 35>). Hence, D2 <21, 24> is made the new *terminator* and D2 <12, 18> is retained as the initiator. They are combined to form the output tuple D2 <12, 24>. Now, *initiator* points to a D2 tuple while *terminator* points to a D3 tuple. Hence, *initiator* is advanced. Now, *initiator* (D3 <40, 47>) lies completely after *terminator* (D3 <12, 19>). Hence, *initiator* and *terminator* are swapped. This makes *initiator* point to D3 <12, 19> and *terminator* point to D3 <40, 47>, which form a proximal pair and are merged to give D3 <12, 47> in the output set. Finally, *initiator* points to D4 <12, 20>, and *terminator* points to D4 <60, 80>. In this case, the distance between them is 40, which is greater than the maximum allowed distance, i.e., 30. Hence, they are not combined, and a lookahead needs to be done to determine which one of them should be discarded, and which one kept.

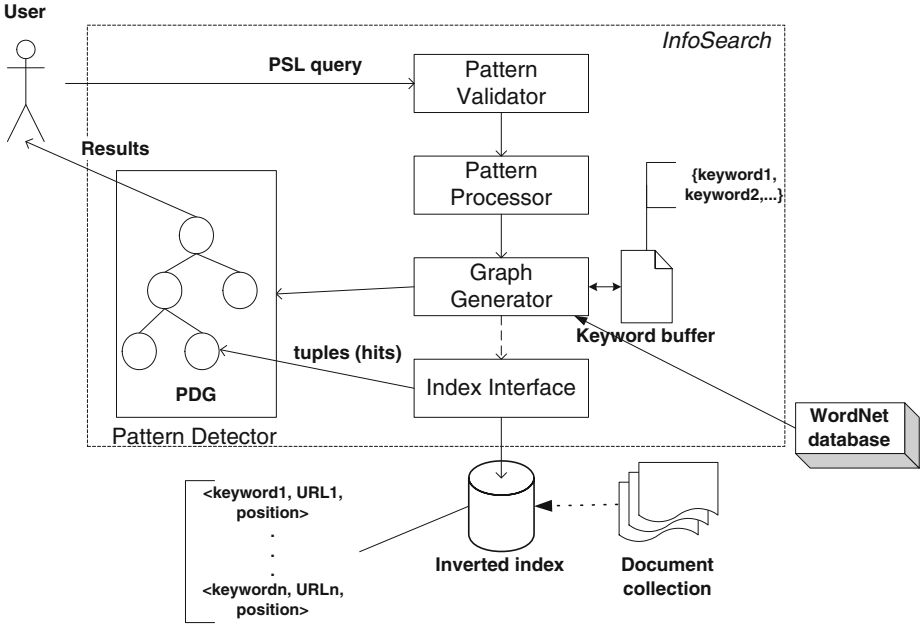

Fig. 6. InfoSearch architecture

4 Design and Implementation of InfoSearch

The InfoSearch architecture is shown in Figure 6. The user query specified in Pattern Specification Language is converted into a Pattern detection graph (or PDG). Leaf nodes of the PDG represent simple patterns such as keywords, phrases or system defined patterns. Higher level nodes represent composite operators on these leaf nodes, or on other composite nodes. To detect and optimize common computations, the *graph generator* shares PDG nodes (and sub-graphs) wherever possible. This is achieved by generating

a single, common PDG or sub-PDG for a common expression or sub-expression. While generating the graph, the *graph generator* stores the keywords specified in the query in a keyword buffer. Once the PDG is generated, the *graph generator* queries the index for each of the keywords it has stored in its buffer. This is done through the *index interface* module, which is responsible for retrieving the "hits" for each keyword from the index. The detection engine of InfoSearch is designed to be generic and capable of working with any kind of index. The "hits" are then wrapped into a set of <docID, start offset, end offset> tuples "tuples" and passed on to the leaf node that represents the keyword. Leaf nodes propagate their input to their parent nodes. The parent nodes, which correspond to one of the operators, merge their input sets according to the appropriate semantics.

4.1 Implementation

Whenever the *graph generator* comes across a token which is a keyword or a phrase, it stores this token in a *Vector* object called the **keyword buffer**. The keywords in the buffer are passed to the *index interface* after the PDG construction is complete, whereas the phrases are passed to the *phrase processor*. The reason for having a keyword buffer is that it is essential that the PDG is completely constructed before the index can be queried for the keywords. If the keywords are passed to the *index interface* or *phrase processor* by the *graph generator* as and when it pops them off the stack, they will return the results from the index to the PDG possibly before it is completely constructed. Thus, the keyword buffer is essential to avoid triggering of PDG nodes by the *index interface* while the PDG is being constructed. If the synonyms option is chosen for any keyword in the query, the *graph generator* queries the WordNet synonym database to get synonyms for the keyword. This is done through an API called the Java WordNet Library (JWNL) [10]. For each synonym, a leaf node is constructed, and finally a SYN operator node is constructed which subscribes to the original keyword and all its synonyms.

