INFOFILTER: COMPLEX PATTERN SPECIFICATION AND DETECTION OVER TEXT STREAMS

The members of the Committee approve the master's thesis of Laali Elkhalifa

Sharma Chakravarthy
Supervising Professor _____

Alp Aslandogan _____

Lynn Peterson _____

Copyright © by Laali Elkhalifa 2004

All Rights Reserved

To my parents, sisters, brother, sister's family, and friends.

INFOFILTER: COMPLEX PATTERN SPECIFICATION AND DETECTION OVER TEXT STREAMS

by

Laali Elkhalifa

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

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2004

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest sincere gratitude to my advisor, Prof. Sharma Chakravarthy, for giving me an opportunity to work on such an interesting and challenging InfoFilter system and providing me great guidance and support through the course of this research work. I would also like to thank Prof. Lynn Peterson and Dr. Alp Aslandogan for serving on my committee.

I am also grateful to Shravan Chamakura and Anoop Sanka for maintaining a well-administered research environment and their commitment to work. Sincere appreciation is due to Raman Adaikkalavan, Manu Aery, Himavalli Kona, Stephen Lobo, Alpa Sachde, Sridhar Reddy Varakala, and all friends in ITLAB for their invaluable help and advice during the course of development of this system. I would also like to thank Azza, Howayda, Nehad, Raja, Rasha and all my friends for their support and encouragement.

I would like to acknowledge the support of the Office of Engineering Counseling and Advising Center, especially my manager Sally Hoelke for this work. This work was also supported, in part, by the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-02-2-0134), and by NSF (grants IIS-0123730 and IIS-0326505).

Last, but not the least, I thank my parents Prof. Mohamed Khalifa and Sameera Idris, brothers Omar and Malik, and sisters Layla, Limiaa, and Nidhal, and my niece Shaden for their endless love and support. Without their encouragement and endurance, this work would not have been possible.

April 8, 2004

ABSTRACT

INFOFILTER: COMPLEX PATTERN SPECIFICATION AND DETECTION OVER TEXT STREAMS

Publication No. _____

Laali Elkhalifa, M.S.

The University of Texas at Arlington, 2004

Supervising Professor: Sharma Chakravarthy

Information filtering deals with monitoring text streams to detect patterns that are more complex than those handled by search engines. Text stream monitoring and pattern detection have far reaching applications such as tracking information flow among terrorist outfits, web parental control, continuous monitoring of rival web sites in e-commerce, and so forth. InfoFilter, a content-based information filtering system presented in this thesis, detects complex patterns in text streams that include but are not limited to news feed, email, web pages and caption text from streaming videos. Pattern characterization requirements of many applications entail an expressive language for specifying patterns than what is currently provided by Information Retrieval Query Languages (IRQLs). In essence, pattern specification and detection play a major role in information filtering. In this thesis, we describe InfoFilter, which allows users to specify complex patterns such as sequential or structural patterns, wild cards, word frequencies, proximity, Boolean operators and synonyms using the proposed Pattern Specification Language Psnoop and

to detect these patterns efficiently using the data flow paradigm over Pattern Detection Graphs (PDGs).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The recent advancements in computer technologies have led to a digitized world with increasing amount of online information. Consequently, users often find themselves swamped with colossal amount of information while retrieving task relevant data. Consider a news analyst who subscribes to an online news-feed in order to perform analysis for an interesting topic, a sales representative who receives email orders for different products, a federal agent who monitors terrorist information correspondence, etc. In all of the above scenarios, information overloading is very likely and hence is considered as a crucial problem and has been addressed in diverse ways.

1.1 Information Filtering

Information filtering is the process of extracting relevant or useful portions of information/documents from large data repositories or continuous streams of textual data based on relatively static user patterns (or queries). In this process, expressiveness of pattern (or query) specification by a user and its detection play a significant role. Typically, a user profile in the form of one or more patterns is created and submitted to the system, and patterns in such a profile are then compared to the incoming text streams for filtering. These patterns can be simple, such as the detection of individual keywords, or complex, such as multiple sequential occurrences of words or patterns. In order to extract useful or meaningful information, the user needs to have the flexibility to specify complex patterns using an expressive pattern specification language.

Current information filtering systems can be classified into content-based/cognitive filtering and social/collaborative filtering systems. Content-based systems filter documents/text streams based on their content or characteristics. These characteristics can be the presence or absence of a given keyword, a phrase, or a particular word sequence. A number of content-based systems have been developed; these include the SIFT [1] information system implemented at Stanford, the Information Lens [2] system from MIT, SCISOR [3] from GE Research and Development Center and InfoScope [2] from University of Colorado. In Collaborative filtering, documents are filtered based on the recommendations or annotations of a number of users who share common interests. Systems using collaborative filtering includes the Tapestry [2] developed at the University of Minnesota, and the MAXIMS [4] developed at the MIT Media Lab Autonomous Agents Group.

1.2 Information Retrieval

Analogous to information filtering, information retrieval [5][6][7] is the process of extracting relevant or useful portions of text from a relatively static collection of documents based on a stream of incoming user patterns or queries. In information retrieval systems, user queries are specified using Information Retrieval Query Languages (IRQLs) [5]. Based on the similarity between information filtering and information retrieval, most of the existing filtering tools such as personalized information filtering systems use IRQLs for user query specification. In addition, traditional information retrieval techniques such as vector space model [7] are used for matching queries against incoming text to extract the relevant documents.

1.2.1 Information Retrieval Query Languages (IRQLs)

A user who is interested in documents that discusses information filtering provides a set of keywords such as *"information"*. It would be likely that the filtering system

would retrieve the documents that contain *"information"*, but those documents might not be relevant except that they have the keyword *"information"*. Thus, many efforts have been dedicated to provide some enhancements to the existing query languages, so that the users can provide more meaningful patterns (or queries). The main focus of IRQLs is to exploit the content and structure of text. The most widely used query languages are keyword-based query languages.

1.2.1.1 Keyword-Based Queries

Keyword-based queries [5] are widely used by most filtering systems because they are characterized as simple, and easy to use. A query can be in the form of a single word, multiple words or a group of words linked together using some operators such as the conventional Boolean operators. Here are few examples of such queries:

- Single-Word Queries - Single word queries are considered to be the simplest form of keyword-based queries. Typically, a single word such as *"bomb"* is defined as a sequence of letters enclosed by separators to form a word. A document/piece of text is considered relevant to a query if it contains the specified word.
- Context Queries - In context queries, users can specify proximal words, (i.e., words that appear closer to each other providing a semantic meaning). Typically, context queries can be of two types: phrase-based or proximity-based. A phrase is a sequence of words, for example *"graduation deadline"*. On the other hand, proximity query is a flexible form of the phrase query. It allows specification of a maximum distance between words. A query consisting of two keywords *"graduation"* and *"deadline"* appearing within ten words (at most) of each other, could match the text *"Application for **graduation** should be submitted before the **deadline**."*
- Boolean Queries - The most popular form of IRQLs is the Boolean queries. In Boolean queries, users can specify a set of keywords or phrases as operands linked

together using Boolean operators AND, OR, and NOT. The functionality of the common Boolean operators are as follows:

- AND - It allows selection of documents/pieces of text that satisfy both the operands of AND as in (*"information" AND "filtering"*).
- OR - It selects the documents/pieces of text that satisfy either of the operands as in (*"information" OR "text"*).
- NOT - It selects documents/pieces of text that do not contain the operand within a document such as (*NOT "audio"*).

Boolean queries are represented by query syntax trees since the results of the operators can be composed over the results of other queries. In the query syntax tree, the leaves of the tree correspond to the words and internal nodes to the operators.

- Natural Language - Natural Language queries as used in information retrieval are a combination of Boolean queries and context queries. It is expressed using a number of words and phrases. There are no syntax rules or conventions restricting the user specification. For example, given the query *"What is the **difference** between **information retrieval** and **information filtering**?"*, the system extracts the noun phrases. If any number of these extracted words appear separately or together in the text, the document/text is retrieved. As noted, noun phrases constitute the context queries (proximity) and the partial matching of the words models the Boolean operators. In other words, *"difference" OR/AND "information filtering" OR/AND "information retrieval"*.

1.2.1.2 Pattern Matching Queries

Pattern matching queries are used to retrieve portions of text that match a specified pattern. In this type of a query, the pattern is comprised of syntactic features that must occur in a text. Most widely used patterns include the following:

- Words - a string that corresponds to a word in the text such as *"filtering"*.
- Prefixes - a string that forms a beginning of a word as in the query *(STARTS "info")* that matches words such as *"**info**rmation"*, *"**info**rmed"*, *"**info**rmative"*, etc.
- Suffixes - a string that appear at the end of a word as in the query *(ENDS "tion")* that matches *"informa**tion**"*.
- Substrings - a string that appears within a word as in the query *(CONTAINS "tal")* that matches *"me**tal**lic"*, *"revi**tal**ize"*, etc.
- Ranges - a pair of strings that match any word that appears lexicographically between them. For example, a range between *"held"* and *"hold"* matches *"**help**"*, and *"**hissing**"*.
- Regular expression - Users can specify patterns that have certain properties using operators such as union(|), concatenation and repetition(*). For instance, the query *"comp(uter|liance)"* matches words like *"computer"* and *"compliance"*.
- Extended pattern - Regular expressions are extended allowing the use of wild-cards that match any word that starts and ends with specified strings. For instance, the query *"info*tion"* matches the word *"**info**rma**tion**"*.

1.2.1.3 Structural Queries

Structural queries incorporate the structural constraints. They include structural information of the text, such as the title field of a web page. Structural queries are restricted to basic queries such as words, phrases and patterns. For example, the query *"commencement" WITHIN TITLE FIELD*, retrieves web pages/documents that has the word *"commencement"* in the TITLE. In structural queries, Boolean queries can be constructed on top of structural constraints.

1.3 Problem Statement

This thesis is motivated by several observations on the expressiveness of patterns and the patterns that cannot be currently specified although they are needed in many applications. As observed from the characteristics of the query languages proposed in the literature, it can be surmised that query languages support single-word, Boolean, context, natural language, pattern matching and structural queries, and their compositions in a restricted way. Typically, a simple user pattern contains only a keyword (e.g., *"information"*). In Boolean queries, a simple pattern can be linked with other simple patterns using AND, OR, and NOT operators. Thus, a composite pattern like *"information AND filtering"* can be specified and detected over a text stream. Basically, in context queries, a user can only specify a set of keywords that co-occur with another keyword. Context queries can be used to detect patterns such as *"information" NEAR "filtering" WITHIN 100 WORDS* (i.e., words *"information"* and *"filtering"* occurring within a distance of 100 words irrespective of the order). However, complex composition of Boolean, context, and some other queries in IRQLs are not supported. For example, an entire complex pattern near another pattern within some distance (e.g., *"information" AND "filtering"* ***NEAR*** *"information" AND "retrieval" WITHIN 100 WORDS*), cannot be expressed in the current systems that use IRQLs. IRQLS currently supports specification of distance between words. Further, in IRQLs the notion of restricting the scope of the operators is not incorporated. For example, if a user is interested in excluding the occurrence of the string *"1998"* only between *"Iraq"* and *"missile"*, it is not possible. In current IRQLs, this cannot be expressed since the Boolean operator NOT is typically applied over the entire text.

Consider a real world example where a federal agent is tracking terrorist-related information streaming from various resources. He is interested in detecting the occurrence of *((*"bomb" *FOLLOWED BY "ground zero") occurring twice) AND "automotive" (or*

its synonyms). This pattern contains keywords, followed by operator, phrase, frequency, synonyms, and a Boolean operator. This pattern cannot be expressed using current query languages since IRQLs do not support the quantification of multiple occurrences (frequency) of patterns. Moreover, a user cannot include the synonyms in his/her pattern, and is required to explicitly list the synonyms as keywords or several patterns specified each with one of the synonyms. Thus, current query languages are quite restrictive in their expressive power and need to be extended and generalized to address the specification and detection of meaningful complex user patterns. This thesis proposes such a framework in the form of an information filtering system that encompasses an expressive language to specify such complex patterns and an efficient computation model for detecting these patterns.

1.4 Contributions

The aim of this thesis is to overcome the limitations mentioned in the previous section. An enhanced user pattern specification language (*Psnoop*) that can be used by the users to specify patterns for filtering out text streams, is proposed. Briefly, this thesis discusses the design and implementation of InfoFilter, a content-based system that utilizes the expressive pattern specification language *Psnoop* and provides an efficient pattern detection mechanism in the form of a PDG (Pattern Detection Graph). *Psnoop* provides pattern specification using a complete set of operators and options that consists of frequency, synonyms, followed by, Boolean operators, structural, wild card and proximity. Once specified in *Psnoop*, the patterns are detected using a data flow approach over PDG. Figure 1.1 outlines the InfoFilter, which handles creation of user patterns using the Pattern Specification Language *Psnoop*, incoming text streams, WordNet [8] database tool (used for finding synonyms) and user notifications.

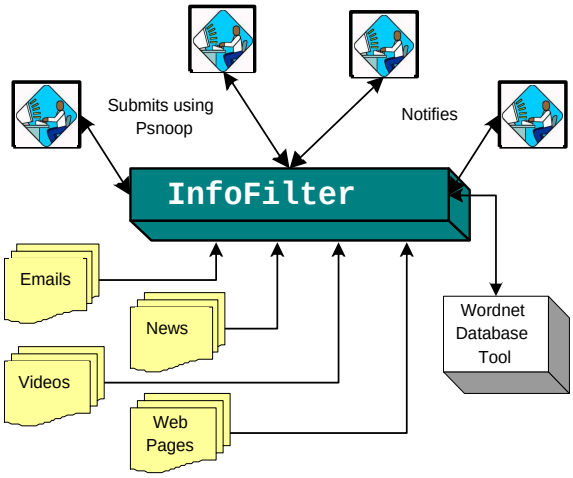

Figure 1.1 InfoFilter with multiple users and incoming text streams.

The organization of this thesis is as follows. *Chapter 2* reviews the related work. *Chapter 3* introduces the Pattern Specification Language (*Psnoop*) and briefly discusses the various *Psnoop* operators and their semantics. *Chapter 4* depicts the architecture of InfoFilter discussing in details the different components of InfoFilter, and pattern detection using *PDG*. *Chapter 5* discusses the implementation of InfoFilter, the optimizations of the *PDG*, and the data structure for storing simple patterns and for string matching. Finally, *chapter 6* outlines the conclusion and future work.

CHAPTER 2

RELATED WORK

During the last decade, we have witnessed an intensive research in the area of information filtering. Commercial and free available information filtering systems were developed to provide solutions that can assist users in extracting relevant information. Various techniques have been applied ranging from simple approaches such as rule-based approaches to complex ones such as machine learning algorithms and probability approaches. In this research, many areas have been targeted including email filtering, selecting interesting news articles, etc. Thereby, examining the work been done on information filtering, it can be surmised that information filtering has evolved over the past few decades.

2.1 Historical Background

The notion of filtering information was around for a long time. It was first introduced by Luhn's early work on Business intelligent systems [2]. The idea was then extended to selective dissemination of information [2]. Basically, the work has focused on the user specification (i.e., constructing user profiles). Later, Denning introduced the term "information filtering", where the work has been expanded incorporating the delivery of information not just the user specification. Initially, the work was aimed at filtering email messages. Denning's approach was utilizing the content of the email to filter the messages. Subsequently, it has been extended to accommodate filtering news articles.

In the eighties, an influential concept emerged when Malone classified information filtering into two prevalent paradigms, content-based filtering and social collaborative filtering. Content-based was an extension of the content filtering discussed by Denning to filter the electronic mail. Malone and his colleagues produced an alternative approach, social/collaborative filtering[2]. Meanwhile, TIPSTER[2] research project funded by DARPA (US Defense Advanced Research Projects Agency) was launched focusing on statistical techniques to preselect messages that can undergo natural language processing.

Since the time Denning introduced the filtering technology, information filtering domain has experienced fundamental changes. In this section, some of the progress in this field with emphasis on the content-based information filtering systems was reviewed.

2.1.1 SIFT

SIFT [1][9] designed at Stanford University, is a content-based filtering system. Boolean queries and Vector Space Model (VSM) are used to construct user profiles, allowing users to specify keywords that are to be included and those that are to be excluded, when filtering documents. To ensure efficiency, SIFT processes groups of similar user profiles, and allows users to apply candidate profiles (or existing profiles) against documents. On the other hand, SIFT does not consider structural information while filtering documents. It makes no distinction between positions of words in a structured text, such as those appearing in a title or the body. In the case of unstructured text, it does not take into consideration the text boundaries, such as sentences and paragraphs.

In SIFT, user subscribes to a server, specifying one or more user profiles. Each profile, comprised of a query and set of parameters such as the time duration for the profile, is stored on a database. On daily basis, USENET News flow are collected by SIFT. The filtering engine reads the articles matching them against the stored queries. After matching the documents, the filtering engine passes the article path name, which

matches a user profile, to the alerter. The alerter, in turn, sends out the notifications to users using their email addresses.

As mentioned above, SIFT uses VSM model [7] to match the documents against the stored queries. Under this model (VSM), documents and queries are represented as vectors. Vectors are constructed using the frequency of the terms available. Based on the content, distinct terms are extracted from the document and their frequency of occurrence is computed within the document. As SIFT queries are specified in natural language, terms are extracted from the queries using their term frequency within the query.

Following the vector construction, query and document vectors are compared against each other. Similarity measure (cosine measure) is computed between the two vectors to compute how far the vectors are close to each other. If the cosine measure is above a threshold, the document is considered to be relevant, otherwise irrelevant. Queries are stored using indexes to improve the performance of matching them against incoming documents. Query indexing method (QI) is used, and this indexing scheme is similar to the indexing scheme used by Information Retrieval (IR) systems. However, instead of storing documents as in IR, queries are stored. In QI, for each term, a list of queries that contain the term are constructed. Each element in the list consists of the query identifier and the associated weight (TD-IDF) of the term in the query. Therefore, when a document D arrives to the system, a vector is constructed and is matched against the stored queries in the index. The cosine measure is computed, and if it is above a specified threshold, the document is considered to be relevant.

2.1.2 InfoScope

Unlike SIFT, InfoScope [10], a content-based filtering system developed by Curt Stevens at University of Colorado, uses adaptive filtering to construct new user profiles

efficiently. It has been developed to filter USENET news articles. Initial user profiles are constructed using Boolean queries. InfoScope applies machine learning techniques to modify the initial user profile to adapt to the changes in user's interest. It also obtains implicit feedback in the form of actions suggested by user, such as the time spent by the user to read a newsgroup.

InfoScope employs a group of agents to monitor the user behavior while he/she is interacting with the system. Suppose the user reads less messages than what has been provided. The InfoScope agent gathers information about the user and makes a suggestion to the user to modify the filter according to his/her interest. If the suggestion is approved by the user, the agent modifies the filter accordingly. InfoScope defines the filters or predicates using Boolean expressions. It does not completely analyze the content but the headers.

2.1.3 GLIMPSE

GLIMPSE (Global Implicit Search) [11], developed at University of Arizona, utilizes indexing and query schemas for retrieving files. It supports Boolean queries and approximate matching such as regular expressions, etc. The method used in the system is called two-level searching. It is a combination of full inverted indexes and sequential search with no indexing. In GLIMPSE, the text is divided into blocks. An index is constructed containing the words and the pointer to the block in which it appears. Multiple occurrences of the words are combined in one entry in the index. Unlike, the full inverted index, this index is comparably small. Essentially, it comprises a set of unique words.

Two-level searching, basically, consists of two phases: In the first phase, a match to the query is looked up in the index to find the list of all blocks. On this phase, a sequential search is applied. Next if there is a match, using the same method, the pattern is looked up within the block for a particular match. This algorithm is used for

searching regular expressions. For Boolean queries, the inverted list is used. For example, for the pattern *"information" AND "filtering"*, the list of blocks in which *"information"* AND *"filtering"* occurs is retrieved. An intersection of the list is computed resulting in the common block numbers appearing in both lists. Those blocks are searched for the exact match. This algorithm is not efficient for Boolean queries when common words are looked up.

2.1.4 Igrep

Igrep [12] was developed at Universidade Federal de Minas Gerais in Brazil. It is an approximate matching tool for large data collections. It accepts words, phrases, and set of characters such as wild cards, ranges, etc. This system uses a full inverted index, which contains the words appeared in the text with their associated positions in the text. For searching single word patterns, the system looks up the words in the table and gets their list of occurrences. Then, an intersection of the list is retained that indicates some similarity with the relative positions of the words in the patterns. Regular expressions are supported to provide approximate matching.

2.2 Search Engines

Search engines [5][9] are used to retrieve documents or Web pages online. Fundamentally, a user submits a query to the search engine and receives the location of the document that matches the query. Most search engines utilize traditional Information Retrieval (IR) techniques. They traverse the web using crawlers (software robots) to create an index based on the textual content of the documents encountered. Crawlers are software programs that run on the local machines and send requests to Web servers available on the net. The mechanism of answering user queries in search engines, however, are different than the typical IR systems. Search engines model the Web as a database.

All queries are answered without accessing the actual text since access to the web pages online is not an easy task. Storing of web pages for retrieval is required.

