INCREMENTAL RETRIEVAL AND RANKING OF COMPLEX PATTERNS FROM TEXT REPOSITIORIES

by

JAYAKRISHNA THATHIREDDY

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

August 2007

To my parents.

ACKNOWLEDGEMENTS

I would like to offer my sincere thanks to Dr. Sharma Chakravarthy for giving me an opportunity to work on this project. It has been a memorable experience, to work with him and the lab. I would also like to thank Dr Mohan Kumar and Dr Nan Zhang for agreeing to serve on my committee.

I am grateful to Aditya Telang, Roochi Mishra for helping me out with the thesis. I would also like to thank Aravind Venkatachalam, Ruchika Marda, Hari Hara Subramanian and Balakumar Kendai for their wonderful support and help.

This work was supported, in part, by the NSF) (grants EIA-0216500, MRI 0421282, IIS 0534611 and IIS-0326505).

July 19, 2007

ABSTRACT

INCREMENTAL RETRIEVAL AND RANKING OF COMPLEX PATTERNS FROM TEXT REPOSITIORIES

Publication No. ______

JAYAKRISHNA THATHIREDDY, M.S.

The University of Texas at Arlington, 2007

Supervising Professor: Sharma Chakravarthy

As the volume of information accessible via the electronic medium (such as the Internet) is staggeringly large and growing rapidly, users have to sift through vast reservoirs of information to retrieve relevant data of their choice. It has been estimated that, the Internet consists of *2.5 billion* unique, publicly accessible web-pages and this figure is growing at an alarming rate of *7.3 million* pages per day[1]. Currently, the only way to wade through such colossal information is by using search engines (such as Google, Live, Yahoo, etc.). Although the popularity of search engines has increased manifold due to – simplicity of usage, speed of retrieval and amount of results generated, their ability to intelligently retrieve relevant information is significantly hampered due to over-reliance on Boolean operators for data retrieval.

Consider searching for complex patterns involving pattern frequency, proximity, sequence, structural patterns and synonyms. Consider the following examples:

(at least 5 occurrences of the phrase "research experiences"),

("metal" near "traders", in any order, within 10 words of each other),

("soya" followed by "plantings", within 5 words of each other),

[1]These statistics are obtained from [1]

all occurrences of the word ("contract" and its synonyms), and

("France" within the occurrence of "sunflower plantings" and "harvest")

The above search patterns are not supported by currently available search engines. Researchers looking for information in domains such as security, biology, legal research etc. need focused, objective and precise semantics to specify such patterns. In order to deal with such complex requirements of specific domain users, the design of a document retrieval system – that consists of a pattern specification language and an efficient detection engine that allows specification of such expressive patterns – is needed.

To address these issues, we have designed a framework (made up of two interdependent yet distinct systems – InfoFilter and InfoSearch) based on an expressive pattern specification language and a set of novel detection algorithms that handle streaming as well as static data. InfoFilter handles pattern detection for streaming data (news feeds, IP packets, etc.) where freshness of search results is paramount. However, in the case of data that resides in the form of large yet static repositories, and when the freshness of data is not critical, the InfoSearch system handles pattern detection using a pre-computed index.

The initial design of InfoSearch, for complex pattern detection, focused on fetching **all** matching occurrences of the pattern in the data repositories. It further processed all the tuples of the operands that constituted the pattern. In this approach, **all** answers are generated even if the user is interested in a small number of answers. Generating all answers when the request is only for **"k"** answers is inefficient in terms of processing time, memory utilization and number of computations. Moreover, the results are generated in the order in which they are detected, thus ignoring the relevancy of results with respect to user preferences. The user may be interested in ranked results which can indicate their quality. In order to address these problems, an incremental approach for complex pattern detection is needed. Moreover, in order to generate results based on the relevancy rather than the order of detection, ranking mechanisms for appropriately filtering the results also needs to be addressed.

In this thesis, we investigate several approaches for incremental detection of complex patterns, retrieval, and ranking of results. We also investigate the need for novel data structures as well as the types of structural meta-data to be associated with the data stored in the index for ranking the fetched results. We propose a novel ranking algorithm that utilizes the structural boundaries of the data to rank results based on the location and occurrence of a complex pattern in a document. We also present algorithms for each operator encountered in a pattern, that are based on ensuring optimal utilization of computational and memory resources in the least possible time. Extensive experiments have been performed to evaluate the scalability, performance, and memory usage of these algorithms on a number of patterns.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Information retrieval is defined as the process of identification and fetching of meaningful information from large collections of documents or data. Before the proliferation of electronic data on the World Wide Web (WWW), information retrieval was an activity that was important to people in specific domains (such as librarians, legal experts, researchers, ...). However, as the volume of accessible data on the Web has grown at an exponential rate, users with minimal knowledge about the structure and/or semantics of the data are interested in querying and retrieving interesting portions of this data. Hence, the problem of retrieving context-relevant information has gained prominence and received considerable attention from the research community.

1.1 Information Access Methods

The simplest and easiest form of information retrieval is to scan and retrieve those documents that match the patterns. The pattern can be either a simple word or can be a complex regular expression, specified using wild card expressions such as *men*r*, *m[a-z]n*. This can be done by using *grep*, *awk*, AND *sed* commands in Unix, Shell and Perl scripting, respectively. Although these approaches work with considerable success for small data sets, they do not scale in the real world due to the enormous size of the data to be searched for.

Another approach to retrieve structured data (E.g., Database Systems or DBMSs, Data Warehouses, etc.) is in the form of queries. However, these systems, require their users to learn query languages (such as SQL) for data retrieval. In addition to these rigid querying mechanisms, the ability to retrieve only structured data in a controlled access manner makes it impossible for users to use these machanisms on other forms of data.

In recent years, the information retrieval paradigm has changed considerably to incorporate the need for accessing increasing amounts of data on the Web. Since, most of the data on the Web is in an unstructured format (documents such as text files, HTML files etc.), Information Retrieval (IR) has evolved as an important mechanism to search these collection of documents based on simple user queries. The Information Retrieval Query Languages (IRQLs) are simpler to use than DBMS query languages. They are used to translate queries into database, where the semantics of the query are defined by an interpretation of the most suitable results of the query instead of a precise rendering of a formal syntax. Additionally, they allow users to specify queries using natural keywords and Boolean operators, thus making it easier to specify the required search pattern. An example of an IRQL is the Common Query Language (CQL), a formal language for representing queries to Information Retrieval systems such as web indexes, bibliographic catalogs and museum collection information. Internet search engines have adapted IR techniques to satisfy information need on the Web, and are discussed below.

1.2 Search Engines

Search Engines for the Web do not search for actual data on the Web in real time [2]. Instead, they search an index of a database of the full text of web pages retrieved from web-servers using crawlers. When the web is searched using a search engine, it is *NOT* looking at the current copy of the original document. Instead, it is using a document that is indexed and stored in the search engine database. When the link provided in a search engine's search results is accessed, the current version of the page is retrieved from the server where it resides.

Search engine databases are built by computer robot programs called *spiders*. They access the pages for potential inclusion in the database by following the links in the pages that have already been stored in their database. If a web page is never *linked to* any other page, spiders cannot find it. The only way a brand new page - one that no other page has ever linked to - can get into a search engine is for its URL to be entered manually for search. After the spiders find the pages, they pass them on to another computer program for *indexing*. This program identifies the text, links, and other content in the page, and the documents are then parsed to extract tokens (keywords). 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 [3]. The results are usually sorted in descending order of relevance, based on sophisticated *ranking algorithms*. The Boolean operators AND, OR, and NOT, as well as exact phrase matching are 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.

1.3 Complex patterns

Current search engines are convenient for performing keyword searches; however, in specific domains such as federal intelligence, legal databases and searching full-text patent information, there is a need to detect more complex patterns. Users in these domains may have more precise requirements in terms of the information they are searching. These patterns may involve *term frequency* (e.g., at least 5 occurrences of the phrase “research experiences“), *proximity with sub-patterns* (e.g., “metal“ near “traders“, in any order, within 10 words of each other), *sequence of sub-patterns* (e.g., “soya“ followed by “plantings“, within 5 words of each other), *all occurrences* of the word “contract“ and its synonyms, or structural patterns (e.g “France“ within the occurrence of “sunflower plantings“ and “harvest“) and so on. Additionally, 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., (“tax“ followed by (“petrol“ or “oil“)) near “retail stations“, 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 quite adequate for several specific applications or domains.

1.4 Data sources

Data sources over which these complex patterns need to be detected can be classified into two basic categories based on the frequency of the change in data: *dynamic* and *static* data sources.

1.4.1 Dynamic Data Sources

In dynamic data sources, the data may be updated very frequently (E.g., news page) or may be sent as streams (E.g., news feeds, data feeds etc.). Monitoring a dynamic source entails streaming in the raw data at the time of pattern detection to detect the required patterns[4]. 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. Further more, if freshness of the search results are important, it becomes necessary to read the data source every time while processing a query. Examples include RSS feeds, news feeds, real time stock information etc..

1.4.2 Static Data Sources

These data sources contain relatively static set of documents that are not updated on a frequent basis [5]. For example, many web and text repositories fall under this category. Since 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 is 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 is then used for answering queries. Since the index would be computed off-line, this approach may result in an occasional out of date search result. However, considering that the data source is not frequently updated, it is assumed to be acceptable to the user. For such relatively static data sources, the gains in terms of efficiency of retrieval that leveraging an index will bring outweigh the slight disadvantage of an occasional out of date result. The index is updated periodically to reflect any changes in the data source.

1.5 Problem statement

Searching for complex patterns (that have well-defined semantics) in dynamic streams of data[4] and over static data sources [5] has been investigated and proved to be possible. A suite of complex operators and algorithms have been developed to detect complex patterns in dynamic and static sources of data. During the detection of complex patterns over static data sources using a pre-computed inverted-index, the system detects all the occurrences of the pattern in a repository or a database. Additionally, all tuples are processed at each operand of the complex pattern, even if the user is not interested in all occurrences of the pattern resulting, in unnecessary wastage of time, computational and memory resources.

A better approach would be to fetch and process certain number of tuples at each operand of the complex pattern. If the processing of tuples results in the detection of patterns that satisfies the users request in the number of patterns detected, then further detection is halted. By adopting such an incremental approach, we could retrieve the patterns in the shortest possible time in addition to saving memory and computational resources.

In InfoSearch [5], the detected patterns are delivered to the user in the order of detection, with no information regarding the relevance of one pattern over another. The user need to sift through all the detected patterns to find the useful information. Hence, there is a need to rank the detected patterns.

In this thesis we investigate various approaches for incremental detection and retrieval of results based on user requirements from text repositories followed by ranking of these search results. Additionally, we investigate the need for novel data structures to incrementally retrieve the tuples, and additional structural information to be stored in the index for ranking of the retrieved patterns. Furthermore, we present a ranking algorithm to rank the search results and algorithms for each operator to incrementally detect and retrieve complex patterns from the index with optimal utilization of computational and memory resources in the least possible time. The ranking approach involves filtering results based on the position of occurrence of the complex pattern in a document, and utilizing the structural boundaries of the data.

1.6 Contributions

The primary goal of this work is to prove that it is possible to detect complex patterns by incrementally retrieving data from large text repositories. Additionally, we present a ranking algorithm to rank the detected patterns, based on proximity and structural information of the data. The existing work [5] has been enhanced to incrementally retrieve and rank the complex patterns from the text data repositories. The complete set of operators, such as frequency, proximity, containment, non-containment, sequence and synonyms, have been redesigned to detect and retrieve complex patterns incrementally. These operators are redesigned to extract appropriate results from the index, and incrementally detect the required number of complex patterns as specified by the user, based on the semantics defined for that operator. Since the information about the occurrence of the patterns is not sufficient to rank the retrieved complex patterns, additional structural information about the pattern is stored in the index. Furthermore, a new GUI interface has been developed to interact with the system by the user. Extensive performance evaluation of algorithms over different data sizes and patterns have been performed to determine scalability, response-time, and memory usage of the proposed algorithms. Furthermore, the proposed algorithms are compared with earlier algorithms as well.

The rest of the thesis is organized as follows: Chapter 2 reviews the related work. Chapter 3 discusses the design and architecture of the system. Chapter 4 describes the working of InfoSearch operators with elaborate examples. Implementation aspects are discussed in Chapter 5. In Chapter 6 time and memory usage of the system are discussed and compared with the InfoSearch system. Chapter 7 concludes the thesis and identifies some potential future directions.

CHAPTER 2

RELATED WORK

The goal of an information retrieval (IR) system is to provide users with the requisite documents that satisfy their information needs. Users need to formulate their information requirements, in a format that can be understood by the retrieval mechanism. To achieve this, the contents of large document collections need to be presented or stored in a format that facilitates the retrieval system to extract the relevant documents quickly and present it to the user. The information may be lost, while transforming the information, and while specifying the information need of the user in a format that the computer understands. Furthermore, the emergence of web as a popular medium of expression since the 1990's has created new challenges for information retrieval. The amount of information on the web is significantly large and is growing as compared to a traditional information system. The problem is further compounded by the increasing number of inexperienced users in the art of information retrieval compared to the domain expert users well-versed with the nature, content and the structure of data in the traditional informational system. To solve this, we need an efficient and scalable index to store the contents of a document, and a retrieval mechanism making efficient use of storage, memory and computational resources. In the following section we will discuss the concept of inverted index and some of the popular IR models.

2.1 Inverted Index

To generate a comprehensive index, the Web needs to be traversed systematically to locate all documents. The traversing of the Web is done by starting with a seed "URLS" to initiate the exploration. After parsing and indexing of the seed document, all the URL's in the seed document are extracted and then are traversed recursively and indexed.

An *inverted index* [6] is an index structure for storing a mapping from words to their occurrences in a document or a set of documents, thus, facilitating full text search. It is one of the most popular data structures used in IR systems. It maintains a sequence of (key, pointer) pairs where each pointer points to a document in the database which contains the key value. The index is then sorted on the key values to allow rapid searching for a particular key value (e.g., binary search). The index is called *inverted* because the key value is used to find a document rather than the other way round.

2.1.1 IR Models

In this section, we give a brief overview of some of the popular IR Models.

2.1.1.1 Boolean model

This model allow users to specify queries using a composition of boolean operators (AND, OR and NOT). Its simplicity and ease of implementation has made it popular amongst many commercial systems [3, 7]. It is based on set theory and hence every document is represented by a set of index terms, each of which is considered as a Boolean variable and is evaluated as True if the term is present in a document. 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 and hence there is no room for partial matches to a query. For example, if there is a query that includes several terms that are linked by the logical operator AND, any document that does not contain all the terms in the query are ignored, even though some of them might contain partial information that the user needs. The Boolean model's retrieval strategy is based on a *binary decision criterion*. A document is either predicted to be relevant or non-relevant, without any notion of relevance.

2.1.1.2 Vector Space models

The vector space model represents the documents and queries as vectors in a multi-dimensional space, whose dimensions are the terms used to build the index that represents these documents [8, 9, 7]. If the vector space is spanned by n normalized term vectors, then each document will be represented by an n-dimensional vector. If a term belongs to a document, it gets a non-zero value along the dimension corresponding to the term. Every document is represented as a vector of keywords, each with associated weights representing the importance of the keyword in the document. The weight of a term in a document vector is determined using the $tf \times idfmethod$, in which the weight of a term is determined by two factors: how often the term j occurs in the document i (the term frequency $tf_{i,j}$) and how often it occurs in the whole document collection (the document frequency df_j). Precisely, the weight of a term j in document i is calculated using the formula 2.1

$$w_{i,j} = tf_{i,j} \times idf_j = tf_{i,j} \times logN/df_j \tag{2.1}$$

where N is the number of documents in the document collection and idf stands for the inverse document frequency. This method assigns high weights to terms that appear frequently in a small number of documents in the document set. The strength of this model lies in its simplicity, but the expressiveness of query specification inherent in the Boolean model is sacrificed. The drawback of the vector-space model is that it assumes the term vectors spanning the space to be orthogonal, and existing term relationships are IGNORED.

2.1.1.3 Probabilistic models

Probabilistic methods generates complex index terms based on term-dependence information and relationships. It is based on the observation that, *the relevance of a document to a query is related to the probability of the query terms occurring in the doc-*

ument. Generation of term-dependence involves consideration of an exponential number of term combinations and, for each combination, estimating the probabilities of coincidences in relevant and irrelevant documents involves considerable effort and work; hence, only certain dependent-term pairs are considered in reality. The probability estimation technique is the key part of the model, and different techniques have been proposed in the literature [10, 11, 7]. Only the basis of the models is described here.

The probability of relevance of a document D can be represented by $P(R|D)$. Documents are ranked based on $\frac{logP(R|D)}{logP(R|\overline{D})}$, where $P(R|\overline{D})$ represents the probability of document being non-relevant. When Bayes' transform is applied to this ratio, $\frac{P(D|R).P(R)}{P(D|\overline{R}).P(\overline{R})}$ is obtained. $P(R)$ and $P(\overline{R})$ can be canceled out, if we assume that $P(R)$ is independent of the document under consideration resulting in $\frac{P(D|R)}{P(D|\overline{R})}$ as the score formula. After this, different systems diverge based on the assumption behind the estimation of $P(D|R)$.

2.1.1.4 Linguistic and Knowledge model

In text retrieval, users typically enter a string of keywords that represents the users information needs which are then used to lookup the inverted indexes. AS this approach retrieves documents based solely on the presence of exact keywords as specified by the user, it often fails to find the information the user actually desires since the words used by the user might be different from the ones used in the relevant documents. To address this problem, linguistic and knowledge-based approaches have been developed by performing a morphological, syntactic and semantic analysis to retrieve documents more effectively [7]. In the morphological analysis, roots and affixes are analyzed to determine the part of speech (noun, verb, adjective etc.) of the words. In the next phase complete phrases ARE parsed using some form of syntactic analysis. Finally, the linguistic methods have been used to resolve word ambiguities and to generate relevant synonyms based on the semantic relationships between words. The development of a sophisticated system is difficult and complex and requires intricate knowledge of semantic information and retrieval heuristics.

2.1.2 Search Engines: Applying IR techniques to the Web

The data available on the web is extremely large, heterogeneous, and IS in the form of uncontrolled collection of documents as opposed to the more controlled, smaller document repositories for which standard IR techniques were originally designed. Search Engines have adapted and extended the existing standard IR models to retrieve the information efficiently and effectively. The ability to search and retrieve information that meets the user's information needs from the Web, is an enabling technology for realizing the full potential of the Web. Standard IR techniques are designed to retrieve documents that closely match the query, given that both the query and the document are represented by their word occurrences. Hence, creating a scalable search engine presents many challenges. Efficient and faster crawling technology is required to retrieve the web documents and keep them up to date. Storage space needs to be utilized efficiently to store indexes and the documents if necessary. The indexing system must be able to process Tera bytes of data efficiently and additionally the queries must be handled quickly, at the rate of millions per second. Search Engines typically searches repositories of full text of web pages selected from the billions of web pages available on the web. When the web is searched using a search engine, it is searching a somewhat stale copy of the real data available on the web. Once the user clicks on the link provided in the search engine's search results, the current version of the page is retrieved from the server and returned to the user. If a web page is never linked to in any other web page, search engine won't be able to find it. The only way a brand new page, one that no other page has ever linked to, can get into a search engine is to request the search engine companies to include the new web page. Search Engines try's to add other factors to rank the retrieved documents including external (meta) information about the documents, references to documents from other documents, etc. Retrieval strategies of some of the Search Engines are further discussed in this section.

