

How To Search for Complex Patterns over Streaming and Stored Data*

S. Chakravarthy[†] L. Elkhalfa[†] N. Deshpande[†] R. Adaikkalavan[†] R. Liuzzi[‡]

Abstract

The colossal amount of digitized information available has resulted in overloading users who need to navigate this information for their routine requirements. Information filtering deals with monitoring text streams to detect patterns and retrieval of documents by searching for patterns over stored data. Although information filtering systems and search engines have been effective in reducing this information overload, they support only keyword searches and queries that use Boolean operators. Consider searching a full text patent database for documents containing more than n occurrences of a particular pattern, or for documents that have a particular pattern followed by another pattern within a specified distance. Such complex patterns involving pattern frequency and sequence of patterns as well as patterns involving proximity, structural boundaries and synonyms are not supported by current filtering systems and search engines. Expressive pattern detection over text streams 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. In this paper, we discuss a novel approach that provides an expressive pattern search over text streams as well as stored data. Our system consists of two main components: InfoFilter, a content-based information filtering module that detects complex patterns over text streams and InfoSearch that retrieves stored documents based on expressive patterns.

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 amounts of information while retrieving task-relevant data. Applying appropriate searching and filtering mechanisms to retrieve only relevant data

becomes critical to avoid information overload (or retrieving very large number or portions of documents). *Information filtering* [1, 2, 3, 4, 5] is the process of extracting relevant or useful portions of documents from continuous streams of textual data based on relatively static user patterns (or queries). On the other hand, *Information Retrieval* [6, 7, 8, 9] is the process of extracting relevant or useful portions of documents from a relatively static collection of documents based on a stream of incoming user patterns (or queries).

Both in information filtering and retrieval, expressiveness of pattern (or query) specification by a user and its detection play a significant role. In other words, in order to extract useful or meaningful information, users need to have the ability to specify complex patterns. In information retrieval systems, user queries are specified using Information Retrieval Query Languages (IRQLs) [7]. On account of 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 [8] are also used for matching queries against incoming text streams or to extract the relevant information from stored documents. However, a critical limitation of these filtering systems and search engines is that they can support only simple patterns or a simplistic combination of patterns using Boolean operators.

Consider a real world example where a federal agent is tracking terrorist-related information streaming from various resources. He/she is interested in the occurrence of the word bomb followed by the word ground zero occurring twice, along with the word automotive or its synonyms (i.e., ((bomb FOLLOWED BY ground zero) occurring twice) AND automotive (or its synonyms)). This pattern contains keywords, sequence (FOLLOWED BY), phrase, frequency, synonyms, and a Boolean operator. This pattern cannot be expressed using current query specifications as they do not support the following: i) quantification of multiple occurrences (or frequency) of patterns and complex compositions, and ii) a user cannot include synonyms in the pattern, and is required to explicitly list all the synonyms as separate patterns.

*This research was supported, in part, by IIS-0326505, EIA-0216500, MRI 0421282, and IIS 0534611

[†]Information Technology laboratory, Dept of CSE, The University of Texas at Arlington; email: sharma@cse.uta.edu

[‡]Raymond Technologies

Thus, current query languages are quite restrictive in their expressive power and need to be extended and generalized to address the specification of meaningful complex user patterns. On the other hand, ability to specify complex patterns will not be meaningful or effective without a correct and efficient mechanism for their detection in real-time. In this paper, we introduce a novel approach, InfoFilter [5] and InfoSearch [9], for expressive pattern matching over text streams and stored data, respectively.

The paper is organized as follows. Section 2 gives a description of the *Pattern Specification Language* supported by our system. Section 3 explains the architecture of our system including the design and working of InfoFilter and InfoSearch. Brief summary of the experiments conducted are discussed in Section 4. We discuss the related work in Section 5. Conclusions are provided in Section 6.