The architecture of most search engines is centralized. It consists of two modules, one comprising user interface and query engines, and other comprising crawler and Indexer. For specifying user queries, a search engine provides a user interface for supporting basic queries such as list of keywords and complex queries including Boolean operators, phrase search, proximity search and wild cards. In some search engines, further filtering is applied by searching for additional words that appear in certain fields such as a title.

Essentially in search engines, the web pages retrieved in response to a user query are ranked using ranking algorithms. Most search engines use variations of the Vector Space Model (VSM). Many ranking algorithms have been introduced and utilized by search engines.

2.3 Information Retrieval Techniques

From the information retrieval perspective, information filtering is considered as a conventional information retrieval task in which the text documents and streams keep arriving to the systems. Most of the information retrieval approaches have been exploited to perform the filtering task such as Latent Semantic Indexing (LSI) [13], Hidden Markov Model (HMM) [14], Association rule mining [15] and visualization techniques [16]. In this section, Latent Semantic Indexing and Hidden Markov model are discussed briefly.

2.3.1 Latent Semantic Indexing (LSI)

Latent Semantic Indexing (LSI) [13] is an information retrieval technique used to filter information based on the associations between the collection of words and textual documents. In latent semantic indexing, the user input is represented by a set of documents provided by the user. LSI uses the association of words or phrases extracted

from these documents to construct a multi-dimensional semantic structure present in these documents. According to the pattern in which these words or phrases co-occur in a single document, LSI represents the structure of relationships between documents and words in form of n-dimensional association matrix (word-document matrix). Singular-value decomposition (SVD) of the association matrix is computed producing a reduced dimensionality matrix containing the best K orthogonal factors to approximate the original matrix as the model of "semantic" space for the collection. The semantic space contemplates the significant associations while ignoring the variations in the usage of words in some documents. LSI produces a representation of the underlying latent/hidden semantic structure present in these documents. Using the SVD algorithm, LSI eliminates the noise from the original word-document matrix, revealing similarities that were latent in the set of documents. SVD is a mathematical technique that maps an n-dimensional matrix to m-dimensional matrix where m is less than n. This reductive mapping allows LSI to correlate semantically related words/phrases.

In LSI search algorithm, a matrix is constructed for the incoming document and the matrix is compared to the reduced dimensionality matrix. Based on a computation of a similarity measure such as cosine measure, the relevance of a document is determined. The documents that have high similarity measure are considered relevant.

2.3.2 Hidden Markov Model (HMM)

In information retrieval, Hidden Markov Model (HMM) [14] exploits probability approach to filter out relevant documents. The probability of document D being relevant, provided that a user query Q is given, denoted by P(D is R/Q), is computed using the Bayes' rule. The HMM model is comprised of a set of states, a set of output symbols, a set of probabilities for transitions between states, and a probability distribution associated with each output symbol. It is used to reflect the generation of a user query. In the

Markov model, the set of output symbols correspond to the words extracted from the queries. The probability P(D is R/Q) using the probability P(Q|D is R) where Q is a user query, D is a document and R states the relevance of the document to the query.

2.4 Summary

SIFT, Infoscope as well as most of the information filtering systems use Boolean queries, and do not support proximity, regular expressions, cardinality, structural, and sequential operators. Boolean queries merely provide three operators, none of which allow the specification of the co-occurrence of words that has a certain order. In addition, number of occurrences cannot be included in Boolean queries. Most of the information filtering systems utilizes IRQLs. IRQLs consist of various query languages such as single-word, context, Boolean, natural language, and pattern matching. A simple query can be formed using the keyword-based approach, where the query is a set of keywords that are to be identified in a document. A simple query can be extended using context query languages that allow the user to specify correlated words, (i.e., words that appear near each other). In addition, Boolean operators are used to compose keyword-based queries. Nevertheless, combinations of these query languages are not fully supported in current content-based filtering systems.

For user specification, search engines and information filtering systems utilize similar techniques such as the IRQLs. Search engines do incorporate proximity, and wild cards. Nevertheless, they do not support complex compositions of the those queries. Furthermore, the task is different, since in search engines, the documents are stored while queries are continuously arriving to the system.

Search engines and filtering systems use IR techniques to retrieve documents that are relevant to the user. IR techniques include the indexing schemes, Latent Semantic Indexing, etc. In search engines, indices of documents are constructed, while in filtering

system, indices of queries are constructed. Even though ranking algorithms are also exploited, it is not crucial in information filtering systems.

CHAPTER 3

USER SPECIFICATION

In information filtering systems, users are required to specify input patterns that are of interest for filtering text streams. In InfoFilter, users can specify simple and composite patterns using the Pattern Specification Language, Psnoop. It supports the following operators and options: *frequency*, *synonyms*, *followed by*, *Boolean operators*, *structural*, *wild card*, and *proximity*. Any arbitrary complex pattern can be composed using the above operators and options. Psnoop has some similarity with SnoopIB [17][18], an event specification language used for the specification of events in a trigger or an Event-Condition-Action (or ECA) rule [19, 20, 21, 22]. SnoopIB has a complete set of operators for supporting complex event specification in ECA rules. The semantics of all the operators are, however, different in Psnoop. Event operators do not include the notion of proximity, which is crucial in information filtering domain. In addition, SnoopIB does not provide operators to support regular expressions, frequency and synonyms. Detailed description of the syntax and semantics for patterns and Psnoop operators are presented in this chapter. Detection of these patterns using Pattern Detection Graphs (PDGs) is illustrated. Finally, some optimizations for efficient pattern detection are outlined.

3.1 Interval-Based Vs. Point-Based Semantics

In information filtering domain, a pattern represents a user's interest within a text stream. A user may be interested in detecting patterns in the form of keywords, phrases, regular expressions, etc. A pattern occurrence is defined as an occurrence of a given pattern, which is assumed to be atomic (i.e., happens completely or not at all). We

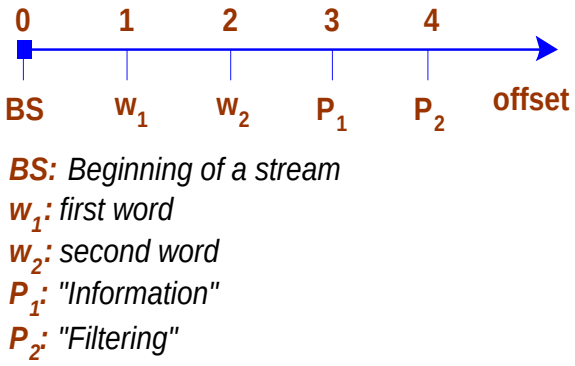

Figure 3.1 Illustration of a pattern occurrence within a text stream.

assume an equi-distant discrete offset domain having "0" as the origin and each offset domain represented by a non-negative integer. Each point in the offset line corresponds to a position of a word occurrence with respect to the beginning of the text stream. For instance, as shown in Figure 3.1, the offset line starts with "0" indicating the beginning of the stream and each offset represents the occurrence of a word such as word w_1 with offset 1. If the offset of a pattern is 3, this translates to the position of the third word in the text e.g. *"information"*. Hence, simultaneous pattern occurrences are not possible as no two patterns can have the same offset.

Computing the offset is intricate when user patterns comprise of more than one word. For example, using the pattern occurrences shown in Figure 3.1, a pattern *"Information" AND "Filtering"* can occur either at the offset 4 (i.e., when Filtering occurs) or can be over an interval [3, 4] (i.e., Information occurs at 3 and Filtering occurs at 4). While the later is termed as *interval-based*, the former is termed as *point-based semantics*. Snoop [23] operators are formally defined for point-based and SnoopIB for interval-based semantics.

Psnoop operators use interval-based semantics, since complex pattern detection requires the entire interval as explained below. Consider the following complex user

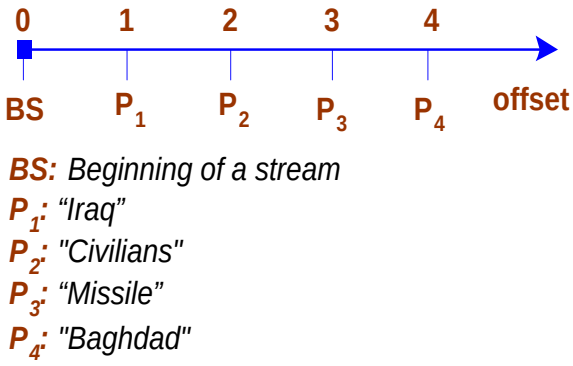

Figure 3.2 Need for Interval-Based Semantics.

pattern where a sub-pattern *("Iraq" occurring prior to "missile")* should **precede** a sub-pattern *("civilians" occurring prior to "Baghdad")*. When point-based semantics is used with the pattern occurrences shown in Figure 3.2, the above example is detected. This is because, for the sub-pattern *"Iraq" occurring prior to "Baghdad"* the offset of detection is 3, similarly for the sub-pattern *"civilians" occurring prior to "Baghdad"* the offset of detection is 4. Since 3 occurs prior to 4, the complex pattern is detected, but which is not intuitively accurate. To detect this pattern accurately (i.e., verify that the first sub-pattern does not overlap with the second pattern), the starting position and ending position of the pattern occurrence should be considered. The occurrence of *"Iraq"* starts the occurrence of the first sub-pattern [i.e., offset 1] while occurrence of *"missile"* ends the occurrence of the first sub-pattern [i.e., offset 3], and similarly *"civilians"* starts the second sub-pattern [i.e., offset 2] and *"Baghdad"* ends it [i.e., offset 4]. In our example from Figure 3.2, offset of first sub-pattern is [1, 3], while the offset of second sub-pattern is [2, 4]. Thus, the complex pattern from the above example is not detected, since [1, 3] overlaps [2, 4], $3 \not< 2$ (i.e., first sub-pattern does not occur prior to second sub-pattern). 3 is the end offset of the first pattern and 2 is the start offset of the second pattern.

3.2 Pattern Specification Language (*Psnoop*)

This section discusses various patterns supported by Psnoop and it also formalizes Psnoop operators.

3.2.1 Pattern

In Psnoop, a pattern $\boldsymbol{P}$ is represented as P_i^j, where i is the pattern identifier and j is the instance of the pattern identifier. O_s is the start offset, and O_e is the end offset. A pattern $\boldsymbol{P}$ is a function that maps from the offset interval domain onto the boolean values, "True/False".

$$\boldsymbol{P} : \mathrm{O} \mapsto \{\text{True, False}\}$$

$$P[O_s,O_e] = \begin{cases} T(rue) & \text{True if a pattern } \boldsymbol{P} \text{ occurs at offset interval } [O_s,O_e] \\ F(alse) & \text{otherwise} \end{cases}$$

The negation of the boolean function $\boldsymbol{P}$ is denoted by $\neg\boldsymbol{P}$. Given an offset interval denoted by $[O_s,O_e]$, it computes the non-occurrence of the pattern within that interval. Furthermore, $\vee$ and $\wedge$ represent the *OR*, and *AND* boolean operators, respectively.

According to the semantics of Psnoop, a pattern is classified into a simple and composite pattern. A simple pattern is either a word such as *"filtering"*, a phrase such as *"information filtering systems"* or a simple regular expression (regular expression on a single word) such as *"info*"*. Prefix and suffix of words can be specified using simple regular expressions. A composite pattern is an expression constructed using simple patterns, previously constructed composite patterns, Psnoop operators and options.

3.2.2 Simple Patterns

Simple patterns form the basic building block of the pattern specification language. A simple pattern is denoted by $\boldsymbol{P[O_s, O_e]}$, where $O_s = O_e$ (i.e., the starting and ending

Table 3.1 Simple patterns in Psnoop

Simple Patterns		Valid Patterns
System-defined	Structural elements for unstructured text	BeginStream, EndStream, BeginPara, , EndPara BeginSent, EndSent
	Structural elements for structured text	BeginHtml, EndHtml, BeginAuthor, EndAuthor
User-defined	Single-word	“information”, “filtering”, “bomb”, “Iraq”
	Phrase	“information retrieval”, “bombing in Baghdad”, “Deadline for graduate application”
	Regular expression	“info*”, “Iraq?”

offset of the pattern is the same). A simple pattern occurrence is an atomic occurrence of a simple or basic pattern. It occurs over an interval $[O_s, O_e]$ and it is detected at the end of the interval (i.e., O_e). Psnoop supports two types of simple patterns, system-defined and user-defined.

3.2.2.1 System-Defined Patterns

System-defined patterns are comprised of pre-defined simple patterns corresponding to structural elements present in the text streams, such as the beginning of a sentence, a paragraph, or a document/stream. For example, two system-defined patterns *BeginPara* and *EndPara* are used to define the beginning and end of a paragraph. In general, the structural elements present in the text streams/documents are domain specific such as structure of a word/latex document is different from the structure of a web page. It will require a different set of pre-defined simple patterns. Hence, *BeginHtml* and *EndHtml*

could be used to define the beginning and end of an html page corresponding to the html tags <html>, </html> respectively.

3.2.2.2 User-Defined Patterns

User-defined patterns are the simple patterns specified by the user describing his/her preference. Possible user-defined simple patterns supported in InfoFilter are:

- **Single-word** - A pattern occurrence is detected if the specified keyword or any of its synonyms (if specified) appear in the text. For example, to detect the word *"filter"*, a user can construct a single-word query as such *"filter"*. Synonyms are specified as an option (SYN) when the user specifies a keyword, such as *"filter"[SYN]*(see section 3.2.3).
- **Multi-word/Phrase** - A pattern occurrence is detected if the specified phrase appears in the text. For example, a user can specify the phrase *"InfoFilter information filtering system"*.
- **Simple Regular expressions** - A pattern is detected if the specified simple regular expression appears in the text. For example, a user can specify *"filter*"* to detect all the forms in which *"filter"* can appear such as *"filtered"* or *"filtering"*.

3.2.3 Composite Patterns

Psnoop provides a comprehensive set of operators, Boolean (OR, NOT, NEAR), followed by (FOLLOWED BY), structure (WITHIN), frequency (FREQUENCY), proximity (NEAR/N, FOLLOWED BY/N) and the option synonyms (SYN) that allow users to compose complex patterns. A composite pattern is a combination of simple patterns, composite patterns, Psnoop operators and options. Table 3.2 provides a summary of the operators and their functionality. The semantics of the composite patterns in SnoopIB [18][24] have been modified and extended to incorporate the semantics of Psnoop

Table 3.2 Summary of Psnoop operators and their functionalities.

Operators	***Purpose***	***Examples***
OR	Provide optional criteria in specifying patterns	***"bomb" OR "explosive"*** ***("bomb" NEAR/3 "automotive") OR ("bomb" NEAR "building")***
NOT	Exclude patterns that are not to be detected Exclude patterns with the number of pattern occurrences exceeds a user specified number Exclude patterns within predefined simple patterns	***NOT ("retrieval") ("information", " filtering")*** ***NOT/2 ("retrieval") ("information", "query language")*** ***NOT(" information" FOLLOWED BY/4 "retrieval") (BeginPara, EndPara)***
NEAR	Detect patterns that occur within *N* words of each other Detect patterns that occur within the same document	***"information" NEAR/2 "filtering"*** ***"information" NEAR "filtering"***
FOLLOWED BY	Detect patterns that occur in certain order within *N* words of each other Detect patterns that occur together in certain order within the same document	***"data" FOLLOWED BY/2 "structures"*** ***"data structures" FOLLOWED BY "algorithm"***
WITHIN	Define the scope of detection within a document, a paragraph or a sentence Define a range for detecting patterns within a document	***("information" NEAR "retrieval") WITHIN (BeginPara, EndPara)*** ***("information" NEAR/2 "retrieval") WITHIN ("InfoFilter", "Psnoop")***
FREQUENCY	Define minimum number of occurrences of a pattern	***FREQUENCY/5("Iraq")***
SYN	Keyword synonyms	***"bomb" [SYN]***

operators. The formal characterization of these semantics formulated using Psnoop operators and options are as follows:

3.2.3.1 OR

Disjunction of two simple or complex patterns P_1 and P_2, denoted by (P_1 *OR* P_2), occurs when either P_1 or P_2 occurs. For example, *"information" OR "filtering"* will be detected whenever either one of the keywords occurs. Since simultaneous occurrences are not possible in a stream (which is essentially a sequence), exclusive-OR semantics is applied to this operator. The formal definition of OR is as follows:

$$(P_1\ OR\ P_2)\ [O_s, O_e] \quad \stackrel{\text{def}}{=} \quad P_1[O_s, O_e] \vee P_2[O_s, O_e]$$

3.2.3.2 NOT

Non-occurrence of the simple or composite pattern P_2 in the range formed by the end offset of P_1 and the start offset of P_3, where P_1 and P_3 can also be simple or composite patterns, is denoted by *(NOT [/F](*P_2*)(*P_1*,* P_3*))*. For example, *NOT ("filtering")("information", "retrieval")* will be detected whenever *"information"* is followed by *"retrieval"* without the word *"filtering"* occurring at least once in between them. "*F*" indicates the minimum number of occurrences and its default value is 1. As another example, *NOT/2 ("filtering")("information", "retrieval")* will be detected whenever *"information"* is followed by *"retrieval"* without the word *"filtering"* occurring at least twice in between them.

$$NOT[/F](P_2)(P_1, P_3)\ [O_{s2}, O_{e2}] \quad \stackrel{\text{def}}{=} \quad ((P_1[O_{s1}, O_{e1}] \wedge (|P_2[O_{s2}, O_{e2}]| < F) \wedge P_3[O_{s3}, O_{e3}]) \wedge (O_{s1} \leq O_{e1} < O_{s2} \leq O_{e2} < O_{s3} \leq O_{e3}))$$

The above formalization explains that pattern P_1 ends at offset O_{e1}, pattern P_3 starts at O_{s3} and pattern P_2 must not occur in the interval defined by *[*O_{e1}*,* O_{s3}*]*, if

the frequency(F) is not specified (i.e., frequency=0). If the frequency is specified, the number of times P_2 must occur in the interval defined by [O_{e1}, O_{s3}] must not exceed or be equal to the value of F.

3.2.3.3 NEAR

Conjunction of two simple or composite patterns P_1 and P_2, denoted by *(P_1 NEAR [/D] P_2)*, occurs when both P_1 and P_2 occur, irrespective of their order of occurrence. "D" is the maximum distance allowed between the two patterns P_1 and P_2. For example, *"information" NEAR/10 "filtering"* will be detected whenever both these words co-occur within a distance of 10. Default value of "D" is the scope of the operator (which can be the entire document), and it refers to the *AND* operator of the Boolean model. The minimum value of D is 1.

$$
\begin{aligned}
(P_1\ NEAR[/D]\ P_2)\ [O_{s1}, O_{e2}] \quad \stackrel{\text{def}}{=} \quad & (((P_1[O_{s1}, O_{e1}]\ \wedge\ P_2[O_{s2}, O_{e2}]) \\
& \wedge \mid O_{s2} - O_{e1} \mid \leq D)\ \vee \\
& ((P_1[O_{s2}, O_{e2}]\ \wedge\ P_2[O_{s1}, O_{e1}]) \\
& \wedge \mid O_{s1} - O_{e2} \mid \leq D)))\ \wedge \\
& (O_{s1} \leq O_{e1} \leq O_{e2})\ \wedge\ (O_{s1} \leq O_{s2} \leq O_{e2})
\end{aligned}
$$

The above formalization states that when the patterns P_1 and P_2 co-occur, they can overlap. They can appear in any order in the interval defined by *[O_{s1}, O_{e2}]*. In other words, the end offset O_{e1} of the initiating pattern and the start offset O_{s2} of the terminating pattern can occur in any order over the interval *[O_{s1}, O_{e2}]*, where O_{s1} is the start offset of the initiating pattern and O_{e2} is the end offset of the terminating pattern. The distance between the pattern occurrences must not exceed the value of D, if specified.

3.2.3.4 FOLLOWED BY

Sequence of two simple or composite patterns P_1 and P_2, denoted by *(P_1 FOLLOWED BY [/D] P_2)*, occurs when the occurrence of P_1 is followed by the occurrence of P_2. The end offset of P_1 is less than the start offset of P_2; that is, the occurrence interval of P_1 and P_2 should not overlap. "D" is the maximum distance allowed between the two patterns P_1 and P_2. For example, *"information" FOLLOWED BY /10 "filtering"* will be detected whenever the word *"information*" precedes "*filtering*" within a distance of 10 words. If "D" is not specified, the distance is bounded by the scope of the operator (can be the entire document). If the value of "D" is 1 (minimum value), this indicates that the patterns P_1 and P_2 form a phrase. Although, technically phrases can be expressed using this operator, direct specification of a phrase is allowed as an intuitive way of specifying phrases and to improve readability of the specification and its detection efficiently.

$$
\begin{aligned}
(P_1\ FOLLOWED\ BY[/D]\ P_2)\ [O_{s1}, O_{e2}] \quad \stackrel{\text{def}}{=} \quad & ((P_1[O_{s1}, O_{e1}] \wedge P_2[O_{s2}, O_{e2}]) \wedge \\
& (O_{s1} \leq O_{e1} < O_{s2} \leq O_{e2}) \wedge \\
& [O_{s2} - O_{e1} \leq D])
\end{aligned}
$$

This formalization of sequence explains that it is detected when the pattern P_2 occurs after P_1 ends. The start offset of P_2 must be greater than the end offset of P_1.

