INFOSEARCH: A SYSTEM FOR SEARCHING AND RETRIEVING DOCUMENTS USING COMPLEX QUERIES

by

NIKHIL PRADEEP DESHPANDE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2005

To my parents, who taught me the ideals in life.

ACKNOWLEDGEMENTS

I would like to thank Dr. Sharma Chakravarthy for giving me an opportunity to work on this exciting project. Dr. Sharma's guidance, advice, support and encouragement have been invaluable in the completion of this work. Sincere thanks are also due to Dr. Gautam Das and Dr. JungHwan Oh for serving on my committee.

I am grateful to Raman Adaikkalavan for helping me out during all stages of this thesis work (and sometimes rescuing me from a tight spot). I also thank Dhawal Bhatia and Hari Hara Subramanian for maintaining a well administered research environment.

This work was also supported, in part, by the National Science Foundation (NSF) (grants IIS-0123730 and IIS-0326505) and the Computer Science and Engineering Department at UT Arlington.

November 18, 2005

ABSTRACT

INFOSEARCH: A SYSTEM FOR SEARCHING AND RETRIEVING DOCUMENTS USING COMPLEX QUERIES

Publication No. _____

NIKHIL PRADEEP DESHPANDE, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Sharma Chakravarthy

The colossal amount of information available online has resulted in overloading users who need to navigate this information for their routine requirements. Although search engines have been effective in reducing this information overload, they support only keyword searches and queries that use Boolean operators. There are certain application domains where more expressive ways of searching are necessary. Consider searching a full-text patent database for documents containing more than n occurrences of a particular pattern, or for documents that have a particular pattern followed by another pattern within a specified distance. Such complex patterns involving pattern frequency and sequence of patterns, as well as patterns involving proximity, structural boundaries and synonyms are not supported by current search engines. Users searching documents based on focused, precise semantics may need to specify such patterns. This calls for a document retrieval system (pattern specification language as well as an efficient detection engine) that allows such expressive patterns.

Presently, we have an expressive pattern specification language and detection algorithms that work on stream data. That is, data coming as a stream of text (or converted to a stream of text) is fed into the Pattern Detection Graph (PDG) using a dataflow approach. This is suitable and required for searching patterns over frequently updated data sources, such as news feeds, IP packets, etc., in order to ensure freshness of the search results. However, this approach is inefficient and unwarranted for searching patterns over data sources that are extremely large or relatively static, or where freshness is not critical (e.g., Web repositories). For such large data sources, streaming approach becomes impractical. One alternative is to use a pre-computed (or computed off-line) index over the stored documents to efficiently detect expressive queries.

This thesis investigates the type of information needed as part of the index to detect patterns that include frequency, proximity and sequence of patterns, phrases, synonyms, etc. We present algorithms for each operator that uses the index and detects the pattern. The thesis also deals with grouping of common subexpressions and their detection in an efficient manner. We describe InfoSearch, a system for accepting complex patterns and retrieving the documents that contain the patterns, by using an inverted index over the document collection. InfoSearch allows specification of patterns that include word frequency, proximity, sequence or structural boundaries using a Pattern Specification Language (PSL). For searching these patterns and retrieving the matching documents, InfoSearch uses Pattern Detection Graphs to filter the results returned by the index.

Interestingly, very little additional information is needed in the index to detect all the patterns that were detected by the streaming approach. Furthermore, the complexity of the algorithms is not very different from the earlier approach.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

A huge amount of information is available today in digital format. With the exponential rate of growth in the amount of electronic information, the problem of how to access context relevant information has become an important one, and has received a lot of attention in the last decade. Users want quick access to information which is relevant to what they are looking for.

1.1 Information Access Methods

Database Management Systems (DBMS) were the first step in the direction of organizing and accessing large amounts of information. They allow efficient storage of data, and provide a query language such as SQL to effectively retrieve information from the data store. However, since DBMSs use a relational model, they restrict the form of the data being stored to be structured in nature. Second, the query languages for DBMSs are structured, and not very easy to learn and use. Also, access to the DBMS is typically permitted to only a small group of users, thus making it a tightly controlled system.

A much larger volume of textual data resides in unstructured documents such as text files, HTML files and documents created in various other formats. As a consequence, Information Retrieval (IR) evolved as a mechanism to search such a collection of documents based on user queries. The Information Retrieval Query Languages (IRQLs) are typically simpler to use than DBMS query languages. They allow users to specify queries

using keywords and Boolean operators, thus making it easier and quicker to find the required documents.

With the advent of the World Wide Web (WWW), the amount of information available in document form has surged exponentially. Searching the WWW effectively became a major research area, and brought IR to the forefront. Web Search engines adapted IR techniques to satisfy information need on the WWW, and are discussed below.

1.2 Search Engines

Search engines generally use a program called a *crawler* to fetch documents from the Web. These documents are parsed to extract tokens (keywords). In some cases, the documents themselves are stored [1]. The tokens generated from each document are used to populate an inverted index. An inverted index essentially stores a mapping from a keyword occurrence to a list of documents that contain that keyword. Billions of documents are indexed by Search Engines, and hence the efficiency of the index is crucial. It has to make efficient use of disk space, and also provide quick lookups for millions of queries per day. User queries, specified mostly in the form of keywords, are evaluated against this index using some variation of the vector similarity model [2]. The results of the lookup are usually sorted in descending order of relevance, according to a ranking algorithm. The Boolean operators AND, OR, and NOT, and exact phrase matching are also allowed in some cases. Search Engines have been instrumental in enabling users to quickly find the information they need on the Web from billions of documents. They have become starting points for information exploration on the Web.

1.3 Complex patterns

Although current Search Engines 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 more complex patterns in data sources. Users in these domains may have more precise requirements in terms of what they are searching for, i.e., they may be searching for patterns that are more specific. These patterns may 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). Current IR systems and Search Engines do not provide a means to specify and detect such complex patterns. In other words, the expressiveness of query specification provided by current Search Engines, although satisfactory for general searches, is not adequate for certain applications.

1.4 Nature of data sources

The data sources over which these complex patterns need to be detected can be divided into two categories. The first category is a dynamic data source, such as a news feed, in which the data inflow is frequent and unpredictable. The second category is a relatively static source of documents, which do not get updated on a very frequent basis, e.g., Web repositories. Monitoring a dynamic source essentially entails streaming in the data to detect the required patterns. In other words, to detect a pattern, the entire data source must be read every time. This is expensive, 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, it is redundant and inefficient to read the entire source each time a pattern is to be detected. A better approach would be to build and leverage some kind of meta data on the source. Specifically, the data source could be indexed, 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 a frequently updated one, 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.

1.5 Problem statement

Searching complex patterns over dynamic streams has been studied and shown to be possible in [3], in which a suite of complex operators and their algorithms were developed that detect complex patterns over stream data. However, for static data sources, none of the known Search Engines support anything more complex than keyword searches and simple Boolean compositions of keyword searches. It is possible to apply the same technique that is used to detect complex patterns in dynamic sources, to detect patterns over stored data repositories. That is, the stored data could be artificially converted into a stream and fed to the detection engine each time a pattern is to be detected. However, this is bound to be inefficient because of redundant reads of the data source which is not updated very often. The inefficiency will be exacerbated as the data source grows larger. *Efficiently* detecting complex patterns over large, static data repositories is certainly required in many application domains like searching full-text patent databases, federal intelligence, analysis of logs, etc.

The technique for searching complex patterns over streams in [3] makes use of the sequential inflow of patterns by reading the entire data source to detect a pattern. However, if we use an index instead of streaming in the data, we 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 objectives of this thesis are to:

- Identify what information is needed in the index to correctly and efficiently detect complex patterns which can presently be detected using a streaming approach.
- Investigate whether complex patterns, which can be detected by streaming in the entire data source, can be detected using information stored in an index.
- Explore the extent of the complexity of the patterns that can be detected using indexed information.

This thesis proposes a framework in the form of a system that allows specification of complex patterns, and an efficient computation model for detecting these patterns over a given document set.

1.6 Contributions

The main aim of this work is to answer the question: Given a collection of documents, how does one support search for complex patterns over the document collection in an accurate and efficient manner? From the point of view of the user, enabling complex pattern search allows for more expressive query specification. A system has been developed that clearly separates pattern detection from index lookup. This facilitates replacement of the index part without affecting the pattern detection part. A complete set of operators, such as frequency, proximity, containment, non-containment, sequence and synonyms, which work on stream data in the predecessor system, InfoFilter [3], have been developed to work on data retrieved from an index. These operators are designed

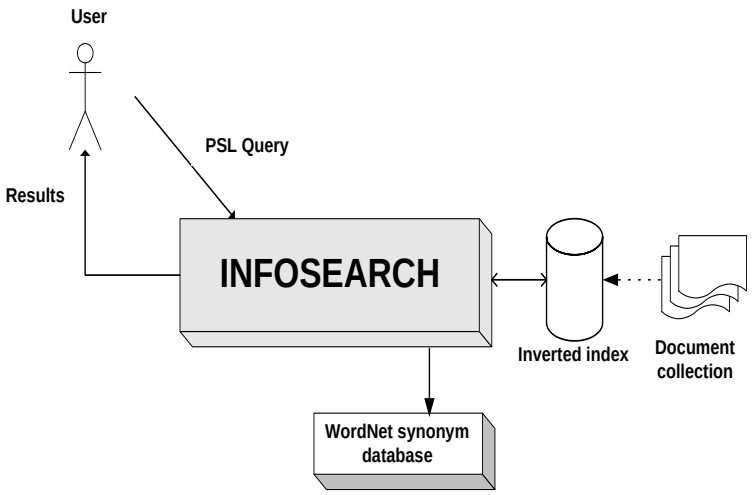

Figure 1.1 High level design of InfoSearch

such that they can take result sets from the index and generate output result sets according to the semantics defined for that operator. Since the order of occurrence of patterns is lost as a result of working with indexed information, the operators reconstruct this order to generate appropriate result sets. Essentially, detection of arbitrarily complex patterns is being enabled on a static document set, with a granularity of pattern specification that is finer than that offered in current IR systems and Web Search Engines. Fig. 1.1 outlines the InfoSearch system, which takes the user query specified in an expressive Pattern Specification Language (PSL), queries the integrated index, uses the WordNet synonym tool to look up synonyms if required, and returns the output to the user.

The organization of the rest of this document is as follows: Chapter 2 reviews the related work. Chapter 3 goes into the details of the working of InfoSearch operators. Chapter 4 discusses the design of InfoSearch. Implementation aspects are discussed in Chapter 5. Chapter 6 concludes the thesis and identifies some potential future directions.

CHAPTER 2

RELATED WORK

Research in the field of Information Retrieval (IR) started in the 1950s, as the need for rapid access to stored documents was felt. In the last decade or so, there has been a tremendous acceleration in research and development in IR, due to the proliferation of electronic information on the World Wide Web. The models studied and developed for traditional IR systems are the backbone for all modern Web Search Engines. The main models are the Boolean model, the Vector Space model and the Probabilistic model, which are described below.

2.1 The Boolean model for IR

Boolean models allow users to specify their queries using a composition of AND, OR and NOT operators. The Boolean model is a clean and simple model, and hence was used in initial IR systems [2]. The model has its basis in set theory - if a document contains exactly the pattern specified by the query, then the document is selected as being relevant. The AND operator essentially performs set intersection, OR does set union and NOT does set difference. The disadvantage of the Boolean model is that it is inherently precise - there is no room for partial matches to a query. For example, if there is a query like *"cancer" AND "treatment"*, a document containing several occurrences of *"cancer"* and *"remedy"*, but no occurrence of *"treatment"* will not be selected as relevant, although it may potentially contain useful information regarding cancer treatment. The Boolean model's retrieval strategy is based on a binary decision criterion, i.e., a document is either predicted to be relevant or non-relevant, without any notion of a grading scale.

The vector space model sought to overcome this shortcoming of the Boolean model, and is described next.

2.2 The Vector Space model

In the vector space model, documents and queries are represented as vectors in an n dimensional space, where n is the number of terms in the entire vocabulary set [4]. Thus, if a term belongs to a document, it gets a non-zero value along the dimension corresponding to the term.

Selection of a document against a query is done by assigning a score for the document against the query. This score is computed by measuring the similarity of the query with the document. Typically, the cosine of the angle between the query vector and the document vector is taken as a measure of similarity: 1.0 implies a perfect match, and 0.0 implies orthogonality. Based on the importance of the terms in the query and the documents, weights are assigned to every term in the query and document. How these weights are assigned differs from system to system. Typically, the term frequency, *tf* (number of times the term occurs in the document) and the inverse document frequency, *idf* (in how many documents does the term occur? The more number of documents it occurs in, the lesser is its discriminating power in identifying a document) are used to compute weights for terms. If $\overrightarrow{D}$ is the document vector and $\overrightarrow{Q}$ is the query vector, the similarity of document D with query Q is given by:

$$Sim(\overrightarrow{D}, \overrightarrow{Q}) = \sum_{t_i \in Q,\ D} w_{t_i Q} \cdot w_{t_i D}$$

where $w_{t_i Q}$ is the weight of term i in the query, and $w_{t_i D}$ is the weight of term i in the document. Hence, the final score for the document depends on how many query terms occur in the documents, and the weight of these terms in the document as well as the query. Importantly, if one or more query terms do not occur in the document, the

document can still have a score because of the presence of some other query terms. This allows for partial matches.

The vector model has been found to be the most effective retrieval model for keyword based searches, and hence a variation of this model is used in most of the current IR systems.

2.3 Probabilistic models

Probabilistic models are motivated by the observation that the relevance of a document to a query is related to the probability of the query terms occurring in the document. Because true probabilities are not available to a system, the system has to estimate the probability of relevance of documents to a query. The probability estimation technique is the key part of the model, and different techniques have been proposed in the literature [5] [6]. Only the basis of the models is described here.

The probability of relevance of a document D can be represented by $P(R|D)$. We can rank documents based on $\frac{logP(R|D)}{logP(R|\overline{D})}$, where $P(R|\overline{D})$ is the probability that the document is non-relevant. By applying Bayes' transform to this ratio, we obtain $\frac{P(D|R).P(R)}{P(D|\overline{R}).P(\overline{R})}$. Assuming that $P(R)$ is independent of the document under consideration, $P(R)$ and $P(\overline{R})$ are simply scaling factors for the final document scores, and hence can be canceled out. This leaves us with $\frac{P(D|R)}{P(D|\overline{R})}$ as the score formula. After this point, different systems diverge based on the assumption behind the estimation of $P(D|R)$.