2. Pattern Specification Language

PSL, an expressive pattern specification language, allows the specification of complex patterns. Occurrence of a *Pattern P* is a Boolean function whose domain is an offset interval and range is TRUE or FALSE, depending upon whether the specified pattern occurs in that interval. According to the semantics of PSL, a pattern is classified into a simple and composite pattern.

Simple patterns form the basic building block of the PSL. 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**. A simple pattern is denoted by P[Os, Oe], where Os = Oe (i.e., the starting and ending offset of the pattern is the same). A simple pattern occurrence is an atomic occurrence of a simple pattern. It occurs over an interval [Os, Oe] and it is detected at the end of the interval (i.e., Oe). PSL supports two types of simple patterns, system-defined (e.g., BeginDoc, EndDoc, BeginPara, EndPara) and user-defined (single word, phrase and regular expression).

A *composite pattern* is an expression constructed using simple patterns, previously constructed composite patterns, PSL operators and options. PSL provides a comprehensive set of operators, OR, non-occurrence (NOT/N), sequential (FOLLOWED BY/N), structural (WITHIN/N), frequency (FREQUENCY/N), proximity (NEAR/N) and the option synonyms (SYN) that allow users to compose *composite patterns*. Table 1 shows various operators and their functionalities with examples. For detailed explanation of PSL, please refer [5, 9].

3. System Architecture

Figure 1 illustrates the architecture of our system. Below, we discuss various modules of our system briefly.

3.1 Pattern Validator

Pattern validator accepts the input patterns in infix notation from users according to the BNF [5] of PSL. Once the patterns are validated they are decomposed into tokens. The tokens in a pattern can be keywords, phrases, system defined patterns, operators and other delimiters allowed by the language. These tokens are sent to the pattern processor.

3.2 Pattern Processor

Pattern processor accepts the patterns or tokens in the infix notation and converts them to postfix notation. Infix notation is easier to specify and the default operator precedence can be altered by using parenthesis. To evaluate the pattern, with emphasis on the operator precedence and minimization of the use of parentheses, the infix notation is converted to postfix notation. Patterns in postfix notation are easier to evaluate as the operands precede the operators. It then sends the patterns in postfix notation to the graph generator.

3.3 Graph Generator

Graph generator performs two main tasks. (i) Generate pattern detection graphs from user-specified patterns by invoking APIs from the pattern generator library. The graph generator constructs the PDGs corresponding to the patterns in the pattern detector, and interacts with the WordNet Database tool to extract the synonyms of single words if specified. It also extracts and stores the keywords, phrases and regular expressions, embedded in these patterns in a shared data structure, suffix trie. (ii) Propagate the extracted simple patterns to the *InfoFilter* and *InfoSearch* modules so that they can be detected from text streams or stored data and propagated to the pattern detection graph. With *InfoFilter*, when simple patterns (keywords, phrases and regular expressions) are extracted, the graph generator stores them in the form of a shared data structure, *suffix trie*, with references to the node corresponding to the original pattern. With *InfoSearch*, the extracted simple patterns are stored in the simple pattern storage for further processing.

WordNet [10] 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.

3.4 Pattern Detection Graph (PDG)

A user pattern is internally represented as a PDG. It maintains 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