2.1.2.1 Google

Google [12] is one of the most popular and successful commercial Web Search Engines. It uses highly optimized data structures to crawl, index, and search the web. It stores the full HTML of every web pages fetched by the crawler in a repository. Each page is compressed using zlib (RFC1950). In the repository, the documents are stored one after the other and are prefixed by doc ID, length, and URL. The document index is a fixed width ISAM (Index sequential access mode) index ordered by doc ID that maintains information about each document. The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. It also makes use of lexicon, hit lists, forward index and an inverted index for fast access of the document lists. Google uses the link structure of the Web to calculate a quality ranking for each web page that is indexed. It maintains much more information about web documents compared to other search engines. Every hitlist includes position, font, and capitalization information. Additionally, it also utilizes information regarding the hits from anchor text and the PageRank [13] of the document. The ranking function is designed such that no particular factor has significant influence. Google supports Keyword queries and also boolean compositions of queries and phrases. However, complex queries based on pattern frequency, proximity, non-occurrence of a pattern within two patterns are currently not supported.

2.1.2.2 Yahoo

Since its inception in 1994, Yahoo manually generated catalogs of the web. It used crawler-based results from its partners only when there were no human-powered matches [14]. This gave Yahoo an edge over other competitors in the initial years until Google used crawlers to generate and retrieve both comprehensive and highly relevant information. Human-maintained lists cover popular topics effectively but are subjective, expensive to build and maintain, slow to improve, and cannot cover all topics. In 2002, it dropped manually generated catalogs in favor of the crawler generated data. The manually generated catalogs, “Yahoo Directory” still exist and are leveraged by the company. Yahoo supports Boolean operators and nested searching with the operators AND, OR, NOT and Phrase searching. Though it does not support proximity based queries, the Wild Card Word in Phrase technique can be combined with OR’s to create a proximity search. For example, to find “addictive semiconscious vice of biblioscopy” when you are not sure of the second word, search “addictive * vice of biblioscopy”. Results are sorted by a relevance algorithm. All the pages in a site are clustered and only one page per site is displayed. Other clustered pages can be accessed using the “More pages from this site” link at the end of the record.

2.1.2.3 Lycos

Lycos operated one of the web’s earliest crawler-based search engines [15]. It uses a breadth-first-search based on the popularity heuristic. Lycos minimizes the ability of authors to manipulate popularity data by only counting one link per server. This approach tends to find home pages rather than subsidiary pages, so the Lycos catalog is biased toward more popular and useful web-pages. Once a document is located and retrieved, an “abstract” of the document is generated and stored. Using standard information-retrieval statistical methods, Lycos identifies the 100 most “weighty” terms. Along with these weighty terms, the titles, header text, and an excerpt of the first 20 lines, an “abstract” is created that is about one-fourth the size of the original document. It then displays the abstracts along with the list of links during the retrieval process, allowing users to quickly determine which of the matched documents they wish to examine. Lycos does not return results that are as relevant as one might expect from other search engines, but better results are obtained for more specific and concise queries. Ranking of the results is 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. It supports queries based on keywords and boolean compositions of keywords; however, complex queries including proximity, containment, etc. are not supported.

2.1.2.4 Ask

The idea behind Ask is its ability to answer questions posed in natural language [16, 17]. It is the first commercial question-answering search engine for the World Wide Web. It supports a variety of user queries in plain English (natural language), as well as traditional keyword searching and strives to be more intuitive and user-friendly than other search engines. It uses subject-specific link popularity to compute "authoritativeness" of a search result. Initially it used editors to monitor what people searched for, then manually select sites that seemed to best answer those queries. This approach worked well for the most popular queries but did not help when users wanted unusual information. In 2002, it shifted over to relying on Teoma search technology, also known as Expert Rank algorithm, for nearly all of its matches. The Expert Rank algorithm searches results by identifying authoritative websites in addition to link popularity, subject-specific popularity. The search engine supports boolean search and limited phrase searching.

2.1.3 Adjacency and Proximity searching

Detection of patterns involving adjacency and proximity in the science citation index (SCI) and google has been explored in [18]. The SCI algorithm uses intercalating stop words in a query phrase, which acts as a placeholder. Such a phrase serves effectively as a fixed adjacency condition determined by the number n of adjacent stop words (i.e., retrieve all records where word A and word B are separated by n words in at least one location). The algorithm integrates over search phrases with different numbers of adjacent stop words to provide a flexible adjacency or proximity capability (i.e., retrieve all records where word A and word B are separated by n or less words in at least one location, where n is the maximum separation desired between A and B in at least one location). To detect the pattern "nutrient" followed by "uptake" separated by a distance

of 3 words, the search algorithm specifies the query as “nutrient” of of of “uptake”, where *“of”* is a SCI stop word. In general, the search algorithm to find records A and B, separated by a distance of n word is
(A[nS]B) OR (B[nS]A) NOT (AB OR BA OR (ASB) OR (BSA) OR (ASSB) OR (BSSA) ...... OR (A[(n-1)S]B) OR (B[(n-1)S]A))
where “S” is a SCI stop word, and “n” specifies the distance between the two words A and B.

The Google algorithm exploits the fact that asterisks (in Google) separating words in a phrase function act like word wildcards. The difference between two such phrases (the first phrase containing one less asterisk than the second phrase) serves effectively as a fixed adjacency or proximity condition, with the number of separating words equal to the number of asterisks in the first phrase. The algorithm integrates over these phrase differentials to provide a flexible adjacency or proximity capability (i.e., retrieve all records where word A and word B are separated by n or less words in at least one location, where n is the maximum separation desired between A and B in at least one location). If A and B are two words in the Google query, then the conditions specified in the table 2.1 holds

Table 2.1 Fixed spacing adjacency conditions

1	Zero word spacing (coherent phrase)	$“AB'' - “A * B''$
2	One word spacing	$“A * B'' - “A * *B''$
2	Two word spacing	$“A * *B'' - “A * * * B''$
3	Three word spacing	$“A * * * B'' - “A * * * *B''$

For example, if the query information * * technology” information * * * technology” is used to search the titles in Google, it will retrieve only those records that contain ”information” preceding, and separated by two words from, “technology”.

2.1.4 InfoSearch

InfoSearch [5], developed at The University of Texas at Arlington, enables complex pattern detection over static data sources. It supports all the operators supported by the earlier system InfoFilter [4], which can detect patterns on streamed data. InfoSearch uses a pre-computed index over large data repositories to efficiently detect and retrieve such complex patterns. It used an inverted index to store the document id and position of occurrence for each keyword in a document. The user specifies a pattern using an expressive Pattern Specification Language (PSL). The user pattern is validated and passed to the graph generator, which represents the pattern as a Pattern Detection Graph (PDG). Similar to the InfoFilter system, the leaf nodes of the PDG represent the simple patterns such as keywords, phrases and system defined patterns such as structural boundaries. The internal nodes in the PDG represent the operators. It extracts the keywords and phrases and inserts them into a keyword buffer. The index interface then extracts the keyword from the keyword buffer and fetches all the tuples of the keyword from the index. Once the tuples has been fetched, the corresponding lead node in the PDG is triggered. Once an operator receives tuples from all its child nodes, it then process them based on the operator semantics. The resulting tuples from the operator is then propagated up the tree for further processing. The tuples available at the root node corresponds to the detected pattern. Once the resulting tuples becomes available, a rule defined on the root node gets triggered. The rule performs the role of notifying the user of the detected patterns.

2.2 Ranking

2.2.1 Google PageRank

Google uses the link structure of the web to calculate quality ranking for each web page that is indexed. It maintains position, font, and capitalization information for each keyword that is stored in the index. Additionally, it also utilizes information regarding the hits from anchor text and the PageRank [13] of the document. To rank a document with a single keyword query, Google looks at that documents hit list for that word. Google considers each hit for the word to be one of several different types like title, anchor, URL, plain text large font, plain text small font, etc., each of which has its type-weight. It counts the number of hits for each type in the hit list and then every count is converted into a count-weight. Count-weights increases linearly with counts initially but quickly taper off, so that more than a certain count will not have significant influence. It then uses the dot product of the vector of count-weights with the vector of type-weights to compute an IR score for the document. Finally, the IR score is combined with PageRank to give a final rank to the document. Hits [19] and TrustRank [20] algorithms also use the information extracted from link structures to evaluate and rank the retrieved web pages. Most of the popular search ranking algorithms depends on the link structure and anchor text to rank the documents.

2.2.2 Ranking in IR

In Probabilistic IR models, the documents are ranked based on the probability of the document being useful to the user, which is calculated based on the Bayesian decision rule. In Vector space model, selection of a document is done by assigning a *score or rank* for the document against the query. This score is computed by measuring the similarity of the query with the document. 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. Typically, the **term frequency** or *tf* (number of times the term occurs in the document) and the **inverse document frequency** or *idf* (in how many documents does the term occur?) are used. The more the number of documents the term occurs in, the lesser is its discriminating power in identifying a document. A high ranking score is assigned to a document that contains only a few of the query terms if these terms occur infrequently in the collection but frequently in the document.

2.2.3 Ranking in DBMS

RankSQL [21] provides a systematic and principled framework to support efficient evaluations of ranking (top-k) queries in relational database systems(RDBMS) by extending relational algebra and query optimization. In normal relational query models, to fetch top-k results for queries, the following steps are performed to produce ranked results. First, all the records of the inputs are consumed. Second, the inputs are joined to materialize the join results. Third, the predicates are evaluated for each of the valid join results. Finally, the fourth step involves sorting of the join results on the predicate. From the sorted result, only top-k results are presented to the user. This approach of materialize-then-sort is inefficient and involves scanning large base tables, joining of large intermediate results, evaluate ranking on every tuple and then sorting on all tuples. To avoid this short coming, the author proposes a split-and-interleave approach, in which the ranking function is split and interleaved with the boolean operations. The inputs are first ranked based on the predicates and then projected and joined. This results in the reduction of the number of intermediate results, and hence reduction in processing costs.

2.2.4 Ranking using proximity

In [22], the ranking of documents or of documents using the properties of the language is explored. Here, the text is treated as a continuous sequence of terms, or tokens each corresponding to a word or number in the text, and an integral position is assigned in sequence to each term. The resulting patterns for the query are ranked based on the assumption *"The smaller the span across which the pattern is detected, the more likely that the corresponding text is relevant"*. In PADRE [23], the usage of proximity relationships to estimate document relevance instead of depending solely on the summing up $tf \times idf$ weights is explored. Here, the ranking scheme is based on the premise *"The closer together a set of interesting terms, the more likely they are to indicate relevance"*

2.3 Summary

Most of the available systems, both traditional IR and Web search engines support keyword queries and boolean composition over keywords and in some cases queries. Though this works in most of the cases, in certain domains like federal intelligence, legal databases and while 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. Current IR systems and search engines do not provide a means to specify and detect complex patterns that are based on term frequency, proximity, sequence and containment. This problem has been addressed for stream data (InfoFilter), and on static data InfoSearch in the previous work on extending the expressiveness of patterns. The index based system developed for static data, processES all the occurrences of the terms that makes the complex query for detection of the pattern. This approach is inefficient and results in wastage of precious computational and memory resources in addition to delayed response it. Hence, there is a need for a better system that can detect the complex patterns in the least possible time with efficient usage of computational resources.

CHAPTER 3

SYSTEM DESIGN

The system allows the user to specify complex queries and returns all occurrences of the pattern. The user can specify whether he would like to incrementally retrieve the patterns, by specifying the number of results required to satisfy his requirements. Additionally, he could also specify whether he want the detected patterns to be ranked. If the user does not enable the ranking of the results, then the patterns are delivered in the order of detection. The process of detection of the pattern is divided into four distinct steps. First, the user specifies the pattern to be detected using the Pattern Specification Language (PSL), through the interface provided by the system. Second, the user query is parsed and validated to ensure that it conforms to the PSL syntax. The parsed pattern is used as input by the graph generator module. The graph generator extracts the tokens from the input and builds two data structures termed Pattern Detection Graph (PDG) and pattern table. Third, the index interface module uses the information provided in the pattern table to look up the index and feed the PDG with data to detect patterns. Algorithms used in InfoSearch have been modified to deal with incremental flow of data. Fourth, the detected patterns are delivered to the user by the notification module. Additionally, if the user has enabled the ranking of the results, a ranking algorithm is run on the detected patterns before notifying the user. In this chapter, we describe the above steps in detail, and the underlying architecture of different modules.

3.1 System Architecture

The system architecture is shown in Figure 3.1. The system has adopted most of the modules from the InfoSearch system [5], after enhancing some of the modules to incrementally retrieve the data. Additionally, the operators were modified extensively to handle the input sets of data available in increments. The modules of the system include the *user interface*, the *pattern parser* and *validator*, *pattern processor*, *graph generator*, the *pattern detection engine*, the *index interface* and the *notifier.* External modules such as the *WordNet synonym database* and the *inverted index* are also shown in the figure. The *pattern parser* and *validator* modules have been adopted completely from the current InfoSearch system. The graph generator has been modified in order to create the pattern table data structure. The pattern table data structure stores information required by the index interface module to look up the index and retrieve the required number of tuples. The *pattern detection engine*, that includes the operator functionality, has been modified extensively to process inputs tuple sets incrementally. The number of tuples to be fetched at a time can either be fixed or determined dynamically, depending on the number of the patterns left to be detected. In the following sections, we will describe the underlying architecture of the different modules of the system.

3.1.1 Inverted Index

The index of a document collection is organized by mapping each document to all the words contained inside it. A keyword is searched by scanning and checking the word-lists associated with each document. This is an inefficient process as it requires a sequential scan of the database for each keyword to be searched. Moreover, as the size of the document collections increase, the inefficiency increases further. An alternative for this problem is to use *inverted indexes.*

Inverted indexes used in search algorithms by search engines, maintain a mapping from a keyword to the set of documents that contain the keyword. Document IDs are assigned to each document in the document collection to uniquely identify them. For each keyword in the document, a *keyword - document ID* mapping is stored in the inverted index. 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 in the index is adequate to answer keyword queries and queries involving composition of Boolean operators.

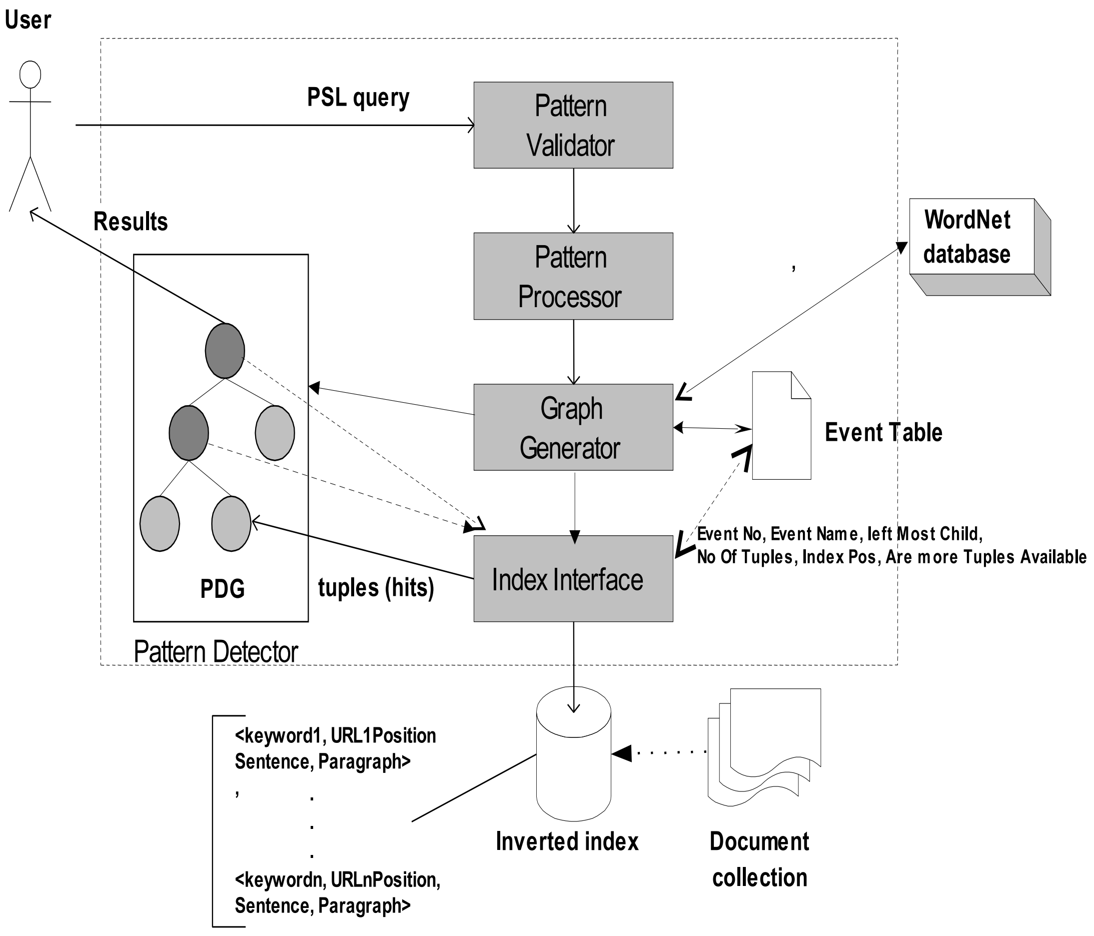

User
PSL query
Pattern Validator
Results
Pattern Processor
WordNet database
Graph Generator
Event Table
Event No, Event Name, left Most Child,
No Of Tuples, Index Pos, Are more Tuples Available
PDG
tuples (hits)
Index Interface
Pattern Detector
<keyword1, URL1Position Sentence, Paragraph>
<keywordn, URLnPosition, Sentence, Paragraph>
Inverted index
Document collection

Figure 3.1 InfoSearch architecture

For example, in the above example, if the user is searching for *"sales" AND "petrol"*, the intersection of the document IDs corresponding to the keywords *"sales"* and *"petrol"* gives us the desired result (documents 1 and 3 in this case).

However, this information is not adequate to answer queries involving *proximity*, *sequence*, *frequency*, and *containment*. Additionally, it is difficult to rank these detected patterns with the available information. The above indexing scheme stores information about the presence or absence of a term in a document but not about *every* occurrence of a keyword and its position in the document. For example, a query such as *"sales" NEAR/5 "petrol"* cannot be answered using information from such an index, because the distance between occurrences of *"sales"* and *"petrol"* within a given document needs

Table 3.1 A sample set of documents

Document ID	**Document Contents**
1	Sales tax on petrol
2	Petrol or Oil
3	Increase in petrol sales

Table 3.2 Inverted Index on Documents in Table 3.1

Keyword	**Documents**
sales	1,3
tax	1
on	1
petrol	1,2,3
or	2
oil	2
increase	3
in	3

to be computed. To support these queries, information regarding the position of *every* occurrence of the keyword needs to be stored [24]. Additionally, to rank the patterns using proximity and positional information, we need to store the sentence and the paragraph in which the keyword occurs. Each keyword is represented in the index as a Tuple *documentid<position information, sentence information, paragraph information>*. Table 3.3 shows an inverted index generated on the documents in Table 3.1 that contains additional information for ranking and proximity determination.

Therefore, the document ID and the positional information of a given keyword from the index is needed to detect and rank these complex patterns. In this thesis, we investigate if this information is adequate to incrementally retrieve and rank the detected patterns.