2.4 Applying IR techniques to the Web: Search Engines

Because the Web is an extremely large, heterogeneous and uncontrolled collection of documents as opposed to the more controlled, smaller document repositories for which standard IR techniques were designed, Search Engines have adapted and extended stan-

dard IR models to be effective. The volume of documents on the Web is much higher than that in controlled, closed IR systems, and hence standard IR techniques by themselves do not produce good results. Standard IR techniques try to return the documents that most closely match the query, given that both the query and document are represented by their word occurrences. On the Web, this strategy returns very short documents that contain the query plus a few words. 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. Retrieval strategies of two popular Search Engines, Lycos and Google are discussed in this section.

2.4.1 Lycos

Lycos was one of the earliest commercial Web Search Engines [7]. It uses a breadth first technique for "foraging" (crawling) the Web to locate documents. Once a document is located, an automated "abstract" of the document is generated and stored. The "abstract" contains a concise description of the documents, by including the 100 most "weighty" words in the document. This results in space saving, because the "abstract" is only about one-fifth of the actual document in size.

Lycos uses inverted file indexing, or a postings file, which stores a mapping from a keyword to a list of documents that contain the keyword, along with additional information such as a list of all positions in the document in which the word occurs. Storing the positions allows the Search Engine to efficiently check proximity or adjacency during retrieval, so that documents in which the query keywords appear closer can be ranked higher. Inverted file indexing allows for much faster retrieval than a sequential scan of the database. Lycos ranks the results according to a relevance ranking based on number of query terms contained in the document, frequency of occurrence of these terms, proximity of query terms, position of occurrence of the query terms, etc. Query support is in

the form of keywords and Boolean compositions of keywords. Complex queries including proximity, containment, etc. are currently not supported.

2.4.2 Google

Google [1] is perhaps the most popular Web Search Engine in existence today. It uses highly optimized data structures in order to crawl, index and search the large collection that the Web is. It 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.

Google maintains much more information about each document than other Search Engines do. It maintains position, font and capitalization information for the keywords. Google computes a PageRank [8] for each page that is indexed. The PageRank algorithm uses the number of sites linking to a page as a measure of quality of that page. Essentially, a hyperlink is like a citation, and the more the number of hyperlinks to a page, the more popular and important it is assumed to be. Thus, a page can have a high PageRank if there are many pages that point to it, or if there are some pages that point to it that have a high PageRank. Google also associates anchor text (description of a hyperlink in Web pages) to the page it is pointing to, because anchors often provide more accurate descriptions of pages than the pages themselves. In order to rank a document with a single word query, Google looks at the document's "hit list" for that word. Each hit can be one of several types (title, anchor, URL, large font, etc.), each of which has its own type-weight. The count of each type of hit is converted into a count-weight. The dot product of the vector of count-weights and the vector of type-weights is taken to compute an IR score for the document. This score is combined with the PageRank to give a final rank to the document. Keyword queries are supported in the main interface, and

there is also a provision to specify Boolean compositions of queries and phrases using an "Advanced Search" screen. However, complex queries based on pattern frequency, proximity, non-occurrence of a pattern within two patterns are not currently supported.

2.5 The INQUERY retrieval system

INQUERY [9], developed at the University of Massachusetts at Amherst, is based on a form of the probabilistic retrieval model called the inference net. Inference nets [10] provide the capability to specify complex information needs, and compare them to document representations. INQUERY uses a Bayesian inference network (or Bayes net), which is a directed acyclic graph (DAG) in which nodes represent propositional variables and arcs represent dependencies. The Bayes net in INQUERY consists of two component networks: one for documents, and one for queries.

The operators supported by INQUERY include *and, or, not*, a phrase operator and also an operator that handles proximity between patterns. In addition, one can specify how the "beliefs" of the specified arguments are to be weighted by the system. In other words, 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.*

2.6 InfoFilter: A system to detect complex patterns over text streams

InfoFilter [3], developed at The University of Texas at Arlington, enables complex pattern detection over dynamic data sources. It supports a number of operators such as proximity, sequence, pattern frequency, non-containment in addition to keyword and phrase searches. It also has an option to look for occurrences of synonyms of a given word, by making use of the WordNet synonym database [11]. InfoFilter continuously

tokenizes incoming streams and uses a graph structure to detect user specified patterns in real time. The architecture of InfoFilter is shown in Figure 2.1.

The user specifies a pattern using an expressive Pattern Specification Language (PSL). The user pattern is validated and passed to the *graph generator*, which internally represents the pattern as a Pattern Detection Graph (PDG). Leaf nodes of the PDG correspond to simple patterns such as keywords, phrases and system defined patterns such as structural boundaries. Internal nodes of the PDG correspond to operators. The *graph generator* also extracts keywords and phrases from the user patterns and inserts them into a data structure called a suffix trie. A suffix trie enables quick lookups of strings or their suffixes [12]. The *graph generator* also interacts with the WordNet database to extract synonyms for a keyword, if necessary. InfoFilter has a *stream processor* for each kind of text stream it handles, such as text, e-mail, HTML, etc. The *stream processor* parses the incoming stream and generates tokens. These tokens are looked up in the suffix trie, and if a match is found, the corresponding leaf node in the PDG is triggered. This means, information regarding the keyword or phrase occurrence is passed to the PDG. The operators combine occurrences of their children to detect complex patterns according to the Proximal-Unique semantics (refer Section 3.2).

This thesis (InfoSearch), extends the capabilities of InfoFilter to search over a stored set of documents by using an index over the documents. Since the same operators are used to specify patterns in InfoSearch, we give an overview of the operators below [3] [13]:
OR: Disjunction of two simple or complex patterns *P1* and *P2* is denoted by *(P1 OR P2)* and occurs when either *P1* or *P2* occurs. For example, *"information" OR "retrieval"* occurs when either one of the keywords occurs.
NOT: Non-occurrence of the pattern *P2* in the interval formed between the end of the pattern *P3* and the beginning of the pattern *P1* is denoted by *NOT[/F](P2)(P1, P3). F* denotes the maximum number of permissible occurrences of *P2* within *P1* and *P3*, for

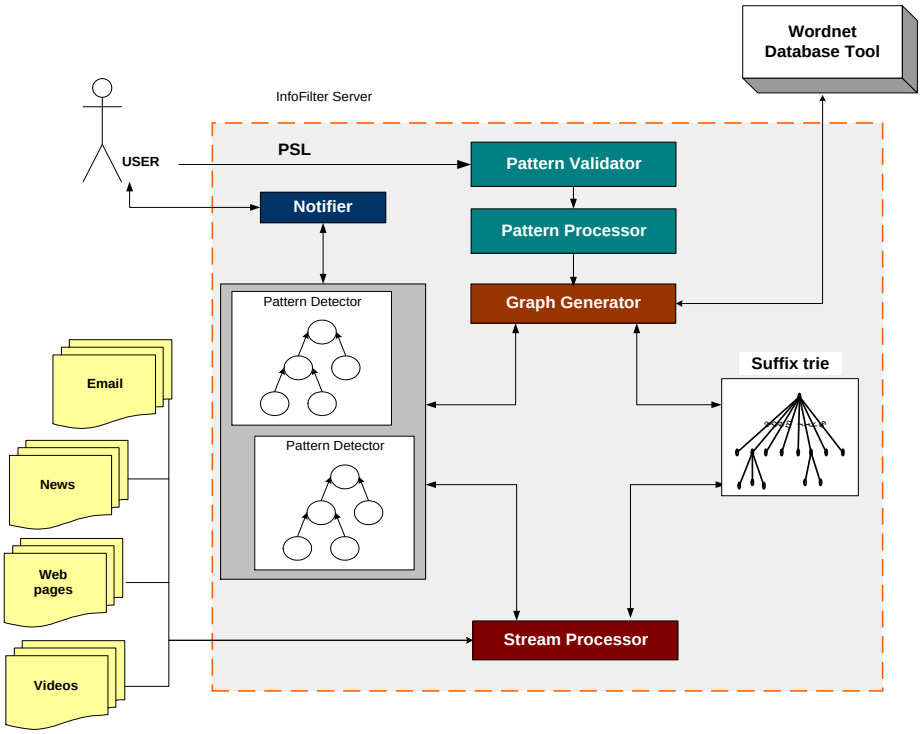

Figure 2.1 InfoFilter architecture

the NOT expression to evaluate to true. For example, *NOT/2("filtering")("information", "retrieval")* occurs when *"information"* is followed by *"retrieval"* with *"filtering"* occurring at most twice in between. If F is unspecified, the system operates with a default value of 0.

NEAR: This operator is similar to a conjunctive (AND) operator, but it allows specification of an optional distance. *(P1 NEAR[/D] P2)* occurs when the simple or complex patterns *P1* and *P2* co-occur, in any order, separated by a maximum of D words. For example, *("information NEAR/10 "retrieval")* will be detected if the two keywords co-occur within a distance of 10. If D is unspecified, the system simply checks for co-occurrence of the operands, without checking for distance (or the whole stream or document is used as the distance).

FOLLOWED BY: Sequence of two simple or complex patterns *P1* and *P2*, denoted by *(P1 FOLLOWED BY[/D] P2)* occurs when the occurrence of *P1* is followed by the occurrence of *P2* within *D* words of each other. If *D* is unspecified, the operator will detect the sequence irrespective of the distance separating the operands.

WITHIN: This operator is used for detecting containment of a pattern within predefined structural boundaries. Occurrence of a simple or complex pattern *P2* within structural patterns *P1* and *P3* is denoted by *(P2 WITHIN (P1, P3))*. The pattern is detected each time *P2* occurs in the range formed by *P1* and *P3*. For example, *("information retrieval") WITHIN (beginPara, endPara)* is detected whenever the phrase *"information retrieval"* occurs within the scope of a paragraph.

FREQUENCY: Multiple occurrences of a simple or complex pattern *P* that exceed or are equal to *F* is denoted by *(FREQUENCY/[F](P))*. For example, the pattern *FREQUENCY/5("information" NEAR "retrieval")* is detected whenever the enclosed *NEAR* expression occurs 5 times or more.

SYN: This is an option used with single word patterns to specify lookup of either the keyword itself or any of its synonyms. For example, *("mining"[SYN])* will result in the same output as would *("mining" OR "excavation")*: the system internally extracts the synonyms and detects the pattern if either the word or any of its synonyms are found.

InfoFilter has been designed for detecting complex patterns over streaming data sources. A naïve approach for detecting complex patterns over stored documents would be to artificially convert the stored documents into a stream by reading the entire data source and tokenizing it, and then feeding the resulting stream to InfoFilter. However, since this approach requires reading of the entire data source every time to detect a pattern, it is redundant for stored, slow-changing documents and inefficient for large document collections. InfoSearch seeks to enable detection of complex patterns with

the same semantics over a stored repository, by detecting patterns over an index on the document store for efficiency.

2.7 Summary

IR systems started out with the intention of providing quick references to documents in a large collection. The Boolean model of IR, although simple, was found to be simplistic for large collections, and was superseded by other models. The vector similarity model was generally accepted as the best model, and is used in the core of most IR systems in use at present. Initially, the document collection was relatively small and controlled. With the advent of the Web, as the nature of the documents to be indexed became heterogeneous and the number of documents soared to millions, new techniques had to be devised to adapt traditional IR to the Web.

However, we observed that neither traditional IR systems nor Web search engines support more than keyword lookups and Boolean operators over keywords, and in some cases, phrases. Although this works fine for general purpose searching, certain applications like searching patent databases demand more expressive query support, which is not provided in the current search systems. This problem has been addressed for stream data in the previous work (InfoFilter) on extending the expressiveness of patterns. There is a need for systems which can allow users to pose complex patterns, and detect these patterns efficiently over a large document collection. Since using an index is an efficient way for searching on keywords, we extend the index based approach to detect complex patterns, hopefully preserving efficiency and at the same time enhancing the expressiveness of queries significantly.

CHAPTER 3

INFOSEARCH OPERATORS

InfoSearch accepts complex queries from the user, searches an index built on a collection of documents, and returns a list of documents that contain the specified pattern. It also returns the starting and ending position of each pattern occurrence within a document. The system can be broadly divided into two components. First, an expressive query language through which the user can specify patterns involving term frequency, sequences, proximity and containment is required. Second, a pattern detection engine is required. This engine should be capable of getting the required information from the index, and processing this information to generate results in response to a user query. InfoSearch adopts the Pattern Specification Language (PSL) and its associated parser and pattern validator used in InfoFilter [3], with some minor changes. Hence, we do not describe the details of the Pattern Specification Language, other than the basic syntax for specifying queries. The focus of this research is on the second part, namely the detection of complex patterns over large document repositories.

3.1 Inverted indexes

One way of indexing a document collection is to store a mapping from each document to all the words it contains. In this case, if we need to search for a keyword, it becomes necessary to scan the word lists corresponding to each document and check if the keyword occurs in the document. This results in a sequential scan of the database, and would be inefficient as the size of the document collection increases. An alternative is to use inverted indexes, which are described next.

The inverted index (also called an inverted list) is the most common mechanism used in Search Engines to maintain a mapping from a keyword to the documents that contain the keyword. Given a collection of documents, document 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 3.1 and the corresponding inverted index is shown in Table 3.2. 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).

Table 3.1 A sample set of documents

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

Table 3.2 Inverted index on documents in Table 3.1

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

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 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 [14]. Table 3.3 shows an inverted index generated on the documents in Table 3.1 with the position information stored.

Table 3.3 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>

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 research 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. In other words, this thesis answers the proposition "Can all the operators that are supported by systems which stream in the data source

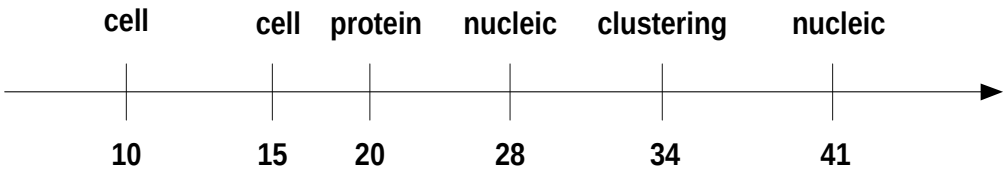

Figure 3.1 Example document for discussion of Proximal-Unique semantics

every time, be supported by leveraging an index over the documents containing the document ID and position information of keywords?" affirmatively and presents the algorithms for each operator which can be used to detect complex patterns.

3.2 Proximal-Unique semantics

Consider a document containing occurrences of words as shown in Figure 3.1. Suppose we want to find occurrences of *"cell" FOLLOWED BY "nucleic"* within this document. As seen 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 $cell^1$ 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 other words, occurrence of a pattern in a document supersedes its previous occurrence in the document as far as semantics for combining with another pattern are concerned. 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. 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 with $nucleic^2$ will result in the detection of another instance of the same pattern using a previously used sub-pattern. The Proximal-Unique semantics has been defined to take this intuitive sense into consideration when detecting a pattern by applying restrictions on the usage of sub-patterns. It is important to understand that the general interpretation of the operators detects patterns that may not be intuitive.

