VIRTUAL QUESTION ANSWERING SYSTEM FOR CINDI

Hong Bing Zhang

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

August 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94758-0
Our file Notre référence
ISBN: 0-612-94758-0

Abstract

Virtual Question Answering System for CINDI

Hongbing Zhang

Modern information retrieval technology is playing an increasingly important role in our daily life. Keyword based search engine system and question answering system are the two main technologies of text information retrieval on the Internet. Current search engine systems do not give enough contexts for their search, and users have to go through the documents to find relevant information. Due to their complexity, question answering systems typically have slow response time.

The Virtual Question Answering System (VQAS) presented in this thesis aims to provide a method to retrieve full-text information from a digital library by targeting the gap between a true question answering system and current search engines. It retrieves the paragraphs where the possible response corresponding to the user's query appear, not just a few words before and after the search terms. In most cases, the user would be able to find the answer directly in the paragraphs of the query result.

Acknowledgements

I express deep gratitude to my supervisor Professor B.C.Desai for his extraordinary and continuous support throughout my studies at Concordia University. Like many others, I consider myself blessed to be under his supervision. He was always gracious in giving me the time and strength even amidst tight schedule.

I would also like to thank all the members in CINDI group and all my friends for their support, patience, encouragement, and understanding throughout my academic endeavors. Without them this thesis would have been impossible.

Last but not the least, I express deep veneration to my dearest parents for teaching me the importance of good education, from which everything else springs.

List of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Information retrieval is the art and science of searching for information from document collections by scanning the documents, searching for metadata which describes the documents, or searching for documents' text. For approximately 4000 years, mankind has organized information for later retrieval and usage [1]. Until recently, information retrieval had been seen as a narrow area of interest mainly for librarians and information experts. The importance of information retrieval has grown dramatically during recent years due to the rapid increase of available storage capacity, increased performance of all types of processors and increased capacity and reliability of networks. This in turn has created an exponential growth of the Internet.

Internet is becoming a universal repository of human knowledge and information, which has allowed unprecedented sharing of ideas and information on a scale never seen before. Despite the many successes, Internet has to face new problems of its own. Finding useful information on the Internet is frequently a tedious and difficult task. These difficulties have attracted renewed interest in information retrieval and its techniques as promising solutions. As a result, information retrieval plays an increasingly important role in modern life.

The most important measures of information retrieval are the quality of the result and the search speed. In order to achieve good performance, a lot of research has been done [1]. This research includes indexing, modeling, classification and categorization, systems architecture, user interfaces, data visualization, filtering, and nature language processing.

1.2 The problem

Currently, keyword based search engine systems and question answering systems are the two main methods of information retrieval on the Internet. Many search engine systems, such as Google, Yahoo and AltaVista, are widely used daily by web users. There are not many question answering systems widely used yet, however many such systems are under development [2].

In search engine systems, an index term is a keyword that has some meaning of its own. Both documents and queries are considered as a set of keywords, and it is assumed that if keyword x occurs in the document, the document is about x. The search engine systems extract all the keywords occurring in the document collection to create an index for them. When a query is made, the index is searched to return a ranked list of documents that contain query keywords. Current search engine systems usually return too many results, and each result only shows several words near the position where the query keyword occurs in the document and a link to that document. Since there is no context in those snippets of text with respect to the query word, users have to go through the document to find whether it contains relevant information; this is extremely time-consuming.

In question answering systems, users make queries in natural language, and the system gives brief answers to the specific question. There are many ways to ask the same question. Likewise, there are many ways of delivering the answer. Such variations form a semantic equivalence class of both questions and answers; many forms of a question can be answered by many forms of the answer. So, the question answering system has to recognize all forms of the question, create a query, retrieve pertinent documents, and find the answer in all possible forms. That is a very difficult task. Current question answering systems seem to typically have slow response time and low accuracy rates [2].

1.3 The Goal of this Thesis

This thesis presents the Virtual Question Answering Subsystem (referred to hereafter as VQAS) of the Concordia Indexing and Discovery System (referred to hereafter as CINDI). VQAS attempts to provide a method to retrieve full-text information from a digital library by targeting the space between a true question answering system and current search engines. VQAS is implemented as a search engine system that supports user queries expressed as one or more words, phrases, and sentences. The difference between VQAS and traditional keyword based search engine systems is that VQAS will retrieve as result the paragraphs where the possible response corresponding to the user's query appear, not just a few words before and after the search terms. In most cases, the user would likely find the answer directly in the paragraphs of the query result.

1.4 The Thesis Outline

Chapter 2 introduces the background information on search engine systems, question answering systems and the CINDI system. Chapter 3 introduces the

system architecture of VQAS and discusses in detail its major components, their inter-operations, algorithms, and its integration into the CINDI system. Chapter 4 discusses the results achieved by VQAS and how these results compare with other search engine systems. Chapter 5 concludes the thesis and discusses the implication for future work in the field.

Chapter 2

State of the Art

2.1 Early Information Retrieval System

Since 4000 years, people have organized information for later retrieval and usage[1]. A typical example is the table of contents of a book. Since the volume of information eventually grew beyond a few books, it became necessary to build specialized data structures to ensure easy access to the stored information. An old and popular data structure for faster information retrieval is a collection of selected words or concepts with associated pointers to the related information (or documents). This structure is called an index. Indexes are at the core of every modern information retrieval system. They provide faster access to the data and allow the query processing task to be speeded up. For centuries, index cards and catalogs were created manually for all the information resources in libraries; the users of a library can use different kinds of index cards to search the information resources by title, author, etc.; the users can also use the catalogs to search the information resources by the catalog number. Currently, most libraries still use the catalog number to classify their documents. Such catalogs have usually been conceived by human subjects from the library science field.

In the early 1990s, a single fact changed the history of information access – the introduction of the World Wide Web (referred to hereafter as WWW). WWW development began at CERN (Geneva) in 1989 [3]. It is defined as an information service on the Internet that has the following properties:

- It uses a common addressing syntax in the form of a Universal Resource Locator (URL).

- It uses Hypertext Markup Language (HTML), a document formatting language. HTML is a language used to describe hypertext resources, in which links with other resources can be defined. It can also describe hypermedia resources, wherein the links are not associated with textual information, but with other resources such as images or sounds.

- It uses the HTTP protocol (HyperText Transfer Protocol) in order to transfer information resources between two computers (a client and a server) on a network. These resources could be texts, menus, hypertexts, images, etc.

WWW makes it possible to access a document anywhere on the Internet using a URL. People can share their ideas and information on the web. The web is becoming a universal repository of human knowledge and culture. Its success is based on the use of a standard user interface no matter what computational environment is used to run the interface. As a result, the user does not need to know the details of communication protocols, machine location, and operating systems. Further, any user can create his/her own web document and make them point to any other web document without restrictions, which turns the web into a new publishing medium accessible to everybody. It is causing a revolution in the way people use computers and perform their daily tasks. It is changing the way that people access information, and has opened up new applications in areas such as digital libraries, general and scientific information dissemination, education, commerce, entertainment, government, health care, etc.

Despite its success, the Web has introduced new problems of its own. The problems include the following:

- Distributed data: Due to the nature of the Web, data spans over many computers and platforms. These computers are interconnected with no predefined topology, and the available bandwidth and reliability on the network varies widely.

- Volatile data: Due to Internet dynamics, new computers and documents can be added or removed easily. It is estimated that 40% of the Web changes every month [4]. We also have dead links and relocation problems when URLs disappear or change.

- Unstructured and redundant data: Web pages are not well structured, and much Web data is repeated (mirrored or copied) or very similar. Approximately 30% of Web pages are duplicated [5]. Semantic redundancy can be even larger.

- Large volumes: The Web is huge and challenging to deal with. Several studies have estimated the size of the Web, and while they report slightly different numbers, most of them agree that over a billion pages are available [6, 7, 8]. Given that the average size of a Web page is around 5-10K bytes, just the size of the textual data is at least tens of terabytes.

- Quality of data: The web can be considered as a new publishing medium. However, documents are not filtered. So, data can be false, invalid, out of date, poorly written, or with many errors. Studies show

that the number of words with typos can range 1 in 200 for common words to 1 in 3 for foreign surnames [9].

- Heterogeneous data: On the Web, data can be of multiple media, multiple formats, and different languages.