3.1.2 User Interface

The current InfoSearch user interface was developed using *java applets* and takes *user E-mail ID*, *query* from the user. The rest of the information such as *data input*

Table 3.3 Inverted index with position information

Keyword	**Documents with position**
sales	1<1,1,1>,3<4,1,1>
tax	1<2,1,1>
on	1<3,1,1>
petrol	1<4,1,1>,2<1,1,1>,3<3,1,1>
or	2<2,1,1>
oil	2<3,1,1>
increase	3<1,1,1>
in	3<2,1,1>

mode, *database index* are read from a file as configuration parameters. Additionally, the user interface can be used only once to enter a query. If the user wants to enter a second query, he needs to restart the interface. Furthermore, the interface does not have the ability to present the user with the detected patterns. Due to the above shortcomings, we have developed a new interface called *InfoClient* using *java swing* technology.

InfoClient runs in either of the two modes *demo mode* or *interactive mode.* The *demo mode* is designed to help the novice users of the system and hence provides a set of pre-determined input patterns to test the system capabilities. The *interactive mode* designed for advanced users, allows users to formulate their own queries for detection. The system processes the query and delivers the detected patterns to the user. The user interface supports the following features:

- Please Enter your EMail Address : Users email address, the system uses it to uniquely identify the user.
- Data Input Mode : The system can be used for detection of the patterns from static data [5] or dynamic data [4]. For detection of patterns from dynamic data, select *Stream* or else select *Index.*
- Return K Results: If the user likes to enable retrieval of patterns incrementally, then select *true* or else *false.*

- Type of Data Stream: If the user choose to detect the patterns from *dynamic data*, the user has to specify the nature of the dynamic data. The available options are *Text_stream*, *Email_stream.*
- Input Stream: Dynamic data stream source, which needs to be parsed for detection of the pattern.
- Database Index: Path of the *static data source index.*
- Pattern: Pattern that needs to be detected.
- Enter Email Address / Addresses for Notification: The system sends notification once the pattern is detected, to all the specified email addresses (separated by comma).
- Required Number of Query Results: Here the user can specify how many patterns he would like to retrieve. This option is available only when the user has chosen *Index* as data input mode and *Return K Results* is enabled.
- Rank Results: Enables or Disables ranking of the detected patterns.
- Input Stream/Output: Displays the detected patterns to the user. Additionally, it also displays the selected input stream, if the user has chosen *Stream* as *Data Input Mode.*

3.1.3 Pattern Parser and Validator

This module takes a user query as its input, and checks if the query is in the proper syntax, as dictated by PSL BNF. Please refer to [4] for psl bnf. If the query does not conform to the PSL syntax, a parser error is returned. After validating the query 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. The extracted tokens are sent to the *pattern processor.*

3.1.4 Pattern Processor

This module receives a set of tokens, in infix notation, as its input. It converts the input into a postfix expression, which is easier to evaluate and to generate a graph. The postfix expression is passed on to the *graph generator*, for further processing.

3.1.5 Graph Generator

To facilitate detection of complex patterns, a data structure called Pattern Detection Graph (PDG) is used. The *graph generator* takes a stack of tokens, which represents the user query in the postfix notation from the *pattern processor* and generates the PDG from it. The PDG is constructed in a bottom-up fashion. The leaf nodes are created first, which represents simple patterns such as keywords, system defined patterns etc.. Internal nodes of the graph correspond to complex patterns and encapsulate the logic of the corresponding operator. When a parent node is created, a reference of the parent is passed to the children, so that data can be passed from a child to its parent. Hence, every node in the PDG has a subscriber list containing references to each of its parents, except for the root node. As an example, the PDG corresponding to the pattern *("metal" FOLLOWED BY 'traders")* is shown in Figure 3.2. For this example, the leaf nodes in the PDG correspond to the keywords in the query, *"metal"* AND *"traders"*, and they have references to their parent nodes. The number in the subscriber list indicates the distance with which the parent has subscribed. User can specify distance for operators such as "NEAR" and "FOLLOWED BY", which indicates the maximum separation between the operands. For example, consider the query *("metal" NEAR/3 "traders") WITHIN("copper", "scrap").* The PDG corresponding to this query is shown in Figure 3.3. In this example, distance "3" is stored in the subscriber list, the WITHIN node subscribes to the NEAR node with a distance of 3. Here, the NEAR node sends only those tuples that are separated by a maximum distance of "3" words to the parent node WITHIN.

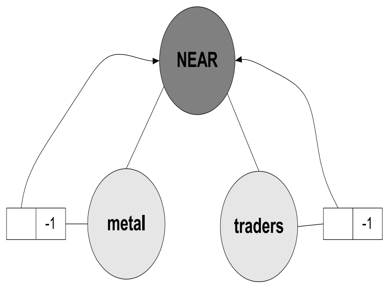


Figure 3.2 PDG corresponding to *(“metal” FOLLOWED BY ‘traders”)*

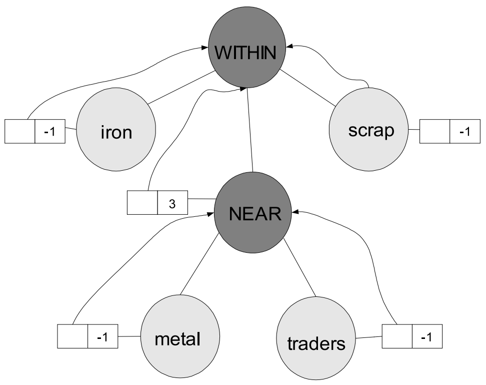


Figure 3.3 PDG with subscriber list containing distance

The input to a leaf node is a set of tuples corresponding to the index lookup for the term represented by the leaf node. This set consists of ***<doc id, start offset, end offset, start sentence, end sentence, start paragraph, end paragraph>*** tuples. For example, the set of tuples for the keyword *“petrol”* 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 known as subscriber nodes, except for the root node. Leaf nodes propagate their input sets to their parent nodes. A parent node, which corresponds to one of the operators such as OR, NEAR, FOLLOWED BY, SYN, FREQUENCY, WITHIN or NOT, gets one or more sets of tuples as its input. The operator then merges its input tuples sets according to its semantics to create a merged result set. The merged result, 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 detected patterns for the query.

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

1<6,6,1,1,1,1>
2<1,1,1,1,1,1>
3<6,6,1,1,1,1>

A tuple corresponding to word occurrences in a document termed a *point tuple*, that is, the start and end values of offset, sentence and paragraph of that tuple are same, because a word occurs at a single position within a document. However tuples corresponding to a more complex pattern are *interval tuples*, that is, their start offset is smaller than its end offset, start sentence is smaller than or equal to its end sentence and start paragraph is smaller than or equal to its end paragraph. This is because a complex pattern such as ("metal" FOLLOWED BY 'traders") occurs in an interval within the document. For example, in Figure 3.2, the tuples corresponding to "metal", "traders" are point tuples, but the tuples corresponding to the combined pattern "metal" FOLLOWED BY "traders" are interval tuples. Thus, the operators may get either point tuples or interval tuples as their input, and their output depending on the semantics of the operator will be point or interval tuples.

Additionally, every node in the PDG has an unique identifier known as *node name*. For leaf nodes, the node name is same as the operand. Whereas, for internal nodes, the name is composed from the operands node names and the operator. FOR the above example shown in Figure 3.3, the node name for the nodes are as shown in the table Table 3.5

3.1.5.1 Sharing of PDG nodes

To optimize the detection of the pattern, the *graph generator* shares PDG nodes wherever possible. It is achieved by reusing common PDG or sub-PDG for a common expression or sub-expression. This avoids creation of a duplicate PDG, when a PDG has already been created for a previous expression or sub-expression for the same user. Each

Table 3.5 Node Names for nodes for the PDG of the pattern “metal” NEAR “traders”

Node	**Node Name**
metal	metal
traders	traders
copper	copper
scrap	scrap
NEAR	metal_near_traders
WITHIN	copper_within_metal_near_traders_scrap

user has a separate memory space in which the the PDG is created. For example, consider that another query *“iron” FOLLOWED BY (“metal” NEAR/5 “traders”)* is specified along with the query shown in Figure 3.3. The graph generator knows that a sub-PDG for the sub-expression *“metal” NEAR/3 “traders”* already exists. Hence, instead of creating a new sub-PDG, the sub-PDG is reused, by having the new FOLLOWED BY node subscribe to it. This results in reduction of time and memory requirements when several queries having common sub-expressions are processed together. More importantly, the sub-pattern is computed once, for all the distances and the corresponding output is generated.

In the “InfoSearch” system, the sharing of the PDG nodes did not happen correctly. When the second query *“iron” FOLLOWED BY (“metal” NEAR/5 “traders”)* arrives, the system checks, whether the leaf nodes *“metal”* and *“traders”* already exists. Since the leaf nodes already exists, it uses the references to these nodes and attempts to create the internal node NEAR with a distance of “5”. Before creation of the internal node, the system checks if a node with the node name *“metal_near_traders”* exists. As the node exists, the system reuses the node. The system does not check if the distance “5” is present in the subscribed list. The distance information is lost and the “FOLLOWED BY” node subscribes to the child node *“metal_near_traders”* with no distance and is shown in Figure 3.4. This results in the detection of patterns that would not have been detected if the distance has been added in the subscriber list. Additionally, consider the two queries *(FREQUENCY/2 (“Iraq” NEAR/10 “Baghdad”) FOLLOWED BY “Missile”)* and *((FRE-*

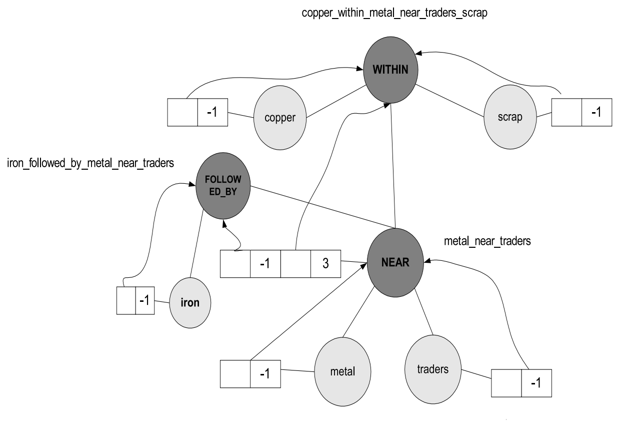


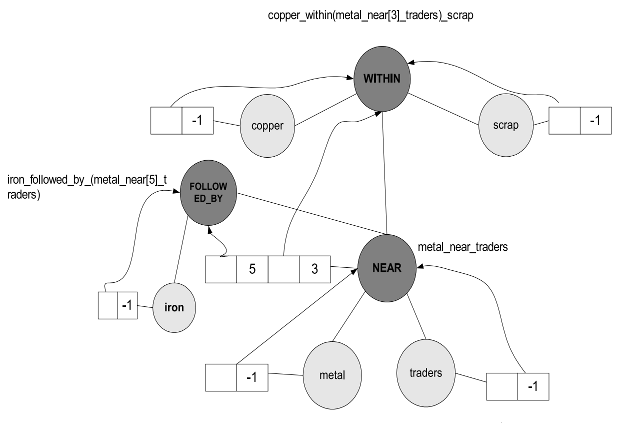
Figure 3.4 PDG with a shared node in InfoSearch system

QUENCY/2 ("Iraq") NEAR/10 "Baghdad") FOLLOWED BY "Missile"). Even though the two queries are different, the node names generated in the InfoSearch system for both the queries is *"FREQUENCY[2]:iraq_NEAR_baghdad_FOLLOWED_BY_missile"*. Hence, when the system attempts to build the PDG for the second query, it finds a PDG with the same name. So it does not create a PDG for the seconds query. To avoid this shortcoming, the node names for the nodes are modified to include the distance and containment information. The node names for the nodes in the PDG for the queries *("metal" NEAR/3 "traders") WITHIN("copper", "scrap")*, *"iron" FOLLOWED BY ("metal" NEAR/5 "traders")* and *(FREQUENCY/2 ("Iraq" NEAR/10 "Baghdad") FOLLOWED BY "Missile")* and *((FREQUENCY/2 ("Iraq") NEAR/10 "Baghdad") FOLLOWED BY "Missile")*are as shown in Figure 3.5, Figure 3.6, Figure 3.7 respectively.

In InfoSearch system, the simple patterns (or leaf nodes) are stored in a list, during the pattern detection graph generation phase. This simple patterns stored are later used to query the *index interface* for retrieving the appropriate "hits" from the index for detecting the pattern.

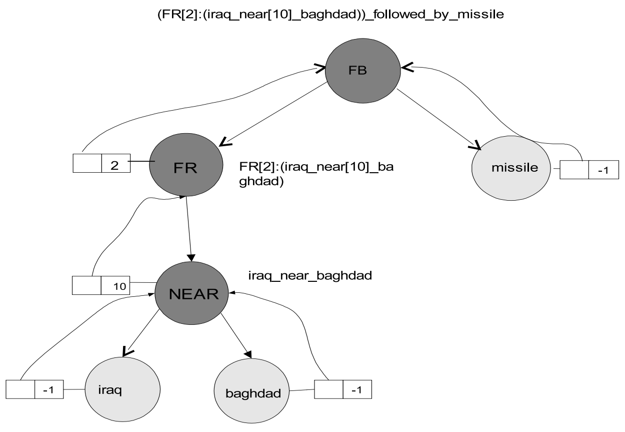
3.1.6 Index Interface

The *index interface* module accepts simple patterns from the *graph generator* and queries the inverted index. It wraps the set of results from the index into a set of



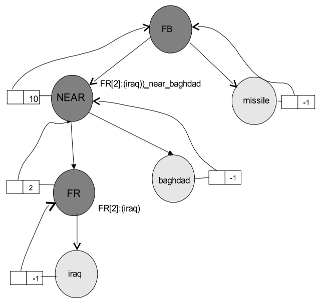
copper_within(metal_near[3]_traders)_scrap
WITHIN
-1
copper
scrap
-1
iron_followed_by_(metal_near[5]_traders)
FOLLOWED_BY
5
3
NEAR
metal_near_traders
-1
iron
-1
metal
traders
-1

Figure 3.5 PDG with a shared node



(FR[2]:(iraq_near[10]_baghdad))_followed_by_missile
FB
2
FR
FR[2]:(iraq_near[10]_baghdad)
missile
-1
10
NEAR
iraq_near_baghdad
-1
iraq
baghdad
-1

Figure 3.6 PDG for Query (FREQUENCY/2 (“Iraq” NEAR/10 “Baghdad”) FOLLOWED BY “Missile”)



((FR[2]:(iraq))_near[10]_baghdad)_followed_by_missile
FB
10
NEAR
FR[2]:(iraq))_near_baghdad
missile
-1
2
FR
FR[2]:(iraq)
baghdad
-1
-1
iraq

Figure 3.7 PDG for Query ((FREQUENCY/2 (“Iraq”) NEAR/10 “Baghdad”) FOLLOWED BY “Missile”)

<docID, start offset, end offset, start sentence, end sentence, start paragraph, end paragraph>tuples and notifies the leaf node, that corresponds to the simple pattern. The inverted index is built using the Berkeley DB java edition free ware [25].

3.1.7 Pattern Detector

The data flow in the PDG and the merging of tuples by the operators to detect patterns, is similar to detection of composite events using Event Detection Graphs (EDG) [26]. In the latter, event occurrences are propagated up the Event Detection Graph, in which the composite nodes merges their inputs based on criteria known as parameter contexts. A node in the graph can have an associated rule, which means that a predefined action can be taken when that event node is triggered. Since there is similarity in the data flow, the Event Detection Engine framework, called the Local Event Detector (LED) [27] is used as the backbone for the Pattern Detector. When a leaf node in the pattern detector receives a set of tuples from the index interface, a reference to the input set is passed to all its parent nodes. Similarly, when internal nodes merge their input sets to create a merged output set, they pass a reference to the merged set to their parents. The root node has a rule associated with it, which handles the notification of the detection of the pattern to the user. The operators in the system use Proximal-Unique semantics to merge their input tuples and is explained in the section 3.1.7.1. Although this is similar to LED in its abstraction, the whole system has been modified/extended to deal with pattern detection including the proximal-unique semantics which is needed for detection. Please refer to [4] for more details on the similarities and differences between EDG and PDG as well as algorithms of operators.

3.1.7.1 Proximal-Unique semantics

Let us consider a document containing words as shown in Figure 3.8. We want to detect occurrences of the pattern *"metal" FOLLOWED BY "traders"* within this document. As seen in the figure, *"metal"* occurs at two positions, one occurring at

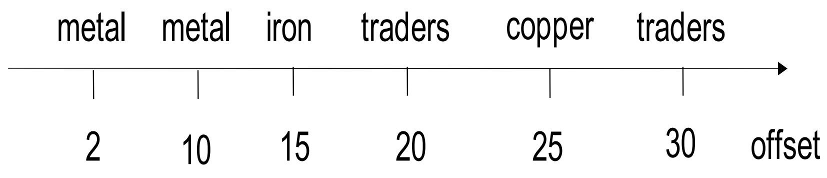


Figure 3.8 Example document for discussion of Proximal-Unique semantics

position 2, say *metal*^{1} and the other at position 10, say *metal*^{2}. The occurrences of *"traders"* are at positions 20 and 30, say $traders^1$ and $traders^2$ respectively. Either *metal*^{1} can be combined with $traders^1$, or *metal*^{2} with $traders^1$, or *metal*^{1} and *metal*^{2} can both be combined with $traders^1$ as occurrences of the combined pattern *"metal" FOLLOWED BY "traders"*. However, it makes intuitive sense to combine the closest occurrences, because closely occurring patterns are more likely to be of interest for a search as the correlation is measured in terms of proximity. Therefore the occurrence of *metal*^{1} is discarded and *metal*^{2} is combined with $traders^1$. In other words, occurrence of a pattern in a document supersedes its previous occurrence in the document while combining with another pattern. In the above example, *metal*^{2} acts as an *initiator* because it initiates the detection of the pattern, and $traders^1$ acts as a *terminator*, because its occurrence results in the pattern being detected.

Sub-patterns once used are not considered for further detection of another instance of the same pattern. For example, it does not make sense to combine *metal*^{2} with $traders^2$, because *metal*^{2} has already been used in the detection of another pattern. Combining *metal*^{2} with $traders^2$ 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.

As another example, suppose we want to find the occurrence of *("metal" FOLLOWED BY 'traders") NEAR ("iron" FOLLOWED BY "copper").* According to the semantics discussed above, *"metal" FOLLOWED BY 'traders"* occurs in the interval

(10, 20) and *"iron" FOLLOWED BY "copper"* occurs in the interval (15, 25). 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 the sub-patterns overlap each other. Hence, the pattern *("metal" FOLLOWED BY 'traders") NEAR ("iron" FOLLOWED BY "copper")* is **not** detected in the document, because intuitively, only sub-patterns that are disjoint are combined. The NEAR operator used here assumes non-overlapping (disjoint) semantics for detection of composite patterns, hence the above pattern is not detected. The operators of the system use the Proximal-Unique semantics to combine patterns to generate result sets.

3.2 Incremental Retrieval

In InfoSearch, the leaf nodes of the pattern detector receives **all** tuples from the index as a single set. Hence, even if the user is not interested in all occurrences of the pattern, all the tuples of the simple pattern are processed resulting in wastage of computational and memory resources. To avoid this, tuples can be fed to the PDG incrementally. If the required number of patterns have not been detected, more tuples can be fetched and processed.

In this section, we discuss various design alternatives considered for addressing this issue.