As another 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 uncombined occurrence of their type. However, it does not make intuitive sense to combine them because they overlap each other. Hence, the pattern *("cell" FOLLOWED BY 'nucleic") NEAR ("protein" FOLLOWED BY "clustering")* does not occur in the document, because intuitively, only disjoint sub-patterns should be combined. The NEAR operator used here assumes non-overlapping (disjoint) NEAR for composite patterns (This problem does not arise for a simple pattern or a word) and hence the above pattern is not detected. It is also possible to define another NEAR operator that allows overlaps for composite patterns. This thesis does not address the overlapping semantics.

InfoSearch operators use the Proximal-Unique semantics to combine patterns to generate result sets.

3.3 Pattern Detection Graphs

To facilitate detection of complex patterns, InfoSearch uses a data structure called a 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 3.2. 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**. For example, the set of tuples for the keyword *"information"* from the index shown in Table 3.3 is shown in Table 3.4. 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 InfoSearch operators such as OR, NEAR, FOLLOWED BY, WITHIN or NOT, thus gets one or more sets of tuples as its input. The operator merges its input sets according to its semantics to create an output set. The InfoSearch operators use Proximal-Unique semantics to merge their input tuples. 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.

An important thing to note is that tuples corresponding to word occurrences in a document are *point tuples*, i.e., the start offset and end offset of such tuples is the same. This is because a word occurs at a single position within a document. On the other hand, a tuple corresponding to a more complex pattern is an *interval tuple*, i.e., its start offset is smaller than its end offset. This is because a complex pattern such as *"Protein" FOLLOWED BY "Clustering"* occurs in an interval within the document. The range of this interval is the start offset of the initiating sub-pattern and end offset

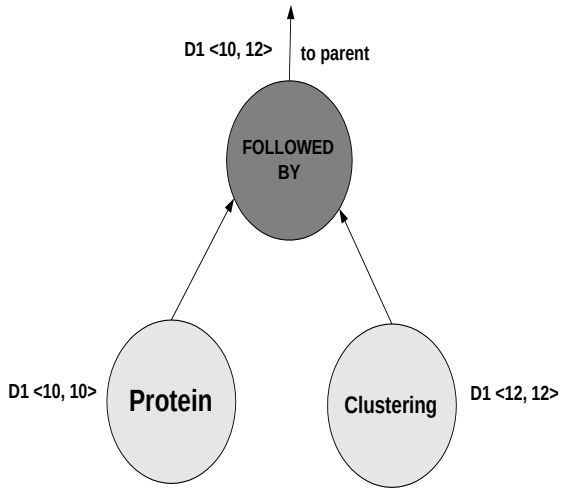

Figure 3.2 PDG corresponding to *"Protein" FOLLOWED BY "Clustering"*

of the terminating sub-pattern. For example, in Figure 3.2, the tuples corresponding to *"Protein"* and *"Clustering"* are point tuples, but the tuple corresponding to the combined pattern *"Protein" FOLLOWED BY "Clustering"* is an interval tuple. Thus, operators in InfoSearch may get either point tuples or interval tuples as their input, and their output will be, in most cases, interval tuples.

Table 3.4 Set of tuples corresponding to occurrences of *"information"*

1<1,1>
3<1,1>
3<3,3>

3.4 Operator algorithms

To allow users to specify more expressive queries, InfoSearch supports the following operators: OR, FREQUENCY, NEAR, FOLLOWED BY, WITHIN and NOT. The semantics of these operators are almost identical to the corresponding InfoFilter operators

described in Section 2.6. However, the working of InfoSearch operators is different from that of the InfoFilter operators. InfoFilter operates by reading in 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. 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. 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. 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.

3.4.1 The OR operator

The input to the OR operator is two sets of tuples, sorted by document ID, corresponding to the left operand and the right operand. The semantics of the OR operator dictate that a pattern is detected whenever either of the operands is detected. Hence, the output of the OR operator should be a union of its input sets, sorted by document ID. It is necessary for the output set to be sorted by document ID and pattern position within a document, because the parent nodes assume their inputs to be sorted. The algorithm of the OR operator is shown below. It generates an output set sorted in ascending order of the document ID. In cases where the document ID is the same, the output is sorted in ascending order of the end offset. This is done because the end point of a pattern determines its occurrence in a document, and the position where the pattern is detected within a document is critical for the semantics of operators using the Proximal-Unique context. The OR operator is a non-filtering operator: the number of tuples in its output set is the sum of the number of tuples in its two input sets. In other words, no tuples from the input are discarded. Essentially, the OR operator generates a sorted union of its input sets. The sorting is done on document ID and end offset of the tuples.

OR (*leftSet*, *rightSet*)

1: $left \Leftarrow$ first tuple in $leftSet$
2: $right \Leftarrow$ first tuple in $rightSet$
3: $resultSet \Leftarrow \{\}$
4: **while** $exists(left)$ OR $exists(right)$ **do**
5: **while** $left.docID < right.docID$ **do**
6: $resultSet \Leftarrow resultSet + left$
7: $left \Leftarrow left \rightarrow next$
8: **while** $left.docID > right.docID$ **do**

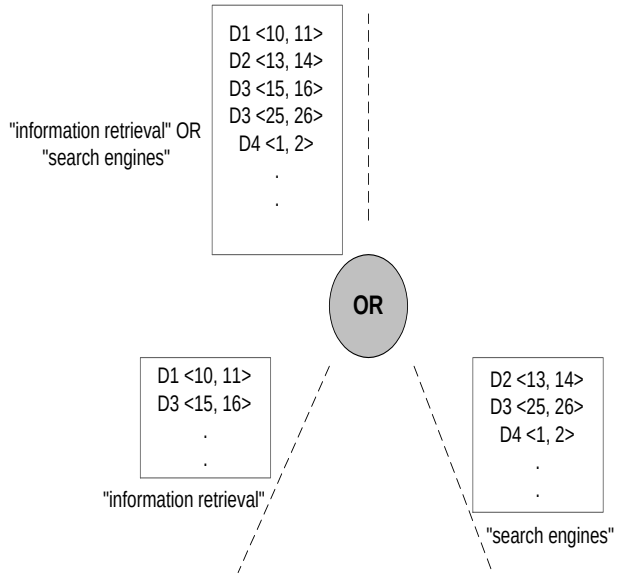

Figure 3.3 An example of the working of the OR operator

9: $resultSet \Leftarrow resultSet + right$

10: $right \Leftarrow right \rightarrow next$

11: **while** $left.docID == right.docID$ **do**

12: **if** $(left.endOffset < right.endOffset)$ **then**

13: $resultSet \Leftarrow resultSet + left$

14: $left \Leftarrow left \rightarrow next$

15: **else**

16: $resultSet \Leftarrow resultSet + right$

17: $right \Leftarrow right \rightarrow next$

As an example, Figure 3.3 demonstrates the working of the OR algorithm. The inputs are sets of tuples corresponding to the phrases *"information retrieval"* and *"search engines"*. The operator merges these input tuples to generate an output set corresponding to *"information retrieval" OR "search engines"*, in which the tuples are the union of the input sets, sorted on document ID and end offset.

3.4.2 The FREQUENCY operator

The FREQUENCY operator is a unary operator; it has a single set of tuples as its input. It is specified as *FREQUENCY/n(P)*, which means all documents containing more than n occurrences of pattern P should be selected. The operator essentially keeps a count of the number of occurrences of P in a given document in the input set. For every n occurrences of P in a given document, it adds a tuple to its output set. The pseudocode for the merging done in the FREQUENCY operator is shown below. It is assumed that if the input set is empty, it is not passed through this algorithm; in such a case, the operator also propagates an empty set as its output. Using this operator, not only can documents containing equal to or more than n occurrences of the pattern P be retrieved, but sections within the document which contain these occurrences can also be identified.

FREQUENCY (*inputSet*, n)

1: $current \Leftarrow$ first tuple in $inputSet$
2: $previous \Leftarrow null$
3: $Integer\ count \Leftarrow 0$
4: $Integer\ startValue \Leftarrow -1$
5: $outputSet \Leftarrow \{\}$
6: $Tuple\ outputTuple \Leftarrow null$
7: **repeat**
8: **if** $current.docID \neq previous.docID$ **then**
9: $count \Leftarrow 0$
10: $startValue \Leftarrow -1$
11: **if** $startValue == -1$ **then**
12: $startValue \Leftarrow current.startOffset$

13: $count++$

14: **if** $count == n$ **then**

15: $outputTuple.docID \Leftarrow current.docID$

16: $outputTuple.startOffset \Leftarrow startValue$

17: $outputTuple.endOffset \Leftarrow current.endOffset$

18: $outputSet \Leftarrow outputSet + outputTuple$

19: $count \Leftarrow 0$

20: $startValue \Leftarrow -1$

21: $previous \Leftarrow current$

22: $current \Leftarrow current \rightarrow next$

23: **until** $current = EOF$

Figure 3.4 shows an example of the working of the FREQUENCY operator. In the example, when the counter for D6 becomes 3, the operator generates an output tuple having the start offset of the first of the three tuples, and end offset of the last of the three tuples. An output tuple will be generated for every three tuples occurring in one document. Thus, if three more occurrences of "saccharide" occur in D6, another tuple will be added in the output set.

3.4.3 The NEAR operator

The NEAR operator is a binary operator. It is specified as *P1 NEAR[/d] P2*, where d is an optional distance. This means that occurrences of *P1* and *P2* within the same document, separated by not more than d words, should be detected. If d is not specified, the *NEAR* pattern is considered to be detected if *P1* and *P2* occur anywhere within a document. The relative order of occurrence of *P1* and *P2* does not matter; *P1* may either follow or precede *P2*, but *P1* and *P2* may not overlap as we are currently using disjoint semantics.

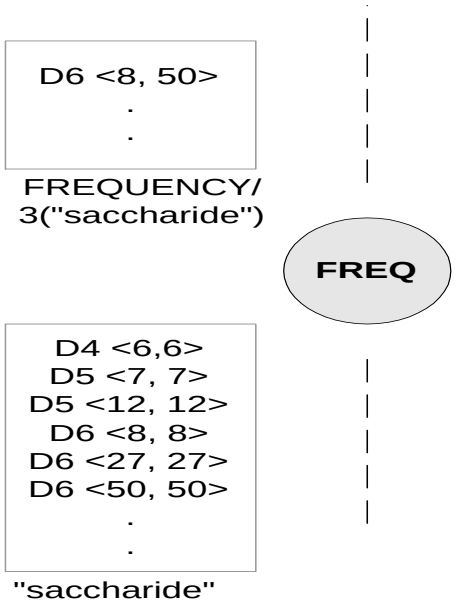

Figure 3.4 An example of the working of the FREQUENCY operator

Since the order of occurrence of the operands does not matter, either one can be the initiator. 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 are interval tuples. We now discuss the different cases possible when we consider two input tuples, and the actions taken in each case.

3.4.3.1 Discussion of 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

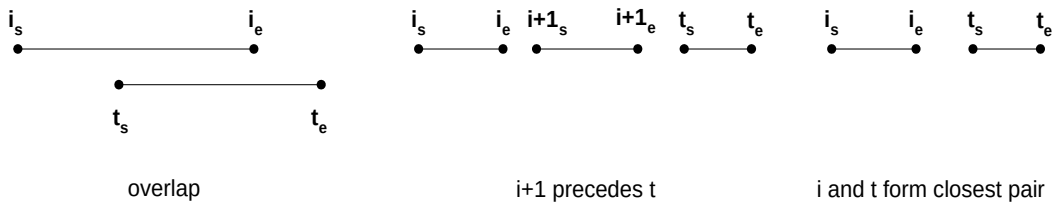

Figure 3.5 Some possibilities when $i_e < t_e$

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 3.5. We perform the following sequence of actions:

if *initiator* and *terminator* overlap **then**

lookahead[1] to determine new *initiator* and *terminator*

go to the beginning of this operation and re-process the new *initiator* and *terminator*

if $i+1_e \leq t_e$ **then**

make $i+1$ the new *initiator*, and re-process the new *initiator* and *terminator*

else

[1]The Lookahead algorithm is explained in subsection 3.4.3.2

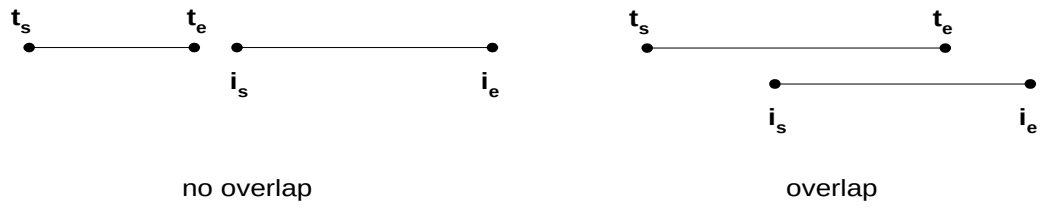

Figure 3.6 Some possibilities when $t_e < i_e$

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 criteria 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 3.6. In this case, we swap the *initiator* and *terminator* pointers, and re-process.

3.4.3.2 Lookahead algorithm

A lookahead is done when the current *initiator* and *terminator* cannot be combined due to an overlap, or because the distance criteria 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*

The NEAR operator pseudocode is shown below. It shows how processing of the input sets is done as per the above discussion. The pseudocode for the **lookahead** routine used in it is shown subsequently.