3.2.3.5 WITHIN

Occurrence of a simple or composite pattern P_2 in the range formed by the end offset of the pattern P_1 and the start offset of P_3, denoted by *(P_2 WITHIN (P_1, P_3))*. The pattern is detected each time pattern P_2 occurs in the range defined by patterns P_1 and P_3. For example, *"information filtering" WITHIN (BeginPara, EndPara)* will be detected whenever the phrase *"information filtering"* occurs within a paragraph. When an expression is specified without a system-defined pattern, the default structure (e.g.,

a document, a web page) is used as the default. This operator is extremely powerful in expressing scopes of the stream being processed.

$$(P_2\ WITHIN\ (P_1, P_3))[O_{s2}, O_{e2}] \quad \stackrel{\text{def}}{=} \quad ((P_1[O_{s1}, O_{e1}] \wedge P_3[O_{s3}, O_{e3}] \wedge P_2[O_{s2}, O_{e2}]) \\ \wedge(O_{s1} \leq O_{e1} < O_{s2} \leq O_{e2} < O_{s3} \leq O_{e3}))$$

The above formal definition explains that a pattern P_1 ends at O_{e1}, a pattern P_3 starts at O_{s3}, and a pattern P_2 must occur at least once in the interval defined by the *[O_{e1},O_{s3}]* to detect this pattern.

3.2.3.6 FREQUENCY

Multiple occurrences of a simple or composite pattern that exceeds or equal to F, denoted by *(FREQUENCY /[F] (P))*. A pattern P is detected each time P occurs at least F times, where "F" is the minimum number of occurrences specified by the user. For example, *FREQUENCY/10 ("information filtering")* will be detected whenever the phrase *"information filtering*" occurs at least 10 times. The first F occurrences are considered for this detection as the stream/document is processed sequentially. The default value of F is 1, which is the minimum value. Multiple occurrences are assumed to be disjoint as instances for a pattern detection are used only once. The same instance is not used for detecting multiple patterns.

$$(FREQUENCY[/F]\ P)\ [O_{sf}, O_{el}] \quad \stackrel{\text{def}}{=} \quad (|P| \geq F)$$

The formal definition explains that a pattern P must occur at least F times within the scope defined. The interval over which it occurs is defined by the start offset O_{sf} of the first pattern occurrence and the end offset O_{el} of the last pattern occurrence.

```
Pattern Specification Language (Psnoop) Syntax

Patterns:= Pattern (";"Pattern)*
Pattern:= Expression  [Options] | "(" Expression ")" [Options]
Expression:= Pattern Binary_Op Pattern |  Term | Unary
Unary:= Not_Expression | Freq_Expression
Not_Expression:=Not_Op "(" Expression ")"
Not_Op="NOT"/Frequency
Freq_Expression:= Freq_Op "(" Expression ")"
Freq_Op:= "FREQUENCY /" Frequency
Frequency:= non-negative  integer
Options:= "WITHIN SENT" | "WITHIN PARA"
Binary_Op:= "OR" | Proximal
Proximal:= Proximal_Op ["/" Distance]
Proximal_Op:= "NEAR" | "FOLLOWED_BY"
Distance:= non-negative integer
Term:= Keyword ["[" SYNONYM "]"] | Phrase | Reg_Exp
Phrase:= Keyword Keyword (Keyword)*
Keyword:= "any quoted string"
Reg_Exp:=  StringLiteral Wildcard StringLiteral | Wildcard StringLiteral
StringLiteral:=any string
Wildcard=*|?
SYNONYM="SYN"

This BNF is left associative.
```

Figure 3.3 BNF for the pattern specification language, *Psnoop*.

3.2.3.7 SYN

Option specified with a single-word pattern, denoted by *(P[SYN])*, to indicate multiple single-word patterns that have the same meaning, in a succinct manner. In Psnoop, specifying a single-word pattern with SYN option is equivalent to specifying N simple patterns that carry the same meaning (synonyms) as the original pattern. For example, if you specify the word *"bomb"[SYN]* is equivalent to specifying *"bomb" OR "explosives device" OR "weaponry" OR "arms" OR "implements of war" OR "weapons system" OR "munition"* . If any of these words or phrases appears in the text, the pattern *"bomb"[SYN]* is detected. This option adds simplicity and flexibility to the specification of single-word patterns. The same is true for composite patterns with embedded syn-

onym specification, e.g. *"missile"[SYN] NEAR "Iraq"*. The formal characterization of the SYN options is as follows:

$$(P[SYN]))[O_s, O_e] \quad \stackrel{\text{def}}{=} \quad P_1[O_s, O_e] \vee ..... \vee P_n[O_s, O_e]$$

Using the above operators, users can specify fairly complex patterns. For instance, in Psnoop, the specification of the following pattern *("bomb" occurring prior to at least two occurrences of "ground zero" with a single occurrence of "automotive" or its synonyms)*, using the BNF provided in Figure 3.3 , is as follows *(FREQUENCY /2 ("bomb" FOLLOWED BY "ground zero")) NEAR "automotive" [SYN].*

3.3 Pattern Detection

Psnoop allows users to specify complex patterns as discussed in the previous section. Efficient detection of these patterns specified by the user is extremely critical in information filtering, especially over streaming textual data. In the previous section, the semantics of Psnoop patterns and operators were defined over an interval on the offset line, which indicates the starting and ending position of a pattern occurrence within a text stream. The detection of a simple pattern is straightforward since the start and end offset of a pattern is the same. However, detection of a composite pattern is complex since it involves the detection of simple or composite sub-patterns. We will explain how simple and composite patterns are detected using pattern histories and PDGs.

3.3.1 Pattern Detection Using Histories

This section explains the pattern detection using the formal semantics of Psnoop operators and pattern histories. Various types of histories are defined below:

Global Pattern-History: It is a set of all simple pattern occurrences denoted by $\boldsymbol{H}$. Each simple pattern occurrence forms a singleton set in the history.

$$\begin{aligned} H \;=\; & \{\{p_i^j\} \mid \forall\, p_i, \text{ the simple pattern } P_i \text{ has occurred} \\ & \text{at instance j with interval offset } [O_{sj},\, O_{ej}]\} \\ & O_{sj} \rightarrow \text{start offset of a pattern that has occurred at instance j} \\ & O_{ej} \rightarrow \text{end offset of a pattern that has occurred at instance j} \end{aligned}$$

Simple Pattern-History: It is a set of all the occurrences of the simple pattern P_i, denoted by P_i[H], present in the global pattern-history $\boldsymbol{H}$.

$$P_i[H] \;=\; \{\{p_i^j\} \mid \forall\, \text{j},\, \{p_i^j\} \in H\}.$$

Composite Pattern-History: A composite pattern $\boldsymbol{P}$ has n sub-patterns P_1,..., P_n. Its pattern-history is a mapping from the global pattern-history $\boldsymbol{H}$ to a subset of $P_1\boldsymbol{[H]} \uplus ... \uplus P_n\boldsymbol{[H]}$, where $\uplus$ is an operator that computes the cross product of two sets and merges the elements of the cross product using the set union operator. For example, consider the pattern histories,

$$P_1[H] = \{\{p_1^1\}, \{p_1^2\}\} \quad \text{and} \quad P_2[H] = \{\{p_2^1\}, \{p_2^2\}\}$$

computing P_1[H] $\uplus$ P_2[H] yields

$$P_1[H] \uplus P_2[H] \;= \{\quad \{p_1^1, p_2^1\}, \{p_1^1, p_2^2\}, \{p_1^2, p_2^1\}, \{p_1^2, p_2^2\} \quad \}$$

Pattern Collection: It is a collection of all simple or composite pattern occurrences of the same pattern within a text stream. It is denoted by the following function:

$$\begin{aligned} C(\boldsymbol{P}, StartOffset, EndOffset) \;=\; & \{p \mid \{p\} \in P[H] \; and \\ & StartOffset < startoffset(p) \\ & \leq endoffset(p) < EndOffset\} \end{aligned}$$

3.3.2 Unrestrictive Pattern Detection Mode

For composite patterns, the pattern occurrence is computed using the occurrences of the sub-patterns present in the global pattern-history. The composite pattern-history $\boldsymbol{H}$ is formulated by computing all the occurrences of the composite pattern in an unrestrictive manner. The operators $\uplus$, *OR*, and *NEAR* are all left associative.

1.

$$(P_1\ OR\ P_2)[H] \ = \{p \mid p \in P_1[H] \cup P_2[H]\}$$

2.

$$\begin{aligned}(P_1\ NEAR[/D]\ P_2)[H] \ = \ & \{\{p^i, p^j\} \mid \{p^i, p^j\} \in P_1[H] \uplus P_2[H] \cup \\ & P_2[H] \uplus P_1[H] \text{ and} \\ & (startoffset(p^i) \leq endoffset(p^i) \leq \\ & startoffset(p^j) \leq endoffset(p^j)) \text{ and} \\ & (startoffset(p^j) - endoffset(p^i)) \leq D\}\end{aligned}$$

3.

$$\begin{aligned}(P_1\ FOLLOWED\ BY[/D]\ P_2)[H] \ = \ & \{\{p^i, p^j\}\} \mid \{p^i, p^j\} \in P_1[H] \uplus P_2[H] \text{ and} \\ & (startoffset(p^i) \leq endoffset(p^i) < \\ & startoffset(p^j) \leq endoffset(p^j)) \text{ and} \\ & (startoffset(p^j) - endoffset(p^i)) \leq D\}\end{aligned}$$

4.

$$
\begin{aligned}
(P_2 WITHIN(P_1, P_3))[H] \;=\; & \{\{p^i, p^j\} \mid \{p^i, p^j, p^k\} \in P_1[H] \uplus P_2[H] \uplus P_3[H] \text{ and} \\
& (startoffset(p^i) \leq endoffset(p^i) \\
& < startoffset(p^j) \leq endoffset(p^j) \\
& < startoffset(p^k) \leq endoffset(p^k))\}
\end{aligned}
$$

5.

$$
\begin{aligned}
NOT[/F](P_2)[P_1, P_3][H] \quad = & \\
if\ F = 0 \quad & \{\{p^i, p^k\} \mid \{p^i, p^k\} \in P_1[H] \uplus P_3[H] \text{ and} \\
& \qquad C(P_2, endoffset(p^i), startoffset(p^k)) = \phi\} \\
if\ F > 0 \quad & \{\{p^i, p^k\} \mid \{p^i, p^k\} \in P_1[H] \uplus P_2[H] \uplus P_3[H] \text{ and} \\
& \qquad \text{or } |C(P_2, endoffset(p^i), startoffset(p^k))| < F)\}
\end{aligned}
$$

6.

$$
\begin{aligned}
FREQUENCY[/F](P)[H] \;=\; & \{\{p^i\} \mid \{p^i\} \in P[H] \\
& \text{and } |p^i| \geq F\}
\end{aligned}
$$

Pertinent to the definitions of the operators discussed previously, consider the article used in the example explained in Appendix A. The detection of the composite pattern using the global pattern history, for the unrestrictive mode, as observed in the article, 8 pattern occurrences for the composite pattern *("Iraq" FOLLOWED BY ((FREQUENCY/2("missile") WITHIN(BeginPara, EndPara)* are detected in the text stream. They are retained in the global pattern-history. Based on the emphasis of the information filtering on proximity (i.e., words that co-occur near each other are considered

highly correlated providing a semantic meaning), not all pattern occurrences are useful or meaningful. Thus, this poses a question as to which pattern occurrences can participate in the detection of composite pattern. The following section introduces pattern detection modes as the means to determine which pattern occurrences are meaningful according to the semantics of the information filtering application domain.

3.3.3 Restrictive Pattern Detection Mode

From the point of view of information filtering domain, detecting a composite pattern in the unrestrictive mode may generate duplicate pattern occurrences and may not be meaningful. A large number of pattern occurrences are generated resulting in a substantial space overhead. To reduce the space overhead and still obtain meaningful pattern occurrences, this section discusses two detection modes, recent and recent-unique.

The detection of a composite pattern requires the detection of its sub-patterns. The sub-pattern that starts the detection of a composite pattern is termed the *"initiating sub-pattern/pattern"*. Similarly, the sub-pattern that ends the detection of a composite pattern is termed the *"terminating sub-pattern/pattern"*.

- **Recent**: The above-mentioned problem associated with detecting composite patterns is similar to the composite event detection using Event-Condition-Action (ECA) rules [25]. According to the event detection literature, various types of parameter contexts were introduced to reduce the space and computation overhead associated with retaining all event occurrences. These parameter contexts were intended to capture the semantics of various applications. In this thesis, the recent parameter context [17] has been utilized to retain the recent occurrences of the pattern. Incorporating this context in pattern detection supports the notion of proximity (words or phrases that appear closer to each other been considered highly correlated). In the illustration shown in figure 3.4, the occurrences are as

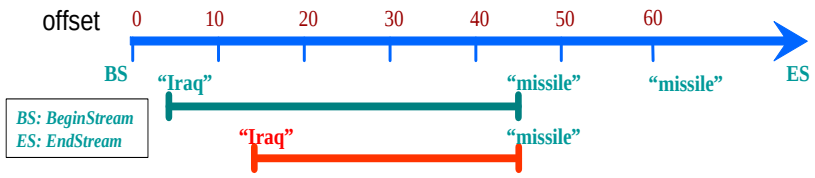

Figure 3.4 Detecting the pattern *"Iraq" FOLLOWED BY "missile"*.

follows: *"Iraq"* occurs at offset 7 ($Iraq^1$) and 15 ($Iraq^2$), and *"missile"* occurs at offset 44. Detecting the composite pattern *"Iraq" FOLLOWED BY "missile"* in the recent mode, the second occurrence of *"Iraq"* (15) is combined with *"missile"* (44) since it is the recent occurrence. In other words, the occurrence of *"Iraq"* that was closer to *"missile"* was used to conform to the emphasis of information filtering on proximity.

In this detection mode, the occurrences that are not used in detecting the patterns are discarded. Each time the pattern occurs, its occurrence replaces the previous occurrence in the history. In the above example, the first occurrence of *"Iraq"* is discarded, and replaced by the second occurrence (i.e., the recent occurrence) in the history. Thus, the history of the composite pattern is $\{Iraq^2\ missile^1\}$, where $Iraq^2$ is the initiating pattern and $missile^1$ is the terminating pattern.

- **Recent-Unique**: Detecting the composite pattern using the recent mode does not generate completely meaningful pattern occurrences. It actually generates duplicate pattern occurrences, since the occurrence of the initiating sub-pattern is used more than once to detect the composite pattern. This may lead to inaccurate results. For instance, if the number of occurrences has been computed, the duplicate pattern occurrence will be counted as a pattern occurrence.

  In recent-unique mode, the recent occurrence of the initiating sub-pattern is discarded immediately after it has been used to detect any pattern, ensuring that no

duplicate pattern occurrences are generated. In other words, this mode entails that an occurrence of a sub-pattern can participate only once in detecting a composite pattern.

For example, consider the pattern *FREQUENCY/2("Iraq" FOLLOWED BY "missile")*. In Figure 3.4, *"missile"* occurs twice at offsets 44 and 60. Detecting the pattern in recent mode, the history contains $\{\{Iraq^2, missile^1\}, \{Iraq^2, missile^2\}\}$ since $Iraq^2$ is the recent occurrence and it is not discarded until another occurrence of *"Iraq"* appears before the second occurrence of *"missile"*. Thus, the occurrence $Iraq^2$ is used to detect the sub-pattern *"Iraq" FOLLOWED BY "missile"* twice, yielding two occurrences. The above pattern is then detected since the specified number of occurrences (2) is equal to the number of times the sub-pattern was detected. However, this detection is not proper. The patterns are partially duplicated since they contain the same initiating pattern. In recent-unique mode, the pattern is not detected. The occurrence $Iraq^2$ is discarded following its participation in detecting the sub-pattern *"Iraq" FOLLOWED BY "missile"*. Since the occurrence $Iraq^2$ is discarded and there is no recent occurrence of *"Iraq"* present in the history, the second occurrence of *"missile"* is not used. Therefore, the sub-pattern is detected once within the stream, not satisfying the specified number of occurrences, and the pattern *FREQUENCY/2("Iraq" FOLLOWED BY "missile")* is not detected.

Further in recent unique mode, the semantics of the operators do not allow detection of overlapping patterns. Suppose the pattern *("March" FOLLOWED BY "Iraq") FOLLOWED BY "missile"* is to be detected and the occurrences in the history are as follows: *"March"* occurs at offset 358, *"Iraq"* at offsets 287 and 399, and *"missile"* occurs at offset 394 and 417. For detecting the sub-pattern *("March" FOLLOWED BY "Iraq")*, the recent occurrences of *"March"* and *"Iraq"*, in which

"March" precedes *"Iraq"*, are required. Therefore, the occurrence *"March"* [358, 358] combined with the occurrence *"Iraq"* [399, 399] forms the composite pattern occurrence $\{March^1, Iraq^2\})[358, 399]$, where [358, 399] is interpreted as the start and end offset of the pattern occurrence. According to the formal semantics of the FOLLOWED BY in recent-unique mode, the occurrence $\{missile^1\}[394, 394]$ is not used since the end offset of the occurrence $\{March^1, Iraq^2\}[358, 399]$ is not less than the start offset of $\{missile^1\}[394, 394]$, i.e., $399 \not< 394$. However, *"missile"* [417, 417] is combined with the composite occurrence, $\{March^1, Iraq^2\}[358, 399]$ since its start offset is greater than end offset of the composite occurrence. This combination, $\{March^1, Iraq^2, missile^2\}[358, 417]$, is used to detect the pattern *("March" FOLLOWED BY "Iraq") FOLLOWED BY "missile"*.

3.3.4 Pattern Detection Graph (PDG)

Unlike information retrieval, in information filtering the text is continuously streaming through the system and is not relatively static. User patterns are required to be detected as the incoming text streams flow into the system. Thus, pattern histories cannot be used for detecting these online patterns. For these patterns, pattern occurrences are computed as they occur using the global pattern-history. For simple patterns, detecting the pattern occurrence is straightforward. However, for composite patterns consisting of complex sub-patterns, the detection is more complicated since the order of sub-pattern occurrences should be preserved. A data flow paradigm in the form of Pattern Detection Graph (PDG) is used to maintain the flow of the pattern occurrences (or maintaining partial histories).

For each pattern or sub-pattern, a corresponding PDG is constructed. Many PDGs are combined to form a complex pattern. In a PDG (Figure 3.5), leaf nodes represent simple patterns and internal nodes represent Psnoop operators. The pattern occurrences

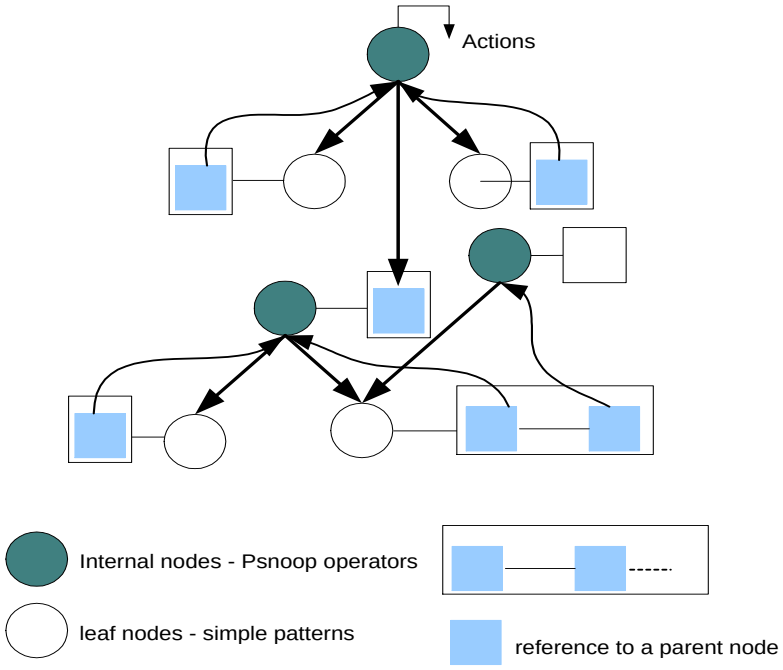

Figure 3.5 Pattern Detection Graph (PDG).

flow through the PDG in a bottom-up fashion. Once a simple pattern is detected in a leaf node, it is propagated to its parent node as leaf nodes have no storage capabilities. Each parent node allocates a space for storing the pattern occurrences that belong to its children nodes.

Typically, in a PDG the parent node subscribes to its children nodes. When a sub-pattern is detected, the corresponding node propagates and notifies the sub-pattern occurrences to the parent node, using the subscriber list, which contains references to the parent nodes. In addition to the subscriber list, the leaf node contains the name of the simple pattern it corresponds to. For composite patterns, the internal node contains the name of the composite pattern, references to the nodes of the sub-patterns and their parameters such as the offset of the pattern occurrence, a reference to the text stream in which it occurs.