3.2.1 Round Robin Approach

In this approach, a fixed number of tuples of each simple pattern are fed in a round robin fashion. If the required number of patterns has not been detected, more tuples are fed until all patterns are detected or the input is exhausted. As an example, let us consider the query *"iron" FOLLOWED BY ("metal" NEAR/5 "traders").* The simple patterns here are "traders", "metal" and "iron". Initially, "n" tuples of "traders", "metal", and "iron" are fed. Once the tuples of "traders" and "metal" are available, the internal node *metal_near_traders* can process the tuples. If the operator has exhausted tuples of one of the operands, say *"metal"*, it needs to wait till the index interface fetches tuples for "iron" and "traders". It is possible that availability of more tuples of *"metal"* might result in the detection of the required number of patterns. In this approach, the number of patterns that were detected were not considered before more tuples are fed. A fixed number of tuples are fed irrespective of how many patterns were detected till all the required patterns are detected. This results in unnecessary processing of extra tuples. Though this approach is straightforward and good when the number of leaf nodes are small, it is not efficient as the number of simple patterns need to be looked up increases. The algorithm for this approach is described in 1.

Algorithm 1 Round Robin Approach

```
1: Input: List of leaf nodes, leafNodes
2: Input: Number of patterns to be detected, k
3: $node \Leftarrow leafNodes(0)$
4: $nodeToQuery \Leftarrow 0$
5: while true do
6:    if $nodeToQuery >= size\ of\ leafNodes$ then
7:       $node \Leftarrow leafNodes(0)$
8:    else
9:       if node is a internal node then
10:         while true do
11:            $nodeToQuery++$
12:            $node \Leftarrow node \rightarrow next$
13:            if node is a leaf node then
14:               break
15:    fetch k tuples of node
16:    propagate the tuples to the node
17:    if $nodeToQuery == -1$ then
18:       break
19:    else
20:       $nodeToQuery++$
21:       $node \Leftarrow node \rightarrow next$
```

3.2.2 Iterative Approach

In this approach, the number of patterns to be detected (“n”) are stored in the root node of the PDG. The root node can start the detection of patterns by requesting its child nodes to fetch “n” tuples. If the child node is an internal node, tuples of the left child are fetched followed by the ones from the right child. The internal nodes request their respective child nodes recursively till the required number of patterns are detected. The request for tuples is percolated down the PDG, till it reaches the leaf nodes. This is done because, only the leaf nodes can interact with the index interface module and request it to fetch “m” number of tuples. The number “m” is computed by pushing the “n” through the PDG and adjusting for FREQUENCY and other operator semantics. If an internal node corresponds to a FREQUENCY operator with associated frequency “d”, then for it to produce “n” tuples as output, it needs a minimum of “$n \times d$” tuples as input. Hence, the value of “m” is adjusted to be $n \times d$. Similarly, if the internal node corresponds to OR operator, then it requires a minimum of *n/2* tuples from each of its child nodes, to produce “n” tuples as output. Hence, the number “m” is adjusted to *n/2*. For internal nodes that corresponds to other operators, the value of “m” is set to be “n”, which is passed to the child nodes.

Let us consider as an example, the PDG described in the Figure 3.3. In the above example, the root node can request the WITHIN operator to fetch “n” tuples. Since WITHIN node is an internal node, it requests its left operand *copper* to fetch “n” tuples first. *copper* is a leaf node and requests the index interface module to fetch the tuples. The WITHIN node now has tuples from the left operator, it then requests the tuples from the middle operand. The middle operand is not an leaf node, so it requests its leaf nodes *metal* and *traders* to fetch the tuples. Next, it requests tuples from the right operand *scrap*. Once the tuples from all the operands are available, the WITHIN node processes the tuples and propagates the result set to the root. The root node recalculates the required number of patterns, and then the processing of tuples continues till the required number of patterns has been detected.

In the round robin approach, the number of tuples fetched is fixed and is independent of the number of patterns to be detected and operator semantics. This shortcoming is addressed in the iterative approach, where the number of tuples to be fetched is calculated based on operator semantics and the number of patterns to be detected. Once some patterns has been detected, the required number of tuples “n” to be detected is recalculated. The updated “n” is propagated down the PDG as described earlier. But in this approach, the flow of control for the detection of patterns is iterative, and does not lend itself to the data flow model used in the pattern detector for the detection of patterns. To address the above shortcomings, we propose a *hybrid approach* that imbibes the advantages of both, round robin and iterative approaches.

3.2.3 Hybrid Approach

Here, the detection of patterns is triggered by feeding, “m” tuples, of one of the leaf nodes. Once the PDG corresponding to the query is built, the number of patterns to be detected “n”, is set in the root node of the PDG. The number “m” is then calculated by pushing “n” down the PDG and adjusting it based on individual operator semantics. The calculation of “m” is described in the algorithm 3. Once the tuples reach the internal node, it can send a request to the index interface module indicating the tuples of the corresponding child node that are needed to continue processing. Additionally the internal node also specifies the number of tuples needed. When tuples for all child nodes are available, the internal node processes them and propagates the resulting set to the parents. If the merging of tuples does not result in any output that can be propagated up the PDG, then the internal node requests additional tuples from its child nodes (left to right manner). The child node can either be a leaf node or another internal node. The index interface can fetch tuples only for the leaf nodes. Hence, it has to translate a request for internal node by feeding tuples for one of lower most leaf nodes of the internal node. The output tuples of the internal node correspond to the resulting merged set obtained by processing of tuples from the child nodes based on the operator semantics.

Once the index interface module detects that there are no more tuples available for a leaf node, it notifies the internal node by sending a special tuple <-1,-1,-1,-1,-1,-1,-1>. When the internal node receives this special tuple, based on the semantics of the operator, it decides whether it has to continue further processing. For instance, the OR and SYN operator can continue processing of tuples as long as tuples of one of the child nodes are available. All other operators require tuples from all its child nodes to continue processing. Hence, in this case, on receiving the special tuple from its child node, it stops processing the input tuples, adds the special tuple to the merged result tuple set and propagates it to the parent. A progressively decreasing set of tuples that arrives at the root node, corresponds to the detected patterns. The root node then recomputes the number of patterns to be detected "n", by taking into account the number of patterns that were already detected. The number of tuples to be fetched "m" is recalculated based on this "n", and this value is propagated down the PDG. This processing continues till the required number of patterns has been detected. Once the root node detects that all the required number of patterns has been detected, a special request is sent to the index interface module, requesting it to stop feeding of tuples.

In contrast to the round robin approach, wherein fixed number of tuples are always fetched and processed, in the hybrid approach tuples are fetched only when necessary. Additionally, the data flow in the hybrid approach is bottom up and lends itself to the detection of the patterns using the PDG, compared to the detection in the iterative approach. The algorithm for this approach is described in 2.

Let us consider as an example, the PDG described in Figure 3.9. To detect "n" patterns of this query, the internal node FOLLOWED BY needs to produce "n" tuples as its output. Hence, the "m" value is set to "n" for the FOLLOWED BY node. The left child of the FOLLOWED BY operator node is an internal node and corresponds to the FREQUENCY operator with associated frequency value "10". For the internal node

Algorithm 2 Hybrid Incremental Retrieval Algorithm

```
1: Input: List of leaf nodes, leafNodes
2: Input: Number of patterns to be detected, k
3: $node \Leftarrow null$
4: $nodeToQuery \Leftarrow 0$
5: $minNoOfTuples \Leftarrow readfromconfigurationfile$
6: while $true$ do
7:     if $node == null$ then
8:         $node \Leftarrow leafNodes(0)$
9:     else
10:        if $node\ is\ a\ internal\ node$ then
11:            while $true$ do
12:                $node \Leftarrow node \rightarrow next$
13:                if $node\ is\ a\ leaf\ node$ then
14:                    $break$
15:        if $k < minNoOfTuples$ then
16:            $k \Leftarrow minNoOfTuples$
17:        $fetch\ k\ tuples\ of\ node$
18:        $propagate\ the\ tuples\ to\ the\ node$
19:        if $nodeToQuery == -1$ then
20:            $break$
21:        else
22:            $node \Leftarrow leafNodes(nodeToQuery)$
23:            $k \Leftarrow required\ number\ of\ tuples$
```

to produce “n” tuples as its output, it needs a minimum of $n \times 10$ tuples as input from its child node. Hence, the “m” value corresponding to the FREQUENCY internal node is set to $n \times 10$. The child node of the FREQUENCY node is an OR operator. For the OR node to produce $n \times 10$ as output, it needs a minimum of $(n \times 10)/2$ tuples from each of it child nodes *iron* and *copper*. Hence the value “m” is set to $(n \times 10)/2$ at the OR node. The algorithm for the computation of the value of “m” is described in 3.

For the query in the above example, the detection of patterns starts by feeding *minNoOfTuples* for one of the leaf nodes. This value is read from the configuration file. Let us assume that *minNoOfTuples* corresponding to the leaf node *traders* is fed first. The input tuples are propagated to its parent, the FOLLOWED BY node, which then

Algorithm 3 **Set required number of tuples**

```
1: Input: Number of tuples to be fetched, k
2: Input: Node for which the tuples has to be set, node
3: $leftNode \Leftarrow null$
4: $middleNode \Leftarrow null$
5: $rightNode \Leftarrow null$
6: $nodeType \Leftarrow node \rightarrow type$
7: $reqdNoOfTuples \Leftarrow k$
8: if $nodeType == leafnode$ then
9:    $return$
10: else
11:    if ( $nodeType == WITHIN$) OR ( $nodeType == NOT$) then
12:       $leftNode \Leftarrow node \rightarrow leftChildNode$
13:       $middleNode \Leftarrow node \rightarrow middleChildNode$
14:       $rightNode \Leftarrow node \rightarrow rightChildNode$
15:       $setRequiredNoOfTuples(leftNode, k)$
16:       $setRequiredNoOfTuples(middleNode, k)$
17:       $setRequiredNoOfTuples(rightNode, k)$
18:    if ( $nodeType == OR$) then
19:       $k \Leftarrow k/2$
20:       $leftNode \Leftarrow node \rightarrow leftChildNode$
21:       $rightNode \Leftarrow node \rightarrow rightChildNode$
22:       $setRequiredNoOfTuples(leftNode, k)$
23:       $setRequiredNoOfTuples(rightNode, k)$
24:    if ( $nodeType == FREQUENCY$) then
25:       $d \Leftarrow cardinality\ of\ the\ node$
26:       $k \Leftarrow k * d$
27:       $leftNode \Leftarrow node \rightarrow childNode$
28:       $setRequiredNoOfTuples(leftNode, k)$
29:    if ( $nodeType == NEAR$) OR ( $nodeType == FOLLOWEDBY$) then
30:       $leftNode \Leftarrow node \rightarrow leftChildNode$
31:       $rightNode \Leftarrow node \rightarrow rightChildNode$
32:       $setRequiredNoOfTuples(leftNode, k)$
33:       $setRequiredNoOfTuples(rightNode, k)$
```

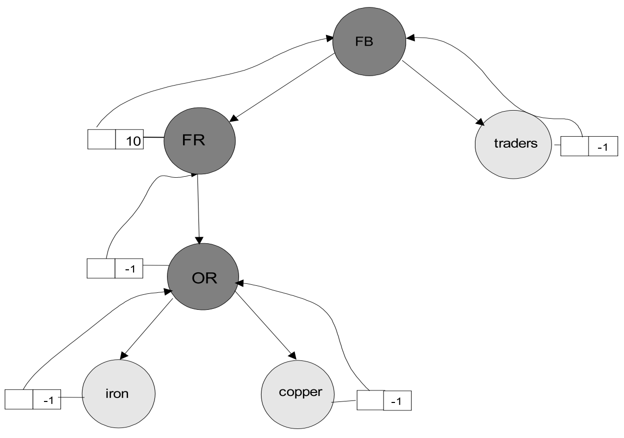


Figure 3.9 PDG for Query ((FRE/10 (“iron” OR “copper”)) FOLLLOWED BY “traders”)

checks if it has tuples from all the child node to process. It detects that tuples of the left child node are not available, hence it sends a request to the index interface module to feed “m” tuples of the OR node. OR is an internal node, hence the index interface fulfills the request by feeding tuples of its leaf child node *iron*. To continue further processing, it needs tuples of its right child node. Hence, it sends a request to feed “m” tuples of the leaf node *copper*. When the tuples of both of child nodes are available, the OR node processes the tuples and propagates the merged result set to the parent. The FREQUENCY operator then processes the input tuples and sends the output set to its parent FOLLOWED BY node. Now, the FOLLOWED BY node has tuples from both the child nodes. Hence, it processes the input tuples and sends the resulting output tuples to the root node. The output tuples produces by the FOLLOWED BY node, arriving at the root node corresponds to the detected patterns. The root node then recomputes the number of patterns left to be detected to satisfy the user requirements, and propagates it down the the PDG. It then sends a request to the index interface module to fetch “n” tuples of FOLLOWED BY node, which is translated to a request to the child node *iron* by the index interface. The changes needed to be made to different modules of the system for supporting incremental detection of patterns and their algorithms are described as follows:

3.2.4 Modification required in Graph Generator Module

In the InfoSearch system, a list of leaf nodes or simple patterns is maintained that is used by the index interface module to query the index and feed tuples of leaf nodes sequentially. This information is not sufficient to incrementally retrieve the tuples. The index interface module needs to know the structure of the PDG. This is required because the index interface needs to identify the leaf node for which are needed to be fed, to satisfy a request for an internal node by the pattern detector. Hence, the graph generator module needs to maintain information on the leftmost child nodes for each and every internal node in the PDG. Once the PDG corresponding to the user query is built, the required number of patterns to be detected is propagated down the PDG after adjusting the value based on user semantics.

To fetch information corresponding to a node, the index interface module has to traverse the entire list to retrieve the required information. This process is time consuming and inefficient since every time a request for tuples of a node arrives, the list has to be scanned. To reduce these scans, every node that is created in the PDG is assigned a unique identifier known as *node number*, in addition to the node name. The node number refers to the position in the list created by the graph generator, where information corresponding to that node is stored. Once the index interface knows the node number, it directly access the information for that node, thus avoiding the traversal through the entire list. The list created by the graph generator module and used by the index interface is termed as *pattern table.* It contains *node number*, *node name*, *left most child*, *index position* and *end of index* information for each node in the PDG and is described in Section 3.2.4.1.

Before processing, the internal nodes needs to check if tuples are available for all the child nodes. If they are available then a merging algorithm corresponding to the semantics of the internal node is called to process them. If tuples of a child node are not available, a request is sent to the index interface module. The request contains the node number for which tuples are required and the number of tuples need to be fetched.

Table 3.6 Pattern Table data structure

NodeNumber	**NodeName**	**LeftMostNode**	**IndexPos**	**EndofIndex**

If the node is a root node, it checks if all the required number of patterns has been detected. If the detected patterns is less than the required number of patterns to be detected, then it recalculates the additional number of patterns needed to be detected. It propagates this information through the PDG. If the required number of patterns has been detected, then it sends a message to the index interface module to stop feeding tuples. The corresponding algorithm is explained in 4

3.2.4.1 Pattern Table

The structure of the pattern table data structure is as shown in Table 3.6. When ever a leaf node or a internal operator node is created by the graph generator, an entry is added in the pattern table. Every node created in the PDG has associated unique identifiers *NodeNumber* and *NodeName*. For leaf nodes, which are simple patterns, as they do not have any child nodes, the value of *Left Most Child* is zero. For internal nodes, the value of the left most child in the graph, is stored. Initially, the index position is null , the index interface updates the index position for leaf nodes after querying the index. This information helps to avoid fetching of duplicate tuples, by starting the search for the tuples from the position where it has let off in the earlier run. Once the index interface discovers that there are no more occurrences in the index, it adds an entry "-1" in the pattern table for the corresponding leaf node. The index interface looks at this entry before accessing the index for retrieving tuples, if the entry is equal to "-1", then it creates a dummy tuple with document ID "-1" and notifies the leaf node to indicate that there are no more tuples in the index.

Algorithm 4 Process Tuples

```
1: Input: tuples of the left node, leftSet
2: Input: tuples of the middle node, middleSet
3: Input: tuples of the right node, rightSet
4: $k \Leftarrow$ number of patterns to be detected
5: $detectedPatterns \Leftarrow 0$
6: $node \Leftarrow currentNode$
7: $nodeType \Leftarrow node \rightarrow type$
8: $leftNode \Leftarrow node \rightarrow leftChildNode$, $middleNode \Leftarrow node \rightarrow middleChildNode$, $rightNode \Leftarrow node \rightarrow rightChildNode$
9: if $nodeType == rootnode$ then
10:    $detectedPatterns \Leftarrow$ size of middleSet
11:    if ( $detectedPatterns > k$) then
12:        return
13:    else
14:        $k \Leftarrow k - detectedPatterns$, $setRequiredNoOfTuples(middleNode, k)$
15: else
16:    if size of $leftSet > 0$ then
17:        if $nodeType$ neq $FREQUENCY$ then
18:            if size of $rightSet > 0$ then
19:                if ($nodeType$ eq $WITHIN$) OR ($nodeType$ eq $NOT$) then
20:                    if size of $middleSet > 0$ then
21:                        if ($nodeType$ eq $WITHIN$) then
22:                            process tuples using $WITHIN$ operator semantics
23:                        if ($nodeType$ eq $NOT$) then
24:                            process tuples using $NOT$ operator semantics
25:                    else
26:                        send $middleNode$ , $k$ to index interface
27:                else
28:                    if ($nodeType$ eq $FOLLOWEDBY$) then
29:                        process tuples using $FOLLOWEDBY$ operator semantics
30:                    if ($nodeType$ eq $NEAR$) then
31:                        process tuples using $NEAR$ operator semantics
32:                    if ($nodeType$ eq $OR$) then
33:                        process tuples using $OR$ operator semantics
34:            else
35:                send $rightNode$ , $k$ to index interface
36:        else
37:            process tuples using $FREQUENCY$ operator semantics
38:    else
39:        send $leftNode$ , $k$ to index interface
```

3.3 Ranking

In “InfoSearch” system, the detected patterns were delivered to the user, in the order of detection. The results delivered to the user, does not offer any information, regarding how “relevant” the detected pattern is, to the user query. Hence, the user needs to go through all the detected patterns. Therefore, there is a need to rank the detected patterns.

In our system, we cannot use any ranking algorithms that relies on link structures, as the input to the system is raw text data and does not contain any link information. Additionally, $“term\ frequency \times “inverse\ document\ frequency''$ used in vector space models of IR, cannot be used for ranking, because our intention is to find, relevancy of the detected patterns to the user query, rather than the document relevancy. Hence, we have come up with a simple ranking algorithm based on the intuitive relevance of patterns which is consistent with the proximal-unique semantics of the language. Hence the span across which the pattern is detected, which extends up on the work done in this area [23, 28, 22, 29, 30, 31], is used in our approach. Since the patterns are detected incrementally, the patterns are ranked *locally* as-and-when they are available. The ranked patterns are combined with the newly detected patterns, and are sorted again to produce combined ranked results.

Semantically and intuitive, a pattern detected in a single sentence is more likely to be relevant to the user query than a pattern detected across multiple sentences in the same paragraph. Similarly, pattern detected in a single paragraph is more relevant when compared to a pattern detected across the paragraphs in the same document. This intuitive reasoning is used to rank the detected patterns because closely occurring patterns are more likely to be of interest to the user. The detected patterns are first sorted based on the paragraph span *“end paragraph - start paragraph”*, across which the pattern is detected. Patterns containing the same paragraph span are further sorted, starting with the lowest value, on sentence span *“end sentence - start sentence”*, in ascending order. The sorted patterns with same sentence span, are sorted again based on the offset span *"end offset - start offset"*. The resulting patterns, sorted based on the decreasing order of relevancy, are then presented to the user. The corresponding ranking algorithm is described in 5.