NEAR(L, R, d)

1: $initiator \Leftarrow$ first tuple in L
2: $terminator \Leftarrow$ first tuple in R
3: **while** $exists(initiator)\ AND\ exists(terminator)$ **do**
4: **while** $initiator.docID < terminator.docID$ **do**
5: $initiator \Leftarrow initiator \rightarrow next$
6: **while** $initiator.docID > terminator.docID$ **do**
7: $terminator \Leftarrow terminator \rightarrow next$
8: **if** $initiator.docID \neq terminator.docID$ **then**
9: $initiator \Leftarrow initiator \rightarrow next$
10: $terminator \Leftarrow terminator \rightarrow next$
11: continue
12: **if** $initiator.endOffset \leq terminator.endOffset$ **then**
13: **if** $overlap(initiator, terminator)$ **then**
14: $lookAhead(initiator, terminator)$
15: continue
16: **if** $initiator \rightarrow next.endOffset \leq terminator.endOffset$ **then**
17: $initiator \Leftarrow initiator \rightarrow next$
18: **else**
19: **if** $(terminator.startOffset - initiator.endOffset) \leq d$ **then**

20: $combine(initiator, terminator)$

21: $initiator \Leftarrow initiator \rightarrow next$

22: $terminator \Leftarrow terminator \rightarrow next$

23: **else**

24: $lookAhead(initiator, terminator)$

25: **else**

26: $swap(initiator, terminator)$

Lookahead

1: **if** $(initiator \rightarrow next).endOffset < (terminator \rightarrow next).endOffset$ **then**

2: $initiator \Leftarrow terminator$

3: $terminator \Leftarrow initiator \rightarrow next$

4: **else if** $(initiator \rightarrow next).endOffset == (terminator \rightarrow next).endOffset$ **then**

5: **if** $initiator.endOffset < terminator.endOffset$ **then**

6: $initiator \Leftarrow terminator$

7: $terminator \Leftarrow initiator \rightarrow next$

8: **else**

9: $terminator \Leftarrow terminator \rightarrow next$

10: **else**

11: $terminator \Leftarrow terminator \rightarrow next$

Figure 3.7 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

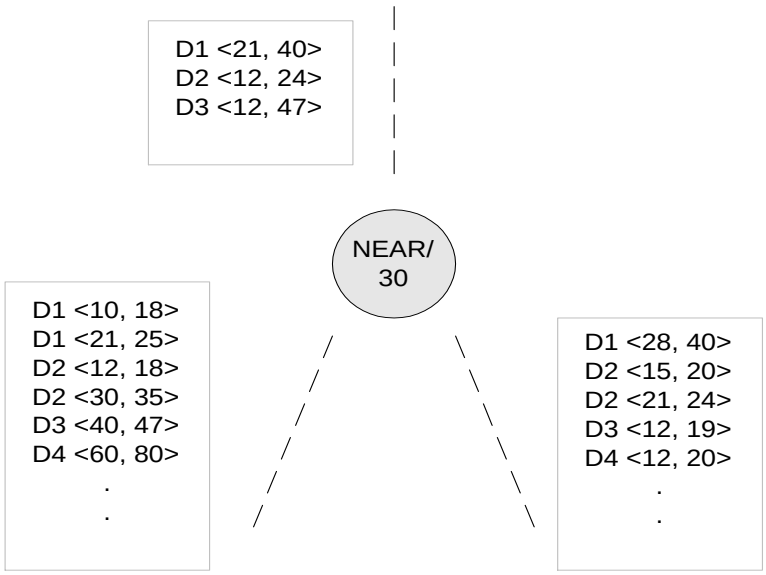

Figure 3.7 Example of the working of the NEAR operator

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 <12, 20>. 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.

3.4.4 The FOLLOWED BY operator

The FOLLOWED BY operator is a binary operator, specified as *P1 FOLLOWED BY[/d] P2*. This means that documents containing *P1* and *P2* both, should be selected, with the restriction that *P1* should occur before *P2*. The occurrences should be separated by at most d words. As with the NEAR operator, d is optional, and in its absence, InfoSearch assumes that any occurrence of *P1* followed by *P2* in a document should be detected, irrespective of the distance separating them.

The inputs to the FOLLOWED BY operator are two sets of tuples corresponding to the left child and the right child. According to the semantics of the operator, the left sub-pattern should occur before the right one in a valid detection; that is, only a tuple from the left set can be the initiator. Hence, the algorithm for the FOLLOWED BY operator is a simplified version of the NEAR operator algorithm. In the NEAR algorithm, we had to perform a lookahead to determine the new initiator and terminator in case of an overlap. This is not required in FOLLOWED BY, because we know that only the tuple from the left set can be the initiator. We do not need to swap pointers at any stage; we simply advance the right set pointer in case of an overlap, or if the distance criteria is not satisfied. The pseudocode for the FOLLOWED BY operator is given below.

FOLLOWED BY(L, R, d)

1: $left \Leftarrow$ first tuple in L
2: $right \Leftarrow$ first tuple in R
3: **while** $exists(left)$ AND $exists(right)$ **do**
4: **while** $left.docID <$ right.docID **do**
5: $left \Leftarrow left \rightarrow next$
6: **while** $left.docID >$ right.docID **do**

```
7:      right ⇐ right → next
8:    if left.docID ≠ right.docID then
9:      left ⇐ left → next
10:     right ⇐ right → next
11:     continue
12:   if overlap(left, right) then
13:     right ⇐ right → next
14:     continue
15:   if left.endOffset < right.endOffset then
16:     if (left → next).endOffset < right.endOffset then
17:       left ⇐ left → next
18:     else
19:       if satisfiesDistance(left, right) then
20:         combine(left, right)
21:         left ⇐ left → next
22:       right ⇐ right → next
23:   else
24:     right ⇐ right → next
```

Figure 3.8 gives a simple example of the working of the FOLLOWED BY operator.

3.4.5 The WITHIN operator

The WITHIN operator is a ternary operator, specified as *P2 WITHIN/[d](P1, P3)*. This means that documents containing *P1* followed by *P3*, with *P2* occurring at least d times in between, should be selected. d is optional, and if it is not specified, the system assumes a default value of 1. The operator gets three sets as its inputs, corresponding to the left, middle and right operand. As with the FOLLOWED BY operator, only a tuple

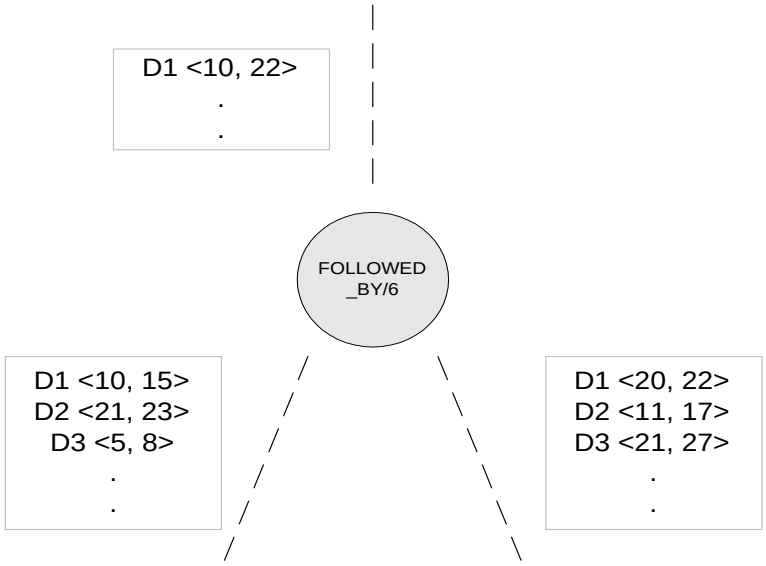

Figure 3.8 Example of the working of the FOLLOWED BY operator

from the left set can be *initiator*. The operator has to look for occurrences of tuples from the middle set occurring between a tuple from the left set and right set, all having the same document IDs.

The pseudocode is shown below. Lines 3 through 10 align the left, right and middle set pointers such that they point to tuples having the same document ID. It takes a left and right tuple from each document, and checks whether the left tuple occurs before the right tuple. If so, it checks if there are at least d complete, non-overlapping occurrences of tuples from the middle set occurring in between. If d middle occurrences are found, it combines the left and right tuple and adds it to the output set. The output tuple thus contains the range of occurrence of the pattern.

WITHIN(L, R, M)

1: $left \Leftarrow$ first tuple in L, $right \Leftarrow$ first tuple in R, $middle \Leftarrow$ first tuple in M

2: **while** $exists(left)$ AND $exists(right)$ AND $exists(middle)$ **do**

3: **while** $left.docID < middle.docID$ **do**

```
4:    left ⇐ left → next
5:  while left.docID < right.docID do
6:    left ⇐ left → next
7:  while left.docID > middle.docID do
8:    middle ⇐ middle → next
9:  while left.docID > right.docID do
10:   right ⇐ right → next
11: if overlap(left, right) then
12:   right ⇐ right → next
13:   continue
14: if left.endOffset < right.endOffset then
15:   if (left → next).endOffset < right.endOffset then
16:     left ⇐ left → next
17:   else
18:     count ⇐ 0
19:     while middle.docID == left.docID do
20:       if middle.liesBetween(left, right) then
21:         count++
22:         if count == d then
23:           combine(left, right)
24:           middle ⇐ middle → next
25:           break
26:         middle ⇐ middle → next
27:       if middle.liesAfter(right) then
28:         middle ⇐ middle → next
29:         break
```

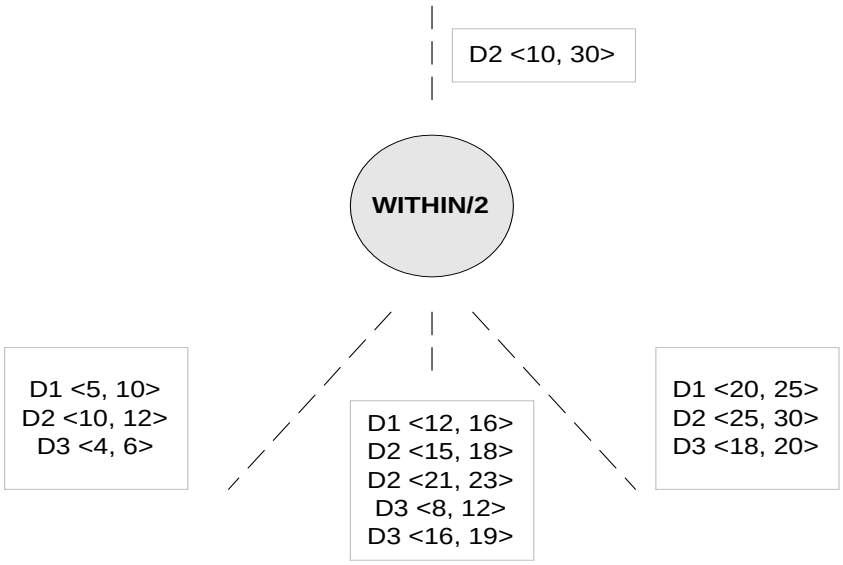

Figure 3.9 Example of the working of the WITHIN operator

30: **else**

31: $middle \Leftarrow middle \rightarrow next$

32: $left \Leftarrow left \rightarrow next,\ right \Leftarrow right \rightarrow next$

33: **else**

34: $right \Leftarrow right \rightarrow next$

An example of the working of the WITHIN operator can be seen in Figure 3.9.

3.4.6 The NOT operator

The *NOT* operator is used to specify a sequence of two patterns with the condition that a certain pattern does not occur between them. It is specified as *NOT[/d](P2)(P1, P3)*, which means that documents containing *P1* followed by *P3*, containing at most d occurrences of *P2* in between, should be selected. d is optional, and specifies the maximum number of occurrences of *P2* that can be allowed for NOT to be true. The system assumes a value of *zero* for d if it is not specified (default).

The NOT operator is a ternary operator: it receives as its inputs three sets corresponding to occurrences of the left pattern, the right pattern and the middle pattern. In addition, it also receives an integer denoting the minimum allowable occurrences of the middle pattern. The operator first tries to find closest, non-overlapping left-right pairs, and then counts the number of occurrences of the middle pattern in between the left-right pair. If there are no middle patterns in between, or if the number of middle patterns are less than the specified number, the left and right pattern are merged and added to the output set. If the number of middle occurrences is equal to or exceeds the specified number, the left and right patterns cannot be merged, and we move on to the next pair. In other words, the pattern is not detected. The pseudocode is shown below.

NOT(L**,** R**,** M**,** d**)**

1: $left \Leftarrow$ first tuple in L, $right \Leftarrow$ first tuple in R
2: **while** $exists(left)\ AND\ exists(right)$ **do**
3: advance $left$, $right$ in sort-merge fashion till they point to tuples with same docID
4: **if** $overlap(left, right)$ **then**
5: $right \Leftarrow right \rightarrow next,\ continue$
6: **if** $left.endOffset < right.endOffset$ **then**
7: **if** $(left \rightarrow next).endOffset < right.endOffset$ **then**
8: $left \Leftarrow left \rightarrow next$
9: **else**
10: $middle \Leftarrow$ first tuple in M
11: $noMiddleTuples = false$
12: **while** $middle.docID \neq left.docID$ **do**
13: **if** $middle.docID > left.docID$ **then**
14: $noMiddleTuples = true,\ break$

15: $middle \Leftarrow middle \rightarrow next$

16: **if** $noMiddleTuples$ **then**

17: $combine(left, right)$

18: $left \Leftarrow left \rightarrow next,\ right \Leftarrow right \rightarrow next,\ continue$

19: $occurrenceCnt = 0$

20: **while** $middle.docID == left.docID$ **do**

21: **if** $middle.liesBetween(left, right)$ **then**

22: $occurrenceCnt++$

23: **if** $(occurrenceCnt > d)\ OR\ (middle.liesAfter(right))$ **then**

24: $break$

25: $middle \Leftarrow middle \rightarrow next$

26: **if** $occurrenceCnt \leq d$ **then**

27: $combine(left, right)$

28: $left \Leftarrow left \rightarrow next,\ right \Leftarrow right \rightarrow next,\ continue$

29: **else**

30: $right \Leftarrow right \rightarrow next$

An example of the working of the NOT operator can be seen in Figure 3.10. It can be seen that for the containing tuples D1 <5, 10> and D1 <20, 25>, there are no tuples from D1 that lie in between. Hence, NOT becomes true for this pair, and a merged tuple D1 <5, 25> is added to the output. For the next pair of containing tuples, D2 <6, 8> and D2 <22, 28>, there is one tuple lying in between, and hence NOT again becomes true, because the maximum number of middle tuples allowed is 1. For the containing tuples D3 <8, 10> and D3 <22, 29 >, there are two tuples lying in between, and hence NOT evaluates to false, and the tuples are not merged. Finally, for D4 <6, 10> and D4 <20, 30>, there is only one tuple that lies in between (D4 <11, 13>). It should be noted

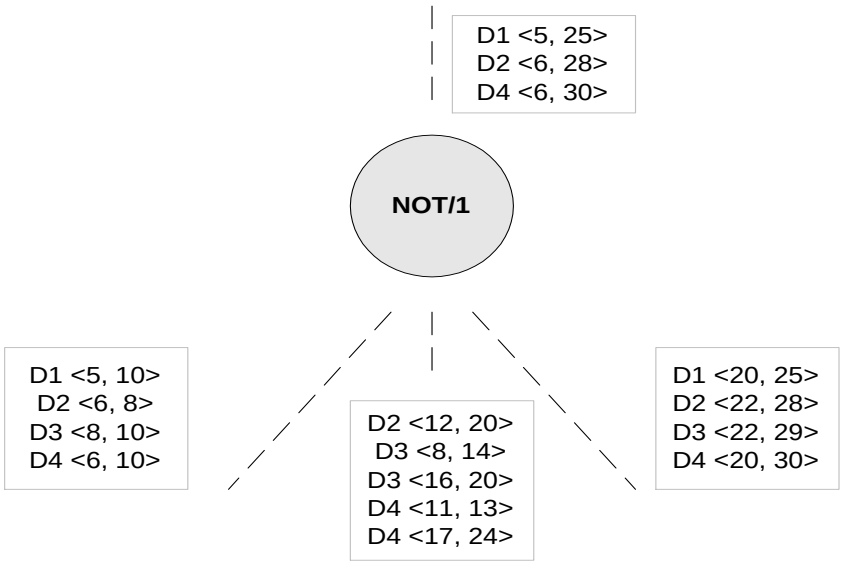

Figure 3.10 Example of the working of the NOT operator

that D4 <17, 24> is not considered to be lying in between, because it overlaps with the right containing tuple. Since there is only one tuple lying in between, a merged tuple D4 <6, 30> is added to the output set.