Finding useful information on the Web is frequently a tedious and difficult task. For example, to find the information he/she needs, the user might navigate the hyperspace searching for information of interest. However, since the hyperspace is vast and almost unknown, such a navigation task is usually inefficient. For those users who are not good at searching, the problem becomes harder, which might waste their efforts. The main obstacle is the absence of a well defined underlying data model for the web, which implies that information definition and structure is frequently of low quality. These difficulties have attracted people's interest to find new solutions. One of these solutions was the introduction of search engine.

2.2 Search Engine System

A search engine is a web-based software tool that enables the user to locate sites and pages on the web based on the information they contain. With a search engine, keywords related to a topic are typed into a search "box." The search engine scans its document collection and returns a ranked list of documents containing the word or words specified with links to the websites of these documents. Search engine system uses statistical methods that rely on the frequency of words in query and document collection. Most search engine systems process search based on the assumption that the more frequently a

keyword occurs in a document, the more relevant the document is to the query. Since these databases are very large, search engines often return thousands of results. Without search strategies or techniques, finding what you need can be like finding a needle in a haystack.

In the days before the introduction of World Wide Web, searching for information on the Internet could be quite difficult. One common method was through FTP. In the beginning, FTP files were publicized and found by word of mouth, email messages, or on message boards. In 1990 Alan Emtage improved FTP search capabilities with a database client called Archie [10]. Archie was able to obtain site listings of FTP files by scouring FTP sites across the Internet and indexing all the files found.

Soon the first robot came. It was called World Wide Web Wanderer [11] and its job was to track the web's growth. Initially it only counted web servers but later it began to gather URLs as well and stored them in the first web database, Wandex. The term robot here is referring to automated computer program that performs an automated task on the Internet. A search engine spider is a type of robot. The Wanderer led the way for programmers to improve upon the idea of robots and spiders. Spiders start on a starter site and begin exploring all of the links from that site.

In October of 1993 Martijn Koster created Archie-Like Indexing of the Web, called ALIWEB [12]. ALIWEB allowed users to submit their pages they want to index with their own page description. This meant it did not need a robot to collect data and was not using up excessive bandwidth. The downside of ALIWEB is that many people did not know how to submit their information resource or were not willing to do so.

In December of 1993, three robot-powered engines are created - JumpStation, World Wide Worm and Repository Based Software Engineering Spider [13]. JumpStation gathered information about the title and header from Web pages and retrieved these using a simple linear search. The WWW Worm indexed titles and URLs. The problem with JumpStation and the World Wide Web Worm is that they listed results in the order that they found them, and provided no discrimination. The RSBE spider did implement a ranking system. Brian Pinkerton of the University of Washington released the WebCrawler on April 20, 1994 [14]. WebCrawler was the first full-text search engine on the Internet.

Lycos [15] was the next major search engine, having been designed at Carnegie Mellon University around July of 1994. On July 20, 1994, Lycos went public with a catalog of 54,000 documents [15]. In addition to providing ranked relevance retrieval, Lycos provided prefix matching and word proximity bonuses. But Lycos' main difference was the sheer size of its catalog: by August 1994, Lycos had identified 394,000 documents; by January 1995, the catalog had reached 1.5 million documents; and by November 1996, Lycos had indexed over 60 million documents -- more than any other Web search engine [16]. In 1995, another search engine, named AltaVista, was created by scientists at Digital Equipment Corporation; it devised a way to store every word of every HTML page on the Internet in a fast, searchable index [17]. AltaVista had led the search engine industry in many areas. It was the first search engine to offer translation services and localize its site for Chinese, Japanese and Korean searchers.

In 1998 the last of the current search super powers, and the most powerful to date, Google, was launched [19]. The Google search engine has two important features that help it produce high precision results. First, it makes use of the link structure

of the Web to calculate a web page's "PageRank", an objective measure of its citation importance that corresponds well with people's subjective idea of importance. Second, Google utilizes link to improve search result. Google improved the quality of web search engines. It has become so popular that major portals such as AOL and Yahoo have used Google.

Figure 2.1 gives a high level overview of how Google system works. Several distributed crawlers download web pages synchronously. There is a server that sends lists of URLs to be fetched into the crawlers. The web pages that are fetched are then sent to another server called the store server. The store server

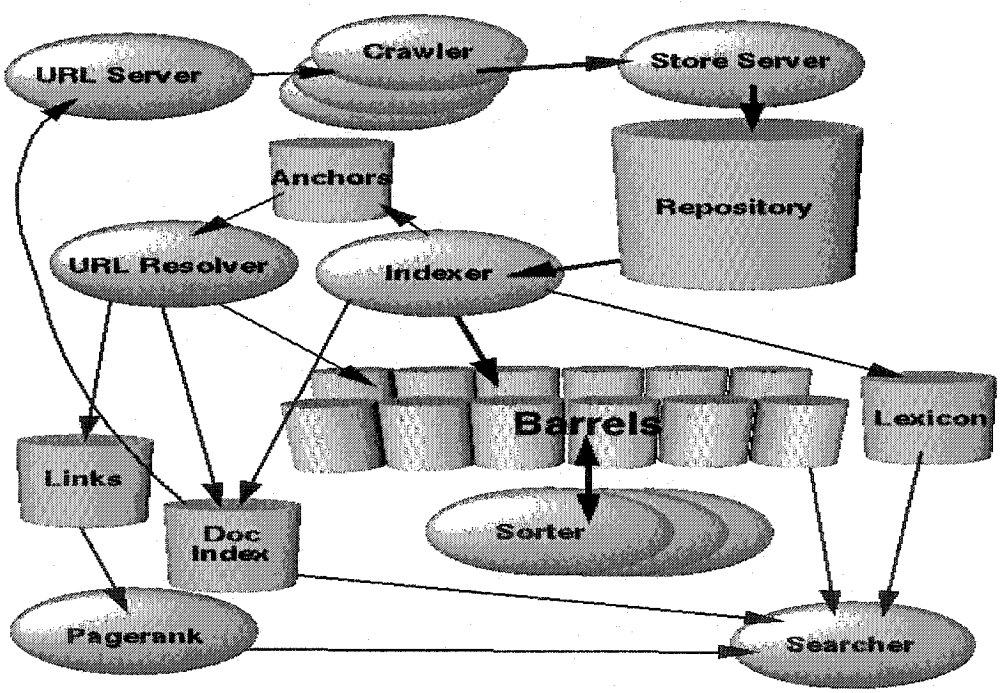


Figure 2.1: Google Architecture [19]

then compresses and stores the web pages into a repository. Every web page has an associated ID number called a docID that is assigned whenever a new URL is parsed out of a web page. The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, decompresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, its position in the document, an approximation of font size, and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parses out all the links in every web page and stores important information about them in an anchors file; the information includes where each link points from and to, and the text of the link. The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links that are pairs of docIDs. The links database is used to compute PageRanks for all the documents. The sorter takes the barrels, which are sorted by docID, and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