Algorithm 5 Ranking algorithm

```
1: Input: Set of patterns, S
2: $S \Leftarrow sort\ tuples\ based\ on\ paragraph\ span$
3: $S \Leftarrow sort\ tuples\ based\ on\ sentence\ span$
4: $S \Leftarrow sort\ tuples\ based\ on\ offset\ span$
```

3.4 Summary

In this chapter, we described the architecture of the system and explained each of its modules. *Pattern parser and validator*, *pattern processor* modules were re-used from the InfoSearch system. We discussed the process of PDG construction by the *graph generator*, and how the distance information is handled. Additionally, we also explained how the sub-PDGs were shared between common sub-expressions. Furthermore, we also discussed the need for a new naming convention for the nodes in the PDG. The new naming convention which includes distance and containment information were described in detail. The *pattern detector* and *index interface* module were briefly described. We also described various approaches for incremental detection of patterns. **Round robin approach**, does not consider the number of patterns already detected while feeding tuples and the same number of tuples are fed at each leaf node in each round, resulting in unnecessary processing of extra tuples. Though the **iterative approach**, takes into account already detected patterns, before requesting more tuples from the index. But, the flow of control for the detection of patterns is iterative, and does not lend itself to the data flow model used in the pattern detector for the detection of patterns. Hence, an alternative approach, namely **hybrid approach**, is proposed that addresses the shortcomings of the previous approaches. In the hybrid approach, pattern detection is triggered by the feeding tuples of one of the leaf nodes. Once tuples are available at the internal nodes that corresponds to the operators, a request containing the tuples of which node to feed and the number of tuples to be fetched, is sent to the index interface module. We presented algorithms for each of the considered approached. An algorithm for calculating the number of tuples to be fetched for each leaf node based on the operator semantics and required number of patterns to be detected is explained. We also described an algorithm that interacts with the index interface module, in addition to the processing of tuples in detail. The modifications to be made for the *pattern detector* module to support incremental detection of patterns are described. Additionally, we presented the *pattern table* data structure, needed to enable fetching of tuples incrementally. We explained the *ranking algorithm* and also discussed why existing ranking algorithms cannot be utilized to rank the detected patterns. The operators that constitute the internal nodes of the PDG, are discussed next in Chapter 4.

CHAPTER 4

PATTERN SPECIFICATION LANGUAGE (PSL) OPERATORS

In this chapter, we present incremental algorithms for the following operators: OR, FREQUENCY, NEAR, FOLLOWED BY, SYN, WITHIN and NOT, supported by the system to allow users to specify more expressive queries. The semantics of these operators is as described in InfoSearch [5]. However, the working of operators is different from that of InfoSearch operators. In InfoSearch the entire result set corresponding to a pattern is propagated at once. However, here the input tuples are processed incrementally until the required number of tuples has been retrieved to detect user-specified number of patterns. Once the required number of PATTERNS have been detected the system stops retrieving and processing additional tuples.

The input to the operators are sets of tuples containing the document ID, start offset, end offset, start sentence, end sentence, start paragraph and end paragraph of the corresponding pattern. Each tuple represents occurrences of the corresponding pattern in the document collection. The input tuple sets are assumed to be *sorted in ascending order of document ID*. The operators processes the input tuple sets, tuple by tuple. Furthermore, the operators ensures that the tuples are merged based on proximal-unique semantics as discussed earlier. Tuples satisfying the semantics of the operator are merged and added to a result set which is then propagated to its parents or subscribers.

4.1 The OR operator

The input to the OR operator is two sets of tuples, sorted by document ID, corresponding to the left and the right operand. According to the semantics of the OR operator, a pattern is detected whenever either of the operands are detected. Therefore, the output of the OR operator is a union of its input sets. The output sets are 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. The position, where the pattern is detected within a document, is critical for the operators using the Proximal-Unique semantics. If there are no more input tuples available at one of the operands, the operator stops processing the tuples, and propagates the available result set to the parents. This is done because once the input set for an operand has been exhausted, there is no way for the operator to know if there is any other tuple of that operand that occurs before the tuple in the other operand. It is necessary for the operator to stop processing of tuples until the tuples are available at both the operands, to produce result sets that are sorted. Once the operator receives another set of input tuples, and when tuples are available at both the operands, it continues processing the tuples along with the ones that were not processed earlier. If the operator encounters a tuple with document ID "-1", then it indicates that there are no more tuples available for that operand. Hence, it appends the tuples from the other operand, that are yet to be processed to the result set and is propagated up to the parent. Essentially, the OR operator generates a sorted union of its input sets and produces a result set sorted on document ID and end offset of the tuples. The algorithm of the OR operator is in 6.

As an example, Figure 4.1 demonstrates the working of the OR algorithm. The operator merges these input tuples to generate an output set corresponding to *"iron" OR "copper"*, in which the tuples are the union of the input sets, sorted on document ID and end offset. The inputs are sets of tuples corresponding to the keyword *"iron"* and *"copper"*. The tuple D1<2,2,2,2,1,1>occurs before D1<3,3,4,4,2,2>, hence D1<2,2,2,2,1,1>is added to the result set,. Next D1<3,3,4,4,2,2>is compared with D1<5,5,6,6,2,2>, since D1<3,3,4,4,2,2>occurs before D1<5,5,6,6,2,2>, it is selected. Similarly D1<5,5,6,6,2,2>, D2<10,10,3,3,1,1>,, D2<11,11,4,4,1,1>are added to the result set. Now there are no more tuples left in the right operand *"copper"* but there is one more tuples D2<15,15,7,7,2,2>in the left operand *"copper"*. The OR operator cannot proceed with the processing of the tuples of *"iron"* operand because there might be tuples

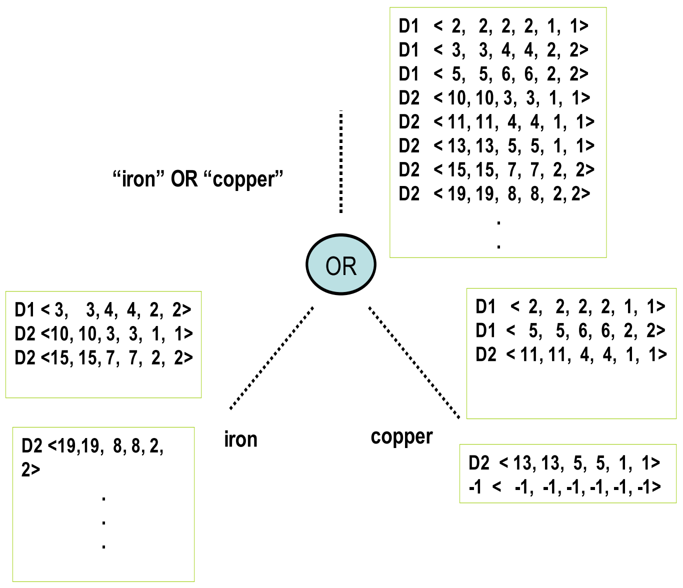


Figure 4.1 An example of the working of the OR operator

of *"copper"* that might occur before D2<15,15,7,7,2,2>. Therefore, it need to fetch more tuples of *"copper"* before processing more tuples. Once it gets another set input tuples for *"copper"*, the operator continues processing of the tuples. Tuple D2<13,13,5,5,1,1>of operand *copper* occurs before D2<15,15,7,7,2,2>, so it is selected. The next tuple -1<-1,-1,-1,-1,-1,-1>, of the right operand *copper*, has document ID "-1", which indicates that there are no more tuples of the operand *copper*. So the remaining tuples of operand *iron* are added to the result set and propagated to the subscribers.

4.2 The FREQUENCY operator

The FREQUENCY operator is an unary operator, it has a single set of tuples as its input. It is represented as *FREQUENCY/n(P)*, which means that all documents containing more than n occurrences of pattern P should be retrieved. The operator 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 result set. If the tuples at the operand are exhausted, the operator propagates the result set to its parent, and then requests the tuple feeder to fetch more tuples. If the operator encounters a tuple with document ID “-1”, then it indicates that there are no more tuples available for the operand. So it appends a dummy tuple <-1,-1,-1,-1,-1,-1>to the result set and propagates up to the parent. Using this operator, one can retrieve all documents containing equal to or more than n occurrences of the pattern P. The pseudocode for the merging done in the FREQUENCY operator is shown in 7.

Figure 4.2 shows an example of the working of the FREQUENCY operator. In the example, there is only one tuple for D1, so the tuple is discarded. When the counter for D2 becomes 3, the operator generates an output tuple, having the start offset, start sentence, start paragraph of the first of the three tuples, and end offset, end sentence, end paragraph of the last of the three tuples. After the counter for D3 reaches 2, the operator has exhausted all the tuples. So it propagates the outputSet to the parent and waits for another set of input tuples to arrive. In the new input set, tuple D3<27,27,4,4,3,3>exists, hence the counter for D2 is incremented to 3, and an output tuple is generated. After processing tuple D4<1,1,1,1,1,1>, it encounters a tuple with document ID “-1”, so it stops further processing of tuples and propagates the output set to the parent, after adding a dummy tuple <-1,-1,-1,-1,-1,-1>indicating that there are no more tuples of the pattern P to be retrieved.

4.3 The NEAR operator

The NEAR operator is a binary operator and it takes two input sets. It is specified as *P1 NEAR[/d] P2*. It retrieves documents containing occurrences of the complex pattern *P1* and *P2* within the same document, separated by a distance not greater than d words. The distance d is optional, and if it is not specified, the *NEAR* pattern is considered to be detected, if *P1* and *P2* occur anywhere within the same document. The relative order of occurrence of *P1* and *P2* is not important, *P1* may either follow or precede *P2*, but *P1* and *P2* may not overlap each other. Hence, either operand one can be the initiator or terminator. While processing the input tuples, the NEAR operator

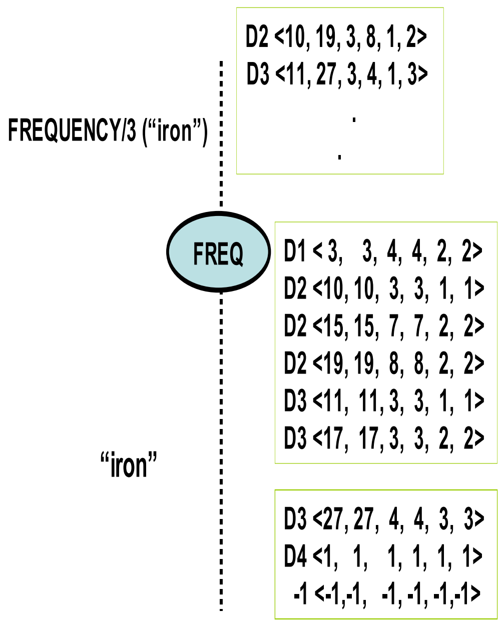


Figure 4.2 An example of the working of the FREQUENCY operator

has to decide whether the tuples are eligible for combination, and if not, decide which tuple to keep and which one to discard.

If the number of tuples available at the left or right operand is 1, and if the document ID of the tuples is not equal to "-1", the operator stops processing of the tuples, and waits till more tuples are available at both operands to continue processing. This is necessary, if the operator is to follow Proximal-Unique semantics. If the operator encounters a tuple with document ID -1, then the operator understands that, there are no more tuples available for that operand. Hence, the operator stops processing of the tuples. Furthermore, it adds a dummy tuple <-1,-1,-1,-1,-1,-1>to the output set and propagates it to the parent. The NEAR operator pseudocode is shown in NEAR:3. It shows how processing of the input sets is done as per the above discussion.

Figure 4.3 shows an example of the working of the NEAR operator. To begin, *initiator* points to D1 <10, 18,1,1,1,1> in the left set, and *terminator* points to D1 <28, 40, 3, 3, 1, 1> in the right set. Since the next tuple in the *initiator* set occurs 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, 2, 3, 1, 1>. When *initiator* and *terminator* point to D2 <12, 18, 2, 2, 2, 2> and D2 <15, 20, 2, 2, 2, 2> 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, 2, 2, 2, 2>ends before the next tuple from the left set D2 <30, 35, 3, 3, 2, 2>. Hence, D2 <21, 24, 2, 2, 2, 2> is made the new *terminator* and D2 <12, 18, 2, 2, 2, 2> is retained as the initiator. They are combined to form the output tuple D2 <12, 24, 2, 2, 2, 2>. Now, *initiator* points to a D2 tuple while *terminator* points to a D3 tuple. Hence, *initiator* is advanced. In this case, the initiator is the last tuple at the left operand. Hence, the operator stops merging of the tuples and propagates the output set to the parent. This is done, because the operator can't decide if there are any more tuples of left set that might occur after initiator. Merging the initiator-terminator pair without looking at more tuples from the input sets will result in incorrect results. Hence, the operator waits till further tuples are available. Once the new input set are available at left operand, the operator finds that the, *initiator* D3 <40, 47, 3, 3, 3, 3>lies completely after *terminator* D3 <12, 19, 1, 1, 1 ,1>. Hence, *initiator* and *terminator* are swapped. This makes *initiator* point to D3 <12, 19, 1, 1, 1 ,1> and *terminator* point to D3 <40, 47, 3, 3, 3, 3>, which form a proximal pair and are merged to give D3 <12, 47, 1, 3, 1, 3> in the output set. Finally, *initiator* points to D4 <12, 20, 2, 2, 2, 2>, and *terminator* points to D4 <30, 35, 4, 4, 2, 2>. Since the initiator is the last tuple at the operand, the operator waits for more tuples to arrive. Once the new input set is available, it finds that the terminator occurs completely before the next tuple D4 <60, 63, 5, 5, 2, 2>in the right set. Hence, the initiator and terminator are swapped to form the new proximal pair, and are combined to form the output tuple D4 <30, 63, 4, 5, 2, 2>. The initiator and terminator are advanced, and now points to the tuples D7 <1, 1, 1, 1, 1, 1 >and D5 <40, 70, 7, 7, 2, 2>respectively. Since the document ID of the terminator is less than that of the initiator, it is advanced. The next tuple in the right set has document ID "-1", indicating that there are no more tuples of the right operand. So the NEAR operator stops processing

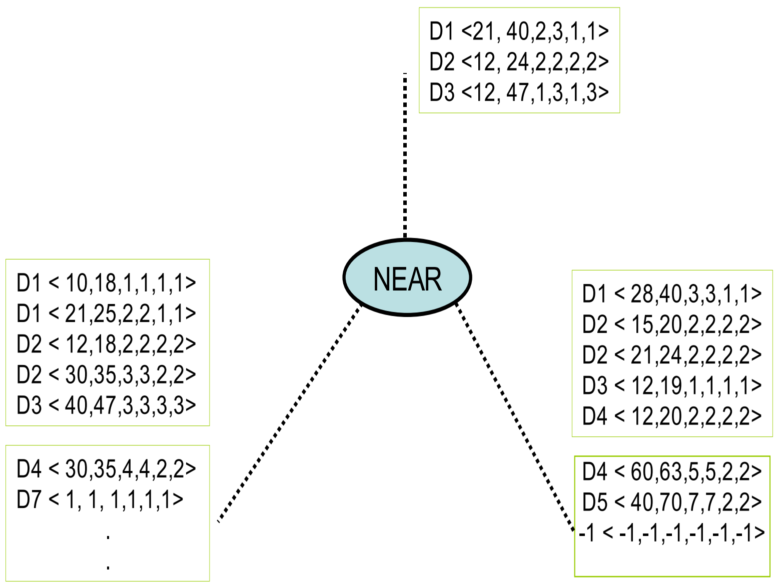


Figure 4.3 Example of the working of the NEAR operator

of the tuples and propagates the output set to its parent after adding a dummy tuple <-1, -1, -1, -1, -1, ,-1, -1 >.

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 both *P1* and *P2*, should be retrieved, with the restriction that *P1* should occur before that of pattern *P2*. The occurrences of *P1* and *P2* should not be separated by more than *d* words. The distance *d* is optional, and in its absence, any occurrence of *P1* followed by *P2* in a document should be retrieved, irrespective of the distance separating them. The inputs to the FOLLOWED BY operator are two sets of tuples corresponding to the left and the right operand. The occurrences of pattern *P1* acts as initiator and occurrences of *P2* acts as the terminator of the pattern *P1* and *P2*. According to the semantics of the operator, the left sub-pattern should occur before the occurrence of the right one, for the pattern to be detected. If the number of tuples available at the left operand is 1, the operator stops processing of the tuples, and waits till more tuples are available at the operand to continue processing. This is necessary, if the operators are to follow Proximal-Unique semantics. If the operator encounters a tuple with document ID -1, then the operator stops processing of the tuples. Furthermore, it adds a dummy tuple <-1,-1,-1,-1,-1,-1>to the output set and propagates the output set to the parent. The algorithm for the FOLLOWED BY operator is given in 9.

Figure 4.4 gives a simple example the input sets of operands *"metal"* and *"traders"*. To begin with, the left tuple D1<2,2,2,2,1,1>acts as an initiator and D1<10,10,4,4,2,2>from the right input set acts as a terminator. Since the next tuple in the initiator set, D1<7,7,3,3,2,2>occurs before the terminator, it is assigned as the new initiator. Now, *initiator* and *terminator* point to a proximal pair of tuples, and hence they are merged and added to the output set, as tuple D1<7,10,3,4,2,2>. Now D2<1,1,1,1,1,1>and D3<25,25,10,10,4,4>are the new initiator and terminator. The initiator document ID is less than that of the terminator, so D3<5,5,3,3,2,2>from the initiator set is made the new initiator. The next tuple in the initiator set occurs before the terminator, so that initiator is advanced to D3<10,10,8,8,3,3>. It is the last tuple in the initiator set, hence the operator can't decide, if there is any other tuple in the initiator set, that occurs before the terminator. Therefore, it stops processing of the tuples and propagates the output set to the parent. Once the operator receives, another input set for the operand *"metal"*, it again starts the processing of the tuples. There exists a tuple D3<15,15,9,9,4,4>in the initiator set, which occurs before the terminator, so it is made the new initiator. The tuples D3<15,15,9,9,4,4>and D3<25,25,10,10,4,4>are the new initiator-terminator proximal pair of tuples, and are combined to produce the output tuple D3<15,25,9,10,4,4>. The next tuple in the terminator set has document ID "-1", hence there are no more tuples of operand *"traders"* are available. So the operator stops processing and propagates the result set to the parents.