3.3.4.1 Illustration of Recent-Unique Mode using PDGs

In section 3.3.3, the pattern detection modes were explained using the global pattern-history. This section, however, illustrates the pattern detection modes using the PDGs. For the example shown in section 3.3.3 and the figure 3.4, a PDG is used to detect the sub-pattern *"Iraq" FOLLOWED BY "missile"* in recent-unique mode. Figure 3.7 shows a PDG constructed for the sub-pattern. When the first occurrence of *"Iraq"* is encountered, the leaf node that corresponds to it propagates the occurrence to the parent node containing FOLLOWED BY. Similarly, the second occurrence of *"Iraq"* appears, it is propagated to the parent node FOLLOWED BY, replacing the first occurrence. Once, the first occurrence of *"missile"* appears, it is propagated by the corresponding leaf node to the parent node to be combined with the existing occurrence of *"Iraq"*. The combined set of pattern occurrences $\{$ *"*$Iraq^2$*", "*$missile^1$*"*$\}$ is then propagated to the parent node. After propagating the combined set, both occurrences of *"Iraq"* and *"missile"* are discarded. When the second occurrence of *"missile"* appears, it is propagated to the parent node to be combined with an existing occurrence of *"Iraq"*. Since *"Iraq"* has not occurred till the end of the stream as shown in figure 3.4. This occurrence of *"missile"* will not be used.

3.3.5 Pattern Detection Graph Optimizations

The previous section discussed the PDG as a mechanism to maintain the flow of the pattern occurrences. This section addresses some of the efficiency issues associated with the PDG. It introduces the naming convention used to allow sharing of nodes in a PDG. It discusses in details the modification of the subscriber list to handle the problem of sharing nodes with different properties/conditions. Finally, it discusses how patterns can be grouped to be detected over a single text stream.

3.3.5.1 Shared Approach

As previously mentioned, a single PDG is constructed for each pattern. This imposes more space and computation requirements for handling large number of patterns. Initially, for common patterns and sub-patterns, a separate PDG is constructed leading to multiple detections of the same pattern over several PDGs. To avoid this, sharing of the corresponding PDGs among the common patterns and sub-patterns is allowed. All the nodes corresponding to patterns are named uniformly using the actual patterns, e.g. the node corresponding to *"information" NEAR "filtering"* is named using the pattern, *"information NEAR filtering"*. This naming convention is used for identifying similar sub-patterns prior to the construction of new PDGs. Thus, an existing pattern's PDG will be shared by a new pattern, if both the existing and the new patterns are identical or comprise identical sub-patterns.

3.3.5.2 Shared Approach with Different Properties

The naming convention allows us to detect identical patterns and share their PDGs. However, in some cases, identical patterns may have different properties associated with them. For instance, the proximity operator, the distance may vary from one pattern to another even though they are identical and the propagation of the pattern occurrences may not be proper. Consider the two patterns *("information retrieval query languages" NEAR ("information" NEAR/2 "filtering"))* and *(("information" NEAR/5 "filtering") NEAR "information retrieval")*. Applying the naming convention, the PDGs constructed for these patterns is shown in figure 3.6. Based on the naming convention, the sub-patterns *("information" NEAR/2 "filtering")* and *("information" NEAR/5 "filtering")* are considered to be identical even though the distance values are different. The method of naming the nodes does not incorporate the values as it may result in constructing

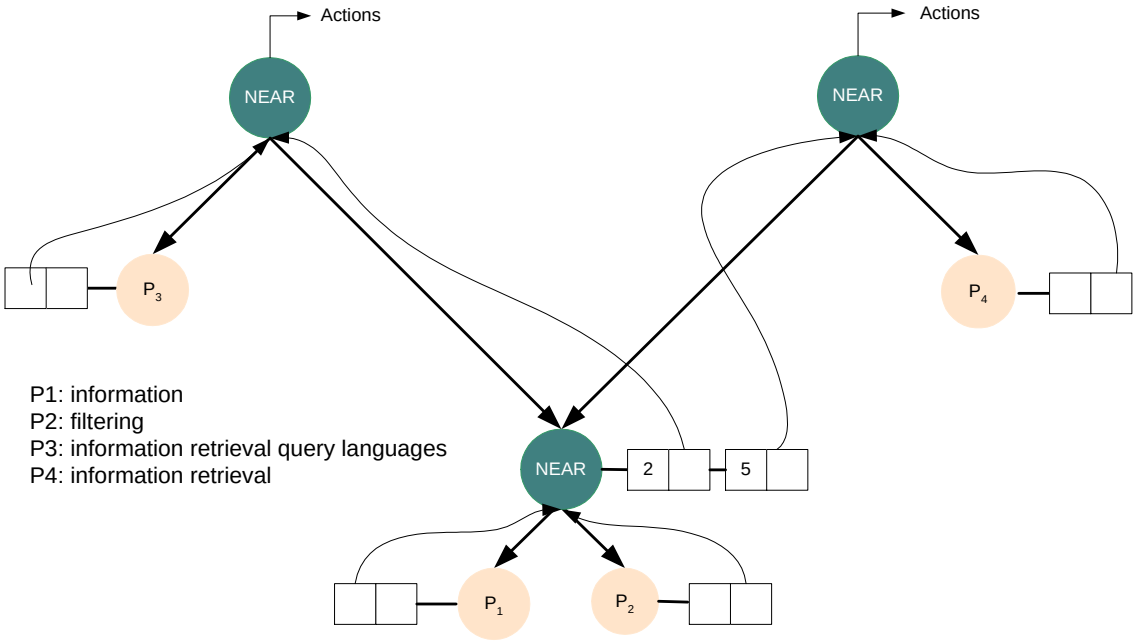

Figure 3.6 Sharing of PDGs/sub-PDGs using the modified subscriber list.

separate NEAR nodes for each sub-pattern. For the detecting the sub-patterns, the distance between the actual occurrences of *"information"* and *"filtering"* is computed separately in each NEAR node to verify whether they satisfy the distance values in the two sub-patterns. Intuitively, if the pattern occurrences of *"information"* and *"filtering"* satisfies the distance of 2, it also satisfies the distance of 5. Hence, if the NEAR node is shared, the distance between the occurrences can be computed once in the best case and twice in the worst case.

To allow the sharing of the NEAR node and incorporate the distance information for the two sub-patterns, the design of the subscriber list has been modified. The subscriber list provides the mechanism to identify which parent is subscribing to a node corresponding to a pattern. The subscriber list is modified to incorporate the properties for propagating the pattern occurrences to the subscribers. It contains the reference to the parent node and the associated condition (i.e., distance value). Prior to propagating

the pattern occurrence to the parent node, the pattern occurrence is checked, if it satisfies the associated property, it is propagated to the appropriate subscribed node.

Using the modified subscriber list, the distance between the pattern occurrences can be computed for the minimum distance available in the subscriber list. If it satisfies the minimum distance, the occurrences can be propagated to all subscribers associated with the distance values greater than the minimum distance. If the computed distance does not satisfy the minimum distance, it is compared with the next minimum distance. In the worst case, it is compared against all distance values available in the list.

Applying this modification to the above example, the NEAR node can be shared. The subscriber list in the NEAR node contains the distance 2 associated with the reference to the NEAR node corresponding to *("information retrieval query languages" NEAR ("information" NEAR/2 "filtering"))*, and distance 5 associated with the NEAR node corresponding to *("information" NEAR/5 "filtering") NEAR "information retrieval")*. Figure 3.6 shows the PDG for the above example with the modified subscriber list.

3.3.5.3 Batch Pattern Detection

In addition, a further optimization is proposed to avoid multiple detections over the same text stream and to enforce the sharing of PDGs among common patterns or sub-patterns. The leaf nodes corresponding to the system-defined patterns, used to define the scope of detection, are shared among the PDGs. If the beginning of a text stream is detected, the corresponding leaf node will notify the parent nodes sharing it, initiating the detection of the corresponding patterns all at once. In other words, the patterns or sub-patterns are grouped together to be detected over the same text stream using the shared PDGs. This reduces multiple scans of the text stream for each pattern to a single scan. Moreover, the pattern or sub-pattern occurrence will be detected once in a

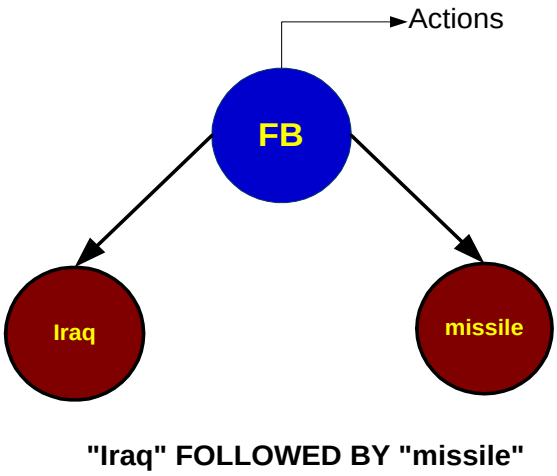

Figure 3.7 for the pattern *"Iraq" FOLLOWED BY "missile"*.

text stream. In all, this reduces the computation and space requirements in detecting patterns over the text streams.

3.3.6 Pattern Deletion

So far, addition and detection of patterns using PDGs have been discussed. In this section the deletion of patterns is addressed. Essentially, to delete user patterns, the corresponding PDGs should be discarded. Unlike pattern detection, the deletion of the patterns is done on a top-down fashion. Deleting of a simple pattern and composite pattern varies slightly.

3.3.6.1 Simple Patterns

If a simple pattern needs to be deleted, the leaf node that contains the name of this simple pattern is deleted, if it is not shared by another pattern. Its references and parameters stored in the parent nodes are deleted as well.

3.3.6.2 Composite Pattern

In a PDG or sub-PDG corresponding to a composite pattern, the deletion is applied in top-down fashion. The internal node that contains the name of the composite pattern unsubscribes (i.e., remove the reference and the associated condition from the subscriber list) from its children nodes. If the children nodes are leaf nodes, they are treated the same way mentioned when deleting the simple patterns. However, if the children nodes are composite, the nodes will be deleted if they are not shared with other patterns, after removing the associated subscriptions from the children nodes. This is applied recursively down the PDG until leaf nodes are encountered.

CHAPTER 4

DESIGN OF INFOFILTER

InfoFilter is a content-based information filtering system that analyzes text streams based on the content and structural information. It accepts patterns from users, detects these patterns and notifies the users when their patterns of interest are detected in the incoming text streams. This chapter briefly discusses the underlying architecture of InfoFilter, and other components external to the system. It also shows how the two important phases of information filtering, user specification and pattern detection have been incorporated into the InfoFilter system.

4.1 Architecture of InfoFilter system

The InfoFilter system is based on a client/server architecture, wherein a group of clients register with a server to specify patterns of interest. Figure 4.1 shows the various modules of the InfoFilter server: Pattern Validator, Processor, Graph Generator, Stream Processor and Pattern Detector. It also shows the WordNet Database tool [8] and the various types of incoming text streams such as documents, web pages, online news articles and video captions.

In the client side, users submit patterns using Psnoop to the InfoFilter server (Figure 4.2). In the InfoFilter server, these patterns are validated and processed upon submission by the pattern validator and pattern processor respectively. Once processed, the pattern processor sends these patterns to the graph generator. The graph generator constructs the PDGs corresponding to the patterns in the pattern detector, and also

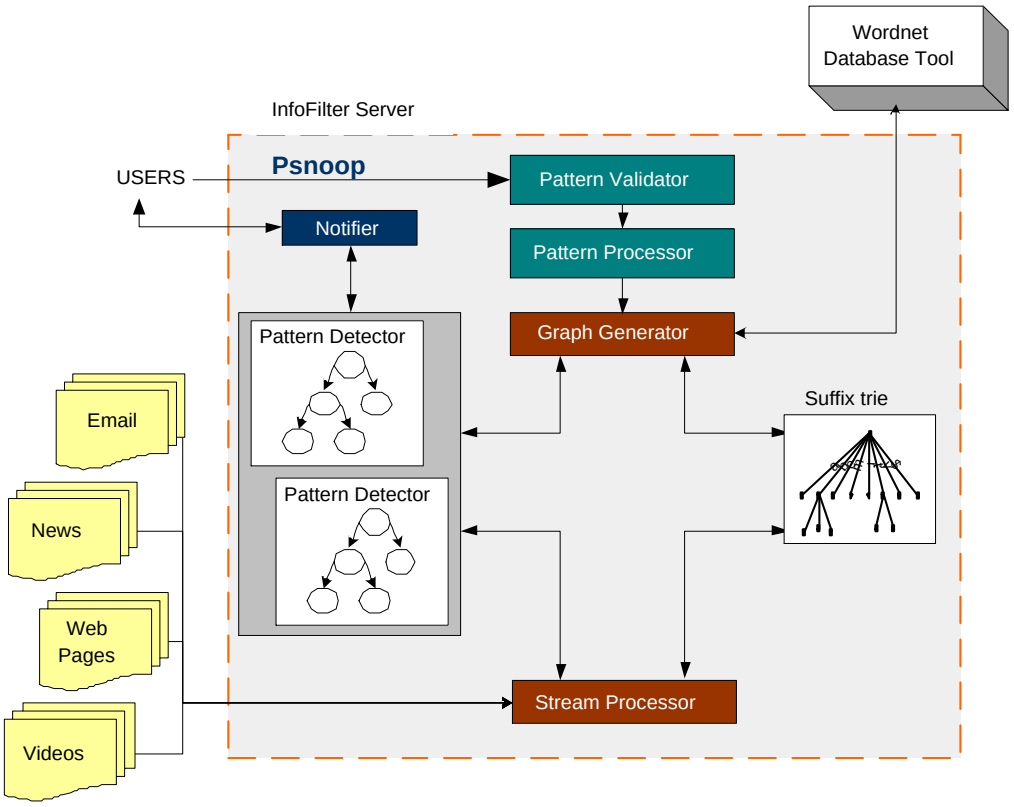

Figure 4.1 Architecture of InFoFilter.

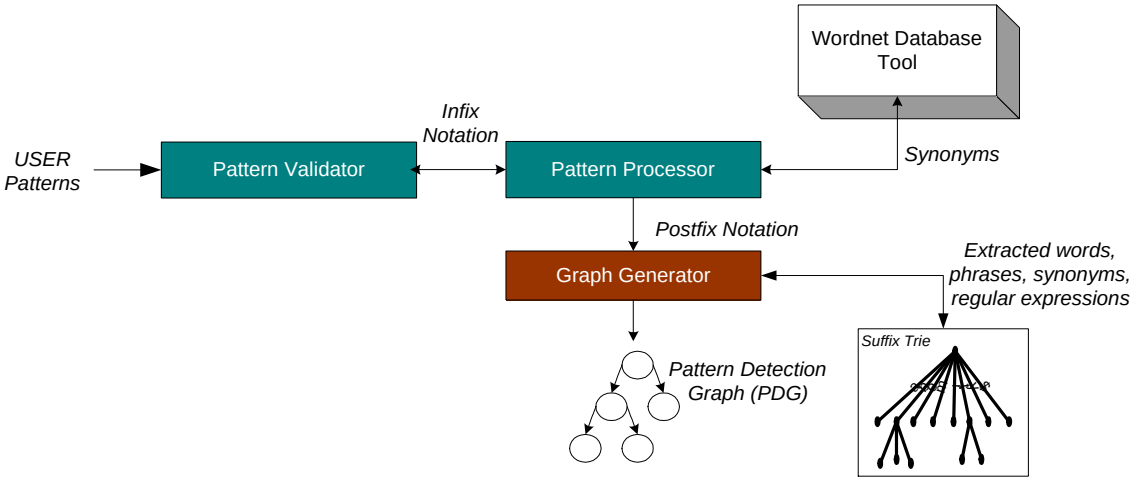

Figure 4.2 Illustration of pattern flow in InfoFilter.

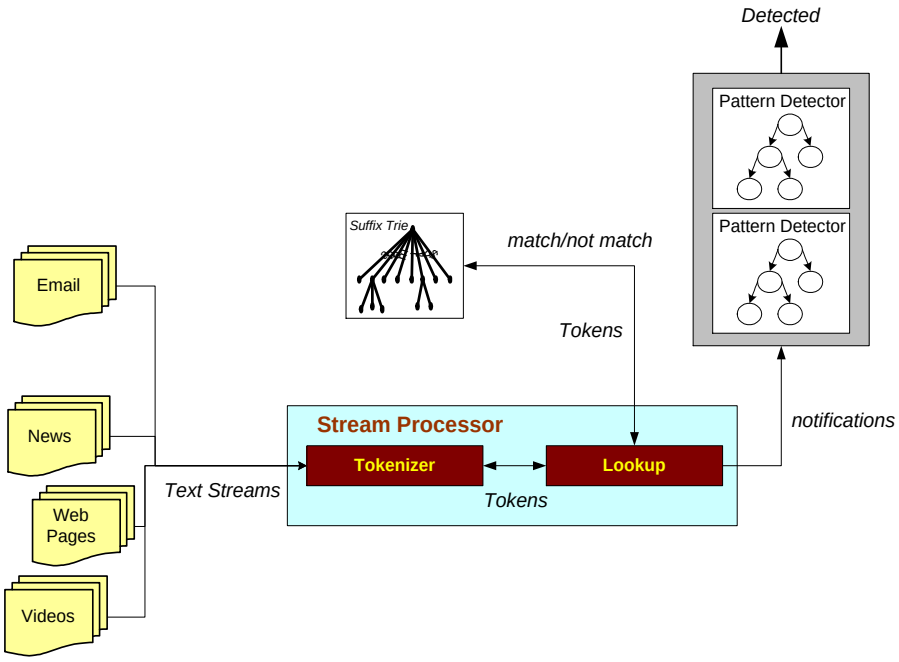

Figure 4.3 Illustration of stream flow in InfoFilter.

extracts and stores the keywords, phrases and regular expressions, embedded in these patterns in a shared data structure, suffix trie.

On the other hand, the stream processor preprocesses the incoming text streams to be matched against the extracted keywords, phrases and regular expressions stored in the shared data structure. The incoming user patterns are associated with different types of streams. That is, a user can specify the type of stream (such as email, video) over which a pattern is to be detected. InfoFilter uses a separate stream parser in the stream processor and a separate pattern detector for each type of input stream. Patterns are accumulated for a stream type in that pattern detector. This allows the system to detect and exploit common patterns over the corresponding PDGs. Figure 4.3 depicts the stream flow in InfoFilter.

InfoFilter continuously monitors multiple types of streaming text, detects simple pattern occurrences and notifies the corresponding pattern detectors. The pattern detec-

tor in turn will detect complex user patterns using the semantics of the Psnoop operators and alerts the notifier to send notifications to a user when his/her pattern is detected. Suppose a user specifies the pattern *"information" NEAR "filtering"* to be detected in an email stream. A separate pattern detector is used to detect the pattern in the email stream. If the pattern is detected, the PDG constructed in this pattern detector is notified.

4.1.1 Pattern Validator (PV)

Users specifying patterns should follow the BNF (see Figure 3.3) provided by the pattern specification language PSnoop. Pattern validator (PV) accepts the input patterns in a linear form (i.e., using infix notation) from the users according to the Psnoop syntax. The PV then checks these patterns for correctness, and once validated these patterns are sent to the pattern processor for further processing.

4.1.2 Pattern Processor (PP)

The Pattern Processor (PP) accepts the patterns in the infix notation and processes the patterns after converting them to postfix notation [26]. In infix notation, the ordering of the operands and operators is subtle. To evaluate the pattern, with emphasis on the operator precedence and minimization in the use of parentheses, the infix notation is converted to postfix notation. Patterns in postfix notation are easier to evaluate. In this notation, the operands precede the operators. PP, then sends the pattern in postfix notation to the graph generator.

4.1.3 Graph Generator (GG)

Graph Generator (GG) interacts with the pattern detector when the pattern processor passes the patterns in postfix notation. Graph generator extracts the keywords,

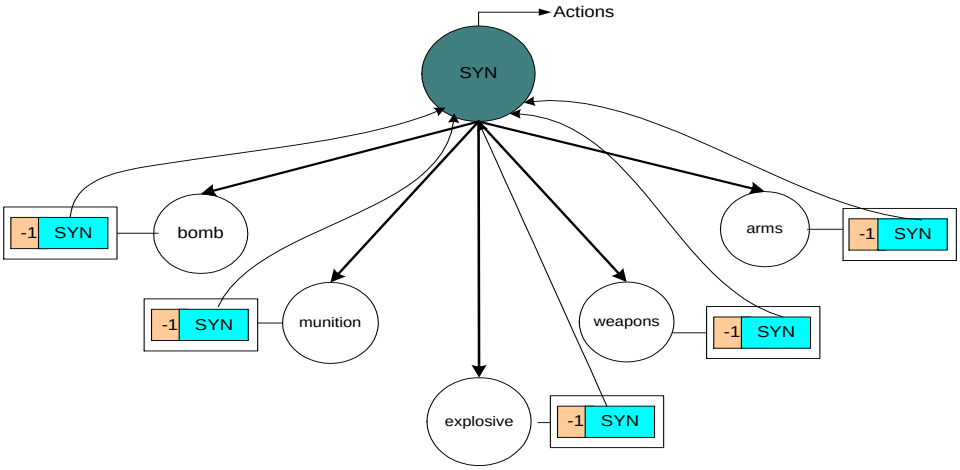

Figure 4.4 PDG for the pattern *"bomb"[SYN]*.

phrases, regular expressions and operators to construct the Pattern Detection Graph (PDG) (see section 3.3.4) in the Pattern Detector (PD). PD is a library that provides the means to define simple and composite patterns in InfoFilter. It provides the necessary APIs to generate the PDGs corresponding to the input patterns.