3.5 Summary

In this chapter, we explained how a basic inverted index is inadequate to detect complex patterns involving proximity, frequency, sequence, containment, etc., although it suffices to answer keyword queries. To support detection of such complex patterns, the index needs to store some additional information. Namely, it needs to store the position of occurrence of every word within a document. We described how the Proximal-Unique semantics corresponds to the intuitive detection of sub-patterns in a document that are typically combined to form a complex pattern. The proximal-unique semantics is based on closeness of the sub-patterns and avoids duplicate combinations. We introduced the Pattern Detection Graph, which represents a user specified pattern.

The operators OR, FREQUENCY, NEAR, FOLLOWED BY, WITHIN and NOT were described along with their algorithms. The InfoSearch operators are presently designed to support non-overlapping semantics, although another set of operators for overlapping semantics can be designed. These operators take sets of tuples as inputs from their children, where a tuple represents a single occurrence of the child pattern in the document collection. To enable efficient merging, these tuple sets have to be in sorted form before being fed to the operators. The operators merge the input sets according to the Proximal-Unique semantics, and generate a sorted set of tuples as their output.

CHAPTER 4

DESIGN OF INFOSEARCH

InfoSearch allows the user to specify complex queries and returns information about every occurrence of the pattern specified in the query. The scope of the search is a pre-indexed document collection (e.g., Web site), and the information returned is the document (e.g., Web page) in which the pattern occurs, and the position of every occurrence of the pattern within each document. This process can be divided into two steps. First, the user query must be parsed and validated to ensure that it conforms to the PSL syntax, following which a PDG corresponding to the query must be constructed. Second, the index must be looked up for detecting the occurrences of the pattern (including complex patterns) that must be detected over the PDG to generate the query result. This chapter reveals how the above steps are performed in InfoSearch, by describing its underlying architecture. We also describe how queries involving phrases are handled. In addition, we describe how queries that include synonym specification are processed.

4.1 InfoSearch architecture

The InfoSearch architecture is shown in Figure 4.1. InfoSearch has adopted some modules from the InfoFilter architecture, added a few modules specific to index-based retrieval and search, and modified the operators extensively to detect patterns over data retreived from an index. The modules of InfoSearch include the *pattern parser* and *validator*, *pattern processor*, *graph generator*, the *pattern detection engine*, and the *index interface*. External modules such as the WordNet synonym database and the integrated index are also shown in the figure. Of these modules, the *pattern parser* and *validator*

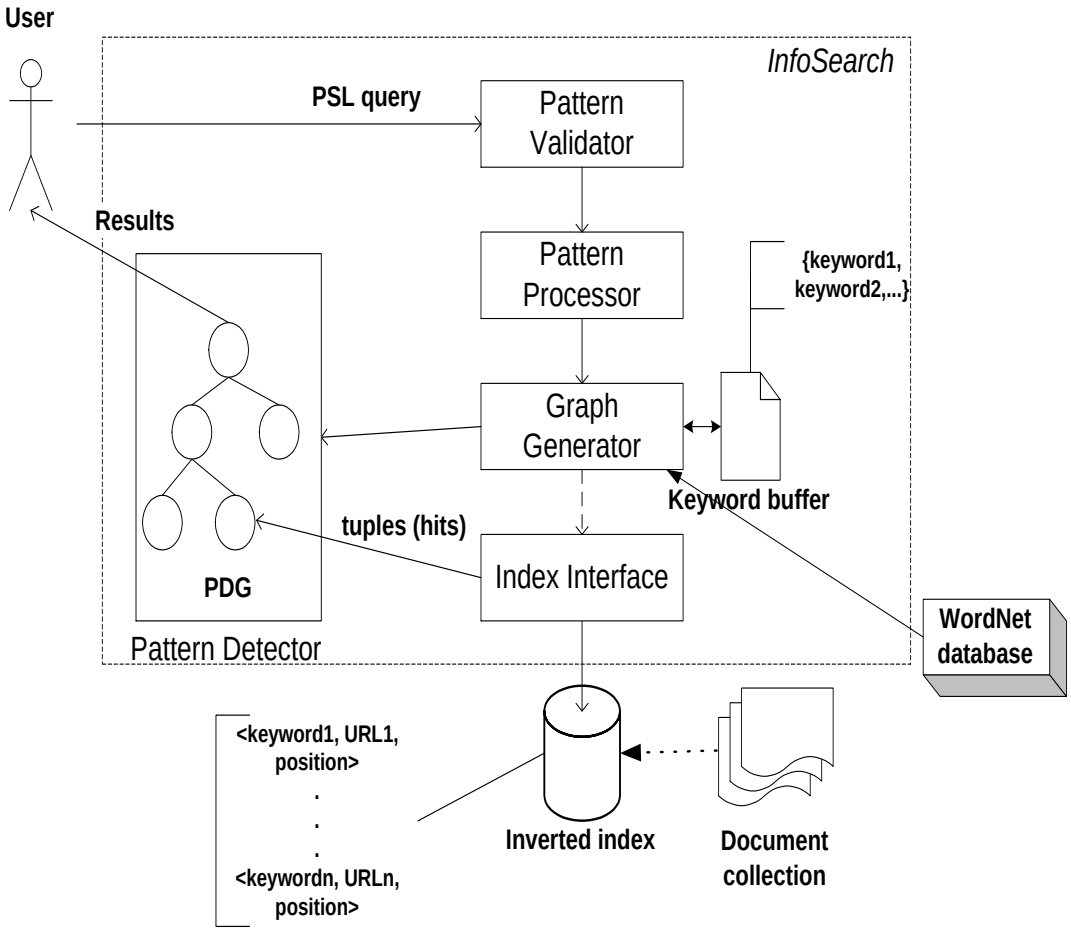

Figure 4.1 InfoSearch architecture

have been almost completely adopted from InfoFilter. The Graph Generator has been modified in order to store the keywords from the user query in a *keyword buffer*. The *pattern detection engine*, which forms the core of the system and includes the operator functionality has been changed to process inputs in the form of sets of tuples. In addition, a generic *index interface* specification and design has been added. These modules are described in more detail in the forthcoming sections.

InfoSearch uses the client-server architecture typically used in Search Engines. The user specifies one or more queries using PSL, which are submitted to the InfoSearch server.

The query is checked for syntax and parsed to extract tokens. The validator passes the extracted tokens to the *pattern processor*, which pushes the tokens on to a stack (Postfix notation) and passes the stack to the *graph generator*. The *graph generator* examines the tokens and generates a PDG. Thus, each user query is represented as a PDG. As mentioned before, 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. 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*. The *index interface* module is responsible for retrieving the "hits" for each keyword from the index. These hits are wrapped into a set of "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 InfoSearch operators such as NEAR, FOLLOWED BY, WITHIN, NOT, etc., merge their input sets according to the appropriate semantics using the algorithms presented in Chapter 3. Thus, a monotonically decreasing set of data propagates up the PDG, and the output of the root node is the answer to the query which is returned to the user.

Pattern Validator

This module takes a user query as its input, and checks if the query is in the proper syntax dictated by PSL. If there is an error in the syntax, a parser error is returned. After the query is validated for syntax, it is decomposed into tokens. The tokens in a query can be keywords, phrases, system defined patterns, operators and other delimiters allowed by the language. These tokens are sent to the *pattern processor*.

Pattern Processor

This module receives a set of tokens, in Infix notation, as its input. It converts the Infix input into a Postfix expression, so that it can be easier to evaluate. The Postfix expression is passed on to the *graph generator*.

4.1.1 Graph Generator

In InfoSearch, a user query is internally represented as a PDG. The task of the *graph generator* is to take a stack of tokens from the *pattern processor* and generate the PDG from it. The PDG is constructed recursively in a bottom-up fashion. The leaf nodes are created first, followed by the operators defined on these leaf nodes. When a parent node is created, it *subscribes* to its children. Essentially, this means that the reference of the parent is given to the children, so that data can be passed from a child to its parent. Thus, every node has a subscriber list that has a reference to each of its parents. As a simple example, the pattern *"protein" NEAR "clustering"* is shown in Figure 4.2. The leaf nodes in this example correspond to the keywords in the query, i.e., *"protein"* and *"clustering"*, and they have references to their parent NEAR node. The number in the subscriber list indicates the distance with which the parent has subscribed, and is described next.

For operators such as NEAR and FOLLOWED BY, the user is allowed to specify a distance, which indicates the maximum separation between the operands. This distance is also stored in the subscriber list. For example, consider the query *("protein" NEAR/3 "clustering") WITHIN("DNA", "modification")*. The PDG corresponding to this query is shown in Figure 4.3. Here, the WITHIN node subscribes to the NEAR node with a distance of 3. In other words, there is a distance associated with the reference to the

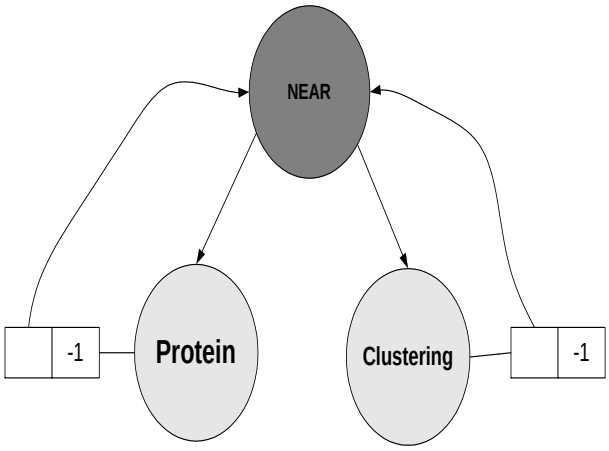

Figure 4.2 PDG for *"protein" NEAR "clustering"* with subscriber lists

parent. In this example, the NEAR node sends only those tuples to the WITHIN node that satisfy the criteria of the operands being separated by at most 3 words.

4.1.1.1 Sharing of PDG nodes

To optimize space requirements, the *graph generator* shares PDG nodes wherever possible. This is achieved by keeping a single, common PDG or sub-PDG for a common expression or sub-expression. This avoids creation of a new PDG, if a PDG has already been created for a previous expression or sub-expression. For example, consider that another query *"peptide" FOLLOWED BY ("protein" NEAR/5 "clustering")* is specified along with the query shown in Figure 4.3. The Graph Generator knows that a sub-PDG for the sub-expression *"protein" NEAR/3 "clustering"* already exists. Hence, instead of creating a new sub-PDG, it re-uses the sub-PDG, by having the new FOLLOWED BY node subscribe to it as shown in Figure 4.4. This results in reduction of memory requirements when several queries having common sub-expressions are processed together in a batch. More importantly, the sub-pattern is computed once for all the distances and

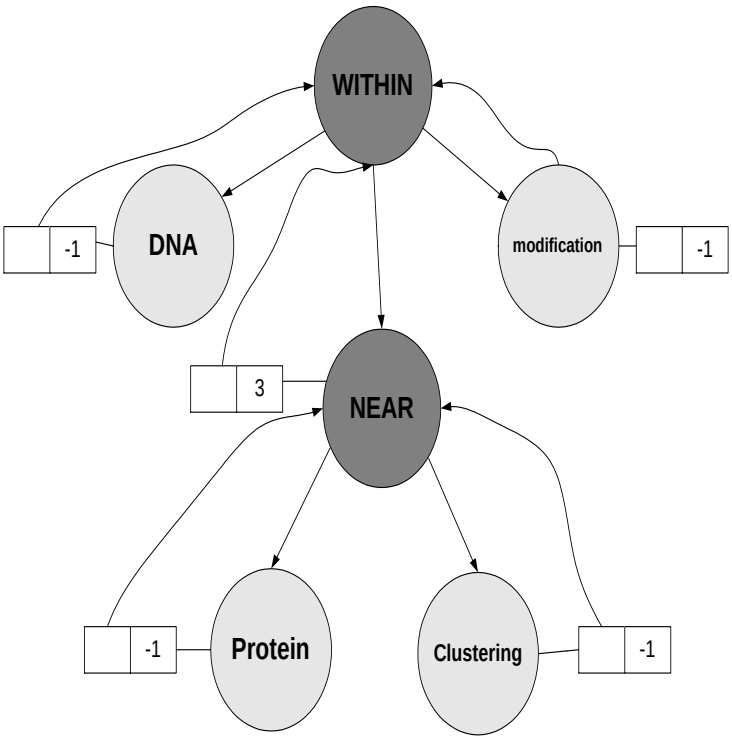

Figure 4.3 PDG with subscriber list containing distance

corresponding output generated. Note that the subscriber list for the NEAR node has been appended with the new distance for the FOLLOWED BY node.

During the Pattern Detection Graph generation phase, all the simple patterns (or leaf node patterns) are stored in a buffer. This buffer is used to feed the *index interface* for retrieving the appropriate "hits" from the index for detecting the pattern.