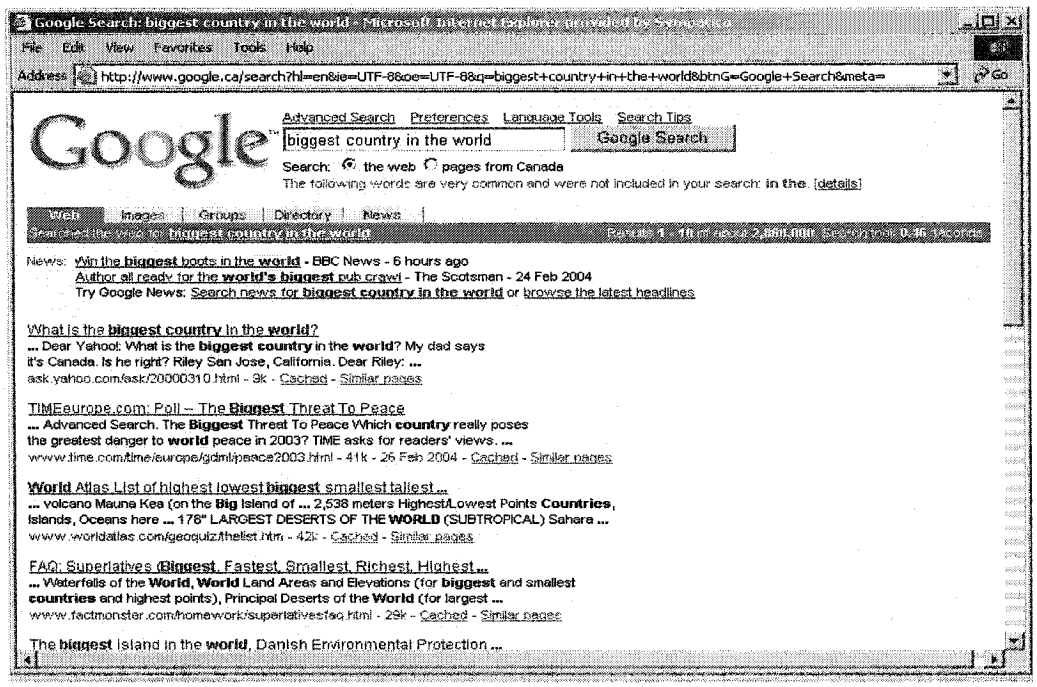


Figure 2.2: Search result for query "biggest country in the world" in Google

Google has indexed a large number of web pages; and is one of the largest in the world, with approximately 3 billion pages [19]. Currently, nearly half of the traffic of the Internet generated by all of the search engines and directories is generated by Google [19]. Despite the big success, current search engine systems have two limitations. Search engine systems answer question indirectly. Search engine systems have traditionally focused on returning reference to URL of the documents that contain the query keywords rather than returning pertinent part of the document. Figure 2.2 shows an example of the search result from Google. Each search result shows a snippet of text in which search terms occur and also a link to the document. In this example, the user wants an answer to the question: "which is the biggest country in the world?". Obviously, the user cannot have the

answer from the snippets of text; he/she has to access the likely documents using the links to find out whether the documents actually contain the answer, which is time consuming.

Search engine systems do not attempt to understand the meaning of user's query and the meaning of documents in the collection. They retrieve document based on term frequency, location of terms, link analysis, popularity, date of publication, length of the document, and proximity of query terms. This usually results in reference to irrelevant documents being returned. A sample query can return thousands or millions of documents, many of them have low relevance to the query and the desired documents may not appear near the top of the list. The main problem is the criteria used by the search engines to rank the relevance of search results; also they always display the paid advertisements on the top of the results. Another problem lies in the unrestricted data source. Google tries to improve the relevance of search result by using a ranking algorithm called PageRank that iteratively uses information from the number of pages pointing to each page. This algorithm works to a certain extent, but there are still a lot of irrelevant results in Google search. One common situation in Google search is that many entries in search result are the e-mails sent by other users who were asking the question with the same terms; this kind of result does not make any sense.

For certain types of questions, users would prefer the system to answer the question than be forced to go through a list of documents looking for the specific answer. This requirement has attracted researches' interest in Question Answering [20]. As a result, question answering has gained a place with other technologies in information access.

2.3 Question Answering System

Question Answering (referred to hereafter as QA) aims at identifying the answer of natural language questions from a large collection of on-line documents. Instead of extracting all related information, a QA system extracts only a short piece of text, accounting for the answer. QA is a resource to address the problem of information overload.

QA research dates back to the 1960s, and has often been confined to domain-specific expert systems [21]. Researchers have experimented with QA systems based on closed, pre-tagged corpora [22], or knowledge bases [23]. Many of these systems focus on Text Retrieval Conference (TREC) tasks. Researchers also have attempted to build QA systems on large collections of documents on the Web by combining information extraction and most advanced information retrieval technology [24]. Recently, researchers' interest has been attracted to the development of open-domain QA systems based on collections of real world documents, especially the WWW [25].

TREC is a series of workshops aiming at developing technologies for information retrieval. The QA track was started in 1999 (TREC-8); it focuses on the evaluation of QA systems, in a competition-based manner that answers questions in unrestricted domains. The TREC QA track is an effort to bring the benefits of large-scale evaluation to bear on the QA problem. The track has run five times so far. The overall goal has remained the same in each of the TREC competitions, which is to retrieve small snippets of text that contain the actual answer to a question rather than a document list. Each TREC QA track introduced new conditions to increase the realism of the task. The latest TREC competition

(TREC-12) includes two separate tasks: the passages task and the main task. The main task consists of the factoid task, the list task, and the definition task.

The passages task tested a system's ability to find an answer to a factoid question within a relatively short (250 characters) span of text. Each text span returned by the system was required to be an extract from a document in the corpus. All processing was required to be completely automatic with no changes to the system permitted once the test questions were released. A passages task run consisted of exactly one response for each of the test questions. A response was either a specification of a document extract or the string "NIL". "NIL" was used to indicate the system's belief that there was no correct answer in the collection. The fraction of questions judged correct, called accuracy, is the main evaluation score for a passages task run. The recall and precision of recognizing no answer are also reported. Precision of recognizing no answer is the ratio of the number of times NIL was returned and correct to the number of times it was returned; recall is the ratio of the number of times NIL was returned and correct to the number of times it was correct.

The factoid task was very similar to the passages task. However, systems were required to return an exact answer rather than an extract containing an answer. The answer strings returned by the systems were not required to be an extract from a document; the response for a question was of the form "query-id run-tag doc-id answer-string". If the system believed there was no correct response in the document collection, doc-id was set to NIL and answer-string was empty. The score for the factoid component of the main task was accuracy, the fraction of responses judged correct.

In the list task, systems assembled a set of instances as the answer from information located in multiple documents. A list question asks for different instances of a particular kind of information to be retrieved, such as "List the names of universities in Montreal". List questions can be thought of as a shorthand for asking the same factoid question multiple times; the set of answers that satisfy the factoid question is the appropriate response for the list question. A system's response to a list question was an unordered set of [document-id, answer-string] pairs such that each answer-string was considered an instance of the requested type. The score for the list task was the combination of the instance precision and the instance recall.

Definition questions are questions such as "What is Ph in biology?" Definition questions occur relatively frequently in logs of web search engines; this suggests that they are an important type of question. However, evaluating systems that answer definition questions is much more difficult than evaluating systems that answer factoid questions because we can not judge a system's response as simply right or wrong. It requires some mechanism for matching the concepts in the desired response to the concepts present in the system's response. The system returned an unordered set of [document-id, answer-string] pairs as a response for a definition question. The judging of the systems' responses was designed in a way to make the evaluation depend only on the content of a system response, and not on the particular structure of a system's response. Assessors ignored wording differences, making conceptual matches between the system's responses and the desired responses, not syntactic matches. The final score for a definition response was computed using the same measure as it was for list questions.

The final score of the main task for a QA system was computed as a weighted average of the factoid task score, the list task score, and the definition task score. Since each of the component scores ranges between 0 and 1, the final score is also in that range. The final score emphasizes the factoid task since it represented the largest number of questions and is the task people are most familiar with. The weight for the other tasks was made large enough to encourage participation in those tasks. Table 2.1 shows the combined scores for the top 15 groups.

The document collection used as the source of answers was the same for all task evaluations. TREC-12 used the AQUAINT Corpus of English News Text. This

Table 2.1 Component scores and final combined scores for main task runs

Run Tag	Submitter	Component Score			Final Score
		Factoid	List	Def	
LCCmainS03	Language Computer Corp.	0.700	0.396	0.442	0.559
nusmml03r2	National University of Singapore	0.562	0.319	0.473	0.479
lexiclone92	LexiClone	0.622	0.048	0.159	0.363
isi03a	University of Southern California, ISI	0.337	0.118	0.461	0.313
BBN2003C	BBN	0.206	0.097	0.555	0.266
MITCSAIL03a	Massachusetts Institute of Technology	0.293	0.130	0.309	0.256
irstqa2003w	ITC-irst	0.235	0.076	0.317	0.216
IBM2003c	IBM Research (Prager)	0.298	0.077	0.175	0.212
Albany03I2	University of Albany	0.240	0.085	0.146	0.178
FDUT12QA3	Fudan University	0.191	0.086	0.192	0.165
UAmsT03M1	University of Amsterdam	0.136	0.054	0.315	0.160
shef12simple	University of Sheffield	0.138	0.029	0.236	0.135
CMUJAV2003	Carnegie Mellon University	0.133	0.052	0.216	0.134
ICTQA2003C	Chinese Academy of Sciences	0.145	0.091	0.149	0.133
uwbqitekat03	University of Wales, Bangor	0.259	0.000	0.000	0.130

collection consists of documents from three different sources: the AP newswire from 1998–2000, the New York Times newswire from 1998–2000, and the (English portion of the) Xinhua News Agency from 1996–2000. There are approximately 1,033,000 documents and 3 gigabytes of text in the collection [26]. The test set of questions contained 413 questions drawn from AOL and MSNSearch logs. Thirty of the questions have no known correct answer in the document collection.