<i>Operators</i>	<i>Purpose</i>	<i>Examples</i>
OR	Provide optional criteria in specifying patterns	<i>“bomb” OR “explosive”</i> <i>(“bomb” NEAR/3 “automotive”) OR (“bomb” NEAR “building”)</i>
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	<i>NOT (“retrieval”) (“information”, “filtering”)</i> <i>NOT/2 (“retrieval”) (“information”, “query language”)</i> <i>NOT (“information” FOLLOWED BY/4 “retrieval”) (BeginPara, EndPara)</i>
NEAR	Detect patterns that occur within <i>N</i> words of each other Detect patterns that occur within the same document	<i>“information” NEAR/2 “filtering”</i> <i>“information” NEAR “filtering”</i>
FOLLOWED BY	Detect patterns that occur in certain order within <i>N</i> words of each other Detect patterns that occur together in certain order within the same document	<i>“data” FOLLOWED BY/2 “structures”</i> <i>“data structures” FOLLOWED BY “algorithm”</i>
WITHIN	Define the scope of detection within a document, a paragraph or a sentence Define a range for detecting patterns within a document	<i>(“information” NEAR “retrieval”) WITHIN (BeginPara, EndPara)</i> <i>(“information” NEAR/2 “retrieval”) WITHIN (“InfoFilter”, “Psnop”)</i>
FREQUENCY	Define minimum number of occurrences of a pattern	<i>FREQUENCY/5 (“Iraq”)</i>
SYN	Keyword synonyms	<i>“bomb” [SYN]</i>

Table 1. Summary of Psnop operators and their functionalities.