4.1.2 Index Interface

The detection engine of InfoFilter is designed to be generic and capable of working with any kind of index. The *index interface* is the agent that cleanly separates the InfoSearch system and the index which is being used. It is the only module which is specific to the index being used. In other words, the *index interface* is the only module that needs to be changed when InfoSearch needs to be integrated with a different kind of

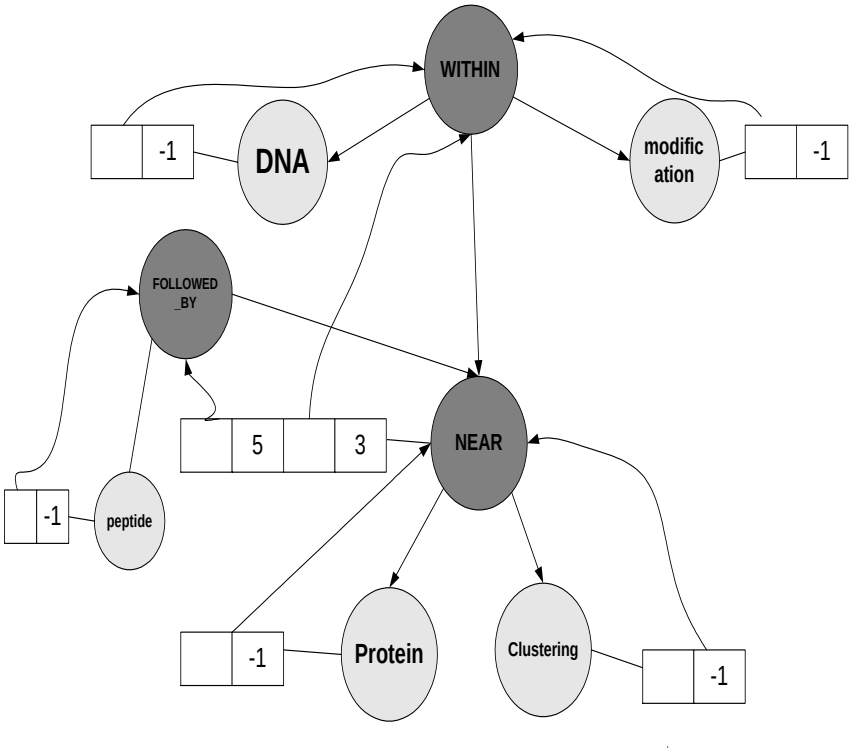

Figure 4.4 PDG with a shared node

index. The *index interface* accepts simple patterns from the *graph generator* and queries the inverted index. It is responsible for wrapping the result returned by the index in a set of <docID, start offset, end offset> tuples. The InfoSearch operators need their input in the form of such tuples, and hence wrapping the output of the index into tuple sets is an important function of the *index interface*. The set of tuples generated as the index lookup for a keyword is then passed to the leaf node in the PDG that corresponds to that keyword. Essentially, the *index interface* needs to have a capability to scan the index for the required keywords, and subsequently wrap the index-specific results into a form required by the InfoSearch system, namely <docID, start offset, end offset> tuples. This is summarized in Figure 4.5.

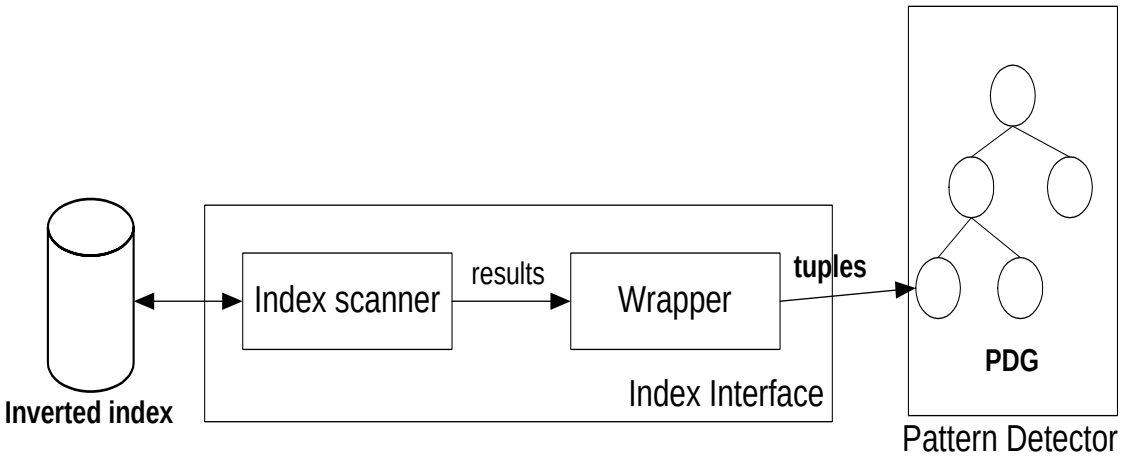

Figure 4.5 Data flow within the Index Interface

4.1.3 Pattern Detector

The flow of data up a PDG and the merging of tuples by the operators to detect patterns in documents has a similarity with composite event detection using Event Detection Graphs [15]. In the latter, event occurrences are propagated up an Event Detection Graph, in which the composite nodes merge their inputs based on criteria known as parameter contexts. A node in the graph can be associated with a rule, which means that a predefined action can be taken when that event node is triggered. Since there is similarity in the data flow in the Event Detection System and InfoSearch, we use the framework of such an Event Detection Engine, called the Local Event Detector (LED) [16] as the backbone for our Pattern Detector. The paradigm of data flow has been kept, with operators and semantics replaced to suit the domain of information searching and retrieval.

When a leaf node in the pattern detector receives a set of tuples from the Index Interface, it passes a reference to this set to its parent nodes. Similarly, when internal nodes merge their input sets to create a merged output set, they pass a reference to this set to their parents. The root node has a rule associated with it, which essentially

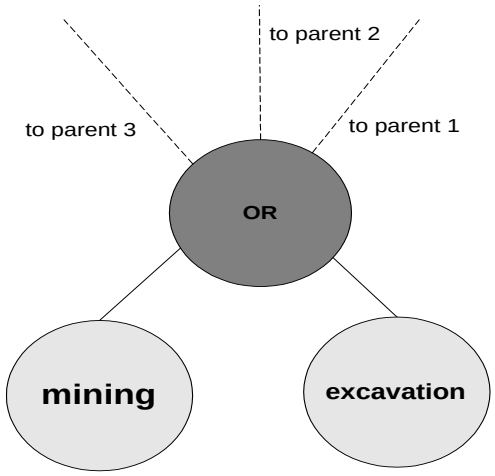

Figure 4.6 Propagation of output to multiple parents by an OR node

directs the output of the pattern detection to the user through e-mail or other forms of notification.

4.1.3.1 Propagation to multiple parents

In the case of leaf nodes, or operators which do not have any distance specification such as OR, the reference to their output set is passed to all their parents. This is illustrated in Figure 4.6. Here, the OR node has three subscribers, and it sends a reference to its output set to all three.

However, when a node having a specified distance such as NEAR or FOLLOWED BY has multiple parents, the results required by each parent may be different, because each parent may have subscribed with a different distance. In this case, the operator generates a different output set for each group of parents who have subscribed with a particular distance. Consider the NEAR node in Figure 4.4, which has two subscribers, one with distance 3 and the other with distance 5. Intuitively, a pattern satisfying the smaller distance also satisfies all distances greater than it. For instance, if two tuples are found to be 2 words apart from each other, they satisfy the distance of 3, and certainly

satisfy the distance of 5 as well. However, those patterns that satisfy the distance of 5 may not satisfy the distance of 3. The larger distance set subsumes the smaller distance set. While the operator is merging its input tuples, it checks the distance between tuples. If it is found to satisfy a particular distance, the output tuple is put in all the output sets which correspond to distances equal to or greater than the satisfied distance. These distances are stored in a non-decreasing order in the subscriber list, making it convenient to output a tuple to all sets corresponding to distances more than a certain distance. After the processing is complete, references to the output sets are passed to the respective subscribers.

4.1.4 Handling phrases as simple patterns

InfoSearch allows the user to specify queries having phrases. Since it is assumed that the index only stores information on words, InfoSearch has to use the information on the words in the phrase to detect whether the complete phrase occurs in the document collection. When a query involving a phrase is given, the *graph generator* receives the entire phrase as a token from the *pattern processor*, along with other tokens such as operators and keywords. The *graph generator* stores the phrases in the *keyword buffer*.

While processing entries from the keyword buffer, the *graph generator* sends simple keywords directly to the *index interface* for lookup, whereas phrases are sent to a separate sub-module called the *phrase processor*.

The *phrase processor* further decomposes the phrase into its constituent keywords, and queries the *index interface* to obtain a set of tuples for each constituent keyword from the phrase. It iterates over these sets and merges only those tuples which conform to the order of keywords in the phrase. In other words, the *phrase processor* generates an intersection of these sets, with the merging criteria being that tuples should belong to the same document ID, and the occurrences should be in the order dictated by the

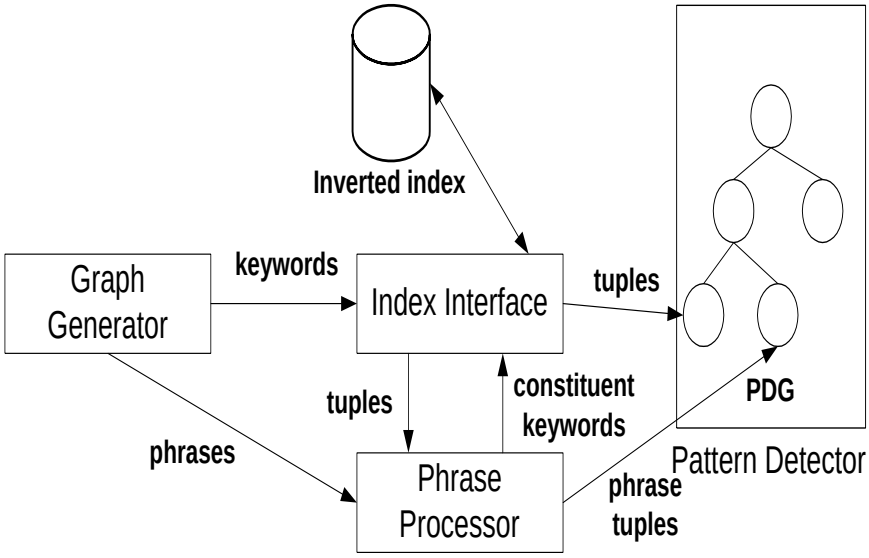

Figure 4.7 Phrase handling by the Phrase Processor

phrase. The high level sequence of operations for generating a set corresponding to the phrase occurrences is shown in Algorithm 1.

In the algorithm, we find the smallest set, and for each tuple in this set, we iterate over all the other sets and check if the tuples are in the sequence dictated by the phrase. If all tuples are from the same document ID and found to be adjacent as required, a tuple is added to the output set with start offset equal to the start offset of the first keyword, and end offset equal to the end offset of the last keyword in the phrase.

After the *phrase processor* generates the set of tuples corresponding to the phrase occurrences, it passes a reference to this set to the leaf node in the PDG corresponding to the phrase. Figure 4.7 shows how the *graph generator* sends simple keywords to the *index interface* and groups of keywords in a phrase to the *phrase processor*, and the *phrase processor*'s interaction with the *index interface* and the PDG.

Algorithm 1 generatePhraseTupleSet

1: **Input**: n sets corresponding to the n keywords in the phrase
2: 1^{st} set corresponds to 1^{st} keyword in the phrase, 2^{nd} set to 2^{nd} keyword and so on
3: **Output**: a set P containing tuples corresponding to the phrase occurrences
4: Find the smallest of the n sets, say S. Assume this is the i^{th} set, where $0 \leq i < n$
5: **for** each set T from the remaining $n - 1$ sets **do**
6: $currentTuple(T) \Leftarrow$ first Tuple in T
7: **for** each tuple s in S **do**
8: $setCnt \Leftarrow 0$
9: **for** each set T from the remaining $n - 1$ sets **do**
10: Let T be the j^{th} set, where $0 \leq j < n$, $j \neq i$
11: **while** $currentTuple(T).docID < s.docID$ **do**
12: $currentTuple(T) \Leftarrow$ next Tuple in T
13: **while** $currentTuple(T).docID == s.docID$ AND $currentTuple(T).startOffset \leq s.startOffset + (j - i)$ **do**
14: **if** $s.startOffset - currentTuple(T).startOffset == i - j$ **then**
15: $setCnt++$
16: $break$
17: $currentTuple \Leftarrow$ next Tuple in T
18: **if** $setCnt == n - 1$ **then**
19: $P \Leftarrow P +$ new tuple$(s.docID,\ s.startOffset - i,\ s.startOffset + n - i)$

4.1.5 Processing queries involving synonyms

InfoSearch provides an option for users to expand their search for a particular keyword by looking up occurrences of synonyms of that keyword in addition to the keyword itself. This can be specified in the query by appending "[Syn]" to the desired keyword. For example, the query *"Cancer" NEAR/3 "medicine"[Syn]* means "Find all occurrences of *"Cancer"* near *"medicine"* or any of its synonyms, not separated by more than 3 words".

As indicated before, InfoSearch uses the WordNet synonym database to look up synonyms for a given word. The number of synonyms to be considered is a configurable parameter in InfoSearch. Since the user is querying for occurrences of the given keyword *or* any of its synonyms, we need to query the index for the synonyms as well as the original keyword. Also, it is required that the final output be sorted by document ID, because it may be the input to another operator. In order to perform this task, InfoSearch has a special operator to process synonyms, called the SYN operator.

Unlike the other operators which have one, two or three sets as their input, the SYN operator can have n sets as its input. Leaf nodes are created for the keyword and its synonyms, and a SYN node is created which subscribes to these leaf nodes. For example, the query *"Cancer" NEAR "medicine"[Syn]* is shown in Figure 4.8. The SYN operator generates a union of its input sets, sorted by document ID and position. The output of the SYN operator is a set corresponding to all occurrences of the keyword or any of its synonyms in the document collection.

The algorithm of the SYN operator is shown in Algorithm 2. Initially, pointers are set to point to the first tuple from each set. The tuple having the smallest docID and end offset is added to the output set, and the pointer for that set is advanced. The process is repeated until there are no more unprocessed tuples in any of the input sets. Thus,

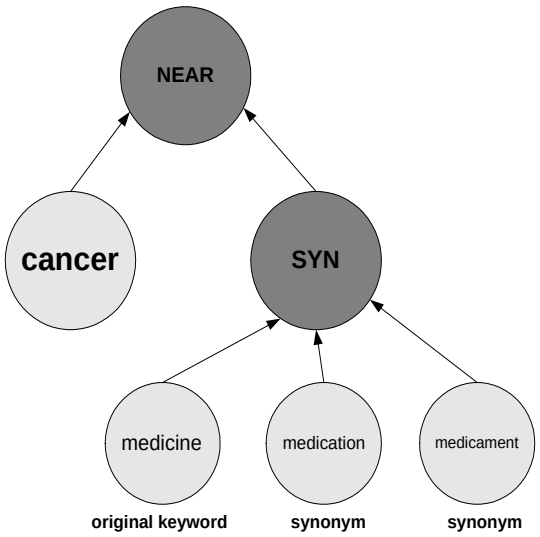

Figure 4.8 Synonym operator with 3 children

the number of tuples in the output set is the sum of the number of tuples in the n input sets.