Most TREC QA systems used a general approach for the past several years. Figure 2.3 shows the system architecture for the general approach for answering passage and factoid questions. In the Question Analysis module, the system attempts to classify a question according the type of its answer as suggested by the question word. For example, a question beginning with "who" implies a person or an organization is being sought, and a question beginning with "when" implies time or date is being sought. Simultaneously, in the Information Retrieval module, the system retrieves a set of possible relevant documents using standard text retrieval technology and the question as the query. In the Answer Candidate Search module, the system performs a shallow parse of the returned documents to detect entities of the same type as the answer. If an entity of the required type is found sufficiently close to the question's words, the system returns that entity as the response. If no appropriate answer type is found, the system falls back to best-matching-passage techniques. While the overall approach has remained the same, individual groups continue to refine their techniques, increasing the coverage and accuracy of their systems.

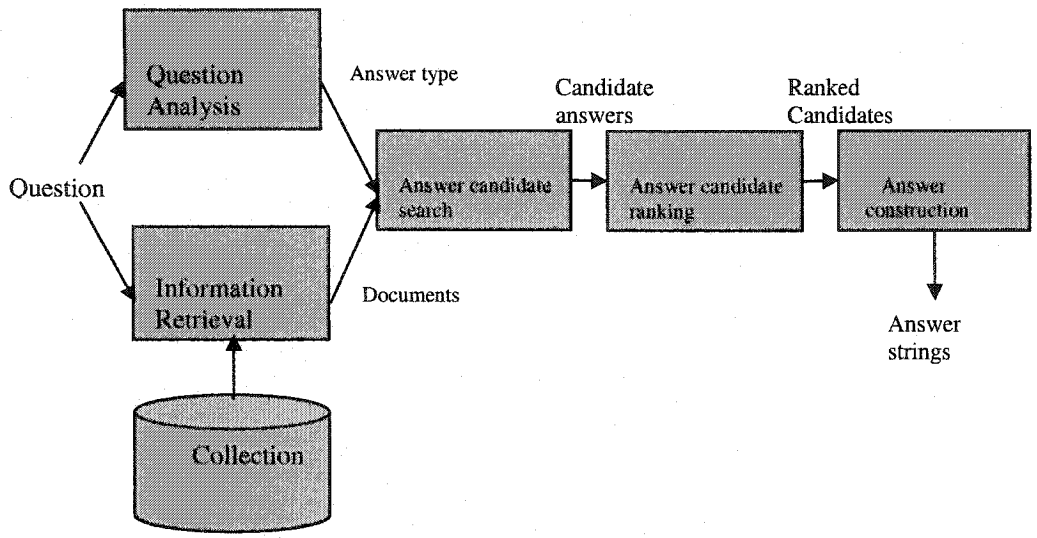


Figure 2.3: The architecture for the general approach to answer passage and factoid questions in TREC QA systems

A similar approach is used to answer list questions. Most groups used their factoid-answering system for list questions; the only difference is the number of responses returned as the answer. The main issue was determining the number of responses to return. Systems whose matching phase creates a question-independent score for each passage returned all answers whose score was above an empirically determined threshold. Other systems returned all answers whose scores were within an empirically determined fraction of the top result's score.

Answering definition questions generally involved using different techniques than those used for factoid questions. Since the definition task did not require "exact" answers, most systems first retrieved passages about the target using a recall-oriented search. Subsequent processing reduced the amount of material returned. Many systems used pattern matching to locate definition-content in text. These patterns, such as looking for copular constructions and appositives, were

either hand-constructed or learned from a training corpus. Systems also looked to eliminate redundant information, using either word overlap measures or document summarization techniques. The output from this step was then returned as the definition of the target.

Despite the positive results that have been gained in QA research, current QA systems seem to typically have slow response time, low accuracy rates, incomplete answers, and irrelevant answers [2]. QA systems are facing big challenges. Researchers are trying hard to improve the current QA systems in the following aspects:

- Timeliness: answer question in real-time, instantly incorporate new data source.
- Accuracy: return exact answer but nothing else; detect no answer if nothing is available.
- Usability: mine answers regardless of the data source format; deliver answers in any format.
- Completeness: return complete answers, not just part of answers.
- Relevance: return relevant answers in context; interactivity to support dialogs.

2.4 CINDI System

CINDI (Concordia Indexing and Discovery System), proposed by Desai et al [18], is a system that enables resource providers to catalogue their own resource

and users to search for documents. For cataloguing and searching, a meta-data description called Semantic Header [26] is used in CINDI to describe an information resource. The intent of the Semantic Header is to include those elements that are most often used in the search for an information resource, such as title, name of the authors, subject, annotation, etc. The creation of Semantic Header solves problems caused by differences in semantics and representation, incomplete and incorrect data cataloguing. The Semantic Header of each document could be either entered by the primary resource provider or by the ASHG (Automatic Semantic Header Generator) [7], a software system that automatically generates the meta-information of a submitted document.

The overall CINDI system includes following subsystems: CINDI Robot system, a Converter and Filter system [35], the ASHG system [26], and the Search system. The data of CINDI system comes from two sources: documents submitted by users and documents download from the Web by the Robot. Before storing into the CINDI system, all documents are converted into PDF format by a Document Converter System (DCS) and filtered by a Document Filter System (DFS). Only theses, academic papers, technical reports and FAQs (Frequently Asked Question) remain in the document collection after the processing by the DFS. Next, the ASHG generates a Semantic Header for each document and informs the primary resource provider to verify the result. If the primary resource provider does not agree with the Semantic Header generated by the ASHG, he/she can log into the system to modify it. Verified semantic header will be inserted into the CINDI database. Using the information in the CINDI database, users can search for documents using typical search items such as author, title, subject, keywords etc.

In addition to the bibliographic search, for pertinent documents users sometimes want to search for answers to their questions. So we need a search system that can retrieve answers based on user queries from the full text of documents in CINDI. Based on this need, VQAS is built. VQAS retrieves as result the paragraphs where the possible response corresponding to the user's query appear, not just a few words before and after the search terms. Similar information retrieval systems have been developed, such as Nova [17] by Sun Microsystems Laboratories, LASSO [18] by Southern Methodist University.

VQAS is a keyword and phrase based search system, and differs from traditional web search engine system in the following two aspects:

- **Data source**

  Traditional web search engine system: The system only stores the URLs of source documents. This causes some problem. First, since the source documents are distributed over remote web servers, it is slow for users to access the source documents through the URLs according to the Internet traffic. Second, the documents could be of any kind; however, many of these documents are useless for most users; examples are: e-mail, advertisement. Third, some URLs may no longer exist, however traditional search engines may maintain a cached version.

  VQAS: Since documents are stored in the local server of CINDI system, they can be accessed rapidly. Since the system converts all the documents into PDF format and filters out documents other than theses, academic papers, technical reports and FAQs, the low quality of data source in traditional web search engine system will be avoided.

- **Search strategy**

Traditional web search engine system: Search results show a snippet of text in which search terms occurs and also a link to the document. Since results do not show the context of search words, it is impossible for users to find answers from those snippets of text, and it is even impossible to know whether the result document is relevant to the user's query. To find the answer of the question, users have to go through the links to the web documents.

VQAS: Documents are broken into semantically coherent segments (paragraphs), and the index is created based on paragraphs. The search result shows paragraphs in which search words occur, therefore gives the context of search words. From the text in the paragraphs of the result, users can understand the semantic meaning of the result, judge the relevance of the result; in most cases, the user would likely find the answer directly in the paragraphs given in the result.

Chapter 3

Virtual Question Answering System (VQAS)

3.1 System Architecture

The VQAS system consists of the following four fundamental modules: document processing module, index creation module, search module, and query interface module. This is demonstrated in the system diagram (Figure 3.1).