In the graph generator, WordNet [8] Database tool is used to determine the synonyms of the extracted keywords, if the synonym option is specified. WordNet is a lexical database tool, partitioned into nouns, verbs, adjectives and adverbs using the parts of speech. Nouns are organized as a lexical hierarchy of nodes. Each node contains the meaning of a word, or a synset (a list of synonymous words). The graph generator sends the keywords to WordNet to extract their synonyms. Once the synonyms are extracted, the graph generator stores them.

The extracted synonyms can be handled in two ways. Synonyms are considered as simple patterns and are incorporated into the construction of the PDG/sub-PDG using leaf nodes. Leaf nodes are then connected to an internal node SYN. If any of the synonyms is detected, the pattern occurrence gets propagated up the SYN node in the PDG resulting in the detection of a pattern/sub-pattern. Nevertheless, this approach

may result in constructing a large number of leaf nodes corresponding to the synonyms. This approach has high computational cost. Refer to the example in section 3.2.3.7, if the pattern *"bomb"[SYN]* is specified. The corresponding PDG shown in figure 4.4, has the synonyms of "bomb" stored in the leaf nodes, and a common node that corresponds to the option SYN. To detect the synonyms, the text stream has to be matched against the leaf nodes corresponding to the synonyms. This leads to multiple traversal of the PDG while detecting a pattern. In the worst case, the comparison is done with all synonyms. In general, if there are N synonyms in a sub-PDG, worst case for matching the text stream will be N comparisons where N is the number of leaf nodes. However, the advantage of this approach is that it allows sharing of synonyms among identical patterns/sub-patterns.

Another alternative is to store the synonyms in the shared data structure, suffix trie, with references to the node corresponding to the original pattern. For example, if the user specifies the keyword *"bomb"* to be detected on a text stream, the graph generator extracts synonyms such as *"bomb", "explosives device", "weaponry", "arms", "implements of war", "weapons system", "munition", "flop", "bust", "calorimeter"* and sends them to the stream processor to be stored in a suffix trie [27]. If any of these synonyms are detected in any incoming text stream, it notifies the node mapped to the word *"bomb"*. The pattern comprising the keyword *"bomb"* is detected. This approach reduces the cost of the search time, matching the token with n leaf nodes. Conversely, the suffix trie has less search time. The limitation involved in this approach is the cost of inserting these synonyms into the suffix trie, which is linear time O(n). The two approaches are supported in InfoFilter.

After generating the PDGs mapped to the patterns and extraction of synonyms, the graph generator passes this information to a shared data structure (suffix trie).

4.1.4 Stream Processor (SP)

For each incoming text stream, the Stream Processor (SP) interacts with the assigned Pattern Detector (PD), according to the type of stream. For example, a stream processor handling the web pages interacts with the PD associated with web pages. The stream processor monitors the text streams. It parses these input streams to detect the occurrence of the stored simple patterns (keywords, phrases, regular expressions and synonyms). For each type of stream, there is a separate parser that analyzes the features associated with the type of text stream. For instance, for web pages, an html parser parses the input stream.

Typically, in information filtering systems the incoming text streams need to be parsed and patterns need to be detected on the fly. Thus, a mechanism for parsing the streams and matching them against the patterns is required. The first approach is to match the tokens generated by the stream parsers against the nodes of the PDGs corresponding to the patterns. As mentioned before, the leaf nodes of the PDGs contain the name of the simple patterns they correspond to. Hence, the generated tokens are matched against the the leaf nodes in the PD. However, this leads to an efficiency issue when a large number of distinct patterns are submitted to the system. The PDGs are traversed for each token generated resulting in at most *O(n x d)* search time, where n is the number of tokens generated by the stream parser and d is the total number of leaf nodes in all the PDGs. Besides, for each token, the stream processor must interact with the PD.

Therefore, to avoid this problem, the simple patterns are stored by the graph generator in a suffix trie/tree. A suffix trie is characterized as space efficient representation and has less search time. Both features are required by InfoFilter to detect patterns on the fly. Using the suffix trie searching algorithms, the tokens are matched against the stored words, phrases, regular expressions and synonyms. For those that do exist,

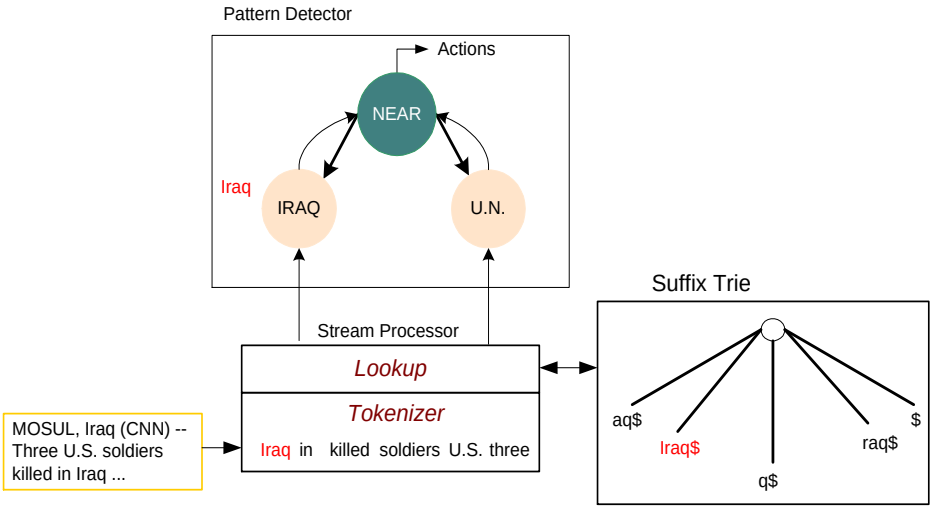

Figure 4.5 Input to PDG from Suffix Tries.

the stream processor notifies the pattern detector. Figure 4.5 shows how a stream is parsed and tokens are matched against keywords stored in the suffix trie. Consider the sentence *"Three U.S. soldiers killed in Iraq"*, the stream processor generates the tokens *"three"*, *"U.S."*, etc. A PDG is constructed for the pattern *"Iraq" NEAR "U.N."* by the graph generator. The simple patterns *"Iraq"* and *"U.N."* have been stored in the suffix trie. Figure 4.5 shows how the word *"Iraq"* is stored in the suffix trie. Following the generation of the tokens, the look up module in the stream processor sends the tokens suffix trie to be matched against the stored simple patterns, resulting in the detection of *"Iraq"* and *"U.N."*. For instance, after receiving a match from the suffix trie for *"Iraq"*, the stream processor notifies the pattern detector that *"Iraq"* has been detected. The pattern detector propagates the detected occurrence of *"Iraq"* from the leaf node to the parent node in the PDG. Same is done for the occurrence of *"U.N."*. If the whole pattern is detected, the pattern detector alerts the notifier.

4.1.5 Pattern Detector (PD)

The Pattern Detector (PD) is a library that provides APIs to construct the PDGs for the user patterns. Once simple patterns are detected in the stream processor, PD passes the notifications to the corresponding leaf nodes leading to the computation of the associated pattern occurrences. As mentioned, the pattern occurrences propagates up the PDG, if the leaf nodes of a PDG have been notified. The pattern occurrences flow up the tree until they reach the topmost node. That node triggers the actions associated with that PDG. Initially, when the PDG is constructed, the topmost internal node of the PDG is mapped to the action required once the entire pattern is detected. The action is typically alerting the notifier to send messages to the user providing him/her with the stream information where the pattern has been detected. The stream information is present in the pattern occurrences.

4.1.6 Notifier

The purpose of the notifier is to handle the notifications sent to users if their patterns are detected. The preferences of the users regarding those notifications are stored in the notifier. Once, the notifier gets alerted that a pattern has been detected and that the corresponding PDG is triggered, it sends notifications to the appropriate users using the pattern information (expiration date, user information, etc.).

CHAPTER 5

IMPLEMENTATION

The InfoFilter server has been implemented as a stand-alone java application. Its underlying architecture has been described in the previous chapter. The server consists of Pattern Validator(PV), Pattern Processor (PP), Graph Generator (GG), Stream Processor (SP), Pattern Detector (PD), Notifier and Suffix trie (shared data structure).

This chapter depicts the implementation details of each component of the InfoFilter server with emphasis on the Pattern Detector (PD). Using Pattern Detector (PD), section 5.3.1 illustrates how the patterns are detected over the Pattern Detection Graphs (PDGs). Section 5.6 explains the data structure, how it has been used and the purpose of implementing it. In section 5.3.2, there is a brief description of the new operators and the modifications implemented for the existing operators. A detailed description including algorithms of the operators are explained in Appendix B. Section 5.8 explains how Pattern Detection Graphs (PDGs) deals with the implementation issues for optimizing the pattern detection. Section 5.8 details the deletion of patterns.

5.1 Pattern Validator (PV)

The pattern validator was implemented using a JavaCC parser [28]. It accepts user patterns submitted in a linear form such as *(FREQUENCY /2 ("bomb" FOLLOWED BY "ground zero")) NEAR "automotive" [SYN].* It parses, validates, and tokenizes the patterns based on the syntax of the Psnoop language. The language syntax such as NEAR, OR, WITHIN_SENT (defines sentence boundaries, begin and end of a sentence) are considered as tokens and based on the token definition, the input is parsed. The

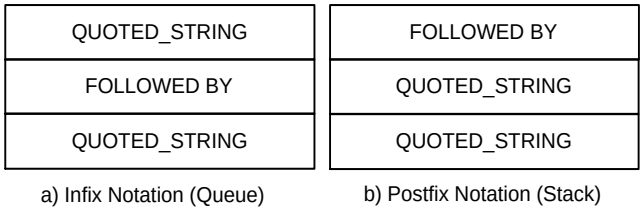

Figure 5.1 Infix and Postfix Notations.

validator outputs the infix notation of the tokens to a queue. Figure 5.1(a) shows an example of infix notation generated for the sub-pattern *"bomb" FOLLOWED BY "ground zero"*. Queue is a data structure that has a concept of order. In the queue, the ordering is based on the order of insertion (First-in-First-out). Thus, if tokens are inserted into the queue, the ordering of the tokens follows the infix notation. The output of PV is then picked up by the processor.

5.2 Pattern Processor (PP)

The pattern processor retrieves the infix notation from the queue and converts it to postfix notation and places it in a stack. Figure 5.1(b) shows an example of postfix notation generated for the sub-pattern *"bomb" FOLLOWED BY "ground zero"*. The ordering in the stack is First-in-Last-out. This allows for the removal of the operators and operands to construct the graph for that subexpression. For instance, if the top of the stack is a binary operator, the following two elements in the stack constitute the operands of that operator, preserving the ordering of the postfix notation. The conversion from infix to postfix is straightforward if parentheses are not used. However if parentheses are used such as in the pattern *(FREQUENCY /2 ("bomb" FOLLOWED BY "ground zero")) NEAR "automotive" [SYN]*. The postfix notation of this pattern is *(("bomb" "ground zero" FOLLOWED BY) 2 FREQUENCY) "automotive" SYN NEAR*. In other

words, the sub-pattern enclosed in parenthesis is considered to be the first operand of the operator *NEAR*.

5.3 Pattern Detector (PD)

Pattern detector (PD) is a Java library that provides the APIs necessary to construct the Pattern Detection Graphs (PDGs) that detect complex patterns. Basically, a java library (LED) [22] had been implemented to create and detect composite events. This thesis uses the LED library with suitable modifications for the algorithms and contexts to detect complex patterns. Several new operators (e.g., Frequency, near with a range) has been added and several operators (e.g., within for use with structural specifications) has been modified and extended to capture the semantic requirements of information filtering.

Local Event Detector (LED) detects events defined in Event-Condition-Action (ECA) rules. In other words, when an event is detected, the rules or actions associated with that event are executed. Events are specific points of interest. LED utilizes an event specification language [29] that provides a group of operators comprised of *OR, AND, NOT, SEQUENCE, APERIODIC, APERIODIC STAR, PERIODIC, PERIODIC STAR* and *PLUS*. LED supports point-based semantics while PD supports interval-based semantics. PD does not support the *APERIODIC STAR, PERIODIC STAR*, and *PLUS*, since they do not suit the requirements of information filtering. In PD, the operators *AND* and *SEQUENCE* were modified and extended to incorporate the properties of NEAR and FOLLOWED BY such as the notion of proximity. The semantics of *NOT* and its implementation have been modified to support the frequency. The *FREQUENCY* operator was added. Furthermore, a SYN operator, a variant of OR, is added to accept n number of operands.

In LED, the parameters required to detect the events are computed in more than one context according to the semantics of the applications. LED supports various parameter contexts, of which none conformed to the requirements of information filtering. In LED, the parameter contexts supported include recent, cumulative, continuous, and chronicle. PD has utilized the recent context as a pattern detection mode for detecting composite patterns. However, as explained in section 3.3.3, the recent context did not produce results that are intuitively correct for the information filtering domain. Thus, in PD, a new pattern detection mode, recent-unique has been implemented to accommodate the requirements of the information filtering domain.

Analogous to the PD, LED generates an event graph that maintains the flow of events while detecting these events. However, in the Pattern Detection Graphs (PDG), the propagation of pattern occurrences varies from the propagation in the event graph (see section 5.8).

5.3.1 Pattern Detection Graphs (PDGs)

In InfoFilter, Pattern Detection Graphs(PDGs) are used to maintain the detection flow of pattern occurrences. As described in chapter 3, the leaf nodes of the PDGs represent simple patterns and internal nodes represent the composite pattern operators. Each node has a subscriber list which is implemented as a vector that contains the parameters (frequency, distance) associated with the patterns and references to the parent nodes. Figure 3.6 illustrates the modification of the subscriber list. Each element of the subscriber list has two objects – the specified distance and the subscriber. For instance, the node that corresponds to the pattern *("Iraq" FOLLOWED BY/30 "missile") FOLLOWED BY/10 ("Civilians" FOLLOWED BY/60 "Baghdad")* subscribes with a distance of 30 in the node that contains ("Iraq" FOLLOWED BY/30 "missile") and distance of 60 in the node that contains ("Civilians" FOLLOWED BY/60 "Baghdad"). If

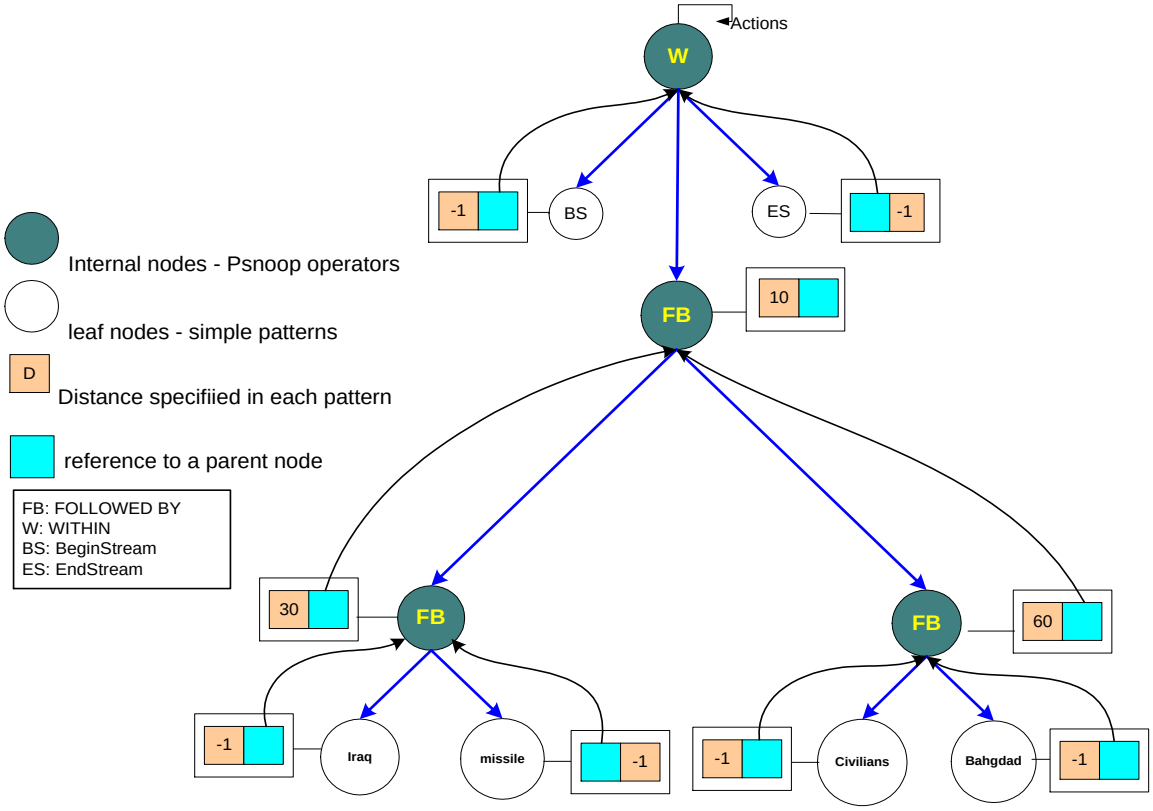

Figure 5.2 *("Iraq" FOLLOWED BY/30 "missile") FOLLOWED BY/10 ("Civilians" FOLLOWED BY/60 "Baghdad").*

a simple pattern is detected in a text stream and the corresponding leaf node is notified, the pattern occurrence is propagated to the parent nodes. The composite pattern is a combination of simple patterns and Psnoop operators. The composite pattern is associated with a list of parameter lists (parameter lists of the constituent sub-patterns). The propagation of the composite pattern occurrences is dependent on the parameter detection mode. The pattern detection mode is indicated by integers in the nodes of a PDG. For instance, the pattern occurrence is propagated in recent-unique if the corresponding integer is non-zero. The integer in the nodes are set using an API that defines the action (sending notifications to the notifier) to be executed if a pattern is detected. The PDG is recursively traversed in a top down fashion to set the corresponding integers in the

nodes encountered. The recent-unique pattern detection mode has been encapsulated into the algorithm that deals with the semantics of each operator. This keeps the system manageable and makes it easy to implement seemingly complex pattern detection modes. Appendix A discusses the algorithms in details.

In a PDG, the leaf receives the notification of an occurrence of a pattern and are immediately passed to the subscriber nodes along with the parameters. There is no need to delay the propagation thereby requiring storage at a leaf node. However, in internal nodes, a table is used to store occurrences of the constituent sub-patterns. This is required as pattern occurrences can occur at different times, and until a matching pattern (or the terminator) for that operator arrives, the pattern needs to be stored and managed (for the recent-unique context). Each element in the table is comprised of a pattern occurrence and a set of bits indicating the pattern detection mode associated with the pattern. For a simple pattern, the pattern occurrence is represented by a singleton set. For composite patterns, the pattern occurrence is represented as a set of pattern occurrences. The size of this set increases as it moves up the graph. When a pattern occurrence participates in detecting a composite pattern in a particular mode, the corresponding mode bit is reset according to its semantics.

The parameters of a simple pattern are the arguments associated with the pattern, such as the values of the pattern whether it is a keyword, a phrase or a regular expression. The offset of the pattern occurrence is considered to be one of the parameters that are stored for computing the proximity of patterns. The name of the pattern or the position of the parameter in the list is used to access the parameters in the list. To support such access, the parameter and its name are both stored in a hashtable that maps the parameter names with the parameter nodes. The parameter node stores the type and value of the parameter. The list of parameter lists in the composite nodes is implemented as a vector to allow both sequential or positional access to the stored parameter lists.

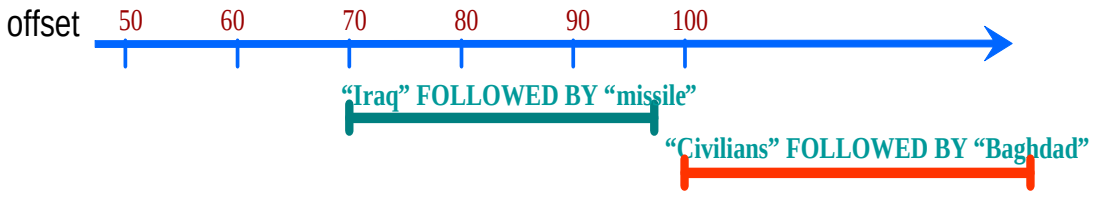

Figure 5.3 Computing distance between two non-overlapping patterns.

5.3.2 Psnoop Operators

PD supports the semantics of Psnoop. It incorporates the newly implemented operators, modified operators and their semantics.

NEAR and FOLLOWED BY: In PD, for *NEAR* and *FOLLOWED BY* operators, the algorithm of computing the distance has been implemented. Distance between simple patterns is computed using the offset of the simple pattern occurrences. For instance, consider the sub-pattern *“Iraq” FOLLOWED BY/10 “missile”*. Let us assume that the word *“Iraq”* occurred at the offset 3 with respect to the beginning of the stream and “missile” at offset 5. For simple patterns, the initiating and terminating patterns are the same. Thus, the offset of the initiating pattern (3) is subtracted from the terminating pattern (5). The result is then compared to the distance, if it is less or equal to that value. The distance is satisfied detecting the occurrence of that pattern.