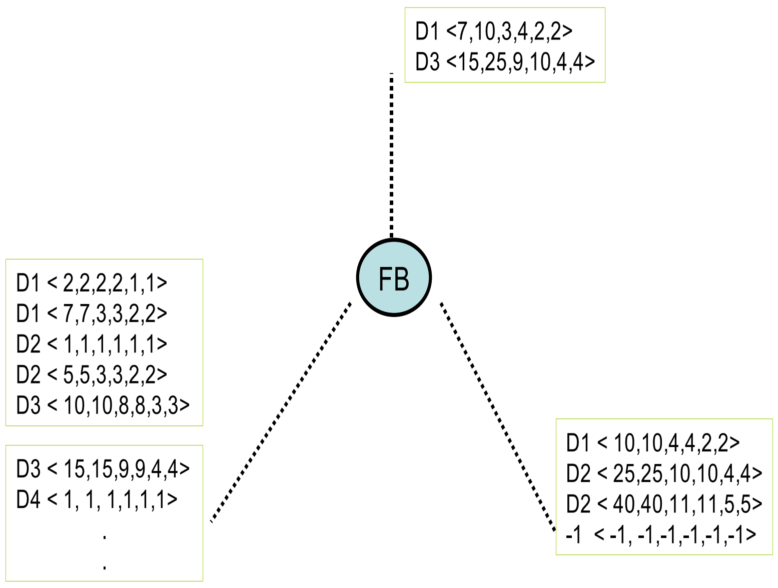


Figure 4.4 Example of the working of the FOLLOWED BY operator

4.5 The WITHIN operator

The WITHIN operator is a ternary operator, specified as *P2 WITHIN/[d](P1 , P3)*. This means that all documents containing *P1* followed by *P3*, with *P2* occurring at least *d* times in between should be selected. The frequency *d* is optional, and if it is not specified, the system assumes a default value of 1. The operator gets three sets of tuples as its inputs, corresponding to the left, middle and right operands. Similar to the FOLLOWED BY operator, only a tuple from the left set can acts as an *initiator* of the pattern. Furthermore, only tuples from the right set can act as a terminator. The operator has to check if there are any tuples from the middle set occurring between a tuple from the left set and right set, all having the same document IDs. The operator 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*, 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. If the number of tuples available at the left operand is 1, the operator stops processing of the tuples, and waits till more tuples of left set are available to continue processing. This is necessary, for the detection of patterns in Proximal-Unique semantics. If the operator encounters a tuple with document ID "-1", then the operator stops processing of the tuples. Furthermore, it adds a dummy tuple <-1,-1,-1,-1,-1,-1>to the output set and propagates the output set to the parent. The pseudocode is shown in 10.

An example of the working of the WITHIN operator can be seen in Figure 4.5. Let us assume that the required frequency of the pattern *P2* be 2. To begin with, the left tuple D1<3,3,1,1,1,1>acts as an initiator and D1<10,10,7,7,2,2>from the right input set acts as a terminator. Since the next tuple D1<5,5,2,2,1,1>in the initiator set occurs before the terminator, it is assigned as the new initiator. Now, *initiator* and *terminator* point to a proximal pair of tuples. Now the operator checks the middle input set. There is only one tuple with document ID 1, so the tuple is discarded, and the initiator and terminator are advanced. The tuple D2<1,1,1,1,1,1>is new initiator and D2<13,13,4,4,1,1>is the new terminator. Since there is no tuple in the initiator set that occurs before the terminator, the tuples D2<1,1,1,1,1,1>and D2<13,13,4,4,1,1>combine to form a proximal pair of tuples. Now the operator looks at the middle set, to check if there are any occurrences of the pattern between the initiator and terminator. The operator founds two non-overlapping tuples D2<4,4,2,2,1,1>, D2<8,8,3,3,1,1>with the same document ID, hence the initiator and terminator are combined to form the output tuple D2<1,13,1,4,1,1>. The tuples D2<25,25,8,8,3,3>and D2<40,40,10,10,5,5>are the new initiator and terminator. They combine to form a proximal pair. The operator then finds a tuple in the middle set with document ID -1, indicating that there are no more tuples of the middle operand. Hence, the operator stops the processing of the tuples and propagates the output set to the parent.

4.6 The NOT operator

The *NOT* operator is a ternary operator, specified as *NOT[/d](P2)(P1, P3).* It is used to specify a sequence of two patterns with the condition that a certain pattern does not occur between them. It receives three sets as inputs, corresponding to occurrences of

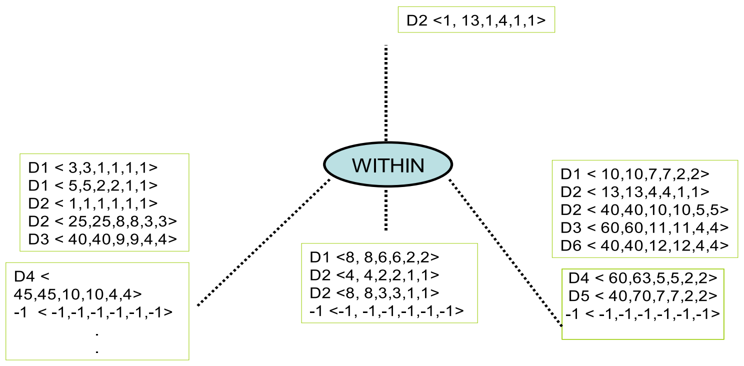


Figure 4.5 Example of the working of the WITHIN operator

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. Documents containing pattern *P1* followed by *P3*, with 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 not specified (default). 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. In other words, the pattern is not detected. The pseudocode is shown in 11.

An example of the working of the NOT operator can be seen in Figure 4.6. Let us assume that the required frequency of the pattern *P2* be 2. To begin with, the tuples D1<3,3,1,1,1,1>and D1<10,10,7,7,2,2>acts as an initiator-terminator pair. Since the next tuple D1<5,5,2,2,1,1>in the initiator set, occurs before the terminator, it is assigned as the new initiator. Now, *initiator* and *terminator* point to a proximal pair of tuples. Now the operator checks if there are 2 or more tuples in the middle input set having the same document ID, that occurs between initiator and terminator pair. Since there is only one tuple in the middle set, the initiator-terminator pair D1<5,5,2,2,1,1>and D1<10,10,7,7,2,2>are combined to form the output tuple D1<5,10,2,7,1,2>. The initiator and terminator are advanced, and the tuples D2<1,1,1,1,1,1>, D2<13,13,4,4,1,1>are the new initiator-terminator pair. Since the operator finds two tuples in the middle set occurring between the initiator-terminator pair, NOT becomes false for this pair. Hence the tuples are discarded and the pointers are advanced. The operator then finds another proximal pairs D2<25,25,8,8,3,3>and D2<40,40,10,10,5,5>. But the next tuple in the middle set has document ID -1 indicating that there are no more tuples of the middle operand. Hence, the operator produces another output tuple D2<25,40,8,10,3,5>. There is only one tuple D3<40,40,9,9,4,4>in the left set. The operator can't decide if there are any more tuples in the left set, that occur before the new terminator D3<60,60,11,11,4,4>. Hence, the operator stops processing of the tuples, and propagates the output set to the parent. Once the operator gets more tuples at the left set, it founds that the tuple D3<45,45,10,10,4,4>occurs before the terminator. Hence they are combined to form the output tuple. The next tuple in the left set has document ID -1, so the operator stops processing of the tuples.

4.7 The SYN operator

The SYN operator lets the user to search for synonyms of a keyword, in addition to the original word itself. It is specified by appending "[Syn]" at the end of the keyword, whose synonyms needs to be searched for. The graph generator creates leaf nodes for the keyword and for each of its synonyms during parse time, and a SYN node is created, which subscribes to these leaf nodes. The number of synonyms to be considered is a configurable parameter and currently only single word synonyms are supported. The system takes n input sets, and generates an output set, which is an union of input

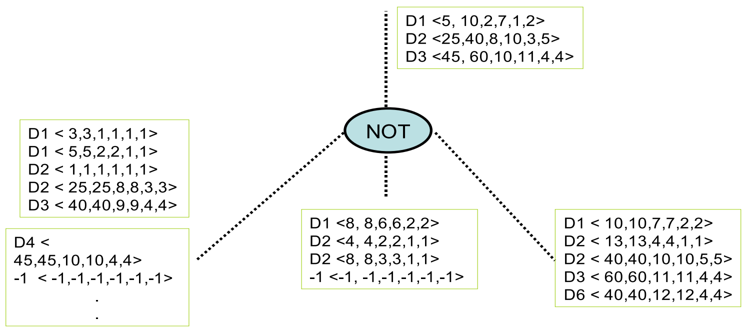


Figure 4.6 Example of the working of the NOT operator

sets, sorted by document ID and position. If the tuples are exhausted at any of the operands, the system stops processing of the tuples till next tuple set are available for that operand. This is done, because the subscribers of the SYN operator, expects the output set to be sorted, and the operator is not sure if there are any tuples of that operand, that occurs before other tuples. If the tuples available at all the operands has document ID "-1", it indicates that there are no more input tuples available. Hence, the operator adds a dummy tuple <-1,-1,-1,-1,-1,-1,-1>to the output set, and propagates it to the parent. The query *"CONTRACT"[Syn] NEAR ("BRITAIN" FOLLOWED BY "ADMINISTRATION"))* is shown in Figure 4.7. The algorithm for the SYN operator is described in 12.

4.8 Summary

In this chapter, we discussed the merging semantics of the operators. The operators OR, FREQUENCY, NEAR, FOLLOWED BY, WITHIN, NOT and SYN were presented along with their algorithms and examples. The operators take sets of tuples (not a complete set) as inputs from their children, where a tuple represents a single occurrence

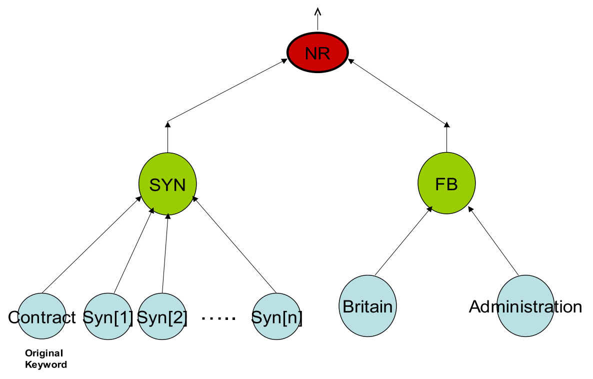


Figure 4.7 Synonym operator with 3 children

of the child pattern in the document collection. The operators merge the input sets according to the Proximal-Unique semantics, and generate a sorted set of tuples as their output, which is then propagated up to the parent for further processing. This process continues till the tuples reach the root node. The tuples available at the root node represents the detected patterns.

The key difference from the previous work is that the size of the sets of tuples given to leaf nodes (and propagated to intermediate nodes) can be arbitrary. The value is also determined by the system to minimize the total amount of tuples retrieved from the index and processed by the system.

Algorithm 6 OR Algorithm

```
1: OR ($leftSet$, $rightSet$)
2: $left \Leftarrow firsttupleinleftSet$, $right \Leftarrow firsttupleinrightSet$, $resultSet \Leftarrow \{\}$
3: while $exists(left)$ OR $exists(right)$ do
4:     if $left.docid == -1$ AND $right.docid == -1$ then
5:         $Add\ dummy\ tuple\ to\ resultSet,\ stop\ processing\ of\ Tuples$
6:     if $left.docid == -1$ then
7:         $resultSet \Leftarrow resultSet + rightSet,\ stop\ processing\ of\ Tuples$
8:     if $right.docid == -1$ then
9:         $resultSet \Leftarrow resultSet + leftSet,\ stop\ processing\ of\ Tuples$
10:    while $left.docid < right.docid$ do
11:        $resultSet \Leftarrow resultSet + left,\ left \Leftarrow left \rightarrow next$
12:        if $left.docid == -1$ then
13:            $resultSet \Leftarrow resultSet + rightSet,\ stop\ processing\ of\ Tuples$
14:    while $left.docid > right.docid$ do
15:        $resultSet \Leftarrow resultSet + right,\ right \Leftarrow right \rightarrow next$
16:        if $right.docid == -1$ then
17:            $resultSet \Leftarrow resultSet + leftSet,\ stop\ processing\ of\ Tuples$
18:    while $left.docid == right.docid$ do
19:        if $left.docid == -1$ AND $right.docid == -1$ then
20:            $Add\ dummy\ tuple\ to\ resultSet,\ stop\ processing\ of\ Tuples$
21:        if $(left.endOffset < right.endOffset)$ then
22:            $resultSet \Leftarrow resultSet + left,\ left \Leftarrow left \rightarrow next$
23:            if $left.docid == -1$ then
24:                $resultSet \Leftarrow resultSet + rightSet,\ stop\ processing\ of\ Tuples$
25:        else
26:            $resultSet \Leftarrow resultSet + right,\ right \Leftarrow right \rightarrow next$
27:            if $right.docid == -1$ then
28:                $resultSet \Leftarrow resultSet + leftSet,\ stop\ processing\ of\ Tuples$
29:        if $left == null$ OR $right == null$ then
30:            $stop\ processing\ of\ Tuples$
```

Algorithm 7 FREQUENCY Algorithm

```
1: FREQUENCY ($inputSet$, $n$)
2: $current \Leftarrow$ first tuple in $inputSet$
3: $previous \Leftarrow null$
4: $Integer\ startValue \Leftarrow -1$
5: $Integer\ startSentence \Leftarrow -1$
6: $Integer\ startParagraph \Leftarrow -1$
7: $outputSet \Leftarrow \{\}$
8: $Tuple\ outputTuple \Leftarrow null$
9: while $exists(current)$ do
10:     if $(current.docid == -1)$ then
11:         $Add\ dummy\ tuple\ to\ outputSet, stop\ processing\ of\ Tuples$
12:     if $current.docid \neq previous.docid$ then
13:         $count \Leftarrow 0$
14:         $startValue \Leftarrow -1$
15:         $startSentence \Leftarrow -1$
16:         $startParagraph \Leftarrow -1$
17:     if $startValue == -1$ then
18:         $startValue \Leftarrow current.startOffset$
19:         $startSentence \Leftarrow current.startSentence$
20:         $startParagraph \Leftarrow current.startParagraph$
21:     $count++$
22:     if $count == n$ then
23:         $outputTuple.docid \Leftarrow current.docid$
24:         $outputTuple.startOffset \Leftarrow startValue$
25:         $outputTuple.startSentence \Leftarrow startSentence$
26:         $outputTuple.startParagraph \Leftarrow startParagraph$
27:         $outputTuple.endOffset \Leftarrow current.endOffset$
28:         $outputTuple.endSentence \Leftarrow current.endSentence$
29:         $outputTuple.endParagraph \Leftarrow current.endParagraph$
30:         $outputSet \Leftarrow outputSet + outputTuple$
31:         $count \Leftarrow 0$
32:         $startValue \Leftarrow -1$
33:         $startSentence \Leftarrow -1$
34:         $startParagraph \Leftarrow -1$
35:     $previous \Leftarrow current$
36:     $current \Leftarrow current \rightarrow next$
```

Algorithm 8 NEAR Algorithm

```
1: NEAR$(L, R, d)$
2: $initiator \Leftarrow$ first tuple in $L$
3: $terminator \Leftarrow$ first tuple in $R$
4: while $exists(initiator)\ AND\ exists(terminator)$ do
5:     if $initiator.docid == -1$ OR $terminator.docid == -1$ then
6:         $Add\ dummy\ Tuple\ to\ resultSet, stop\ processing\ of\ Tuples$
7:     if $sizeof(left) == 1$ OR $sizeof(right) == 1$ then
8:         $stop\ processing\ of\ Tuples$
9:     while $initiator.docid < terminator.docid$ do
10:        $initiator \Leftarrow initiator \rightarrow next$
11:    while $initiator.docid > terminator.docid$ do
12:        $terminator \Leftarrow terminator \rightarrow next$
13:    if $initiator.docid \neq terminator.docid$ then
14:        if $initiator.docid < terminator.docid$ then
15:            $initiator \Leftarrow initiator \rightarrow next$
16:        else
17:            $terminator \Leftarrow terminator \rightarrow next$
18:        continue
19:    if $initiator.endOffset \leq terminator.endOffset$ then
20:        if $overlap(initiator, terminator)$ then
21:            $lookAhead(initiator, terminator)^1$
22:            continue
23:        if $initiator \rightarrow next.endOffset \leq terminator.endOffset$ then
24:            $initiator \Leftarrow initiator \rightarrow next$
25:        else
26:            if $(terminator.startOffset - initiator.endOffset) \leq d$ then
27:                $combine(initiator, terminator)$
28:                $initiator \Leftarrow initiator \rightarrow next$
29:                $terminator \Leftarrow terminator \rightarrow next$
30:            else
31:                $lookAhead(initiator, terminator)$
32:    else
33:        $swap(initiator, terminator)$
```

Algorithm 9 FOLLOWED BY Algorithm

```
1:  FOLLOWED BY($L$, $R$, $d$)
2:  $left \Leftarrow$ first tuple in $L$
3:  $right \Leftarrow$ first tuple in $R$
4:  while $exists(left)$ $AND$ $exists(right)$ do
5:      if (($left.docid == -1$) OR ($right.docid == -1$)) then
6:          $Add\ dummy\ tuple\ to\ resultSet, stop\ processing\ of\ Tuples$
7:      else
8:          if ($sizeof(L)$ == 1) then
9:              $stop\ processing\ of\ Tuples$
10:     while $left.docid$ < right.doc id do
11:         $left \Leftarrow left \rightarrow next$
12:         if ($sizeof(L)$ == 1) then
13:             $stop\ processing\ of\ Tuples$
14:     while $left.docid$ > right.doc id do
15:         $right \Leftarrow right \rightarrow next$
16:     if $left.docid \neq right.docid$ then
17:         $left \Leftarrow left \rightarrow next$
18:         $right \Leftarrow right \rightarrow next$
19:         if ($sizeof(L)$ == 1) then
20:             $stop\ processing\ of\ Tuples$
21:         $continue$
22:     else
23:         if $overlap(left, right)$ then
24:             $right \Leftarrow right \rightarrow next$
25:             $continue$
26:         if $left.endOffset < right.endOffset$ then
27:             if $(left \rightarrow next).endOffset < right.endOffset$ then
28:                 $left \Leftarrow left \rightarrow next$
29:             else
30:                 if $satisfiesDistance(left, right)$ then
31:                     $combine(left, right)$
32:                     $left \Leftarrow left \rightarrow next$
33:                 $right \Leftarrow right \rightarrow next$
34:         else
35:             $right \Leftarrow right \rightarrow next$
```

Algorithm 10 WITHIN Algorithm

```
1: WITHIN($L$, $R$, $M$)
2: $left \Leftarrow$ first tuple in $L$, $right \Leftarrow$ first tuple in $R$, $middle \Leftarrow$ first tuple in $M$
3: while $exists(left)\ AND\ exists(right)\ AND\ exists(middle)$ do
4:     if (($left.docid == -1$) OR ($right.docid == -1$) OR ($middle.docid == -1$) ) then
5:         $Add\ dummy\ tuple\ to\ resultSet,\ stop\ processing\ of\ Tuples$
6:     else
7:         if $sizeof(L)\ ==\ 1$ then
8:             $stop\ processing\ of\ Tuples$
9:     while $left.docid\ <\ middle.docid$ OR $left.docid\ <\ right.docid$ do
10:         $left \Leftarrow left \rightarrow next$
11:         if $sizeof(L)\ ==\ 1$ then
12:             $stop\ processing\ of\ Tuples$
13:     while $left.docid\ >\ middle.docid$ do
14:         $middle \Leftarrow middle \rightarrow next$
15:     while $left.docid\ >\ right.docid$ do
16:         $right \Leftarrow right \rightarrow next$
17:     if $overlap(left, right)$ then
18:         $right \Leftarrow right \rightarrow next,\ continue$
19:     if $left.endOffset\ <\ right.endOffset$ then
20:         if $(left \rightarrow next).endOffset\ <\ right.endOffset$ then
21:             $left \Leftarrow left \rightarrow next$
22:             if $sizeof(L)\ ==\ 1$ then
23:                 $stop\ processing\ of\ Tuples$
24:         else
25:             $count \Leftarrow 0$
26:             while $middle.docid\ ==\ left.docid$ do
27:                 if $middle.docid == -1$ then
28:                     $Add\ dummy\ tuple\ to\ resultSet,\ stop\ processing\ of\ Tuples$
29:                 if $middle.liesBetween(left, right)$ then
30:                     $count++$
31:                     if $count\ ==\ d$ then
32:                         $combine(left, right),\ middle \Leftarrow middle \rightarrow next,\ break$
33:                     $middle \Leftarrow middle \rightarrow next$
34:                 if $middle.liesAfter(right)$ then
35:                     $middle \Leftarrow middle \rightarrow next,\ break$
36:                 else
37:                     $middle \Leftarrow middle \rightarrow next$
38:             $left \Leftarrow left \rightarrow next,\ right \Leftarrow right \rightarrow next$
39:     else
40:         $right \Leftarrow right \rightarrow next$
```