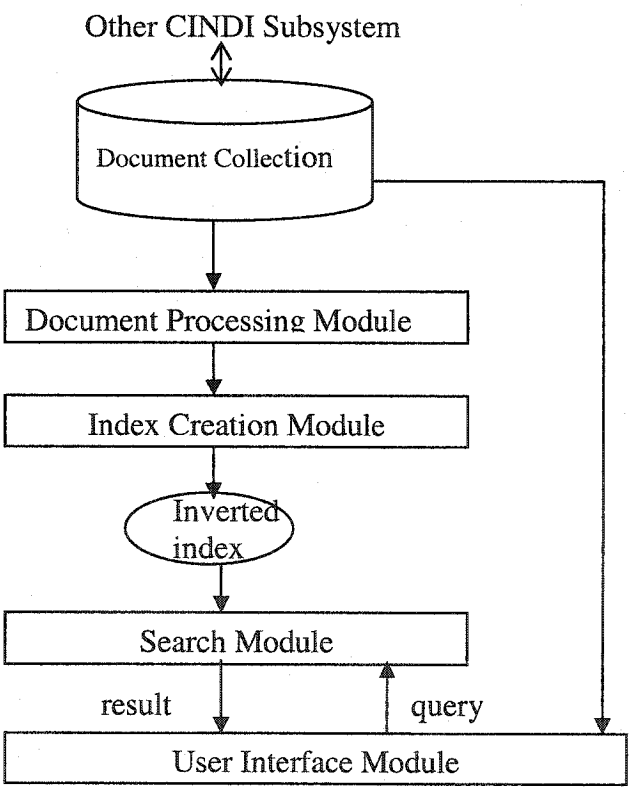


Figure 3.1 System diagram

The document processing module lexically analyzes the text of the document, filters the content of the document, breaks the document stream into paragraphs, identifies potential index terms, and extracts the information of index entries.

The index creation module uses a B+ tree data structure to temporarily store and manage the index entries provided by the document processing module for each document, and finally creates an inverted index for all documents in the collection. In order to get better search result, complete information of search terms is included in the index.

The search module tokenizes the query stream, parses the query stream into query terms, matches the query terms to the inverted file generated by the index creation module to get search result, and stores the search result by the relevance of the paragraph to the query. Single term query, multiple term query and phrase query are supported in this module. In order to speed up phrase query, a B-tree phrase index is created for frequently searched phrases.

The user interface module provides users a graphical web interface, passes users' queries to the search module, and displays the search result to users.

3.2 Document processing module

As mentioned in section 2.4, the CINDI system stores all the documents in PDF format. Documents with other formats are converted into PDF format before being put into the CINDI document collection. Therefore, before analyzing documents in word, phrase and sentence levels, VQAS has to convert the documents from PDF format to text format. The ASHG [26] subsystem for CINIDI had already done this

conversion for automatic semantic header generation. So we integrated VAQS with this subsystem using a table called ASHG, one of the relations of the CINDI database that maintains the information about documents conversion. Details are in Table 3.1.

Table 3.1 Schema of relation "ASHG"

Attribute Name	Attribute Type	Comments
ASHG_ID	integer	The id of semantic header generated.
DocID	integer	The id of source PDF document.
create_time	date	The date when the semantic header is generated.
update_time	date	The date when the semantic header is updated.
Ashg_directory	string	The pdf file directory.
txt_directory	string	The text file directory.
Ashg_filename	string	The filename of the pdf file.
txt_filename	string	The filename of the text file.
Index_flag	boolean	Indicate whether the document has been indexed

Only those documents that have not yet been indexed will be processed in document processing module where the processing procedure can be divided into four steps: lexical analysis of the text, content filtering, elimination of stopwords, and wordId lookup of words.

3.2.1 Lexical analysis of the text

Lexical analysis is the first step of document processing. It converts a stream of characters, which are the texts of the documents, into a stream of words, which are the candidates to be adopted as index terms. The major objective of the lexical

analysis phase is the identification of the words in the text. Several operations are done in this phase.

First, punctuation marks are replaced by spaces. However, there are some exceptions. Usually, a period is used to finish a sentence. But the period can also be part of a search word, such as "java 1.4.1". The replacement of the period in the documents will affect the accuracy of information retrieval. Therefore, in VQAS, we only replace those periods followed by the space. Another exception is a hyphen. Hyphens are used very often to connect a word split into two lines. However, there are words that have hyphens as an integral part. For example, gilt-edge, B-49, etc. Hence, in VQAS, we only remove those hyphens occurring at the end of lines. After removing, we also join the two split parts into one word. There might be the case that the hyphen appears at the end of one line and it is also the integral part of a word. In this scenario, VQAS could mistakenly generate a new word and produce a wrong analysis. To avoid this error, before joining two split parts that are separated by the hyphen at the line end, we check the two parts in the VQAS lexicon. If these two parts have been verified as two words, the hyphen between these two parts is kept.

Second, all the texts are converted into lower cases for the document processing. There might be some case sensitive words, such as China and china. Since VQAS mainly deals with technical articles, this condition seldom happens. Therefore, this approach is still valid.

Third, words are extracted by using spaces as word separators, in which case, multiple spaces are reduced to one space. In this step, some of the word information, such as the word position, the word length and the start position of the paragraph are stored and will be used in the index creation module.

3.2.2 Content Filtering

Some parts of the document cannot give an answer to user's query, such as title, subtitle, table of content, keywords list, acknowledgements, and references. Users do not want to see them in the search result. Therefore, we have to eliminate them to save system resources and to decrease the size of the search result. In this step, VQAS does a filtering of document content by analyzing the document layout and keywords. For example, the reference list of a document usually appears at the end of the document and follows the keyword "references" or "bibliography".

3.2.3 Elimination of stopwords

This step aims to eliminate terms that have little value in finding useful documents in response to a user's query. Since stopwords may comprise up to 40 percent of text words in a document, elimination of stopwords has values in real applications to save system resources [31]. In VQAS, a stopword list is loaded at the beginning of the program. It consists of those word classes known to convey little substantive meaning, such as articles (*a, the*), conjunctions (*and, but*), interjections (*oh, but*), prepositions (*in, over*), pronouns (*he, it*), and forms of the "to be" verb (*is, are*). The contents of the stopword list used in VQAS are taken from Google's implementation choice [27] and have been described in Appendix A. Each word in the document is compared with the stopword list and will be removed if it is in the list.

3.2.4 WordId lookup

In VQAS, a word is uniquely represented by its id, which has been used in document processing module, index creation module, and search module. A word in

the document may appear in different forms such as singular, plural, tense, etc., but all these forms stand for similar semantics. Therefore, query result should be flexible of word forms. For example, if users query for *computer*, they may also want documents that contain *computers* as well; here, *computer* is the base form of *computers*. Traditional stemming might be used for this task since it removes the affixes of words [32]. However, this approach reduces the precision of search since all the forms of a stem will be matched. For example, the word *comput* is the stem for the variants *compute, computation, computing, computer, computers,* etc. When we query for the word *computer*, documents that contain *computers* may be acceptable but documents that contain *compute, computation,* and *computing* are unlikely in the required set. However, applying the traditional stemming technique, documents that contain *compute, computation,* and *computing* will be fetched since the system will do the word matching with the stem *comput*.

In VQAS, we developed an algorithm to get the base form of a word. It takes over the role stemming acts in traditional search engine and at the same time reduces the shortcoming of traditional stemming approach. Converting words into their base form reduces the number of unique words in the index, which in turn reduces the storage space required for the index and speeds up the search process.

In order to look up and match the words, a B-tree lexicon called main lexicon is built in VQAS to contain the pairs of a word in base form and its id, denoted as wordId. The wordId starts from 1 and will be increased by 1 when a word is inserted into the main lexicon. Each word and its variants are given the same wordId. For those regular words, a series of rules are applied to retrieve their base form. However, for those irregular words, such as *give, gave and given*, applying transforming rules cannot work at all. So, another B-tree lexicon called exc_lexicon

is built to contain the pairs of words and their exceptional word forms. Words in main lexicon and exc_lexicon are extracted from the WordNet online lexical reference system [33] by an automatic selection and filter program written in C++.

The algorithm of the wordId lookup process in VQAS is as shown below:

Step1: For a given word, look up the wordId in the main lexicon

If it is found, return its wordId, done.