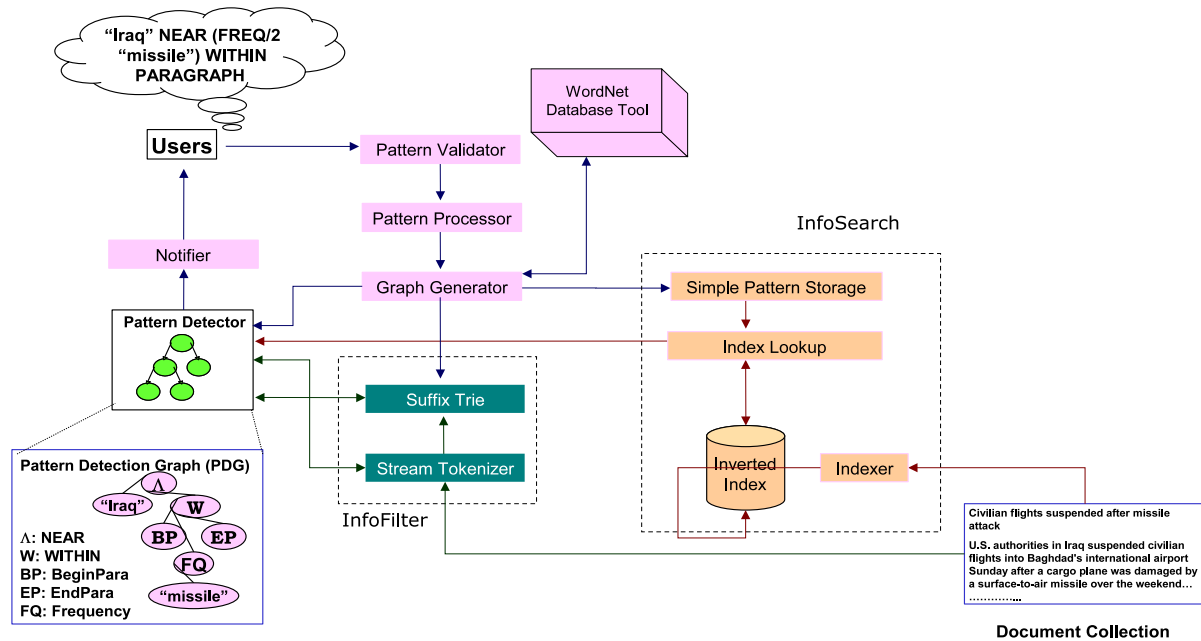


Figure 1. System Architecture

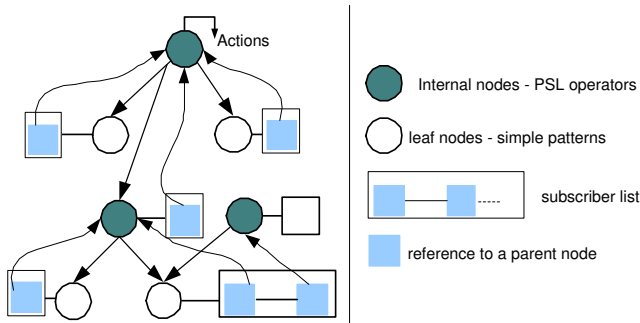


Figure 2. Pattern Detection Graph

to form a complex pattern as shown in Figure 2. The PDG is constructed recursively in a bottom-up fashion. The leaf nodes are created first, followed by the operators defined on these leaf nodes. When a parent node is created, it subscribes to its children. Essentially, this means that the reference of the parent is given to the children, so that data can be passed from a child to its parent. Thus, every node has a subscriber list that has a reference to each of its parents. To optimize space requirements, the graph generator shares PDG nodes wherever possible. This is achieved by keeping a single, common PDG or sub-PDG for a common expression or sub-expression. This avoids creation of a new PDG, if a PDG has already been created for a previous expression or sub-expression.

3.5 Pattern Detector

Efficient detection of user-specified complex patterns is extremely critical both for streaming as well as stored data. 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. One approach to detection is by using *Histories* (Global and Composite). The other approach for detection is usage of *Detection Modes* (Restricted and Unrestricted). These approaches are explained in detail with the help of examples in [5, 9].

The pattern detector is a library that provides APIs to construct the PDGs for the user patterns. Once simple patterns are detected in the stream processor (for streaming data) or in the index lookup (for stored data), pattern detector passes the notifications to the corresponding leaf nodes in the PDG leading to the computation of the associated complex pattern occurrences. As mentioned, the pattern occurrences propagate 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. A node in the graph can be associated with

a rule, which means that a predefined action can be taken when that pattern node is triggered. 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 document information is present in the pattern occurrences.

The flow of data up a PDG and the merging of tuples by the operators to detect patterns in documents has a similarity with composite event detection using event detection graphs [5, 9] in active databases. Event occurrences are propagated up an event detection graph, in which the composite nodes merge their inputs based on criteria known as parameter contexts. Since there is similarity in the data flow in the event detection system and our system, we use the framework of such an event detection engine as the backbone for our pattern detector. The paradigm of data flow has been kept, with operators and semantics replaced to suit the domain of information filtering and retrieval.

In *InfoFilter*, when a leaf node is notified, it receives the start and end offset of a *simple pattern*. On the other hand, in *InfoSearch* when a leaf node in the pattern detector receives a *set of tuples* from the Index Interface, it passes a reference to this set to its parent nodes. In *InfoFilter*, in the internal nodes, the simple patterns are merged using the operator semantics and the child patterns offsets. With *InfoSearch* when internal nodes merge their input set of tuples to create a merged output set, they pass a reference to this set to their parents. The root node has a rule associated with it, which essentially directs the output of the pattern detection to the user through e-mail or other forms of notification. Even though the set of operators used for both stream data and stored data are same, the computation at each node (or operator) are different. With streaming data, operators operate on **single** pattern and their offset at a time, whereas with stored data they operate over a **set of tuples** obtained using the index lookup. The stream data is processed as they come in; fairly static stored data is indexed first and then used for pattern detection. Both have their utility in different application domains.

3.6 InfoFilter

InfoFilter [5] analyzes text streams based on the content and structural information, and notifies the users when their patterns of interest are detected.

Simple patterns are stored by the graph generator in a suffix trie. A suffix trie is characterized as a space efficient representation taking significantly less search time. Both features are required by *InfoFilter* to detect patterns. Using the suffix trie searching algorithm, the tokens are matched against the stored words, phrases, regular expressions and synonyms. For those that do exist, the stream processor notifies the pattern detector accordingly. The suffix trie receives the patterns from the graph generator after the pat-

terns are mapped to graphs. For each incoming text stream, the stream tokenizer interacts with the assigned pattern detector, according to the type of stream. For example, a stream processor handling the web pages interacts with the pattern detector associated with web pages. The stream tokenizer 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. Once processed, the pattern processor sends these patterns to the graph generator.