Similarly, the distance between non-overlapping composite patterns is computed by subtracting the offset of the initiating sub-pattern from the terminating sub-pattern. As another example, consider the pattern *(“Iraq” FOLLOWED BY/10 “missile”) FOLLOWED BY/15 (“Civilians” FOLLOWED BY “Baghdad”)*. Assume that the sub-patterns *(“Iraq” FOLLOWED BY/10 “missile”)* and *(“Civilians” FOLLOWED BY “Baghdad”)* have occurred with the Iraq at offset 70, missile

at 96, Civilians at 100, and Baghdad at 131, as shown in figure 5.3. The distance between the two sub-patterns is computed by subtracting 96 (end offset of the first pattern) from 100 (start offset of the second pattern). The result is 4 which is less than the distance 15, thus detecting the pattern *("Iraq" FOLLOWED BY/10 "missile") FOLLOWED BY/15 ("Civilians" FOLLOWED BY "Baghdad").*

FREQUENCY: It is a unary operator that has the parameter frequency, a user specified number that represents the minimum number of occurrences. The operator has a counter associated with the pattern occurrence, if the number exceeds the specified frequency, the operator propagates the pattern occurrences.

patternOccurrenceCounter++;

if(patternOccurrenceCounter == frequency)

Propagate the pattern occurrences to the subscribed parents

NOT: In LED, NOT operator does not support the frequency. According to Psnoop semantics in PD, if the frequency is not one (the default) in the NOT operator, then similar to the FREQUENCY operator, a counter is used to keep record of the number of times a pattern has occurred in between two patterns defined by the NOT operator. If the counter exceeds the specified frequency, the pattern is not detected. NOT operator is a ternary operator that has three patterns, two used to define the scope/range of detection. Those two patterns can be simple or composite. In the actual implementation, the default patterns that defines the scope of detection are *BeginStream* and *EndStream* indicating the beginning and end of the text stream respectively.

SYN: SYN has been incorporated in PD as an option. The algorithm for the SYN option has adopted the semantics of the OR operator. It has been extended to accept N operands. The N operands are extracted by the graph generator using the WordNet Database tool. The WordNet Database tool extracts

5.4 Graph Generator (GG)

The graph generator reads the postfix notation, inserted into the stack by the pattern processor, to construct the PDG in the pattern detector. As mentioned earlier, the graph generator interacts with WordNet database tool to extract synonyms of words. The graph generator uses Java WordNet Library (JWNL) [30] to provide this interaction. In JWNL, the method call to extract the synonyms of a word is given as:

```
IndexWord wordIndex = Dictionary.getInstance().
                          getIndexWord(POS.NOUN, "bomb");
Synset[ ] synonyms = wordIndex.getSenses();
```

Essentially, *Synset* is a set of synonyms, where each synonym is considered a different sense of the original word. In WordNet, a single word can have many senses, but not all senses are considered to be accurate. In information retrieval, this problem is considered as ***Word Sense Disambiguation*** [31]. According to the user patterns specified using Psnoop, it is difficult to determine which word sense is required. In InfoFilter, this problem has been tackled by allowing the extraction of the synonyms appearing in the first sense of the word.

To construct the PDGs, the graph generator uses the appropriate PD APIs according to the tokens read from the postfix notation stored in the stack. For instance, if the token is a binary operator such as NEAR, the graph generator pops the operands. If any of the operands is a simple pattern, the graph generator uses the API for creating the leaf node in a PDG, and then passes the reference of the leaf node. The reference is used by the API for constructing the internal node corresponding to the binary operator. Similarly, if any of the operands is a composite pattern the reference to the internal node in the sub-PDG that corresponds to that pattern is used by the API for constructing the internal node corresponding to the binary operator.

While constructing the PDGs, the graph generator sends the extracted keywords, phrases, regular expressions and synonyms to be stored in the suffix trie.

5.5 Stream Processor (SP)

The stream processor parses, and tokenizes the incoming streams. A parser for handling unstructured text has been implemented using the BreakIterator Class in java. The BreakIterator Class can detect text boundaries such as word boundaries, sentence boundaries, etc. The Java DOM parser is used to handle xml documents embedded in html pages and an html parser for parsing web pages.

The parser should extract the textual information present in the text streams. The textual information is then tokenized using the BreakIterator. The tokens are then matched against the extracted keywords, phrases, and regular expressions stored in the suffix tries (see section 5.6). If there is a match, the stream processor notifies the PD.

In addition to generating the notification when simple pattern occurrences are detected, the stream processor is also responsible for detecting structural pre-defined simple patterns such as begin-sentence and end-sentence.

5.6 Suffix Tries (Shared Data Structure)

Suffix tries are constructed over all the suffixes of the text. Suffix tries are characterized as space efficient and have less search time for small text [5][32][33]. Furthermore, the search algorithm of the suffix trie can be used to search words, prefixes, suffixes, substrings and regular expressions. In the suffix trie, each position in the text is considered to be a text suffix. The start and end position of the suffixes, represented by the subtries, are stored in the nodes of the trie [34]. Suffix trie for the keywords *"bomb"* and *"Iraq"* are shown in Figure 5.4. In this suffix trie, the branches connected to the root

node are labeled in alphabetical order. For instance, the branch that is labeled *"b"* is connected to the node that contains pointers and indices of the suffixes *"omb$"*, and *"$"* respectively. The dollar symbol($) is used to mark the end of the word. The suffix trie used in InfoFilter has been implemented in java.

5.6.1 Inserting patterns into a suffix trie

The construction algorithm starts with the root node that represents empty string. This is the node from where the construction begins. Consider the insertion of the string *"Iraq$"*. Number the word from left to right, from 1 to n, where n is the length of the word. In this example, it is 5 (for *"Iraq$"*). The suffixes are inserted into the trie in the order suffix(1,5), suffix(2,5), and so on. 1 represents the location from where the suffix start. So suffix(2,5) starts at 2 and ends at 5 (i.e., *"raq$"*). The word is scanned from left to right.

The insertion starts from the root since there is no edge/branch from the root node that starts with "I". Therefore, an edge labeled "Iraq$" is added and in the node a reference pointer to the PDG corresponding to the string "Iraq" is stored. Next, the suffix (2,5) is inserted, starting from the root. There is no edge that starts with r, hence another edge is added to the root that is labeled "raq$". This is repeated until the end of the string (i.e., insert suffix(3,5), suffix(4,5), suffix(5,5)). Note that if there is a branch that has the same label as the suffix, a pointer is added to the node to refer to the corresponding PDG.

5.6.2 Searching for patterns in a suffix trie

To look for the patterns in the input stream, stream processor matches the tokens, generated by the parsing process, with the patterns stored in the suffix trie. Consider the text in Figure A.1, the stream processor processes the first line of the text *"Three U.S.*

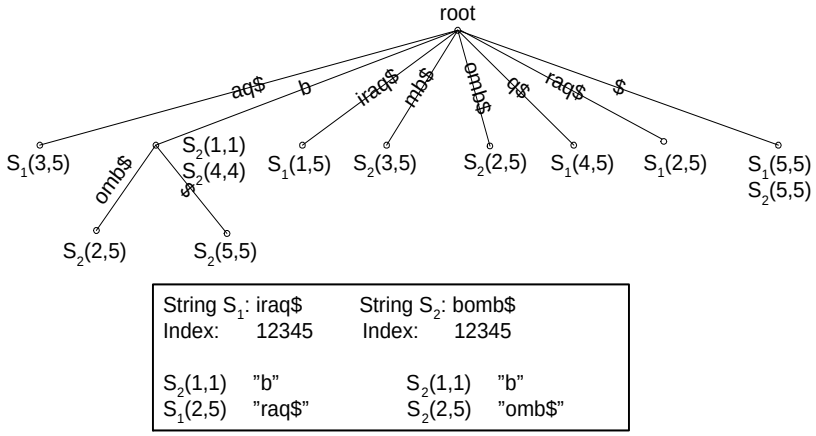

Figure 5.4 Suffix tries for the words *"Iraq"* and *"bomb"*.

soldiers killed in Iraq" and generates the following tokens, *"three"*, *"U.S."*, *"soldiers"*, *"killed"*, *"in"*, and *"Iraq."*. To search for *"Iraq"* in the suffix trie, the token *"Iraq"* gets numbered from left to right starting at 1. The search starts with the root node and it follows the branch that is labeled with *"i"*. Then, it compares the remaining indices with the pattern indices. If the indices are equal, it compares the second character of the token *"r"*, if there is a match it continues comparing the remaining characters. Hence, *"r"* matches with *"r"*, *"a"* with *"q"*, and so forth, until the label and the token are consumed. Thus the match is considered to be successful, since the node has a pointer that corresponds to the word *"Iraq"*. Finally, the stream processor triggers the PDG mapped to the word *"Iraq"*.

So far this section has explained the suffix tries and how they are used to store and match keywords and phrases. However, in this system, regular expressions are also supported. To represent a regular expression in a suffix trie, the wild-cards are considered as characters. Thus, the construction of the suffix trie for a regular expression is similar to words and phrases. Figure 5.5 shows a suffix trie for the regular expression "info*". However, when searching, the functionality of the wild-cards are considered.

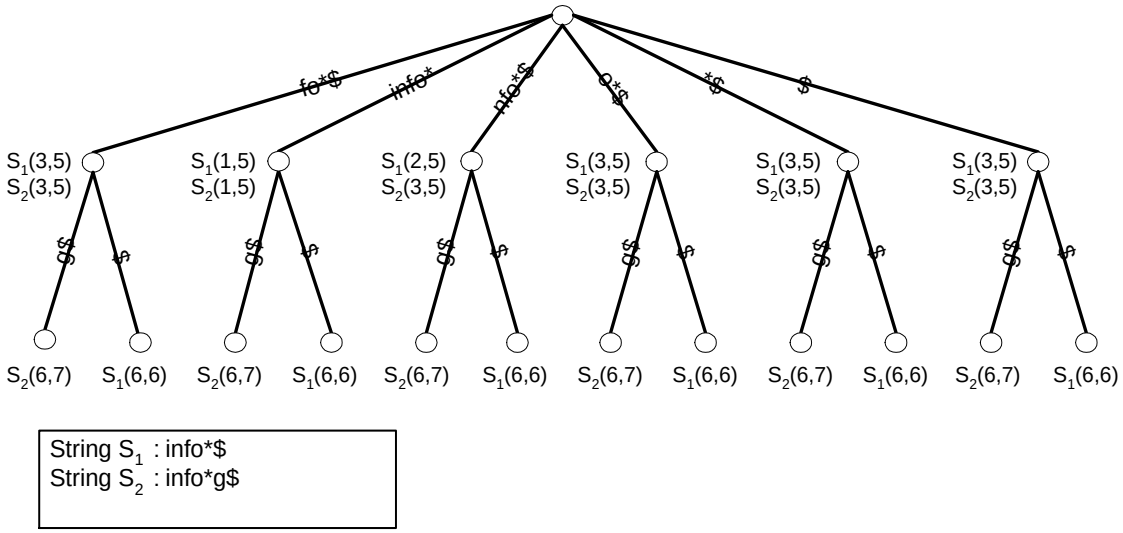

Figure 5.5 Suffix tries for the words *"info*"* and *"info*g"*.

While searching in the suffix trie, if a wild-card is encountered, the matching is done according to the type of wild-card (i.e., ? matches a single character). For instance, to search for "information" in the suffix trie shown in figure 5.5, the characters match until the wild-card "*" is encountered. Since there is no character after the wild-card, the * matches the rest of the word "information".

If there is a character after the wild-card *, the length n of the remaining branch label is computed. If the last n characters of the token matches the characters, the search is successful.

5.7 Notifier

The notifier sends notifications to users using email messages. The notifier is implemented using Java Mail APIs [35]to send messages to users subscribing to the system. The following code illustrates how the emails are sent.

public void postMail(String recipients[ ], String subject, String message,
String from) throws MessagingException {

```
String from = args[1];
String to = args[2];
// Get system properties
Properties props S̄ystem.getProperties();
// Setup mail server
props.put("mail.smtp.host", host);
// Get system properties
Properties props = System.getProperties();
// Setup mail server
props.put("mail.smtp.host", host);
// Get session
Session session = Session.getDefaultInstance(props, null);
// Define message
MimeMessage message = new MimeMessage(session);
// Set the from address
message.setFrom(new InternetAddress(from));
// Set the to address
message.addRecipient(Message.RecipientType.TO,
new InternetAddress(to));
// Set the subject
message.setSubject("Hello JavaMail");
// Set the content
message.setText("Welcome to JavaMail");
// Send message
Transport.send(message);
// Get session
```

```
        Session session S̄ession.getDefaultInstance(props, null);
        // Define message
        MimeMessage message n̄ew MimeMessage(session);
        // Set the from address
        message.setFrom(new InternetAddress(from));
        // Set the to address
        message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
        // Set the subject
        message.setSubject("Hello JavaMail");
        // Set the content
        message.setText("Welcome to JavaMail");
        // Send message
        Transport.send(message);
}
```

5.8 Pattern Detection Optimizations

Following the initialization of PD, an instance of the PD is created for each type of input stream. This instance stores the name of all patterns and their associated handles in a Hashtable. As mentioned earlier, for each simple pattern a leaf node is created in a PDG. The pattern detector maintains a hashtable that stores the a mapping between the pattern names and pattern nodes.

In Chapter 3, a naming convention was introduced to allow sharing of nodes in PDGs in order to reduce space and computation cost. The naming convention is based on the patterns. The nodes are named using the actual patterns or sub-patterns they correspond to, with space being replaced as an underscore character. For instance, Figure 5.6

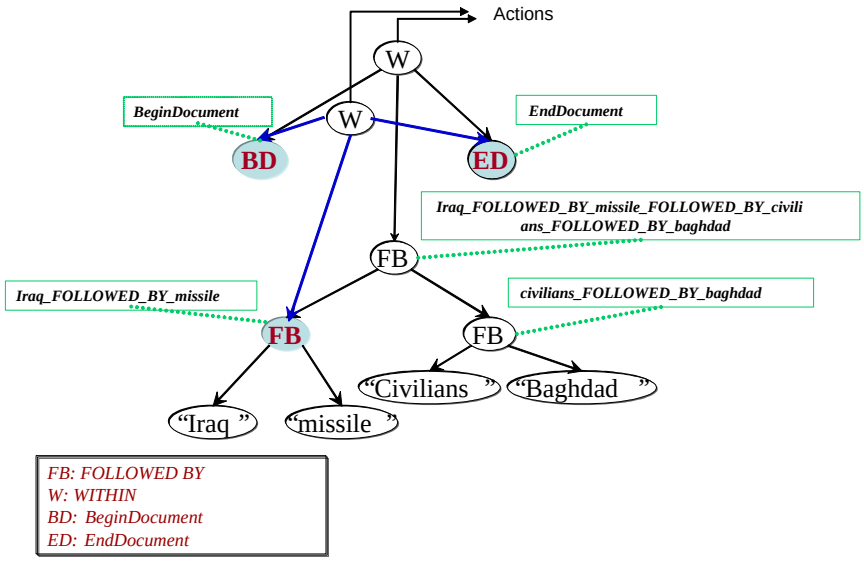

Figure 5.6 Sharing of nodes using the naming convention.

shows the sub-pattern *"Iraq" FOLLOWED BY "missile"*, the name of the internal node mapped to this composite sub-pattern carries the name, *"iraq_FOLLOWED_BY_missile"*. Essentially, this naming convention allows the graph generator to detect if a node exists (i.e., mapped to an identical pattern). For this purpose, the graph generator uses the hashtable that maps the nodes to the patterns in the PD.

Another optimization for sharing the nodes in the PDG has been implemented. When sharing is allowed, identical patterns can be represented by the same PDG. However, if a pattern consists of identical sub-patterns (i.e., a binary operator having the left and right patterns as identical patterns), this raises a problem of which pattern is detected, the left or the right pattern. Moreover, whether to propagate the pattern occurrence as a left pattern occurrence or a right pattern occurrence. To differentiate between a shared node that has the same parent node, a left and right label is added to the node. If a pattern occurrence is propagated from the left node, the pattern occurrence contains the label left, and the same for the right node. In ternary operators such as NOT, and

WITHIN, the middle node has a third label middle indicating a pattern occurrence propagated by the middle node. This optimization has been incorporated into the semantics of the operators supported by PD.

As can be observed from Figure 5.6, the internal node corresponding to the pattern *("Iraq" FOLLOWED BY "missile") FOLLOWED BY/10 ("Civilians" FOLLOWED BY "Baghdad")* is named using the pattern. However, the distance 10 is not included. The parameters associated with the patterns are not included to reduce the computation cost as explained previously in chapter 3. Based on the LED framework, the PD does not support the sharing of nodes. During the construction phase, if a node exists in a PDG, the PD does not create a new one. However, after the adding the optimized approach of sharing the nodes, a mechanism is required to add distance values to existing nodes. The construction of PDGs is done from leaf nodes to the root. When a NEAR / FOLLOWED BY operator is constructed or shared, a distance value is added. At the point of construction, the children node has no information about the parent nodes/subscribed nodes unless a parent node subscribes to it. The subscription is done when the parent node is constructed. Therefore, two temporary variables, implemented as an array list of 2 elements, is used to store the distance values in the order they are added. The first element is mapped to the left operand, and second element is mapped to the right operand. Typically, one parent node subscribes to a children node during the construction, unless a pattern contains two identical sub-patterns,i.e., left and right operands are equal.

Consider the pattern *("Iraq" FOLLOWED BY/10 "missile") NEAR ("Iraq" FOLLOWED BY/30 "missile").* First, construct the PDG for the sub-pattern *("Iraq" FOLLOWED BY/10 "missile").* Two leaf nodes are constructed representing "Iraq" and "missile". The FOLLOWED BY operator is constructed with the value 10 stored in the first element of the temporary variable. Then, a PDG for the second sub-pattern *("Iraq" FOLLOWED BY/30 "missile")* is constructed. Note that based on the naming conven-

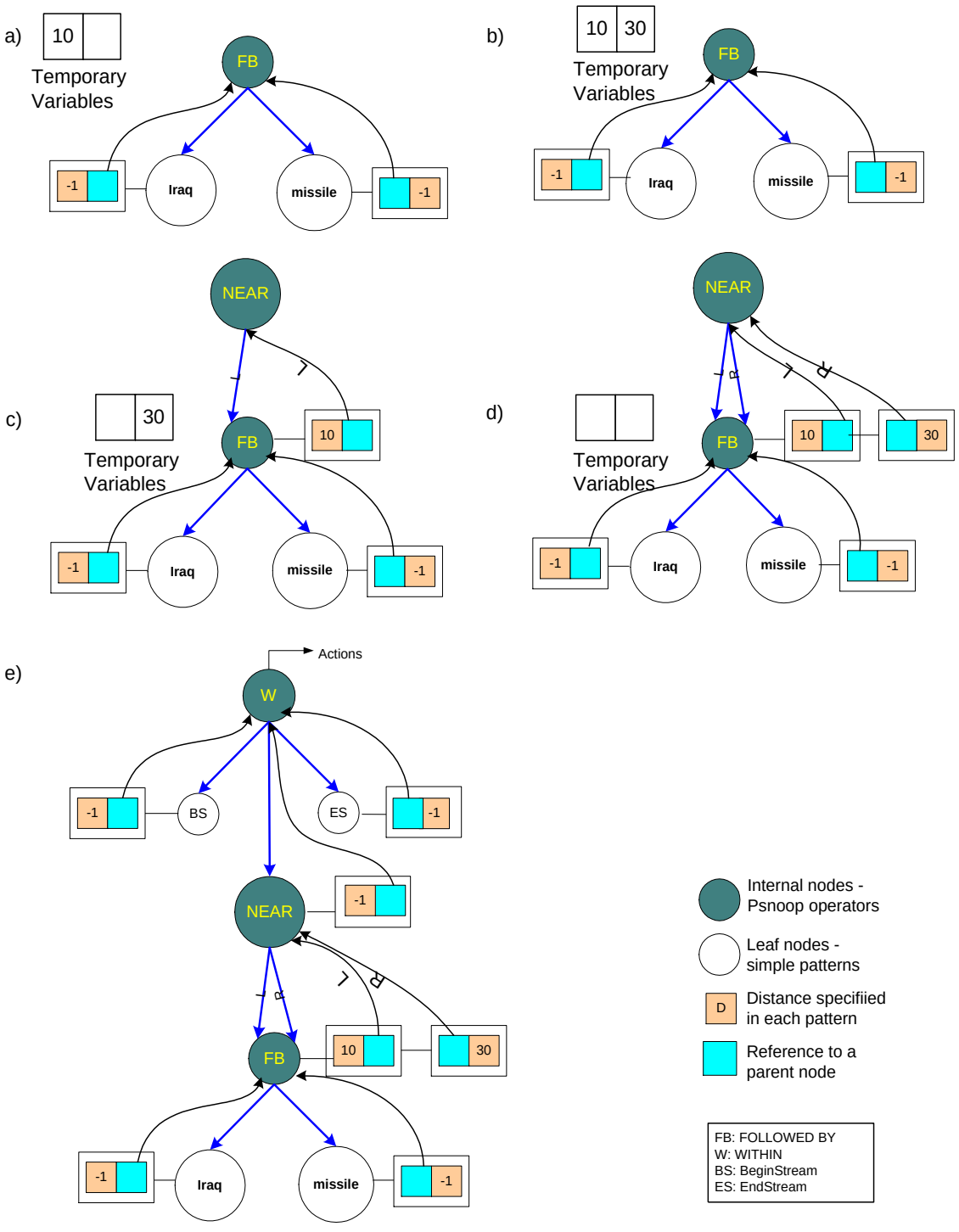

Figure 5.7 Using the temporary variable during construction of the PDG.

tion, the PDG for the second sub-pattern already exists. Instead of constructing a new one, the new distance value is added and it is stored in the second element of the array list. When the NEAR node is constructed, it subscribes to the same node, first with distance 10 and the next with distance 30. It extracts the distance in the first element since it is mapped to the left operand and add it to the subscriber list associated with a reference to itself. Same thing is applied to the second element. Figure 5.7 illustrates the construction of the PDG for the above example.