Else, go to step 2.

Step2: Look up the base form of the word in the exc_lexicon

If it is found, look up its wordId in the main lexicon using the base form and return its WordId, done.

Else, go to step 3

Step3: Apply the following rules to replace the suffix of the word by a string to get the base form of the word, then look up the wordId in the main lexicon using the base form.

If it is found, return its wordId, done.

Else, go to step 4.

Rules: s→Ø, es→Ø, d→Ø, ed→Ø, ing→Ø, ing→e, ies→y

men→man, er→ Ø, er→e, est→ Ø, est→e

s→Ø means to remove letter 's' if it appears at the end of a word.

men→man means to replace suffix "men" by "man".

Step 4: Assign the maximal wordId plus one to the wordId of the word, then insert the pair of the word and its wordId into the main lexicon, and return its wordId.

3.3 Index creation module

VQAS aims to retrieve paragraphs that contain query words or phrases. Therefore we need to create an index of word occurrences. For each word, the index contains a number of entries that includes the identification of a document, the position of the paragraph in the document, and the position of the word in the paragraph. The identification of a document, denoted in VQAS as docid, is used to locate the document. The position of the paragraph records the position where the paragraph starts in the document so that we can locate it. The position of the word is used to highlight the query word in the result. Since the position of a paragraph and the position of a word indicate their significance to the document, they are also used to calculate the rank of a retrieved paragraph.

There are a number of indexing approaches that have been used in different search engines, such as inverted index [28], bitmaps [29], signature file [30]. None of them is optimal for all applications. For large text collection, [28] stated that the inverted index provides reasonably better performance than any other index approaches. Therefore, we choose the inverted index as the approach to create the index structure for VQAS.

The index creation module can be divided into three sub-modules that are the B+-tree creator, the forward index creator, and the inverted index creator. The B+-tree creator creates a B+-tree in the memory to temporarily store the word occurrences

for a particular document. In the forward index creation, the information of word occurrences for a particular document stored in the B+-tree is appended to the forward index file. Afterwards the B+-tree is cleaned and prepared for processing another document. The forward index will be created when all the documents are processed. Finally the inverted index creator creates an inverted index based on the forward index.

3.3.1 B+-tree creation

In this module, we use a B+-tree to temporarily store the information of word occurrences for a document, such as the identification of the word, the position of the paragraph and the position of the word. This information is provided by the document processing module described in section 3.2. After the system scans all words in the document, the information stored in this B+-tree will be appended to the forward index file, and the B+-tree will be initialized for next document.

Data structures

The data structure of B+-tree, as demonstrated in Figure 3.2, has the following major components:

B+-tree: B+-tree is a balanced search tree in which the real data is stored in the leaf nodes, and the internal nodes are used to store the keys for navigation. This B+-tree is used to store the word occurrences in a given document being processed. The word occurrences in a document are stored in the leaf node of the B+-tree. Once all the word occurrences are stored, they will be appended to the forward index file, and the B+-tree will be reset for the next document.

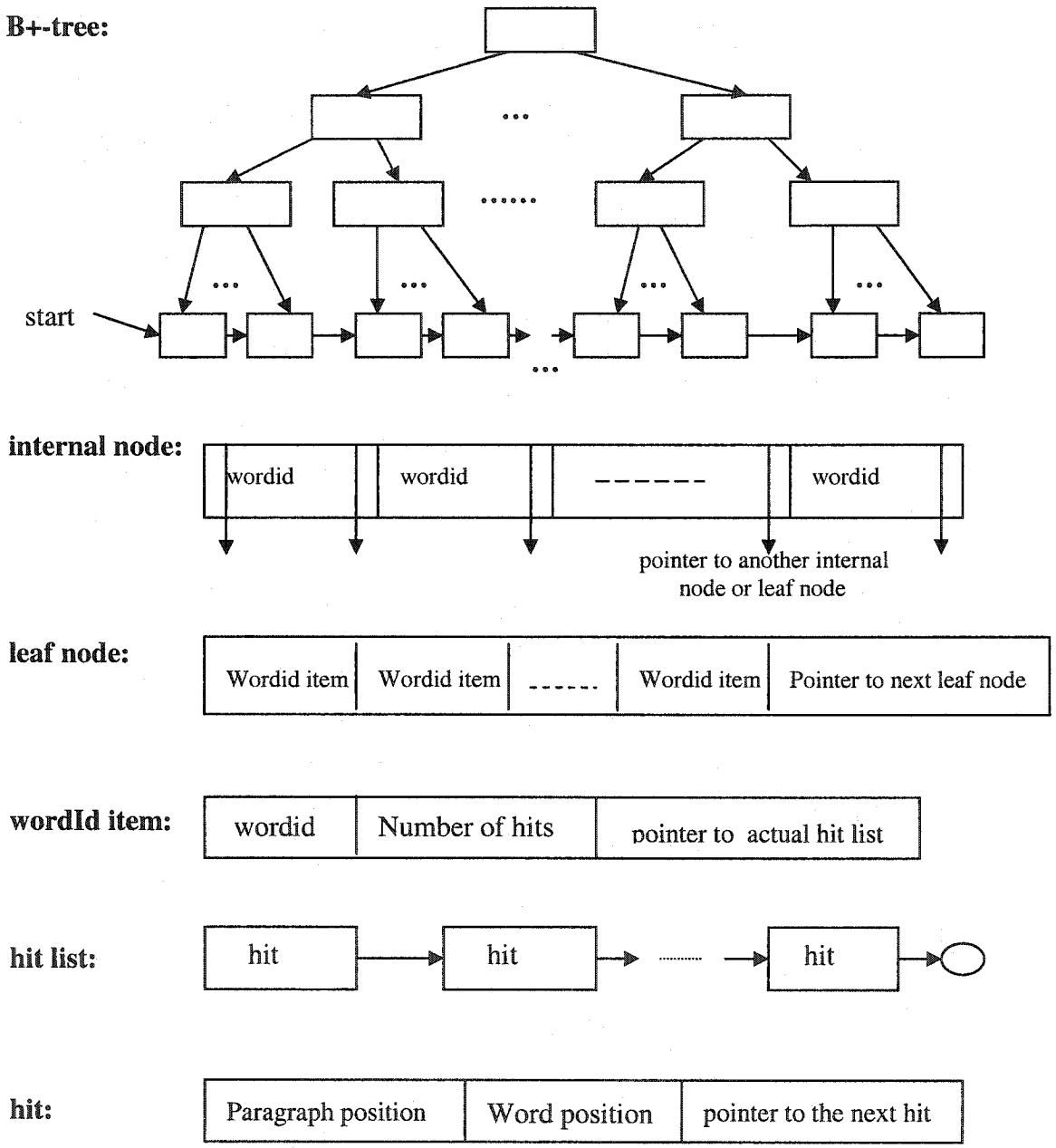


Figure 3.2 Major data structures used in the B+-tree index creator

Internal node: Internal nodes in B+-tree are used to store keys to navigate to the leaf nodes. Each internal node contains an array of selected keys and an array of pointers. In our B+-tree, each internal node contains an array of size 128 for wordIds and an array of size 129 for pointers. The ith pointer in a internal node will point to those nodes whose wordIds are less than the ith wordId in this internal node but greater or equal to (i-1)th wordId in this internal node.

Leaf node: Leaf nodes contain the real data. Each leaf node contains an array of items and a pointer to the next leaf node by the order of the key of the item. In our B+-tree, the leaf nodes contain all the information being used to build the forward index. Each leaf node contains an array of of size 128 for wordId items and a pointer to the next leaf nodes by order of wordId.

Start: Start is a pointer that points to the first leaf node in the B+-tree.

WordId item: WordId item is used to store the id of the word, which is the only identification of a word, the number of occurrences of this word in a particular document, and a list of those occurrences.

Hit list: A hit list corresponds to a list of occurrences of a particular word in a particular document.

Hit: A hit corresponds to one occurrence of a particular word in a particular document. It includes the position of the paragraph in the document that contains the word, the position of the word in the paragraph, and a pointer to the next hit.