Algorithm 11 NOT Algorithm

```
1: NOT($L$, $R$, $M$)
2: $left \Leftarrow$ first tuple in $L$, $right \Leftarrow$ first tuple in $R$
3: while $exists(left)\ AND\ exists(right)$ do
4:    if $((left.docid == -1)$ OR $(right.docid == -1))$ then
5:       $Add\ dummy\ tuple\ to\ resultSet, stop\ processing\ of\ Tuples$
6:    else
7:       if $sizeof(L)\ ==\ 1$ then
8:          $stop\ processing\ of\ Tuples$
9:    advance $left$, $right$ till they point to tuples with same doc id
10:    if $overlap(left, right)$ then
11:       $right \Leftarrow right \rightarrow next,\ continue$
12:    if $left.endOffset\ <\ right.endOffset$ then
13:       if $(left \rightarrow next).endOffset\ <\ right.endOffset$ then
14:          $left \Leftarrow left \rightarrow next$
15:       else
16:          if $sizeof(L)\ ==\ 1$ then
17:             $stop\ processing\ of\ Tuples$
18:       else
19:          $middle \Leftarrow$ first tuple in $M$
20:          $noMiddleTuples\ =\ false$
21:          while $middle.docid\ \neq\ left.docid$ do
22:             if $middle.docid\ >\ left.docid$ then
23:                $noMiddleTuples\ =\ true,\ break$
24:             $middle \Leftarrow middle \rightarrow next$
25:          if $noMiddleTuples$ then
26:             $combine(left, right)$
27:             $left \Leftarrow left \rightarrow next,\ right \Leftarrow right \rightarrow next,\ continue$
28:          $occurrenceCnt\ =\ 0$
29:          while $middle.docid\ ==\ left.docid$ do
30:             if $middle.liesBetween(left, right)$ then
31:                $occurrenceCnt++$
32:             if $(occurrenceCnt\ >\ d)\ OR\ (middle.liesAfter(right))$ then
33:                $break$
34:             $middle \Leftarrow middle \rightarrow next$
35:          if $occurrenceCnt\ \leq\ d$ then
36:             $combine(left, right)$
37:             $left \Leftarrow left \rightarrow next,\ right \Leftarrow right \rightarrow next,\ continue$
38:    else
39:       $right \Leftarrow right \rightarrow next$
```

Algorithm 12 SYN Algorithm

```
1: Input: $n$ sets corresponding to base keyword and $n - 1$ synonyms
2: for each set $S$ do
3:     $currentTuple(S) \Leftarrow$ first Tuple in $S$
4: $minTuple \Leftarrow currentTuple(0)$
5: $min \Leftarrow 0$
6: while $exists(minTuple)$ do
7:     for $(i = 0 ; i < n, i \neq min ; i++)$ do
8:         if $currentTuple(i).docid == -1$ then
9:             $tuples\ exhausted \Leftarrow true$
10:        else
11:            $tuples\ exhausted \Leftarrow false$
12:        if $minTuple.docID < currentTuple(i).docID$ then
13:            $continue$
14:        if $minTuple.docID == currentTuple(i).docID$ then
15:            if minTuple.endOffset < currentTuple(i).endOffset then
16:                $continue$
17:            else
18:                $minTuple \Leftarrow currentTuple(i)$
19:                $min \Leftarrow i$
20:        if $minTuple.docID > currentTuple(i).docID$ then
21:            $minTuple \Leftarrow currentTuple(i)$
22:            $min \Leftarrow i$
23:    $resultSet \Leftarrow resultSet + minTuple$
24:    if set $min$ has more tuples then
25:        $minTuple \Leftarrow$ next Tuple in $min$
26:    else
27:        arbitrarily assign current tuple of some unexhausted set to $minTuple$
28:        assign that set number to $min$
29:    if $tuples\ exhausted == true$ then
30:        $stop\ processing\ of\ Tuples$
```

CHAPTER 5

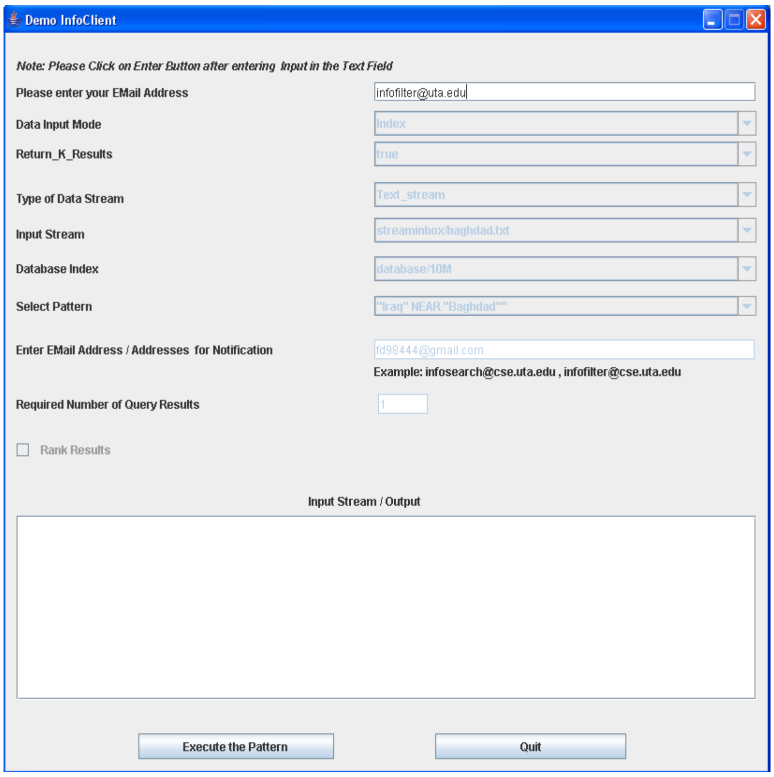
IMPLEMENTATION OF INFOSEARCH

This chapter describes the implementation aspects of various system modules in InfoSearch namely, *graph generator*, *index interface*, *pattern detection engine* and the *notifier* module. The additional data structures needed for incremental detection of the patterns has been described in detail. In addition we also discuss data structures that are used to pass information up the Pattern Detection Graphs (PDGs) and the corresponding changes made to the operators.

The implementation of InfoSearch system is integrated with InfoFilter, and the two systems share some common modules. *Pattern input client*, and *pattern validator and processor* modules are same in both systems. Since the current system is an extension of InfoSearch, it should be able to run in "InfoFilter", "InfoSearch" mode, in addition to the mode to incrementally detect and retrieve the patterns. The user inputs determine the mode for the system to run. A snap shot of the user interface is shown in the Figure 5.1.

5.1 User Input

In the previous work, the mode in which the server runs is specified in the configuration file and is decided during system startup. Once the system is started in one mode, it can't be changed to detect patterns in another mode. The only way to change it, is by changing the parameter in the configuration file and restarting the server. In the current system, this shortcoming has been fixed. The user can detect patterns from static or dynamic data sources or choose to incrementally retrieve the patterns by choosing appropriate inputs in the user interface. The user can specify to the system, to incrementally detect and retrieve the patterns by selecting "Index" as *Data Input mode*,"true" for *Re-*



Note: Please Click on Enter Button after entering Input in the Text Field

Please enter your EMail Address	infofilter@uta.edu
Data Input Mode	Index
Return_K_Results	true
Type of Data Stream	Text_stream
Input Stream	streaminbox/baghdad.txt
Database Index	database/10M
Select Pattern	"Iraq" NEAR "Baghdad""
Enter EMail Address / Addresses for Notification	fd98444@gmail.com
	Example: infosearch@cse.uta.edu , infofilter@cse.uta.edu
Required Number of Query Results	1

☐ Rank Results

Input Stream / Output

Figure 5.1 User Interface

turn_K_Results in addition to *Database Index*, *pattern* and *email for notification*. The user also needs to specify how many patterns needs to be detected by entering a value for *Required Number of Query Results* field. If 'false" is selected for *Return_K_Results*, then the system runs in the "InfoSearch" mode. In this mode, the server returns the detected patterns to the user at once. If the user wants to detect patterns from *dynamic data sources*, he can choose "Stream" for *Data Input mode*. The user also needs to specify *Type of Data Stream*, *Input Stream* in addition to *pattern* and *email for notification* information. Additionally, the user needs to enter his "mail id" in *Please enter your EMail Address* field. This information is used by the system to create a named ECA Agent [27]. Once all the required inputs has been entered, the query is sent to the server for detection.

5.2 Pattern Parser, Validator, and Processor

Once the server receives user input, it unwraps the input, and initializes various systems configuration parameters based on the values of *Data Input mode* and *Return_K_Results*. The user query is then passed to the *pattern parser* module. The *pattern parser* implemented using JavaCC parser [32], extracts the tokens from the user query. The tokens include simple patterns (words, system defined patterns), operators and other delimiters allowed by the language. If the query does not conform to the specifications of the Pattern Specification Language (PSL), a Java Exception called *parse exception* is thrown. The extracted tokens are passed to the *pattern validator* module, it enqueues the tokens in infix notation and passes it to the *pattern processor* as input. The server also initializes the notifier module and passes the user input information. Let us consider as an example, the query *("STUDENT" NEAR/3 (FREQUENCY/2 ("GRADUATE" FOLLOWED_BY "ADMISSIONS")))* and lets assume the user wants "10" patterns to be detected.

The *pattern processor* takes the input in infix notation and converts it into postfix notation preserving the precedence of tokens and operators as specified in the user query. The postfix notation allows for easier processing of operands. Additionally, generation of a graph from postfix notation is easier than from the infix notation. The *pattern processor* sends the stack of tokens in postfix notation to the *graph generator* for further processing. The postfix notation for the above example is *["STUDENT", [["GRADUATE", "ADMISSIONS", FOLLOWED_BY], 2, FREQUENCY], 3, NEAR].*

5.3 Graph Generator

The Event Specification and Detection framework called the Local Event Detector (LED), that has been used in InfoSearch system, was adopted with some modifications. The *graph generator* uses the Event Specification API of the LED [27] to generate the PDGs. The API provides methods for the creation of leaf nodes, which correspond to leaf nodes in a PDG. It also provides methods for creation of internal nodes, which correspond to internal or parent nodes in a PDG. The *graph generator* pops a token from the top of its input stack, and depending on the type of token, calls the appropriate LED API to create that particular node. If the token is a simple pattern or keyword or a system defined pattern, a leaf node is created for the token. The node is named after the keyword or a system defined pattern that it corresponds to. If the token is an operator, a internal node is created by the the *graph generator*, which subscribes to the children of the operator. The children of a internal node can either be a leaf node or other internal nodes. The internal node is named uniquely, and is derived from the operator name, distance with which it subscribed to its children and the names of the children. The names of these nodes are stored in a hash table, with a reference to the node.

In addition to the node name, a node number is assigned to each node. Whenever a node is created, an entry is added into the pattern table. The information stored in the pattern table includes node name, node number and the left most node. For leaf nodes, a value of “0” is stored for “left most node ” field, as they do not have any child nodes. For a internal node, the node number of the left most leaf node of the sub-PDG is stored is stored in “left most node ” field. The node name of the leaf nodes are same as the simple patterns, hence they are used by the index interface module to look up for the “hits” in the index. For internal nodes, the node name is big and complex and there is no use for it. Hence, a string “NULL” is stored, for internal nodes as “node name”. Further more, the “required number of patterns” to be detected, specified by the user, is stored for each node in the pattern table. If the nodes being created were to be subscribed by internal node FRE, then “required number of patterns” is multiplied by the frequency and used.

The graph generator checks the node names hash table to see if a node with the same name already exists. If the node exists, then a reference is obtained and is used to represent the node. If the node does not exist in the hash table, a new node is created and an entry is made into the hash table. This allows for sharing of the sub-PDGs resulting in efficient utilization of resources and has been explained in detail in Chapter 3.

When the graph generator receives the stack as input, it pops the token at the top of the stack and examines it. In the example, the popped token "NEAR", is a system operator, so it knows the next tokens are either distance followed by the operands or operands if no distance information is specified. The token "3" is popped and finds that token represents the distance. The next token is a right operand of "NEAR" operator. On examination, the graph generator finds the token is a stack and hence represents a sub-PDG. The token on top of the stack is popped, and it finds the system operator "FREQUENCY". The unary operator "FREQUENCY" has one operand or child and has a frequency information associated with it. So the graph generator expects the next token to be frequency information and the next token to be its operand. It finds the frequency information to be "2". The *required no of tuples* is multiplied by frequency "10", hence becomes "20". The next token corresponding to the operator is a stack, hence represents a sub-PDG. The token at the top of the stack is popped, and on examination founds the token to be a system operator "FOLLOWED_BY". It knows that the operator can have a distance information associated with it, if there is no distance associated then the next two tokens represents the child nodes. The next token in the stack "admissions" is popped and is found to be a simple pattern. Since the token "admissions" is a simple pattern, the graph generator tries to create a primitive event representing the leaf node of the PDG using the APIs provided by the LED framework. Before creating the node, the node name hash table is checked if a node with the same name already exists. Since it does not exist, a primitive node with the name *admissions* is created representing the simple pattern "admissions". A entry is added to the pattern table for this node. Since this is the first node to be created the node number of the node is "1". The node *admissions* is a leaf node, and does not have any child nodes, hence a value of "0" is stored for "left most node" entry. The value "20" is stored as the *required no of tuples*in the pattern table for the node *admissions*. Next token *graduate* is a simple pattern, hence a leaf node is created. Now both the operands of the system operator are created, so a internal node representing the "FOLLOWED BY" is

Table 5.1 Pattern Table for the Query "student" NEAR/3 (FREQUENCY/2 ("graduate" FOLLOWED_BY "admissions"))

Node Number	**NodeName**	**LeftMostNode**	**IndexPos**	**EndofIndex**
1	admissions	0		
2	graduate	0		
3	NULL	2		
4	NULL	2		
5	student	0		
6	NULL	5		

created with the name *graduate_followed_by_admissions* next. It subscribes to the leaf nodes *admissions* and *graduate*. The node created is a internal node, hence has a child node. The left most node of the internal node is *graduate* with node number "2". So "2" is stored as the entry in the left most node. There is no distance information associated with the "FOLLOWED BY" node, so a value of "-1" is stored in the subscriber list. The node has an node number "3" associated with it. For leaf nodes, the node name is same as simple pattern and hence used by index interface module to lookup the tokens. Since the node *graduate_followed_by_admissions* is a internal node, the index interface does not need this information and hence "NULL" is stored as node name. Next a internal node corresponding to "FREQUENCY" operator is created, with node name *FREQUENCY[2]:graduate_FOLLOWED_BY_admissions*. It subscribes to its child, internal node *graduate_followed_by_admissions*. The node number corresponding to the node *FREQUENCY[2]:graduate_FOLLOWED_BY_admissions* is "4". The left most node for the internal node, is the left mode node of its child, *graduate_followed_by_admissions*. Hence, the graph generator looks up the pattern table and finds that node "2" is the left most node for *graduate_followed_by_admissions*, and is stored as "left mode node" for the internal node *FREQUENCY[2]:graduate_FOLLOWED_BY_admissions*. The node corresponding to node name "6" has node name *student_NEAR_(FREQUENCY[2]:(graduate_FOLLOWE*
The information stored in the *node table* for each node, is shown in Table 5.1

The *pattern table* data structure is created only when server is running in the mode to incrementally retrieve patterns. If the server is running in InfoSearch mode, then a keyword list consisting of simple patterns is created as described in [5]. Also, every PDG constructed in the system is encapsulated under a *WITHIN(BeginIndex, EndIndex)* as described in [5].

5.4 Index Interface

The index interface of InfoSearch system was adopted after modifying it to feed the tuples incrementally. It receives the data structure pattern table from the graph generator module as input. The index interface looks at the pattern table, and retrieves the node corresponding to the node number "1". Since the node with node number "1" is the first node to be created in the PDG, it corresponds to a leaf node, hence simple pattern. The index interface also retrieves the "node name" and "required number of tuples" that corresponds to node "1" information from the pattern table. It checks if the value in the "required number of tuples" is greater than "minNoOfTuples" configuration parameter. If true, then then number of tuples to be fetched is equivalent to the value of "required number of tuples", else "minNoOfTuples" will be fetched. It then looks up the index for the occurrences of the node *admissions*, the node with node number "1", starting at the beginning of the index. Every time a "hit" is detected, the entry is fetched from the index. The hit contains document ID, offset, sentence and paragraph information. It wraps the information in the form of tuple <document ID, start offset, end offset, start sentence, end sentence, start paragraph, end paragraph >. Since the node corresponds to a leaf node the start offset is same as end offset, start sentence is same as end sentence and start paragraph has same value as that of end paragraph. The tuple is then added to a vector. Every time, it detects a hit, it increments the counter and checks if the required number of tuples has been fetched. If the required number of tuples has not been fetched, it continues with the retrieval. Else, the index interface stops fetching the tuples, copies a reference to the index position and stores in the pattern table. This is done, so that the next time, the index interface need not fetch the tuples from the beginning of the index, and hence avoid fetching of duplicate tuples.

Once the index interface module fetches the required number of tuples, it notifies the leaf node, corresponding to that leaf pattern. Once the tuples reaches the internal node "FOLLOWED BY", it finds that it does not have the tuples of the lead node. Hence, it calls *setNodeInfo* method in the index interface module with the node number of the left child and the required number of tuples. The index interface module, looks up the pattern table corresponding the node number. It finds that the node corresponds to a child node, and also there is no index information available. Hence, it starts looking for the occurrence of the pattern from the beginning of the index. Once the required number of tuples has been fetched the leaf node is notified. Additionally, it also stores the index position for that node in the index table.

As the tuples for both the child nodes are available, the internal node starts processing the tuples as per the semantics explained in previous chapter 4. The resulting tuples if any are added to result vector. Once the processing is done, it checks the size of the vector. If the size of the vector is greater than "0", the result vector is propagated up to the parent node "FREQUENCY", else the "FOLLOWED BY" node calls the *setNodeInfo* method requesting it to feed tuples of its left child *graduate* with node number "2". Once the result vector reaches the "FREQUENCY" node, it process them, and sends the resulting tuples if any to the parent. If there are no tuples to propagate, it calls *setNodeInfo* of the index interface to send tuples of it child with node number "3".

The index interface on receiving request to fetch tuples of node number "3", looks up the node table for more information. It finds that node with node number "3" corresponds to a internal node. Hence, it looks up the value in the "left most node" and finds that node "2" is the left most node. The information pertaining to node "2" is fetched from the pattern table. The index interface finds that index position is available, it retrieves the reference, and starts searching from the referred position, instead of from the beginning of the index.

If the index interface reaches the end of the index while searching for tuples, it stores "-1" in the pattern table for that node, to indicate that there are no more tuples to be found for that node. When the index interface receives a request to fetch more tuples for an node, it first checks if there is "-1", stored in the pattern table. If it finds a "-1", then the index interface creates a dummy tuple <-1,-1,-1,-1,-1,-1>and notifies the leaf node.