5.9 Pattern Deletion

In InfoFilter, for each user pattern a PDG is constructed to detect the pattern. When a pattern is deleted, the PDG should also be deleted to prevent the system from detecting the pattern. However, sharing of nodes imposes a problem while deleting a pattern. If a PDG is deleted and it is shared by another existing pattern, it leads to loss of information. Unsubscribing the pattern overcomes these problems. In figure 5.8, two patterns *("iraq" FOLLOWED BY (FREQUENCY/2("missile"))* and *(FREQUENCY/2("missile") NEAR "nuclear")* share nodes in their PDGs. Consider deletion of the first pattern *("iraq" FOLLOWED BY (FREQUENCY/2("missile"))*. In the shared node that corresponds to the pattern *FREQUENCY/2("missile")*, the entries in the subscriber list associated with the deleted pattern are removed. Pattern occurrences are propagated to the parent nodes present in the subscriber list. If the subscriber list is empty, the propagation of pattern occurrences is stopped, as shown in figure 5.8. The leaf node *"iraq"* and the internal node *"FB"* are both deleted from the PDG after they unsubscribe from the node *"FQ"*. A node is deleted by removing its mapping between the name of the pattern and the node in the hashtable present in the PD instance.

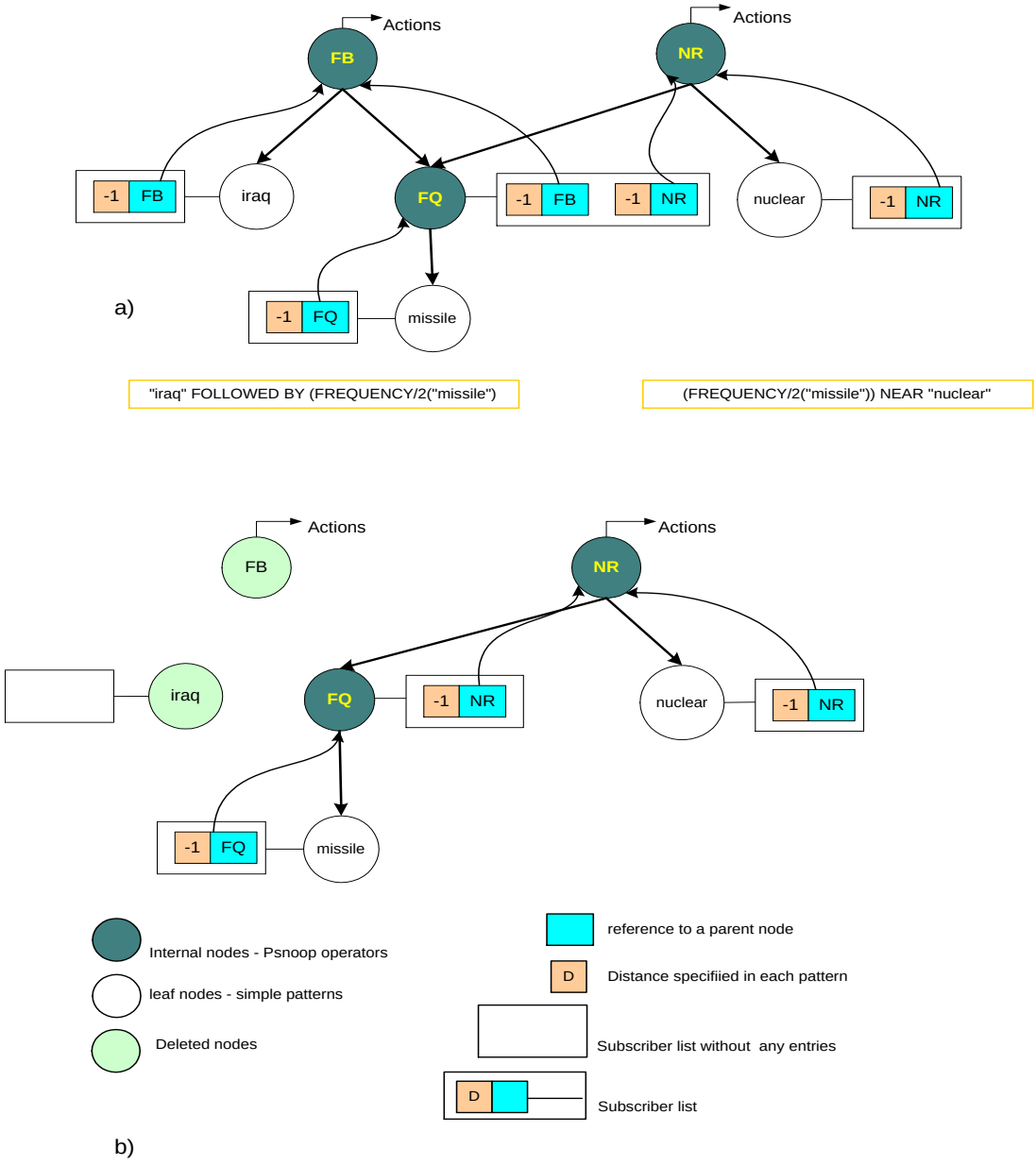

Figure 5.8 Deleting Patterns and their corresponding PDGs.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis, we have presented InfoFilter, a content-based system for filtering text streams. InfoFilter has been developed with an intent to support expressive user patterns using Psnoop and to provide on the fly filtering and notification on various streams. We have implemented an InfoFilter server that consists of Pattern Validator (PV), Processor (PP), Detector (PD), Graph Generator (GG), and Stream Processor (SP).

Psnoop, proposed and developed in this thesis, overcomes the limitations of the Information Retrieval Query Languages currently used for specifying user patterns. It provides a complete set of pattern operators and options such as *frequency*, *synonyms*, *followed by*, *Boolean operators*, *structural*, *wild card*, and *proximity*. *FREQUENCY* operator allows the user to specify minimum number of occurrences for a specific pattern. *SYN* option is a shorthand of specifying single-word patterns that carry the same meaning. As mentioned in the thesis, SYN adds flexibility to user specification and reduces detection cost. Boolean operators support the capabilities of the conventional Boolean operators with some extensions such as the frequency option added to the *NOT* operator. Using proximity operators *(NEAR and FOLLOWED BY)*, users can specify correlated patterns. To our best of knowledge, correlation among patterns has not been yet provided in information filtering literature. Context queries allows specification of correlated words and not complex patterns.

The semantics of Psnoop operators are interval-based, where a pattern is considered to occur over an interval defined by a start offset and end offset. The formal characterization of Psnoop operators were defined in unrestricted detection mode. In unrestricted

mode, not all pattern occurrences retained were needed. A naive approach adopting the recent context, introduced in event detection literature, was applied to enforce the notion of proximity while detecting patterns. It considers the recent pattern occurrences. This mode has further been extended to *recent-unique* mode where patterns that are consumed in detecting composite patterns are discarded. This capability does not allow generation of duplicate pattern occurrences as in the naive approach.

Efficient mechanism for detecting these complex patterns using the Pattern Detection Graphs, and their optimizations has been discussed. Sharing of PDGs and subPDGs has been allowed among identical patterns to reduce the detection cost including storage, propagation and merging of pattern occurrences. By detecting similar patterns over a single PDG, pattern occurrences are retained in a single PDG instead of separate PDGs. In addition to the above, sharing allows detection of multiple patterns over a single text stream, i.e. batch processing of patterns. This is done by sharing the nodes corresponding to the structural elements of text streams such as the beginning of a stream or end of stream. Once the beginning of a stream is detected, the shared node corresponding to it is notified initiating the detection of the PDGs sharing it. Also, a further optimization was introduced to handle sharing of nodes incorporating distance values. The subscriber list, used to retain references to parent or subscribed nodes, was modified to include the distance values associated with the references to the parent nodes. This increases the efficiency of computing distance between patterns. For detecting composite patterns that have identical sub-patterns, with the reference to the parent node, a label is attached to identify the pattern occurrence whether it is a left pattern occurrence or a right pattern occurrence.

In this thesis, a suffix trie was used to speed up the search time taken to look up the tokens. Extracted words, phrases, synonyms and regular expressions are stored by the graph generator in the suffix trie. A modification of the suffix trie has been implemented

to incorporate the features required in this system. In the nodes of the suffix trie, pointers to the corresponding leaf nodes of the PDGs are stored. Suffix trie is shared by the Graph Generator (GG) and the Stream processor (SP).

InfoFilter can be extended in a number of ways. Psnoop can be extended by allowing complex regular expressions and synonyms for phrases. Higher level specification of patterns that can be converted into Psnoop is another direction. We plan on linking InfoFilter with the web monitoring system (WebVigiL) [36] to filter web contents in a selective manner. In order to process XML streams, only the stream processor and pattern validator modules of the InfoFilter architecture need to be extended to handle the structure of XML elements. The system can also be extended to search for patterns in an entire web and for the generation of web ontology based on the patterns detected in web pages in a web server. Handling of approximate pattern matching can be explored further. Psnoop operators can also be formalized using the recent unique context. In the long term, this work can be extended to incorporate adaptive filtering, discovering patterns that can be of an interest to the user.

APPENDIX A

A DETAILED EXAMPLE

In this appendix, we present an example for illustrating how patterns are detected using the Pattern Detection Graphs (PDGs) in the Recent Unique mode.

A.1 Example

Consider a real-world example where a news analyst is analyzing the latest news. The analyst is interested in detecting the word *"Iraq"* occurring prior to two occurrences of *"missile"* appearing within a paragraph. This pattern can be specified in Psnoop as follows:

"Iraq" FOLLOWED BY (FREQUENCY/2("missile") WITHIN PARAGRAPH))

The pattern is processed and the corresponding PDG is constructed in the pattern detector. Figure A.3(a) shows the PDG for this pattern.

Suppose the article shown in figure A.1 has been processed and tokenized by the stream processor. An offset line shown in figure A.2 illustrates the occurrences of the simple patterns corresponding to the pattern that is to be detected (i.e., *"Iraq"* and *"missile"*).

Detecting these patterns over the PDG is illustrated in figure A.3. In recent-unique mode, the first occurrence of Iraq is propagated from the leaf node corresponding to *"Iraq"* to the parent node FOLLOWED BY. Then, it is replaced by the second occurrence as shown in figure A.3(b). Again, the second occurrence is replaced and also the third occurrence. At offset 400 the first occurrence of *"missile"* is detected, it is propagated to the FREQUENCY operator. The number of occurrences is computed, it does not equal or greater than the frequency specified. Thus, it is retained. The second occurrence of missile is propagated when it occurs at offset 421. The number of occurrences accumulated in the FREQUENCY node is computed and compared to the specified frequency. It has satisfied the frequency this time. Hence it is propagated to the WITHIN operator. The WITHIN operator in turn propagates the occurrences of the

Three U.S. soldiers killed in Iraq

MOSUL, Iraq (CNN) -- Attacks on U.S. convoys killed three American soldiers Sunday, including two who were dragged from a car and looted after they were shot in the northern Iraqi city of Mosul, according to witnesses.

Despite the losses, a U.S. military spokesman in Baghdad said guerrilla attacks on occupation forces in **Iraq** were becoming less significant.

"This is an enemy that cannot defeat us militarily, and in engagement after engagement we see the enemy breaking off, running away," Brig. Gen. Mark Kimmitt told reporters.
The soldiers who died in Mosul were hit by gunfire while riding in a civilian vehicle, two witnesses told CNN. An Army spokesman said the soldiers were from the 101st Airborne Division, which is based in Mosul.

A military spokesman disputed the accuracy of eyewitness reports that after the soldiers were shot, men came and cut their throats while they remained in the vehicle. U.S. Army Maj. Trey Cates, a spokesman for the 101st Airborne Division, told CNN the Army's investigation showed their bodies had no stab wounds or slash wounds.

Military sources who wished to remain anonymous confirmed other accounts that a crowd of Iraqis, including children, dragged the bodies from the vehicle and stripped them of personal effects and weapons.

Another U.S. soldier was killed and two were wounded Sunday morning near Ba'qubah, north of Baghdad, when a military convoy hit a roadside bomb, according to a spokesman for the 4th Infantry Division.

The wounded soldiers are in a stable condition, the spokesman said.
In addition, vehicle accidents claimed the lives of three other soldiers Friday and Saturday, and Iraqi sources said an Iraqi police colonel charged with security at oil installations was shot and killed in northern **Iraq**.

The deaths bring the total number of U.S. soldiers killed in the **Iraq** war to 432 -- 300 of them under hostile conditions.
Since May 1, when President Bush declared the end of major combat operations, 290 U.S. troops have died, 185 from hostile fire.

No reliable estimate of Iraqi deaths over the course of the conflict is available. The Associated Press reported an estimated 3,240 civilian Iraqi deaths between March 20 and April 20, but the AP said the figure was based on records of only half of Iraq's hospitals and that the actual number was thought to be significantly higher.

Civilian flights suspended after **missile** attack: U.S. authorities suspended civilian flights into Baghdad's international airport Sunday after a cargo plane was damaged by a surface-to-air **missile** over the weekend.

Military air traffic into Baghdad will continue, Coalition Provisional Authority spokesman Dan Senor said.

A DHL flight was struck by a surface-to-air **missile** shortly after takeoff Saturday from Baghdad, military officials said. The **missile** struck one of the jet's engines, and the aircraft returned safely to the airport, its left wing ablaze. No one was hurt.

The U.S. Air Force is conducting an investigation into the attack, Kimmitt said.
DHL and passenger carrier Royal Jordanian Airlines are the only companies flying into Baghdad.

Figure A.1 Detected patterns in an article posted in CNN website.

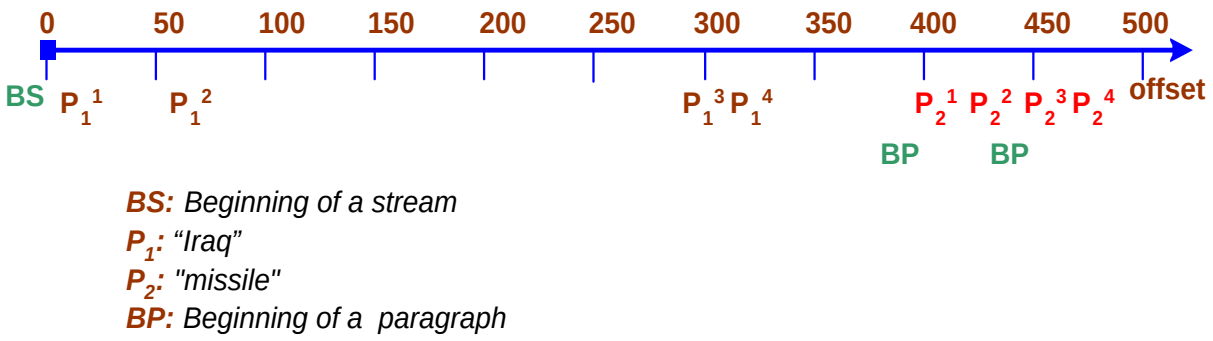

Figure A.2 Offset line representing the pattern occurrences in the article.

beginning of the paragraph which occurs at offset 395 and the two occurrences of missile to the FOLLOWED BY operator. In the FOLLOWED BY operator, the fourth occurrence of *"Iraq"* is combined with the occurrences propagated by the WITHIN operator. They are combined since the end offset of *"Iraq"* (306) is less than the start offset of the sub_pattern *FREQUENCY/2("missile")*(395)(i.e., they are not overlapping). Once they are combined, the pattern occurrences are propagated to the WITHIN operator that starts the detection of the patterns over a text stream. The WITHIN operator notifies the notifier that this pattern has been detected providing the name of the text stream.

Note in figure A.4 that *"missile"* occurs twice at offsets 447 and 458 respectively after another paragraph started. Nevertheless, as shown in the figure A.5(m), the pattern occurrences are not consumed since there is no occurrence of *"Iraq"* to combine it with. Thus, they are not used.

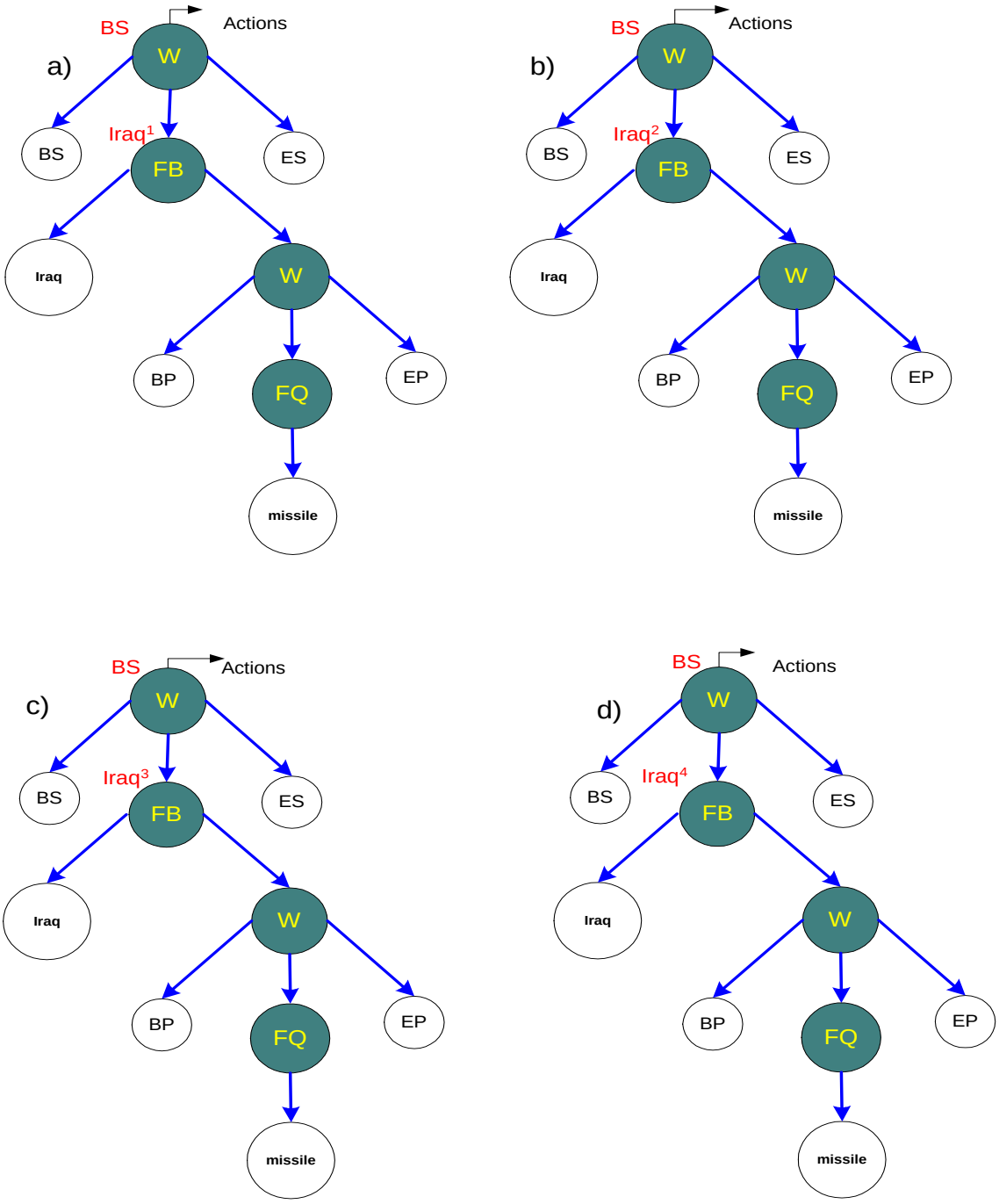

Figure A.3 Detection of the pattern *"Iraq" FOLLOWED BY (FREQUENCY/2("missile") WITHIN PARAGRAPH)*).