Advantage of this data structure

The advantages of choosing the B+-tree structure over other structures (such as sorted array, hash table, B-tree, etc.) lie in the following three aspects:

- Search and insertion operation

Search and insertion are used very frequently. For every occurrence of a word, the system needs to determine if the word is already in the B+-tree. If the answer is yes, the new hit will be added into the hit list of this word. If the answer is no, a new wordId item will be created and inserted into the B+-tree. For both search and insertion, the system needs to locate the appropriate leaf node, which is very efficient for a B+-tree. If there are N words in the lexicon and each leaf node can contain at most 128 wordId items, since each node is usually 2/3 full, each leaf node will contain $\lceil 256/3 \rceil$ wordId items; then there will be $\lceil 3N/256 \rceil$ leaf nodes. If each internal node can also contain at most 128 words, there will be at most 129 points in each internal node. Since each node is usually 2/3 full, there will be 86 points in each internal node. So by using a B+ tree structure, the system only needs $\lceil \log_{86} 3N/256 \rceil$ accesses to locate the leaf node. For example, if there are 500,000 words in the lexicon, it only needs 2 accesses to locate the leaf node.

- The way to write the information of word occurrences into the forward index file

In the B+-tree we constructed in VQAS, all the information that is required to create the forward index is stored in the leaf nodes. A pointer called *start* is used to point to the first leaf node in the tree, and each leaf node has a pointer to its next leaf node in the order of wordId. Since the forward index uses exactly the

same order to store the information of word occurrences in a particular document, we can easily access all the leaf nodes to create the forward index by following the *start* pointer and the *next* pointer in each leaf node.

- Use of memory

For some structures, such as array and hash table, the memory allocation is static; we have to give the length of the array or the hash table. Since the number of words in documents varies a lot and is unpredictable before the document is processed, it is difficult to choose the length of the array or the hash table to store the information of word occurrences. In B+-tree, the memory is dynamically allocated to create a new node as needed and is released when all the information in the leaf nodes is written to the forward index.

3.3.2 Forward index creator

In the `B+-tree` creator module, all the information of word occurrences for a particular document is stored in the B+-tree after the system finishes scanning the document. The main task of the forward index creator module is to create a forward index using the information stored in the B+-tree. As mentioned in section 3.3.1, we can easily access all the leaf nodes to create the forward index by following the *start* pointer and the *next* pointer in each leaf node. During the creation process, the identification of the current document (known as docid), a list of words and their hit lists in the document are appended to the end of the forward index file.

Figure 3.3 shows an example of forward index. In this figure, the number in the third column indicates the number of times the word appears in the document

while each hit consists of the start position of the paragraph that contains the word and the position of the word in this paragraph. The line in which both

Doc 001	Word 001	61	hit hit hit ……hit
	Word 002	50	hit hit hit ……hit
	Word 005	4	hit hit hit hit
	Word 008	2	hit hit
	Word 010	15	hit hit hit ……hit
	Word 517	51	hit hit hit ……hit
	Word 980	41	hit hit hit ……hit
	0	0	
Doc 002	Word 001	31	hit hit hit ……hit
	Word 002	1	hit
	Word 003	4	hit hit hit hit
	Word 008	5	hit hit hit hit hit
	Word 057	11	hit hit hit ……hit
	Word 110	3	hit hit hit
	0	0	
Doc 003	Word 001	5	hit hit hit hit hit
	Word 002	2	hit hit
	Word 003	1	hit
	Word 007	4	hit hit hit hit
	Word 110	10	hit hit hit ……hit
	0	0	

Figure 3.3 A forward index example

wordId and number of hits are equal to 0 indicates the end of the forward list for the document. In this example, there are three documents in the document collection. Doc 001 contains 7 words, in which word 001 appears 61 times in the document, and word 002 appears 50 times in the documents.

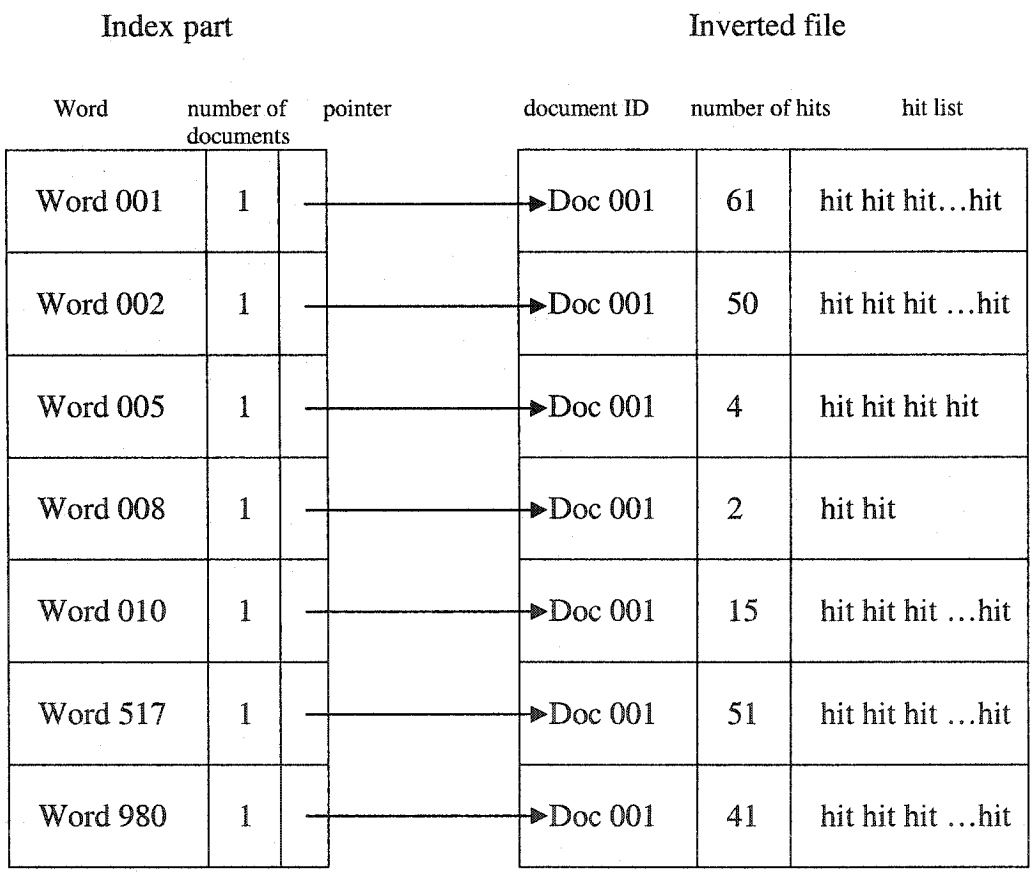
The forward index file is stored as a binary file in VQAS; in this file, we use 6 bytes for document id, 4 bytes for wordId, 2 byte for number of hits, and 5 bytes for each hit. For the hit, we use 3 bytes for position of paragraph and the other 2 bytes for position of word. The forward lists of documents are written to the forward index file in the order that the documents are processed, while in the forward lists of a particular document, the word and its hit list are written to the forward index file in the order of wordId.

3.3.3 Inverted index creator

The task of this module is to create an inverted index for all documents in the collection. The input of this module is the forward index file created in the forward index creator module, and the output is the inverted index. An inverted index consists of an inverted file and an index for inverted file. In VQAS, the inverted file contains a list of units that includes the identification of a document (denoted as docid), number of hits in this document for a particular word, and a hit list. The index of the inverted file is a linked list in which each element contains the identification of a word (denoted as wordId), number of documents in which this word occurs, and a pointer to the inverted list of this word in the inverted file.

Recall the forward index example in Figure 3.3; the following steps explain how an inverted index is created based on this example. First, an index and an inverted

file are created based on the data stored in the forward list of document 001; the result is shown in Figure 3.4. Next, the forward list of document 002 is merged into the index and the inverted file as shown in Figure 3.4, which creates the index and the inverted file for document 001 and 002; the result is shown in Figure 3.5. Next, the forward list of document 003 is merged into the index and the inverted file as shown in Figure 3.5, which creates the index and the inverted file for document 001, 002 and 003; the result is shown in Figure 3.6. Since there



Figue 3.4 Inverted file after inserting document 001

are only three documents in the collection for this example, we are done; if there are more documents, we continue the merge process until the end of the forward index file. Finally, the index part is stored in a disk file. The detailed algorithm can be found in Appendix B.