4.2 Summary

In this chapter, we described the architecture of the InfoFilter system and explained the details of its modules. Some modules such as the *pattern validator* and *pattern processor* were re-used from the predecessor system. The process of PDG construction by the *graph generator* was explained with details on how the distance specification is incorporated, and also how common sub-patterns are grouped together in a single PDG. This grouping results in a space saving, and also reduces the number of computations by propagating output tuples in a batch, based on a distance group. The *index interface*, which is a link between InfoSearch and the inverted index being used was described. The *pattern detection engine* receives sets of tuples from the *index interface*, which get progressively filtered as they propagate up the PDG. An algorithm for phrase detection which performs a constrained intersection of tuple sets corresponding to the constituent

Algorithm 2 Syn

```
1: Input: n sets corresponding to base keyword and n − 1 synonyms
2: Required output: A sorted union of these n sets
3: for each set S do
4:    currentTuple(S) ⇐ first Tuple in S
5: minTuple ⇐ currentTuple(0)
6: min ⇐ 0
7: while exists(minTuple) do
8:    for (i = 0 ; i < n, i ≠ min ; i++) do
9:       if minTuple.docID < currentTuple(i).docID then
10:         continue
11:      if minTuple.docID == currentTuple(i).docID then
12:         if minTuple.endOffset < currentTuple(i).endOffset then
13:            continue
14:         else
15:            minTuple ⇐ currentTuple(i)
16:            min ⇐ i
17:      if minTuple.docID > currentTuple(i).docID then
18:         minTuple ⇐ currentTuple(i)
19:         min ⇐ i
20:   resultSet ⇐ resultSet + minTuple
21:   if set min has more tuples then
22:      minTuple ⇐ next Tuple in min
23:   else
24:      arbitrarily assign current tuple of some unexhausted set to minTuple
25:      assign that set number to min
```

words was explained. To detect patterns specifying synonyms, a special operator called SYN was described, which generates a sorted union of its input sets.

CHAPTER 5

IMPLEMENTATION OF INFOSEARCH

This chapter elaborates on the implementation aspects of the InfoSearch modules. The *pattern validator* and *pattern processor* modules have been almost entirely adopted from the previous work, and hence will not be described in much detail. The focus will be on the details of implementation of the *graph generator*, *index interface* and the *pattern detection engine*. The data structure used to pass information up the Pattern Detection Graphs (PDGs) and the operations on it will be described in detail. In addition, we will also look at the choice of inverted index for the first cut of this system, reasons for that choice and how the chosen index is integrated with the InfoSearch system. We will describe how documents are indexed and the mechanism used to retrieve information from the index.

Since there are a number of common modules in InfoSearch and InfoFilter, the implementation of InfoSearch has been integrated with InfoFilter. Modules such as the *pattern input client*, *pattern validator and processor* are almost exactly identical for both systems. In other modules such as the *graph generator* and *pattern detection engine* which have some parts specific to InfoFilter and some to InfoSearch, a configuration parameter is used which determines whether the system is being run in "filter" mode or "search" mode. This parameter can be set in the system configuration file, which is loaded at system start-up (refer Appendix A).

5.1 Pattern Parser, Validator, and Processor

The *pattern parser* is implemented using a JavaCC parser [17]. It generates the appropriate tokens from the input query as per the specifications of the Pattern Specification Language (PSL). The tokens include simple patterns (words, phrases), operators, and other options allowed by the language. If the input query does not conform to the PSL syntax, a Java Exception called *ParseException* is thrown. The *pattern validator* enqueues the tokens in Infix notation and passes this queue to the *pattern processor*.

The *pattern processor* takes the queue of tokens and converts it into Postfix notation preserving the precedence of operators specified by the user. The Postfix notation allows for easier processing of operands. The *pattern processor* sends this stack to the *graph generator*.

5.2 Graph Generator

As mentioned in Chapter 4, the framework of an Event Specification and Detection system called the Local Event Detector (LED) has been used in InfoSearch. The *graph generator* uses the Event Specification API of the LED [16] to generate the PDGs. This API allows creation of primitive events, which correspond to leaf nodes in a PDG. It also allows creation of composite events, which correspond to parent nodes in a PDG. The *graph generator* from InfoFilter has been used here with minor extensions. The *graph generator* pops a token from its input stack, and depending on the type of token, calls the LED API to create that particular node. If the token is a keyword, phrase or system defined pattern, a leaf node is created for that token. This node is named after the keyword, phrase or system defined pattern that it corresponds to. If the token is an operator, the *graph generator* creates a composite node for this operator, which subscribes to the leaf nodes which are the children of this operator. This node is also

named uniquely. The operator type and the name of the children are used to compose the name for the node. The names of these nodes are stored in a hashtable, with a reference to the node.

Before creating a primitive or composite node, the *graph generator* checks the hashtable to see whether a node with the given name already exists. If the node does not exist in the hashtable, a new node is created and an entry is made into the hashtable. If the node already exists, the reference to the node is obtained from the hashtable and used to represent the PDG or sub-PDG. This allows for sharing of nodes as described in Chapter 4. Thus, common sub-expressions in a given group of queries correspond to a single PDG or sub-PDG in the system.

The main extension to the *graph generator* for this thesis is the addition of a buffer to store keywords. 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) [18]. The number of synonyms to be considered can be configured in InfoSearch (using a parameter in the system

configuration file). This is done to limit the number of synonyms to be processed[1]. 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.

Every PDG constructed in the system is encapsulated under a *WITHIN(BeginIndex, EndIndex)* construct introduced by the *graph generator*. This is shown in Figure 5.1. Here, *BeginIndex* and *EndIndex* are system defined patterns, and are internally triggered when an index scan begins. This encapsulation of a PDG is done to identify the scope of the pattern detection with respect to which index the detection originated from. Although the current implementation uses only one index, it can be imagined that several indexes can be integrated into the system. In such a scenario, it might be useful to know which index is providing the information to the Pattern Detector. This encapsulation is done after the PDG corresponding to the query is constructed. The top WITHIN node also handles the notification of the result set to the user. When it receives the output of the pattern detection from the root of the PDG corresponding to the query, which is its middle child, the rule associated with the WITHIN node sends this result to the user who has issued the query. Note that rules are associated with the root node of the PDG. Multiple rules can be associated if independent actions need to be taken.

5.3 Index Interface

The *index interface* is the bridge between the (inverted) index and the InfoSearch system. This module is specific to the index being integrated with the system. From the point of view of the rest of the InfoSearch system, it can be seen as an *index driver*. The *index interface* has to provide standard methods to access data from the integrated index, and deliver the results to the *pattern detector* is a specific format. As such, it does

[1]Currently, only single word synonyms are processed

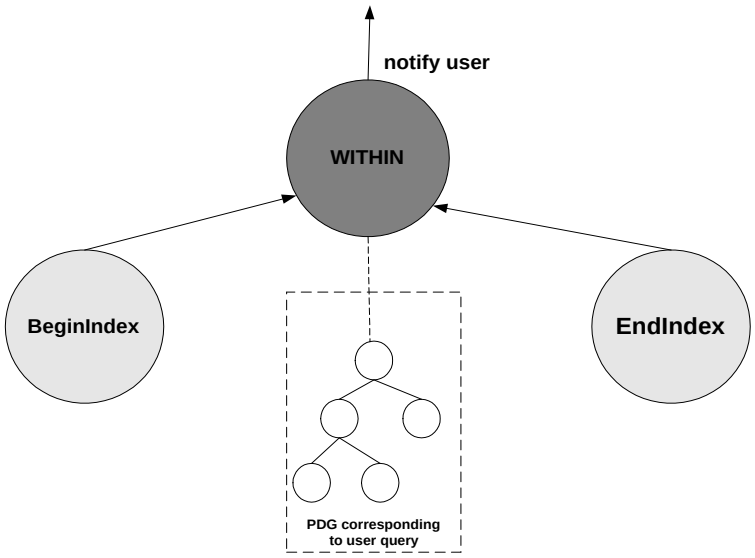

Figure 5.1 Encapsulation of query PDG by a top level WITHIN node

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 following standard methods are essential in the *index interface*:

initialize. This can be a part of the *index interface* constructor, or a separate method. This method initializes the inverted index, i.e., essentially brings it "online" with the configured parameters. InfoSearch calls this method at start up, and passes it a reference to the *pattern detector*. In terms of the LED API, this is a reference to the *ECAAgent*. In the LED API, *ECAAgent* encapsulates the event specification and primitive event detection methods [16]. A reference to the *pattern detector* is needed by the *index interface* to send data to leaf nodes of PDGs in response to keyword queries.

query. This routine accepts a keyword, and scans the index for the keyword. The data returned by the index has to be wrapped in a *Vector* of <*docID, start offset, end offset*> tuples. This is the standard data format expected by the PDG nodes. Hence, this routine may need to convert the native format of data returned by the index into the standard tuple format. This routine creates this *Vector* of tuples, and passes a reference to this object to the leaf node corresponding to the keyword in the *pattern detector*. This is done by calling the *raiseBeginEvent* method in the given *ECAAgent*, which is used to indicate detection of a primitive event in the LED [16]. A reference to the *Vector* of tuples is passed as a parameter in the *raiseBeginEvent* call. This call results in the primitive (leaf) node, which corresponds to the occurrence of the keyword, passing this reference to its parent node. It is important to note that *references* to result sets from the index are propagated up the PDG, and not the result sets themselves. This is very important from the view of efficiency, because passing entire result sets would be very inefficient as the size of these result sets increases.

getTuples. A method is also needed which can take in a keyword, generate the tuple *Vector* and return this object to the caller. This method does not trigger the PDG nodes itself but just returns the *Vector* of tuples to the calling routine. This method is useful for modules like the *phrase processor*, which needs the tuple sets corresponding to the constituent keywords of a phrase, subsequently merges these sets and then triggers the leaf node in the PDG corresponding to the phrase.

5.4 Tuple operations

Because the tuple is an important data structure used to represent information about a pattern occurrence, a Java class called *Tuple* has been defined, which encapsulates the tuple properties and methods. A *Tuple* has a numeric *docID*, which uniquely identifies a document in the collection. It has a numeric *startOffset* and *endOffset*, which represent

that starting position and ending position of the occurrence that the *Tuple* represents. Obviously, for a tuple representing a keyword occurrence, *startOffset* and *endOffset* will be the same.

Some commonly performed operations and checks on tuples have been provided as methods in the *Tuple* class. These methods are:

endsBefore: It is very often required to know if a *Tuple* occurs before another *Tuple*. Essentially, this method returns an appropriate value based on whether the *Tuple* has a smaller, same or greater offset than another *Tuple*.

overlaps: A key consideration in merging tuples in operators is that the tuples should not be overlapping. If the *Tuple* has a *startOffset* greater than the *endOffset* of the other *Tuple*, or an *endOffset* smaller than *startOffset* of the other *Tuple*, the two tuples do *not* overlap. This method uses this criteria to check for overlap between two *Tuple*s and returns an appropriate Boolean value.

liesBetween: For containment operators such as WITHIN and NOT, it becomes necessary to know if a pattern occurrence lies between two other patterns completely, that is, without any overlap. A *Tuple* lies between two other *Tuple*s (termed first and second) if *startOffset* of the *Tuple* is greater than *endOffset* of the first *Tuple*, and *endOffset* of the *Tuple* is smaller that *startOffset* of the second *Tuple*. This assumes that the tuples between which we are checking for containment are in the required order. *liesBetween* returns a Boolean value based on its decision.

liesAfter: This method checks if the *Tuple* lies completely after a given *Tuple*. If *startOffset* of the *Tuple* is greater than *endOffset* of the given *Tuple*, it means that the *Tuple* lies after the given *Tuple*.

5.5 Pattern Detection Engine

The *pattern detection engine* is responsible for processing the result sets from the index. This is done over PDGs: every internal node of a PDG is an operator, which encapsulates the logic to process its input according to the operator semantics. Leaf level nodes represent simple patterns such as keywords, phrases and system-defined patterns. The *index interface* passes a reference to a *Vector* of *Tuple*s corresponding to a keyword to the leaf node corresponding to the keyword. Similarly, leaf nodes corresponding to phrases are triggered by the *phrase processor*, and leaf nodes corresponding to system defined patterns are internally triggered by the system. Leaf nodes do not have any storage or processing capability; they simply propagate their input, which is a reference to a *Vector* of *Tuple*s to the parent nodes. By passing just references, we avoid entire result sets being propagated at every level of the PDG.

Internal nodes of the PDG correspond to one of the InfoSearch operators such as OR, NEAR, FOLLOWED BY, WITHIN, NOT, FREQUENCY and SYN. They get references to one or more *Vector*s from their children. They operate on these *Vector*s and merge them to produce an output *Vector*. This merging is done as per the operator semantics described in Chapter 3. A reference to the output *Vector* is passed to the parent node. In this manner, conceptually, a non-increasing set of data propagates up the PDG, getting progressively filtered at each level. The root node has a rule associated with it, which is to return the results to the user.

5.6 The Inverted Index

The focus of this thesis was on how to use information stored in an index to detect complex patterns over a document collection. We did not place too much emphasis on the actual index being used, since the system was designed to be capable of working with any

kind of index. For the first release of this system, it was necessary that the inverted index to be used should be easy to use and integrate, and yet provide the required information needed by InfoSearch. Hence, we built a simple inverted index built using the Berkeley DB Java Edition [19], 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.

5.6.1 Overview of Berkeley DB Java Edition

Berkeley DB Java Edition is an in-process storage manager written in Java. This means that unlike the commonly known database servers which run as a separate process, a Berkeley DB database runs in the same process as the application which uses it. All interaction with the storage manager is done using the API provided by BerkeleyDB. It supports B-trees and transactions. It is a replacement for a traditional database management system, when an efficient in-process solution is required.

5.6.2 Creating the inverted index

To create the inverted index, we 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. The index initially appears as a single file in a pre-specified directory, and can span multiple files as its size increases.

To index a given collection of documents, the documents must be placed in a single directory, and the path of the documents (essentially the path of the containing directory) must be passed to *DocumentIndexer*. This will result in the documents being

indexed, and the index file will be placed in the directory specified in the Berkeley DB configuration file (refer Appendix A).

The index created above stores the entire path of a document in each hit. If we use the index in this form, a tuple would consist of the document path along with the position information. The path of a document can be fairly large (e.g., a Web page reference), and hence the size of a set of such tuples would be correspondingly large. Also, at the PDG operator nodes, comparisons are done between tuples to check if they belong to the same document. If this comparison has to be done between document paths, which are essentially Java *String*s, they will be computationally very expensive. Comparisons between numbers are always computationally cheaper than lexicographic comparison of strings. Hence, we need to have a numeric document ID for each document to reduce the size of a tuple, and also to enable fast comparison. We also need to maintain a mapping from a document ID to the document it corresponds to.