Once the required number of patterns has been detected, the root nodes requests the index interface module to send tuples of node with node number "-1". Once it receives such a request, the module understands that required number of patterns has been detected, and hence stops fetching tuples and returns the control to the graph generator module.

5.5 Pattern Detection Engine

The *pattern detection engine* process the tuple sets received from the index. It is done over the PDG, every leaf node corresponds to a simple pattern. The internal nodes of the PDG corresponds to one OR, NEAR, FOLLOWED BY, WITHIN, NOT, FREQUENCY or SYN operators, hence incorporates their logic for processing the tuples. The leaf node receives a reference, to a vector of tuples corresponding to the simple pattern. The leaf node copies the tuples in the vector, into another vector and passes the reference to the parent. Once the internal node, receives the references from the child nodes, it process the tuples from the received vector. Once a tuple has been processed, it is removed from the vector. Once the leaf node receives more tuples, it appends the received tuples, to the tuples that were not processed earlier and the reference is passed to the parents. Once the root nodes receives the tuples, it checks if the required number of tuples has been detected, if it does, then it send a request to the index interface to feed tuples of node with node number "-1", indicating to stop feeding of tuples. If the required number of tuples has not been found, it deducts, the number of detected patterns from the required no patterns, and propagates it to the child nodes recursively. Additionally, it also instructs the index interface to feed tuples of its child node. When the root node receives tuples, it executes a rule, which passes the vector of the detected tuples to the notifier module along with the node name of the root node.

5.6 Notifier

The notifier looks up its data structure, and extracts information corresponding to the root node. The information includes the query entered by the user, e-mail IDs to which notification has to be sent regarding the detection of patterns. The notifier extracts the tuples from the vector, and access the inverted index, to fetch the document name corresponding to each document ID. It translates the document IDs to the corresponding document names. If the user has requested the detected patterns to be ranked, then a ranking algorithm is run on the detected patterns, and the ranked patterns is then delivered to the user.

5.7 Ranking

The ranking algorithm receives tuples as a input vector. It extracts tuples from the vector, and calculates the paragraph span, *end paragraph - start paragraph*, across which the pattern is detected. It then inserts it in map object with the paragraph span as key and tuple as the corresponding value. Once the sorting of tuples by paragraph span is done, it sorts them based on sentence span, *end sentence - start sentence.* It extracts values corresponding to each key, and calculates the sentence span for each tuples. The tuples are then inserted in a map object, with sentence span as key, and tuples as value. When all tuples are processed, the map objected is again sorted based on offset, *end offset-start offset.* The processed tuples are inserted in a map object, with offset span as the key and the tuples as the value. This map object contains tuples that are ranked. The notifier extracts the tuples from each object and delivers them to the user in order.

5.8 The Inverted Index

To create the inverted index, the Java program *DocumentIndexer* from InfoSearch [5] was used with modifications. It takes a directory as input, reads directories and documents, and builds an inverted index over those documents. From every document, it creates a document ID – a numeric integer, and stores a mapping from the document ID to document path in a separate database. The document ID is used for all processing, because the computations done with numbers is computationally cheaper then comparisons with strings. It maintains a paragraph counter, sentence counter and offset counter to keep track of the positional information.

It reads a paragraph of data from a document at a time. The read data is then passed as input to the *getSentenceInstance()* method of the "BreakIterator" class, which creates an iterator with sentence-breaks. The iterator is used to extract individual sentences from the paragraph. The sentence counter is incremented after processing every sentence. Each sentence is then passed to the *getWordInstance()* method, and a iterator for word-breaks is created. Each word is extracted from the iterator and converted into text. The extracted word is stored as a "hit" in the inverted index. The offset counter is incremented after every word is extracted and stored. Each hit stored in the index, contains document ID, offset, sentence and paragraph information. Once all the paragraph has been processed, the counters paragraph, sentence and offset counters are reset. How the tuples are stored in the index, and how they are retrieved from the index given a simple pattern, is explained in detail in [5].

5.9 Summary

In this chapter, we discussed the implementation details of various modules of the system. Modules like *pattern validator* and *pattern processor* were adopted completely from the previous system InfoSearch. A new user interface has been developed using *Java Swing* technology, to overcome the shortcomings of the previous system. The *graph generator* module was modified extensively, and a new naming convention for internal nodes has been developed to handle sharing of sub-pdg corresponding to sub-patterns. Additionally, it was also modified to create a data structure *pattern table* for every node that is created. The *index interface* module was modified to take, an pattern table as input, and feed tuples incrementally to the PDG. The ranking algorithm was incorporated into the *notifier* module, which sorts the detected patterns based paragraph span of the detected pattern, then sentence span followed by the offset span.

CHAPTER 6

EXPERIMENTAL EVALUATION

The design and implementation of the system were discussed in earlier chapters. In this chapter we explain the experiments, data sets and analyze the results that were obtained. The experiments were conducted on a machine running Redhat Enterprise Linux Application Server 4 with four dual core AMD Opteron 2GHz processors and 4GB of RAM per processor using data sets of sizes 10MB, 25MB, 50MB, 100MB, 150M, 200MB and 250 MB.. The maximum allowed heap for Java runtime was kept at 3.5GB. The Java version used is 1.5, update 11. In the following sections, we will explain the nature of the data sets and the complex queries used.

6.1 Data Set

The data used for the experiments are taken from Reuters-21578 [33] and NSF Research Awards [34] text collections. Since the size of the Reuters data set is 27MB, we incorporated the text collections from NSF research awards to show the scalability of the system. We could have used just the NSF data, since its size is 650 MB, but the data content does not generate enough number of patterns to show clear performance gains of incremental search over InfoSearch.

6.2 Experiments

The experiments were run using few complex queries containing different operators and varying number of operands. The patterns were selected in a way, such that the number of detected patterns does not vary significantly between data sets of different sizes. To achieve this, initially we have selected 7MB of data from Reuters and 3 MB from NSF to form the 10MB data set. To this data, we have added more data incrementally, such

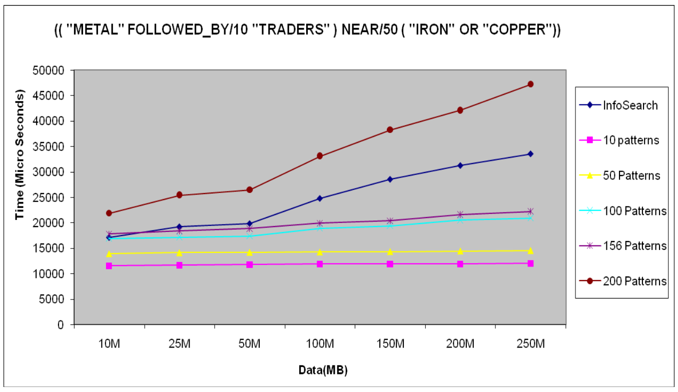


Figure 6.1 Time Analysis for query (“METAL” FOLLOWED_BY/10 “TRADERS”) NEAR/50 (“IRON” OR “COPPER”)

that the addition of the new data does not result in the detection of large number of new patterns. The experiments were run using complex queries involving a combination of operators, ((“metal” FOLLOWED_BY/10 “traders”) NEAR/50 (“iron” OR “copper”)), ((“damping”) NOT/2 ((“spring” FOLLOWED_BY “stiffnesses”) , “coefficients”)), and (“tax” FOLLOWED_BY (“petrol” OR “oil”)) NEAR (“retail” FOLLOWED_BY “stations”). The experimental results for the first two queries are discussed in the following sections.

6.2.1 Query1

The experiments were run with the query (“metal” FOLLOWED_BY/10 “traders”) NEAR/50 (“iron” OR “copper”). There are 156 occurrences of the pattern in the data set. The time taken to process all the tuples, to detect 156 patterns using the InfoSearch were calculated. Additionally, we also measured the total memory consumed by tuples of the operand that constitute the complex pattern, during the detection of patterns is also calculated. We compared those results, with the those taken to detect 10, 50, 100, 156 and 200 patterns using our system.

From Figure 6.1, we can see that the time taken to detect 10, 50, 100 and 156 patterns were significantly less than the time taken by the InfoSearch system. We would have expected the time taken to detect 156 patterns will be same or more than that of

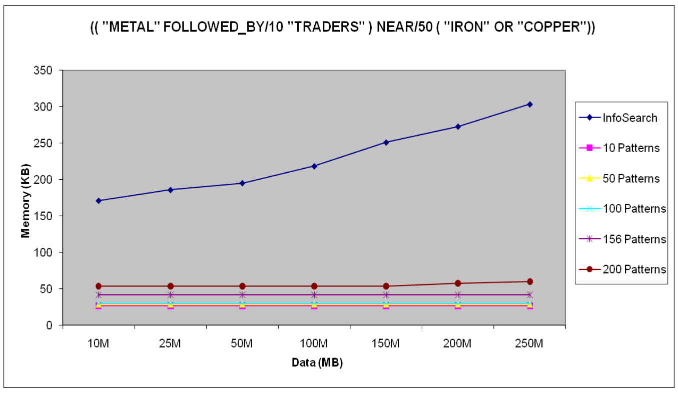


Figure 6.2 Memory Analysis for query (“METAL” FOLLOWED_BY/10 “TRADERS”) NEAR/50 (“IRON” OR “COPPER”)

Table 6.1 Tuples required for detection of the pattern (“metal” FOLLOWED_BY/10 “traders”) NEAR/50 (“iron” OR “copper”)

NoOfPatterns/Operands	**10**	**50**	**100**	**156**	**200**
traders	100	200	500	624	1900
metal	300	400	700	780	1095
copper	100	200	500	624	1163
iron	100	200	473	473	473

the time taken by the InfoSearch system. The response time significantly better than InfoSearch, because once the 156 patterns were detected, the system stops processing of further tuples. In the case of InfoSearch, it process all the tuples irrespective of the number of patterns to be detected, resulting in increase response time. As an example, from the Table 6.1, we can see that to detect 156 patterns from data set of size 200MB, the system needs 624 tuples of “traders” out of 1900, 780 tuples of “metal” out of the total 1095, 624 tuples of “copper” of 1163 and finally “473” tuples of “iron” out of 473.

To detect 200 patterns of the query, the system needs to process all the tuples of the operand or till the tuples of one of the operands that is necessary for the detection of the pattern are exhausted. The response time taken to detect all the patterns is greater than that of the InfoSearch system, because of the overhead in terms of maintaining extra data structures, and propagating control information from up and down the PDG, and extra index access calls.

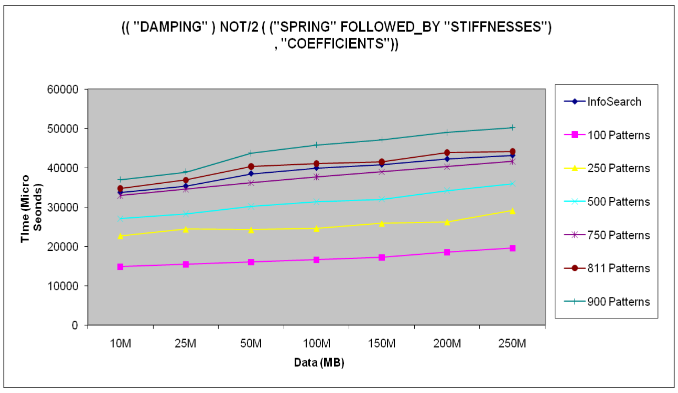


Figure 6.3 Time Analysis for query (“DAMPING”) NOT/2 ((“SPRING” FOLLOWED_BY “STIFFNESSES”) , “COEFFICIENTS”)

The analysis of memory consumption is shown in Figure 6.2. We can see from the plot that the memory consumed by the system is 20-30% from that required by the InfoSearch system. This is because, in InfoSearch system, all the tuples are fetched from the index, at a time leading to the consumption of large amount of memory at the nodes. Whereas, in our system, the tuples are fetched in incrementally, on a need basis, resulting in huge savings in memory.

6.2.2 Query2

Here, we tried with the complex query ((“damping”) NOT/2 ((“spring” FOLLOWED_BY “stiffnesses”) , “coefficients”)), involving two operators *NOT* and *FOLLOWED BY*. We have detected that there are 811 occurrences of the patterns. We have measured the time and memory resources consumed for the detection of 100, 250, 500, 750 and 811 patterns.

In Figure 6.4, we see that the time taken to detect 100, 250 and 500 tuples is significantly lower than the time taken to detect the patterns using the InfoSearch system. Whereas, for the detection of 750 patterns, the time taken is slightly less than that of InfoSearch system. The time to detect 811 patterns is higher compared to InfoSearch system, most of the tuples of the operands that constitute the complex query are involved in the detection in the pattern. In the Figure 6.3, the time taken to detect 900 patterns,

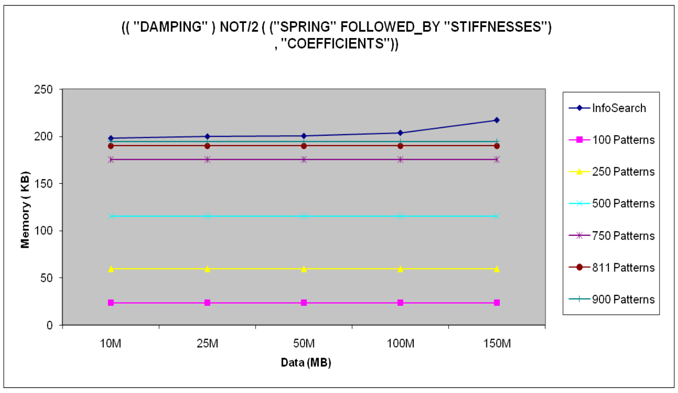


Figure 6.4 Memory Analysis for query (“DAMPING”) NOT/2 ((“SPRING” FOLLOWED_BY “STIFFNESSES”) , “COEFFICIENTS”)

which is 10-16% higher than response time of InfoSearch system is shown. The memory analysis for the query is shown in the Figure 6.4. From the Figure, we see that the memory requirements for incremental approach is significantly lesser than that of the InfoSearch system.

6.3 Summary

The experiments were performed using complex queries, composed of a combination of different operators and the results were analyzed. For each of the queries, we have measured the time taken, and memory consumed for the detection of various number of patterns, using data sets of sizes 10, 25, 50, 100, 150, 200, 250 MB. The memory consumed by incremental approach is 25-30% compared to the InfoSearch system, and does not increase with the size of the data sets, enabling the system to scale. We have also seen that the time taken to detect all the patterns is higher by a factor of 10-35% depending on the queries, because of the overheads.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, we discussed the incremental approach for detection of pattern over text repositories. Currently available retrieval tools are restricted in their expressiveness of the query, and there is a need for a system that could handle queries involving proximity, sequence, frequency and containment operators. We described the disadvantages of processing all the occurrences of the operands, that constitute a complex query, in terms of memory and response time. Furthermore, we presented various design approaches for the incremental detection of patterns, and the advantages of one over other. Additionally, we also a described a ranking algorithm, to sort the detected patterns.

To achieve the incremental approach for detection of patterns, we investigated the need for new data structures, and additional information to be stored. Algorithms, developed for the detection of patterns over static data in InfoSearch system [5] were modified to process tuples arriving incrementally. Since this work extends, previous work [5], some of the modules were incorporated completely. Few modules of the system, has been changed extensively, for detection of patterns incrementally.

7.2 Future Work

The detected patterns were ranked by the system based on the “interval span”, across which the pattern has been detected, and the position of occurrence in the document. If there is a better approach to rank the detected patterns, by any other criteria can be researched upon. Furthermore, to improve performance, caching strategies can be developed. The merging of tuples at the nodes is being done sequentially, strategies based on distributed and parallel computing can be developed to enhance the performance of the system.

REFERENCES

[1] How much information. [Online]. Available: http://www2.sims.berkeley.edu/research/projects/how-much-info/internet.html

[2] How do search engines work. [Online]. Available: http://www.lib.berkeley.edu/TeachingLib/Guides/Internet/SearchEngines.html

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

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

[5] N. Deshpande, "Infosearch : A system for searching and retrieving documents using complex queries," Master's thesis, University of Texas at Arlington, Arlington, 2005.

[6] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, p. 6, 2006.

[7] A. Spoerri, "Infocrystal: a visual tool for information retrieval & management," in *CIKM '93: Proceedings of the second international conference on Information and knowledge management.* New York, NY, USA: ACM Press, 1993, pp. 11–20.

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

[9] D. L. Lee, H. Chuang, and K. Seamons, "Document ranking and the vector-space model," *IEEE Softw.*, vol. 14, no. 2, pp. 67–75, 1997.

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

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

[12] 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.

[13] 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.

[14] Review of search engines. [Online]. Available: http://www.searchengineshowdown.com

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

[16] Ask. [Online]. Available: http://sp.ask.com/en/docs/about/webmasters.shtml

[17] teoma. [Online]. Available: www.teoma.com

[18] R. N. Kostoff, J. T. Rigsby, and R. B. Barth, “Brief communication adjacency and proximity searching in the science citation index and google,” *J. Inf. Sci.*, vol. 32, no. 6, pp. 581–587, 2006.

[19] J. M. Kleinberg, “Authoritative sources in a hyperlinked environment,” *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, 1999. [Online]. Available: citeseer.ist.psu.edu/kleinberg99authoritative.html

[20] Z. Gyngyi, H. Garcia-Molina, and J. Pedersen, “Combating web spam with trustrank.” in *VLDB*, 2004, pp. 576–587. [Online]. Available: http://dblp.uni-trier.de/db/conf/vldb/vldb2004.html

[21] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song, “Ranksql: query algebra and optimization for relational top-k queries,” in *SIGMOD ’05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2005, pp. 131–142. [Online]. Available: citeseer.ist.psu.edu/article/li05ranksql.html

[22] C. Clarke, G. Cormack, and F. Burkowski, “Shortest substring ranking multitext experiments for trec.” [Online]. Available: citeseer.ist.psu.edu/84012.html

[23] D. Hawking and P. Thistlewaite, “Proximity operators - so near and yet so far,” 1995. [Online]. Available: citeseer.ist.psu.edu/hawking95proximity.html

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

[25] Berkeley db java edition. [Online]. Available: http://www.oracle.com/technology/products/berkeley-db/index.html

[26] 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.

[27] R. Dasari, “Events and rules for java: Design and implementation of a seamless approach,” Master’s thesis, University of Florida at Gainesville, Gainesville, 1999.

[28] M. Beigbeder and A. Mercier, “An information retrieval model using the fuzzy proximity degree of term occurences,” in *SAC ’05: Proceedings of the 2005 ACM symposium on Applied computing.* New York, NY, USA: ACM Press, 2005, pp. 1018–1022.

[29] E. M. Keen, “Some aspects of proximity searching in text retrieval systems,” *J. Inf. Sci.*, vol. 18, no. 2, pp. 89–98, 1992.

[30] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval.* New York, NY, USA: McGraw-Hill, Inc., 1986.

[31] E. M. Keen, “The use of term position devices in ranked output experiments,” *J. Doc.*, vol. 47, no. 1, pp. 1–22, 1991.

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

[33] Reuters-21578 data collection. [Online]. Available: http://www.daviddlewis.com/resources/testcollections/reuters21578/

[34] Nsf research awards 1900-2003. [Online]. Available: http://kdd.ics.uci.edu/databases/nsfabs/nsfawards.html

BIOGRAPHICAL STATEMENT

Jayakrishna ThathiReddy was born in India, in 1980. He obtained his M.Sc (Hons) in Chemistry and B.E (Hons) in Electrical and Electronics from BITS-Pilani in 2003. Subsequently, he worked as a Member Technical Staff for Sun MicroSystems India (Pvt) Ltd, in Bangalore, India upto 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 2007.