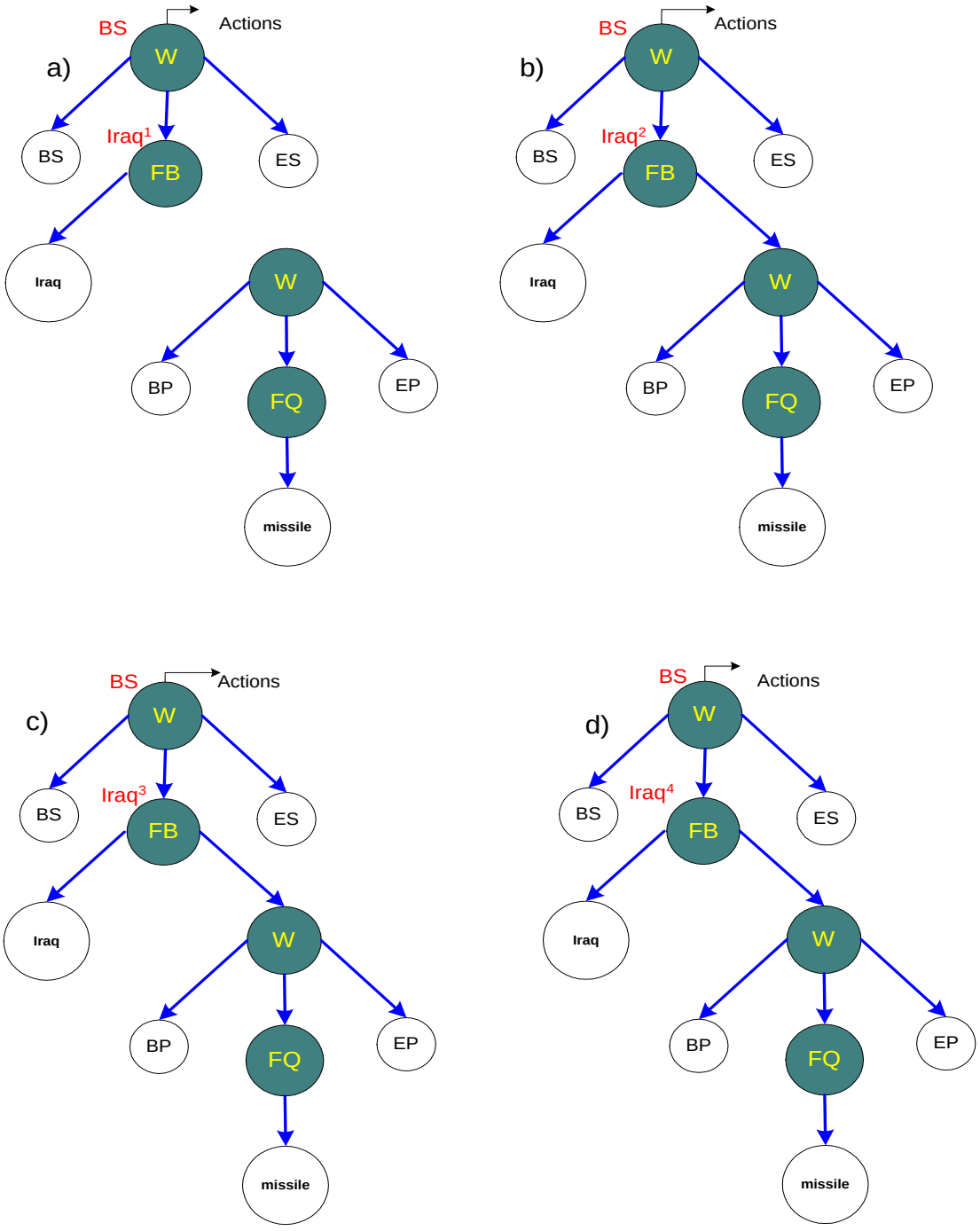

Figure A.4 Detection of the pattern *"Iraq" FOLLOWED BY (FREQUENCY/2("missile") WITHIN PARAGRAPH*))(contd..).

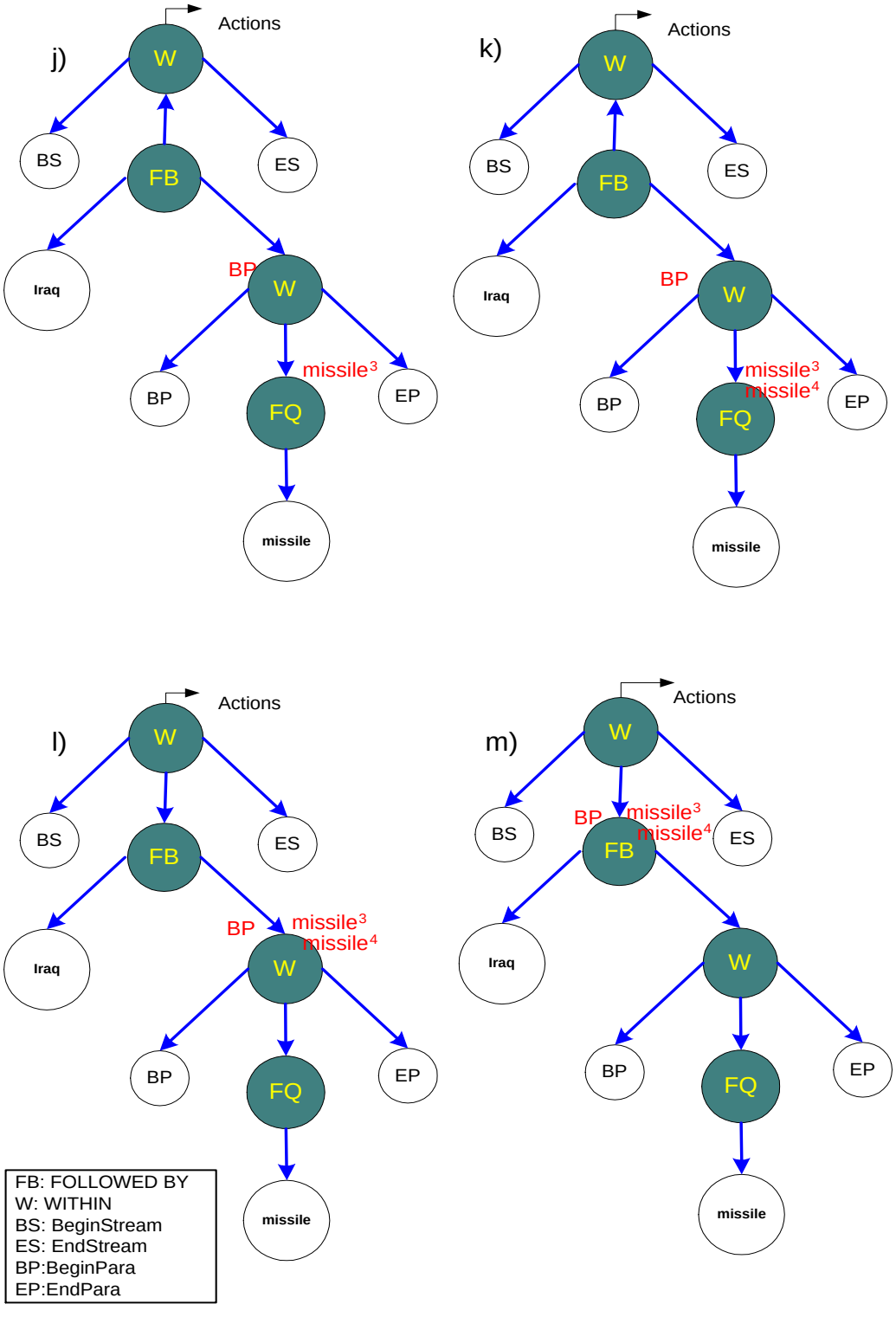

Figure A.5 Detection of the pattern *"Iraq" FOLLOWED BY (FREQUENCY/2("missile") WITHIN PARAGRAPH)*)(contd..).

APPENDIX B

COMPOSITE PATTERN DETECTION ALGORITHMS

In this appendix, the algorithm for detecting the composite patterns using the Recent Unique pattern detection mode is illustrated.

B.1 Algorithms for Recent-Unique Pattern Detection Mode

OR(P_1, P_2): In the recent-unique mode, if P_1 or P_2 is detected, it is propagated to the parent nodes with the start offset and end offset. For simple patterns, the start offset and end offset are the same.

```
PROCEDURE or_recentunique( P_i, parameter_list)
    For any pattern <p> detected
        Propagate to the parents with startoffset(p) and endoffset(p)
```

NEAR/D(P_1, P_2): In the recent-unique mode, if an instance of P_1 is detected, P_2's list is checked for a recent occurrence of P_2. If P_2's list contains the recent occurrence, the NEAR node propagates the occurrences of P_1 and P_2. To do that the NEAR node checks the subscriber list. If a distance value has been specified for a parent node, the distance is computed between the two occurrences. The difference between the end offset of P_2 and start offset of P_1 is computed. If the distance computed is less than the specified distance then the pattern occurrences are propagated to the corresponding parent node. Then, P_1's and P_2's lists are flushed. Since the subscriber list is ordered based on distance values. The first subscriber in the list has the minimum distance. Thus, if the distance computed between the pattern occurrences satisfies the minimum distance value, then they are propagated to all the parent nodes that have distance values greater than the minimum distance. If the distance does not satisfy the minimum distance, then the

computed distance is compared against the next minimum distance. All the above is repeated if a pattern occurrence P_2 is detected.

```
PROCEDURE near_recentunique(P_i, parameter_list)
  if pattern p_1 is detected
    if P_2's list is not empty and startoffset(p_2)≤startoffset(p_1)
        and endoffset(p_2)≤endoffset(p_1)
      while subscriberList is not empty
        for i:1 to size(subscriberList)
          if subscriberList[i].distance is specified
            if (startoffset(p_2)- endoffset(p_1))≤ subscr̄iberList[i].distance
              for j:i to size(subscriber list)
                Propagate p_2, p_1 to the parent with startoffset(p_2)
                        and endoffset(p_1)
              Remove p_2 and p_1 from P_2, P_1 lists
              break
            else
              increment i by one
          else
            for i: 1 to n
              Propagate p_2, p_1 to the parent with startoffset(p_2)
                and endoffset(p_1)
            Remove p_2 and p_1 from the P_2, P_1 lists
    else
      Replace p_1 in P_1 lists
```

```
if pattern p2 is detected
    if P1's list is not empty and startoffset(p1)≤startoffset(p2)
        and endoffset(p1)≤endoffset(p2)
      while subscriberList is not empty
        for i:1 to size(subscriberList)
          if subscriberList[i].distance is specified
            if (startoffset(p2)- endoffset(p1))≤ subscriberList[i].distance
              for j:i to size(subscriber list)
                Propagate p1, p2 to the parent with startoffset(p1)
                        and endoffset(p2)
              Remove p1 and p2 from P1, P2 lists
              break
            else
              increment i by one
          else
            for i: 1 to n
              Propagate p1, p2 to the parent with startoffset(p1)
                and endoffset(p2)
            Remove p1 and p2 from the P1, P2 lists
```

FOLLOWED BY/D(P_1, P_2): In the recent-unique mode, if an instance of P_1 is detected, it replaces the recent instance in P_1's list. If an occurrence of P_2 is detected, P_1's list is checked for a recent occurrence of P_1. If P_1's list contains the recent occurrence, the FOLLOWED BY node propagates the occurrences of P_1 and P_2. To do that the FOLLOWED BY node checks the subscriber list. If a distance value has been specified for a parent node, the distance is computed between the

two occurrences. The difference between the end offset of P_1 and start offset of P_2 is computed. If the distance computed is less than the specified distance then the pattern occurrences are propagated to the corresponding parent node. Then, P_1's and P_2's lists are flushed. Since the subscriber list is ordered based on distance values. The first subscriber in the list has the minimum distance. Thus, if the distance computed between the pattern occurrences satisfies the minimum distance value, then they are propagated to all the parent nodes that have distance values greater than the minimum distance. If the distance does not satisfy the minimum distance, then the computed distance is compared against the next minimum distance.

```
PROCEDURE followedby_recentunique(P_i, parameter_list)
  if pattern p_1 is detected
     Replace p_1 in P_1 lists

  if pattern p_2 is detected
    if P_1's list is not empty and startoffset(p_2)≤endoffset(p_1)
      while subscriberList is not empty
        for i:1 to size(subscriberList)
          if subscriberList[i].distance is specified
            if (startoffset(p_2)- endoffset(p_1))≤ subscriberList[i].distance
              for j:i to size(subscriber list)
                Propagate p_1, p_2 to the parent with startoffset(p_1)
                        and endoffset(p_2)
              Remove p_1 and p_2 from P_1, P_2 lists
              break
            else
```

```
            increment i by one
        else
            for i: 1 to n
                Propagate p1, p2 to the parent with startoffset(p1)
                    and endoffset(p2)
                Remove p1 and p2 from the P1, P2 lists
```

NOT/F(P_2)(P_1,P_3): In the recent-unique mode, if an instance of P_1 is detected, it replaces the recent instance in P_1's list. If an occurrence of P_2 is detected, it is inserted into P_2's lists. In NOT operator, all occurrences of P_2 are accumulated provided that there they occur after the endoffset of the recent occurrence of P_1 and prior to the startoffset of the pattern occurrence of P_3. When an occurrence of P_3 is detected, P_1's list is not empty, and the number of occurrences of P_2 is less than the specified frequency, the occurrences of P_1 and P_3 are propagated to the subscribers. After detection, P_1's and P_3's lists are flushed and if P_2's list is not empty, it is also flushed.

```
PROCEDURE not_recentunique(Pi, parameter_list)
  if pattern p1 is detected
      Replace p1 in P1 lists

  if pattern p2 is detected
     if P1's list is not empty and endoffset(p1)<startoffset(p2)
         Insert p2 into P2 lists

  if pattern p3 is detected
     if P1's list is not empty and endoffset(p1)<startoffset(p3)
```

if P_2's list is not empty

for all p_2's in P_2's list

if endoffset(p_2)>startoffset(p_3)

Remove p_2 from the P_2's lists

if size(P_2's list) $\geq$ F

if P_1's list is not empty

Flush P_1's list and P_3's lists

else

Propagate p_1, p_3 to the parent with startoffset(p_1)

and endoffset(p_3)

if P_2's list is not empty

Flush P_2's lists

Flush P_1's and P_3's lists

(P_2)**WITHIN(P_1,P_3):** In the recent-unique mode, if an instance of P_1 is detected, it replaces the recent instance in P_1's list. If an occurrence of P_2 is detected after the end offset of the recent occurrence in P_1's list, the pattern occurrences of both P_1 and P_2 are propagated to the subscribers. If an occurrence of P_3 is detected after the recent occurrence in P_1's lists (i.e., end offset of p_1 is less than p_3), both P_1's and P_3's lists are flushed.

PROCEDURE within_recentunique(P_i, parameter_list)

if pattern p_1 is detected

Replace p_1 in P_1 lists

if pattern p_2 is detected

if P_1's $\bar{\text{l}}$ist is not empty and endoffset(p_1)<startoffset(p_2)

Propagate p_1, p_2 to the parents with startoffset(p_1)

and endoffset(p_2)

Remove p_1 and p_2 from the P_1's, P_2's lists

if pattern p_3 is detected

if P_1's list is not empty

if endoffset(p_1)<startoffset(p_3)

Flush P_1's

Flush P_3's lists

FREQUENCY(P): In the recent-unique mode, if an instance of P is detected, it is inserted into the list. If the number of occurrences of P is equal to the frequency specified, the set of occurrences is propagated and the first occurrence is discarded.

PROCEDURE frequency_recentunique(P_i, parameter_list)

if pattern p is detected

Insert p into P's list

if size(P's list) $\geq$ frequency

Propagate p_f, p_l to the parents with startoffset(p_f)

and endoffset(p_l)

Remove p_f from the P_1's list

REFERENCES

[1] T. Yan and H. Garcia-Molina, "The sift information dissemination system," *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 4, pp. 529 – 565, December 1999.

[2] D. Oard and G. Marchionini, "A conceptual framework for text filtering," University of Maryland, December 2001.

[3] P. Jacobs and L. Rau, "Scisor: extracting information from on-line news," *Communications of the ACM*, vol. 33, no. 11, pp. 88 – 97, November 1990.

[4] Y. Lashkari, M. Metral, and P. Maes, "Collaborative interface agents," *Proceedings of the twelveth national conference on Artifical Intelligence*, vol. 1, pp. 444–449, October 1994.

[5] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. New York: ACM Press / Addison-Wesley, 1999.

[6] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, Inc., 1983.

[7] M. W. Berry, *Survey of text mining : clustering, classification,and retrieval*. New York: New York : Springer-Verlag, 2004.

[8] C. Fellbaum, "Wordnet: An electronic lexical database," MIT press, 1998.

[9] K. Aas, "Survey on personalized information filtering systems for the world wide web," December 1997.

[10] C. Stevens, "Knowledge-based assistance for accessing large, poorly structured information spaces," Ph.D. dissertation, , Department of Computer Science. University of Colorado, Boulder, 1993.

[11] U. Manber and S. Wu, "Glimpse: A tool to search through entire file system," University of Arizona, Tuscon, October 1993, pp. 92–94.

[12] M. Araújo, G. Navarro, and N. Ziviani, "Large text searching allowing errors," in *Proc. of WSP'97*. Carleton University Press, 1997, pp. 2–20.

[13] P. W. Foltz, "Using latent semantic indexing for information filtering," in *Proceedings of the conference on Office information systems*. Cambridge, Massachusetts, United States: ACM Press, 1990, pp. 40–47.

[14] D. R. H. Miller, T. Leek, and R. M. Schwartz, "A hidden markov model information retrieval system," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. Berkeley, California, United States: ACM Press, 1999, pp. 214–221.

[15] "Mondou: interface with text data mining for web search engine," in *Proceedings of the Thirty-First Hawaii International Conference*, vol. 5, Hawaii, Jan 1998, pp. 275 – 283.

[16] P. C. Wong, P. Whitney, and J. Thomas, "Visualizing association rules for text mining," in *Information visualization Proceedings. IEEE symposium on Information Visualization*, San Francisco, California, USA, October 1999, pp. 120 – 123, 152.

[17] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts, and Detection," 1994, pp. 606–617.

[18] R. Adaikkalavan and S. Chakravarthy, "SnoopIB: Interval-Based Event Specification and Detection for Active Databases," in *Advances in Databases and Information Systems (ADBIS)*, September 2003, pp. 190–204.

[19] S. Chakravarthy, "Design of sentinel: An object-oriented dbms with event-based rules," *Information and Software Technology*, vol. 36, no. 9, pp. 559–568, 1994.

[20] M. L. Anwar, E. and S. Chakravarthy, "A new perspective on rule support for object-oriented databases," *ACM SIGMOD Conf. on Management of Data*, pp. 99–108, 1993.

[21] H. Lee, "Support for temporal events in sentinel: Design, implementation, and preprocessing," Gainesville, 1996.

[22] R. Dasari, "Events and rules for java: Design and implemenation of a seamless approach," Gainesville., 1999.

[23] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data and Knowledge Engineering*, vol. 14, no. 10, pp. 1–26, 1994.

[24] A. Galton and J. Augusto, "Two approaches to event definition," in *In proceedings of 13th International Conference on Database and Expert Systems Applications*, Aix en Provence, France, 2002.

[25] V. Krishnaprasad, "Event detection for supporting active capability in an oodbms: Semantics, architecture, and implementation," Gainesville, FL 32611, 1994.

[26] A. Drozdek, *Data Structures and Algorithms in Java.* New York: McGraw-Hill, Inc., 2001.

[27] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, New York, 1999.

[28] "Java compiler compiler (javacc) - the java parser generator." [Online]. Available: http://www.webgain.com/products/java_cc

[29] S. Chakravarthy and D. Mishra, "Towards an expressive event specification language for active databases," in *5th International Hong Kong Computer Society Database Workshop on Next generation Database Systems*, Kowloon Shangri-La, Hong Kong, 1994, pp. 606–617.

[30] "Jwnl (java wordnet library)." [Online]. Available: http://sourceforge.net/projects/jwordnet

[31] C. Stokoe, M. P. Oakes, and J. Tait, "Word sense disambiguation in information retrieval revisited," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM Press, 2003, pp. 159–166.

[32] S. Wu and U. Manber, "Fast text searching allowing errors," October 1992.

[33] M. Nelson, "Fast string searching with suffix trees," *Dr. Dobb's Journal*, August 1996.

[34] S. Sahni, "Data structures, algorithms, and applications in java," 1999. [Online]. Available: http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/suffix.htm#reference

[35] "Sunmicrosystems, javamail api specification v 1.3.1." 2003.

[36] S. Chakravarthy, "Webvigil: An approach to just-in-time information propagation in large network-centric environments," *Second International Workshop on Web Dynamics*, August 2002.

BIOGRAPHICAL STATEMENT

Laali Elkhalifa was born in Medani, Sudan, in 1976. She received her B.S. degree from Omdurman Ahlia University, Sudan, in 1997. In the Summer of 2001, she started her graduate studies in Computer Science at The University of Texas, Arlington. She received her Master of Science in Computer Science from The University of Texas at Arlington, in May 2004. Her research interests include active databases, data mining, text mining, information retrieval, and information filtering.