InfoFilter processes 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 can be associated with different types of streams. That is, a user can specify the type of stream (such as email, video caption, HTML, word) 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. InfoFilter continuously monitors multiple types of streaming text, detects simple pattern occurrences and notifies the corresponding pattern detectors. Parsers and tokenizers have been developed for HTML, XML, text, and word document types.

3.7 InfoSearch

InfoFilter [9] allows the user to specify complex queries and returns information about every occurrence of the pattern specified in the query. The scope of the search is a pre-indexed document collection (e.g., root of a Web site), and the information returned is the document (e.g., Web page) in which the pattern occurs, and the position of every occurrence of the pattern within each document. This is a two step process in which the first step involves forming a PDG corresponding to the query. The next step involves a lookup for the index to detect the occurrences of the pattern (including complex patterns) that must be detected over the PDG to generate the query result.

InfoSearch [9] accepts complex queries from the user, searches an index built on a collection of documents, and returns a list of documents that contain the specified pattern. It also returns the starting and ending position of each pattern occurrence within a document. The system can be broadly divided into two components: First, an expressive query language through which the user can specify patterns involving term frequency, sequences, proximity and containment is required. Second, a pattern detection engine capable of getting the required information from the index, and processing this information to generate results in re-

sponse to a user query. The InfoSearch modules are specific to index-based retrieval and search, and the operators use algorithms that have been modified to detect patterns over data retrieved from an index. The pattern detection engine, which forms the core of the system and includes the operator functionality has been changed to process inputs in the form of sets of tuples. In addition, a generic index interface specification and design has been added. The architecture, generation of PDGs and other aspects are common to both InfoFilter and InfoSearch.

In InfoSearch, unlike InfoFilter, while generating the graph, the graph generator stores the keywords specified in the query in a keyword buffer. Once the PDG is generated, the graph generator queries the index for each of the keywords it has stored in its buffer. This is done through the index interface. The index interface module is responsible for retrieving the *hits* for each keyword from the index. These hits are wrapped into a set of *tuples* and passed on to the leaf node that represents the keyword. Thus, a monotonically decreasing set of data propagates up the PDG, and the output of the root node is the answer to the query which is returned to the user.

The detection engine of InfoFilter is designed to be generic and capable of working with any kind of index. The index interface cleanly separates the InfoSearch system and the index which is being used. It is the only module which is specific to the index being used. In other words, the index interface is the only module that needs to be changed when InfoSearch needs to be integrated with a different kind of index. The index interface accepts simple patterns from the graph generator and queries the inverted index. It is responsible for wrapping the result returned by the index in a set of $\langle \text{docID}, \text{start offset}, \text{end offset} \rangle$ tuples. The InfoSearch operators need their input in the form of tuples with offsets, and hence wrapping the output of the index into tuple sets is an important function of the index interface. The set of tuples generated as the index lookup for a keyword is then passed to the leaf node in the PDG that corresponds to that keyword.

4. Experiments

The primary reason for developing operators to detect complex patterns over indexed data was to support efficient searching of stored documents for complex patterns. For streaming data, it does not make sense to store, index, and process them disregarding the quality of service (QoS) requirement of near real-time pattern detection. On the other hand, it does not make sense to *stream already stored documents* to use InfoFilter for detecting patterns. Hence, both approaches are needed for different scenarios. Our effort separates these two approaches by a few specific modules and keeps the bulk of the system and the architecture common. This is an important and desirable characteristic of