The *index interface* has to provide standard methods to access data from the integrated index, and deliver the results to the *pattern detector* in a specific format. As such, it does not matter if the index being integrated is an inverted index, or any other kind of index, say a B-tree index, as long as an index interface for it is developed. In other words, if a new index has to be integrated with InfoSearch, an *index interface* for that index has to be created which will support the required calls from InfoSearch, and return data to it in the expected format.

The *pattern detection engine* is responsible for processing the result sets from the index. The *index interface* passes a reference to a *Vector* of *Tuple*s corresponding to a keyword to the leaf node corresponding to the keyword. Internal nodes of the PDG correspond to one of the operators. They get references to one or more *Vector*s from their children and merge them to produce an output *Vector*. This merging is done as per the operator semantics described earlier.

For the first release of this system, we built a simple inverted index using Berkeley DB Java Edition [11], and integrated it with InfoSearch. Since the Berkeley DB API is in Java, it was convenient to develop an *index interface* for it, because the rest of the InfoSearch system was also developed in Java. To create the inverted index, we

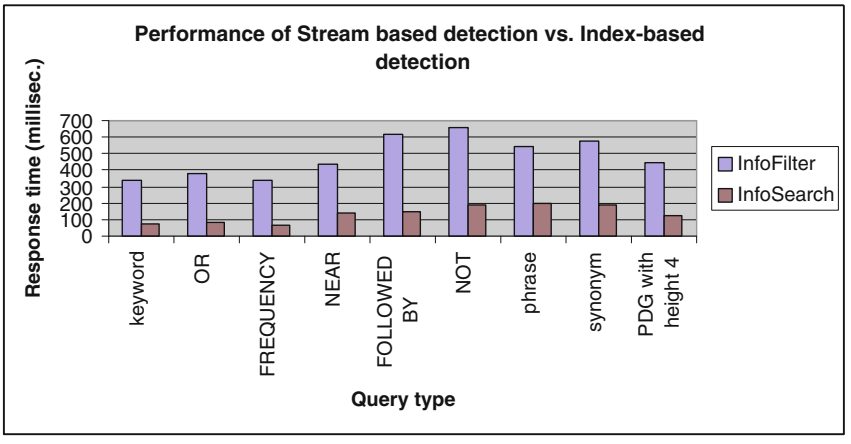

Fig. 7. Comparison of system performance over 2600 words

Operators \ Dataset Size	1MB		2MB		5MB		10MB		20MB		50MB	
	IS	IF	IS	IF	IS	IF	IS	IF	IS	IF	IS	IF
Keyword	5.33	6395	4.67	24651	4	18507	4	7779	3.33	7955	4.5	7456
OR	38	12249	92.5	24704	69	62832	68	50665	72.5	49271	73	119486
FREQUENCY	21.67	12379	18.67	23875	17	40575	18	34081	19.67	11173	19.5	32713
NEAR	23	10054	39	24184	53.67	62064	44.5	11970	39.33	30332	50	28767
FOLLOWED BY	21	12652	39.33	24528	63	61419	54	122912	54	260296	67.5	659407
NOT	50	12662	64.67	24347	93	62378	94	124080	96	250709	103.33	678606
PHRASE	32	12265	29.67	24417	63.5	62365	44.5	116913	43.5	248182	64.3	633933
SYN	40.67	12748	42	24169	54	15503	40.67	114311	42	113739	47	654118
PDG WITH HEIGHT 4	44.67	12849	47.33	15845	76.5	61585	82.5	124865	82	257901	105	845303

Fig. 8. Response Time of InfoSearch (IS) and InfoFilter (IF) Systems in milliseconds for each Operator

have created a Java program called *DocumentIndexer* which takes a given folder of documents, reads the documents, and builds an inverted index over those documents. For every keyword in each document, it stores a "hit" in the inverted index, which contains the path of the document the keyword is from, and the position of the keyword in that document.