To achieve this, as the inverted index is being constructed, a parallel index (using Berkeley DB itself) is built, which stores a unique document ID for each document. This is stored as a separate database in Berkeley DB. We call this parallel index the *documentID map*, because it stores a mapping from DocumentID to the document path. In the actual inverted index, we now store just the document ID along with the position information, for each keyword. Thus, only the numeric document ID information is incorporated in tuples which are propagated up a PDG. After the result is obtained from the root node of the PDG, the *documentID Map* is looked up to get the document path from the document ID, and this path along with the position information, is returned to the user.

BerkeleyDB has the capability to store the hits internally sorted by document ID and position. Hence, the tuples generated from the index results are automatically sorted by documentID and position. This property of the tuples being sorted is very useful while

merging tuples in the operators. The operators have to ensure that their output is also sorted, in order for the higher level nodes in the PDG to function properly.

5.6.3 Retrieving information from the inverted index

The interaction of InfoSearch with the inverted index is through the *index interface*. The *index interface* is invoked at server start-up to bring the inverted index "online". This essentially entails reading of the configuration file to look up the location of the index file, loading the necessary environment, etc. The index is brought online in read-only mode, which means that the transactional mode of the index is turned off. This is done to make the index performance as high as possible.

When the *index interface* receives a keyword to query the index with, it opens a *Cursor* on the index. The *Cursor* is part of the API provided by Berkeley DB JE. Using the *Cursor*, we get a handle on the result set from the index, which is zero or more "hits". We iterate through the result set using the *Cursor*, convert each hit retrieved into a *Tuple*, and keep adding *Tuple*s to a *Vector*. When the *Cursor* is done iterating over the result set, the *Vector* of *Tuple*s is returned.

5.7 Summary

In this chapter, we described the implementation aspects of the various InfoSearch modules. The *pattern validator* and *pattern processor* were adapted from the predecessor system, InfoFilter, with almost no changes. The *graph generator* was modified to incorporate a *keyword buffer* to store words and phrases from the user specified pattern before they can be retrieved from the index. An *index interface* module, its method specifications and an implementation for an inverted index using Berkeley DB were developed. Essentially, a different *index interface* is required for each type of index being integrated, but the methods supported by each one are the same to ensure a uniform interface to

InfoSearch. The first inverted index chosen to be integrated was an inverted index using Berkeley DB, owing to its simplicity of use. The said index was successfully integrated with InfoSearch.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this thesis, we discussed InfoSearch, a system designed to search complex patterns over a document collection. It was observed that current search systems are somewhat restrictive in the expressiveness of patterns that can be specified by the user. Such expressive patterns may need to be specified in certain domains such as searching full-text patent databases, analysis of logs, intelligence and tracking applications, etc. InfoSearch is an attempt to facilitate searching of complex patterns involving proximity, frequency, containment and sequences over a given document collection.

One alternative for detecting complex patterns over stored documents is to read the entire data source, tokenize it, and feed it into the Pattern Detection Graph to detect the pattern. This has been explored earlier in [3]. Although this is useful in certain domains, such as news feeds and other applications where data is generated inherently as a stream, there are other domains where artificial streaming is inherently inefficient, especially when the source does not change frequently or accuracy can be traded off for response time. For such relatively static documents, it is much more efficient to use an index over the document collection (as has been demonstrated by IR and Web Search Engines) to search for patterns.

The question that we raised in the beginning as to what kind of information is needed as part of the index and what information would be sufficient to detect complex patterns specified in PSL so as to obtain the same result as stream processing, has been answered in the affirmative in this thesis. We have shown that the offset information is

sufficient for all the operators of PSL to detect patterns and that the Proximal-Unique context can be applied in conjunction with the operator semantics (which are independent of whether we process a stream or a static data set).

Complete algorithms were developed for complex operators including OR, NEAR, FREQUENCY, FOLLOWED BY, NOT, WITHIN and also a SYN operator for processing synonyms. Phrases can be handled as well. These operators work on sets of "tuples" of pattern occurrences. Appropriate semantics for pattern detection were reviewed to ensure that the pattern detection made intuitive sense. The Proximal-Unique context of pattern detection, used by all InfoSearch operators, ensures that duplicate pattern occurrences are not reported, and only the closest constituents of a pattern are combined. It was found that it is indeed possible to detect complex patterns using indexed information, if the index provides the document ID and position of every word occurring in the document collection.

This work extends the previous work and has modified a few modules and added several modules that are specific to index-based pattern detection. Various modules of the system including the *pattern detector* and *index interface* were developed as part of this thesis using the Java platform, and were integrated with the *pattern parser*, *pattern processor* and *graph generator* (with minor modifications), of the predecessor system, InfoFilter. The WordNet synonyms database was used for handling the synonym option. Since the focus of the research was on the process and algorithms for pattern detection from indexed information, the actual index chosen for the first cut was a simple inverted index. The objective was to choose an index which would be easy to use and integrate, but still store and provide the desired information. An *index interface* was developed for the index, and the index was successfully integrated with the rest of the system.

6.2 Future Work

There are two areas where a search system can always be improved upon: quality of the results, and performance. Presently, InfoSearch gives the result of the query in sorted order of document ID. A ranking strategy can be developed, which delivers the results in descending order of how "closely" the results match the specified query. The criteria and quantification of "relevance" of a result to a query can be researched upon. Also, partial matches to a query can be reported.

To improve performance, strategies to selectively cache the result sets can be developed. The operator algorithms need to be more rigorously analyzed to assess their complexity, and improved, if possible, for efficiency. Also, presently the propagation of result sets up the Pattern Detection Graphs is through main memory data structures. Using file based propagation of result sets can certainly increase the scalability of the system. The merging of input sets at the operator nodes, which is presently being done sequentially, can be sped up using parallel processing.

APPENDIX A

SYSTEM CONFIGURATION PARAMETERS

In this appendix, we describe some of the important system configuration parameters used in the integrated InfoFilter / InfoSearch system. These parameters are specified in a file called *System.config*, which resides in the same directory from which the InfoFilter / InfoSearch server is being launched.

We also describe some configuration parameters for Berkeley DB.

System.config parameters:

- *PORT:* This parameter specifies the port number on which the InfoFilter / InfoSearch server will listen. If unspecified, the system listens on port 7000 (default).
- *SYN:* This parameter indicates the maximum number of synonyms that will be processed by the system. The *graph generator* uses this value while creating leaf nodes corresponding to the synonyms, for the SYN operator.
- *MODE:* This parameter decides whether the system is being run as InfoFilter, or InfoSearch. It can be set to either FILTER or SEARCH. If this parameter is not specified, InfoFilter is launched by default.

There are some other configuration parameters in *System.config*, which are self explanatory.

berkeley_db.config parameters:

This file also resides in the same folder from which the server is launched.

- *DATABASE_DIR:* This parameter specifies the folder in which the database files for BerkeleyDB reside. In InfoSearch, these database files contain the inverted index and the *documentID Map*.
- *FILE_MODE:* This parameter is used to launch the *index interface* in debug mode. In this mode, the *index interface* does not query BerkeleyDB, but instead gets input from a dummy input, a text file called *pseudo.idx*. This file resides in the same

folder from which the server is launched, and contains keyword - documentID and position mappings. The syntax for entering mappings in *pseudo.idx* is as follows: Each line in the file represents "hits" for a keyword. Each line has a keyword followed by a tab, followed by a documentID, followed by the positions of occurrence of the keyword in that document, separated by commas. Any number of such document IDs can be entered on a line. Any number of lines can be present in the file.

APPENDIX B

EXPERIMENTAL RESULTS

The primary reason for developing operators to detect complex patterns over indexed data was that it was inefficient to detect the same patterns by streaming in the same data instead of using an index. Certainly, after the size of the document set increases beyond a certain point, we would natually expect the indexed approach to perform better. The crossover point (at what point, in terms of number of words in the document collection, would the indexed approach perform better than the streaming approach?) was the first parameter of interest. An initial experiment was performed with a very small number of documents. This was like a sanity check, to prove that the indexed approach outperforms the streaming approach for even small document collections.

A set of 20 documents of around 1.5 kB each were selected from the Reuters dataset. The total number of words in this collection was around 2600. The documents were artificially converted into a stream, fed to InfoFilter and patterns involving all the operators were detected over this stream. The time taken to process the stream was noted. Subsequently, the same documents were indexed, and InfoSearch was used to detect the same patterns as in the previous case. In this case, the time taken for the result set to reach the root node was noted. The time taken by the two systems, for each of the operators, is shown in Figure B.1.

This experiment shows that using an index to detect complex patterns outperforms the streaming approach even for relatively small document collections. Of course, the time taken to index the documents is not considered in the above comparison, but that can be considered as a pre-processing step, and does not come into the picture once the documents are indexed and the index is brought online.

Other experiments related to performance and scalability of the current implementation of InfoSearch are currently in progress at the time of writing.

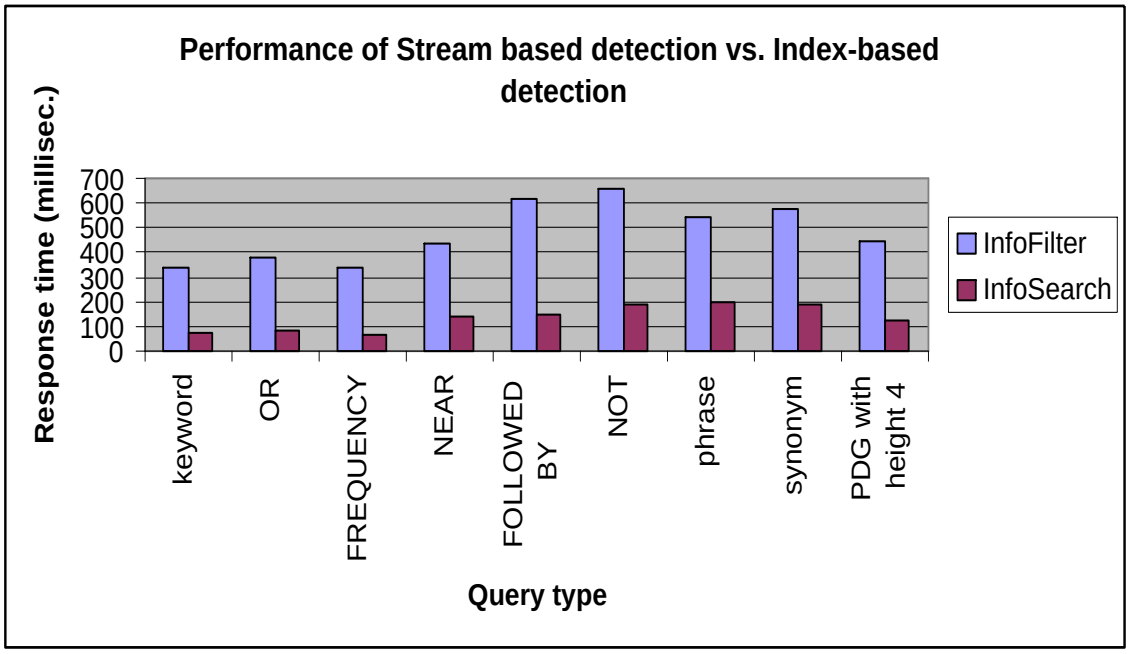

Figure B.1 Comparison of system performance over 2600 words

REFERENCES

[1] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the 7th World-Wide Web Conference (WWW7)*, Brisbane, Australia, Apr. 1998, pp. 107–117.

[2] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Essex, England: Pearson, 1999.

[3] L. Elkhalifa, "Infofilter : Complex pattern specification and detection over text streams," Master's thesis, University of Texas at Arlington, Arlington, 2004.

[4] G. Salton, A. Wong, and C. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, pp. 613–620, 1975.

[5] M. Maron and J. Kuhns, "On relevance, probabilistic indexing and information retrieval," *Journal of the ACM*, vol. 7, pp. 216–244, 1960.

[6] S. Robertson, "The probabilistic ranking principle in ir," *Journal of Documentation*, vol. 33, pp. 294–304, 1977.

[7] M. L. Mauldin. (1997) Lycos : Design choices in an internet search service. IEEE Expert. [Online]. Available: http://lazytoad.com/lti/pub/ieee97.html

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford Digital Library Technologies Project, Tech. Rep., 1998.

[9] J. Callan, B. Croft, and S. Harding, "The inquery retrieval system," in *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, 1992, pp. 78–83.

[10] H. Turtle and B. Croft, "Evaluation of an inference network-based retrieval model," *ACM Transactions on Information Systems*, vol. 9, pp. 187–222, 1991.

[11] Wordnet online lexical reference system. Princeton University. Princeton, NJ. [Online]. Available: http://wordnet.princeton.edu/w3wn.html

[12] Suffix tries. [Online]. Available: http://www.cise.ufl.edu/ sahni/dsaaj/enrich/ c16/suffix.htm

[13] L. Elkhalifa, R. Adaikkalavan, and S. Chakravarthy, "Infofilter: A system for expressive pattern specification and detection over text streams," in *Proceedings of SAC*, Santa Fe, NM, Mar. 13–17, 2005.

[14] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kauffman, 1999.

[15] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite events for active databases: Semantics, contexts and detection," in *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994, pp. 606–617.

[16] R. Dasari, "Events and rules for java: Design and implementation of a seamless approach," Master's thesis, University of Florida at Gainesville, Gainesville, 1999.

[17] Java compiler compiler. [Online]. Available: https://javacc.dev.java.net/

[18] Java wordnet library. [Online]. Available: http://sourceforge.net/projects/jwordnet

[19] Berkeley db java edition. [Online]. Available: http://www.sleepycat.com/jedocs/

BIOGRAPHICAL STATEMENT

Nikhil Deshpande was born in Mumbai, India, in 1979. He obtained a B.E. degree in Electronics from the University of Mumbai in 2001. Subsequently, he worked as Analyst for OrbiTech Solutions Limited, in Chennai, India upto 2002, where he got his first exposure to large scale information systems. He also worked as Database Co-Ordinator for Research In Motion Corp., in Seattle, Washington in the summer of 2005. His interest in Database Systems brought him to the University of Texas at Arlington, where he obtained his M.S. degree in Computer Science and Engineering in 2005. His research interests include database systems, query optimization, information retrieval, Web search engines and large scale data processing systems. He is a member of the Tau Beta Pi National Engineering Honor Society.