our system. It is true that the index-based approach will perform better for larger volume of documents where as stream-based approach will be better for smaller volumes that are not static. The crossover point, in terms of number of words in the document collection, where the indexed approach perform better than the streaming approach was the first parameter of interest. A set of 20 documents of around 1.5 KB each were selected from the Reuters data set. The total number of words in this collection was around 2600. The documents were artificially converted into a stream, fed to InfoFilter and patterns involving all the operators were detected over this stream. The time taken to process the stream was noted. Subsequently, the same documents were indexed, and InfoSearch was used to detect the same patterns as in the previous case. In this case, the time taken for the result set to reach the root node was also noted. It was observed that using an index to detect complex patterns outperforms the streaming approach even for relatively small document collections. Of course, the time taken to index the documents is not considered in the above comparison, but that can be considered as a pre-processing step, and does not come into consideration once the documents are indexed and the index is brought online. The results of the experiments can be found with detailed analysis in [9].

5. Related Work

During the last decade, we have witnessed significant research in the area of *Information Filtering*. Commercial and freely 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 probabilistic approaches. Many areas have been targeted for filtering including emails, selecting interesting news articles, etc. Information Filtering was first introduced on Business Intelligent Systems by Luhn [11]. Since its inception, a lot of fundamental changes have been introduced in its implementation. Some of the Information Filtering Systems that have been developed are: SIFT [1], InfoScope [2], GLIMPSE [3] and IGrep [4].

On the other hand, the field of *Information Retrieval* [6, 7, 8] originated in the late 1950s and has undergone radical changes over the years. The models studied and developed for traditional IR systems are the backbone for all modern Web Search Engines. The main models [7] are the Boolean model, the Vector Space model and the Probabilistic model. Some of the famous search engines designed over the last few decades include: Google [12], Lycos [13], and INQUERY [14].

6. Conclusions

PSL – introduced in this paper, with its expressiveness and well-defined semantics, overcomes the limitations of the current information filtering systems used for specifying and detecting 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. Complete algorithms were developed for complex operators and the synonym option. These operators work on both on sets of tuples of pattern occurrences for stored data and on streaming data. An index interface was developed for the index, and the index-based algorithms were successfully integrated with the rest of the system. The system has been completely implemented in Java and is available for use.

7. Acknowledgements

The authors would like to thank Mr. Aditya Telang for helping us in the preparation of this paper.

References

- [1] T. Yan and H. Garcia-Molina, "The sift information dissemination system," in *ACM Transactions on Database Systems (TODS)*, Dec. 1999.
- [2] C. Stevens, "Knowledge based study for accessing large, poorly structured information spaces," Ph.D. dissertation, University of Colorado at Boulder, 1993.
- [3] U. Manber and S. Wu, "Glimpse: A tool to search through entire file system," in *USENIX Winter 1994 Technical Conference*, 1994.
- [4] M. Arafiujo, G. Navarro, and N. Ziviani, "Large text searching allowing errors," in *Proceedings of WSP*, 1997.
- [5] L. Elkhalfifa, "Infofilter: Complex pattern specification and detection over text streams," Master's thesis, The University of Texas at Arlington, 2004. [Online]. Available: <http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Laali.pdf>
- [6] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1983.
- [7] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.
- [8] M. W. Berry, "Survey of text mining: clustering, classification, and retrieval," *Springer - Verlag*, 2004.
- [9] N. Deshpande, "Infosearch: A system for searching and retrieving documents using complex queries," Master's thesis, The University of Texas at Arlington, 2006. [Online]. Available: <http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Des05MS.pdf>
- [10] C. Fellbaum, "Wordnet: An electronic lexical database," in *MIT Press*, 1998.
- [11] D. Oard and G. Marchionini, "A conceptual framework for text filtering," in *University of Maryland*, 2001.
- [12] S. Brin and L. Page, "The anatomy of a large-scale hyper-textual web search engine," in *Proceedings of the 7th World-Wide Web Conference*, April 1998.
- [13] M. L. Mauldin, "Lycos: Design choices in an internet search service," in *IEEE Expert*, 1997.
- [14] J. Callan, B. Croft, and S. Harding, "The inquiry retrieval system," in *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, 1992.