5 Experimental Results

The primary reason for developing operators and algorithms to detect complex patterns over indexed data was to support efficient searching of stored documents for complex patterns. It does not make sense to stream *already stored* documents and use InfoFilter [5] for detecting patterns. Since InfoSearch uses an index-based approach, it is expected to be efficient for large volumes of data. A set of 20 documents of around 1.5 KB each were selected from the Reuters-21578 dataset[3] and the documents were artificially

[3] Available at http://www.daviddlewis.com/resources/testcollections/reuters21578

converted into a stream, fed to InfoFilter [5] and patterns involving all the operators were detected over this document stream. The time taken to process the stream was noted in milliseconds. Subsequently, the same documents were indexed for testing the efficiency of InfoSearch and the time taken for the result set to reach the root node was noted in milliseconds. For each of the operators, the performance of these systems are noted and are shown in Figure 7. The experiments were again repeated with document sets of sizes 1MB, 2MB, 5MB, 10MB, 20MB and 50MB and the results are shown in Figure 8. The last row of the table indicates a pattern whose PDG is height 4 (has 4 operators and at least 5 leaf nodes). From that row it is clear that the improvement for data sizes as little as 1MB is more than a factor of 100. This is even better with the improvement being more than a factor of 1000 for data volumes of 50MB. As can be seen, the detection for index-based algorithm grows sub-linearly whereas the detection time for the streaming approach grows super-linearly. The time taken to index the documents is not considered in the above comparison (which is negligible as compared to the improvements in detection time), since it is a pre-processing step, and is amortized over multiple searches on the repository.

6 Related Work

Most search engines use a variation of the vector space model [1] to select documents against a query from a document collection. In addition, search engines try to add other factors to the ranking process for documents including external (meta) information about the documents, references to documents from other documents, etc. Google [2] stores the pages fetched by the crawler in compressed form in a repository. It has a document index, which is a fixed width ISAM index, to keep information about each document. It also has a lexicon, forward index and an inverted index to facilitate rapid access to document lists. However, it support queries only in the form of keywords and Boolean compositions of keywords. INQUERY [3] is based on a form of the probabilistic retrieval model called the inference net. Inference nets [4] provide the capability to specify complex information needs, and compare them to document representations. The operators supported by INQUERY include *and, or, not*, a phrase operator and also an operator that handles proximity between patterns. In addition, specification of a particular argument as being more important that the others can be done. *However, there are no operators for sequence of patterns, pattern frequency, synonyms and containment.*

7 Conclusions

It was observed that current search systems are somewhat restrictive in the expressiveness of patterns that can be specified by the user. InfoSearch facilitates searching of complex patterns involving proximity, frequency, containment and sequences over a given document collection. The use of pattern operators and its modified semantics to provide an expressive pattern specification mechanism and to develop algorithms for an index-based approach are the main contributions of the paper. We have demonstrated that there is no loss of detection capability from stream mode to index mode for the pattern specification language. The overhead of additional information is quite small

in the form of offsets which can be readily obtained while indexing. The index-based algorithms are quite different from their counterparts in stream processing.

We are currently working on incremental algorithms where the computation can be stopped after detecting k patterns efficiently without having to use and compute all patterns. With this it is also possible to consider ranking the results for their utility from a users' viewpoint.

References

1. Salton, G., Wong, A., Yang, C.: A vector space model for automatic indexing. Communications of the ACM 18, 613–620 (1975)
2. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proc. of the WWW, Brisbane, Australia, pp. 107–117 (April 1998)
3. Callan, J., Croft, B., Harding, S.: The inquery retrieval system. In: Proc. of the DEXA, pp. 78–83 (1992)
4. Turtle, H., Croft, B.: Evaluation of an inference network-based retrieval model. ACM Transactions on Information Systems 9, 187–222 (1991)
5. Elkhalifa, L., Adaikkalavan, R., Chakravarthy, S.: Infofilter: A system for expressive pattern specification and detection over text streams. In: Proc. of the ACM SAC, Santa Fe, NM (March 13-17, 2005)
6. Chakravarthy, S., Elkhalifa, L., Deshpande, N., Adaikkalavan, R., Liuzzi, R.A.: How To Search for Complex Patterns Over Streaming and Stored Data. In: IC-AI, pp. 17–22 (2006)
7. Mauldin, M.L.: Lycos : Design choices in an internet search service. IEEE Expert (1997), `http://lazytoad.com/lti/pub/ieee97.html`
8. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kauffman, San Francisco (1999)
9. Deshpande, N.: Infosearch : A system for searching and retrieving documents using complex queries, Master's thesis, University of Texas at Arlington, Arlington (2005), `http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Des05MS.pdf`
10. Java wordnet library, `http://sourceforge.net/projects/jwordnet`
11. Berkeley db java edition, `http://www.oracle.com/us/products/database/berkeley-db/je/index.html`