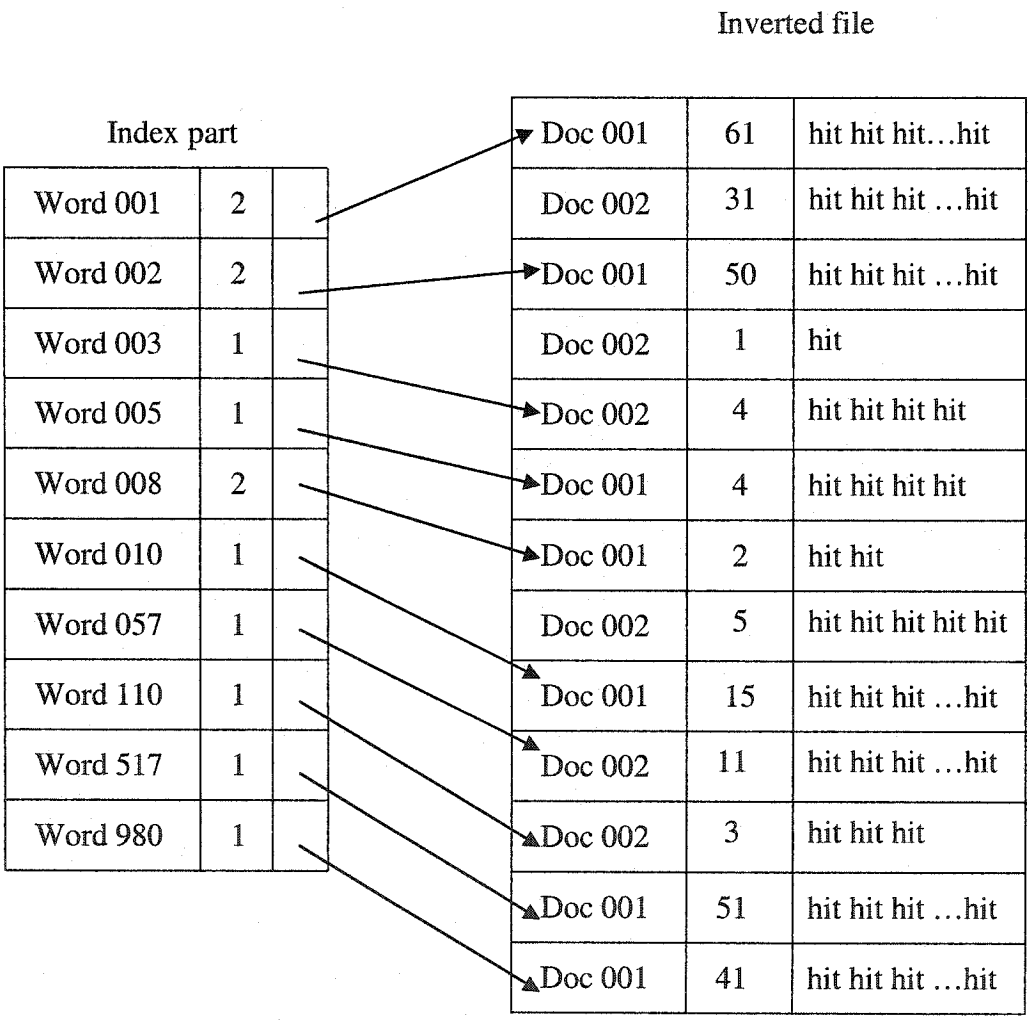


Figure 3.5 Inverted file of Figure 3.4 after inserting document 002

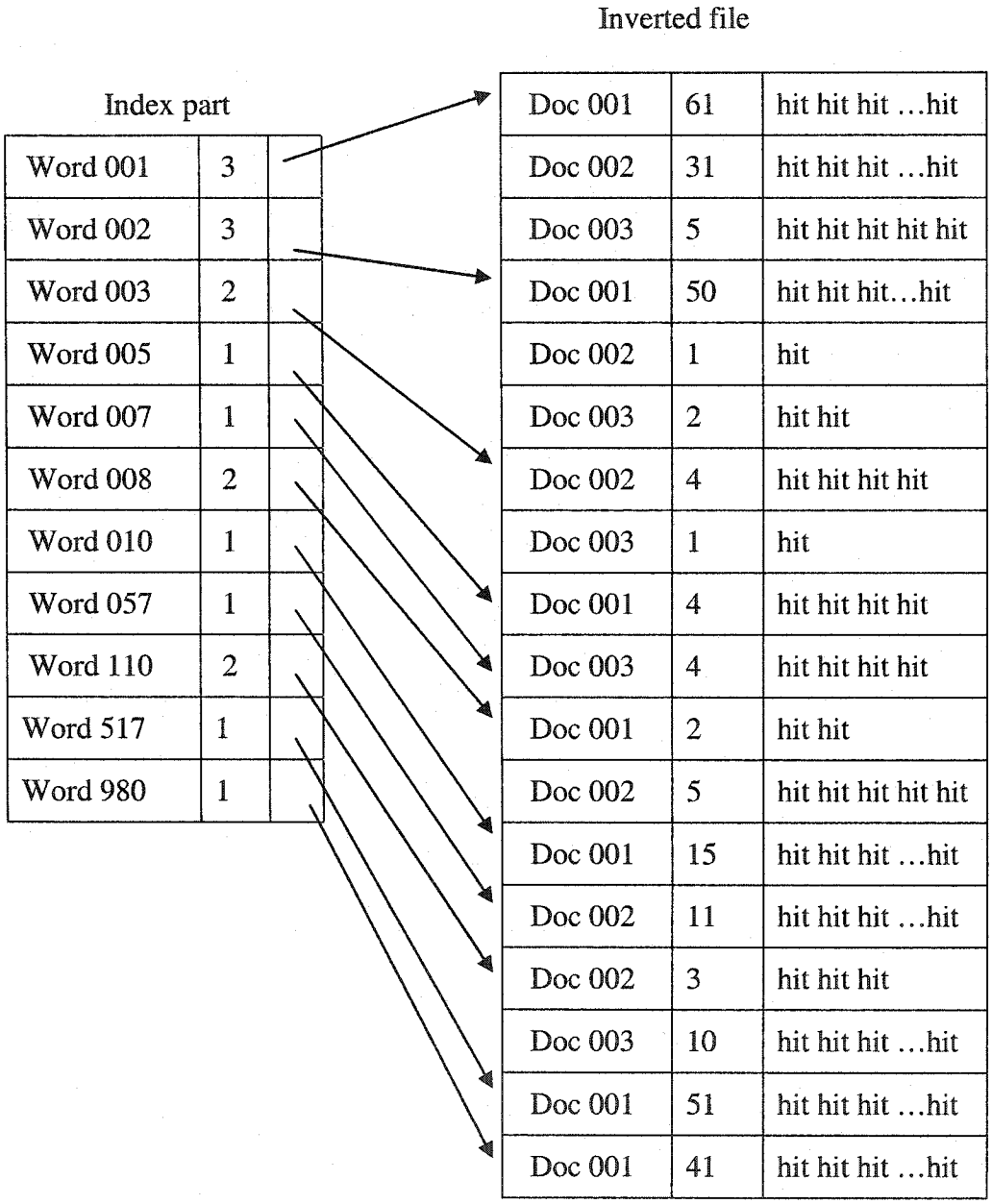


Figure 3.6 Inverted file of Figure 3.5 after inserting document 003

New documents are put into the collection continuously. To make the new documents searchable by users, the inverted index has to be updated frequently; the inverted index creator is run automatically once a week.

3.4 Search module

The objective of the search module is to retrieve the relevant paragraphs for user queries. The input of this module is the query stream from a user, and the output of this module is a file that contains a ranked list of the metadata of the paragraph relevant to the query; this metadata includes the identification of the document (denoted as docid) that contains the paragraph, the position of the paragraph in that document, and a hit list of those query terms in the paragraph. The processing steps of this module are described in Figure 3.7.

3.4.1 Query input

In the user interface module, a query is input by a user via the web interface. After getting this query, VQAS will insert this new query into a table called current_query in CINDI database for backend processing. Different users may have the same query input; therefore, we have to distinguish query input from users. Since VQAS uses the HTTP protocol, each request from a web client has a unique number called session id that is used to distinguish different user requests. The user interface module inserts a user's query along with a session id in the current_query table. There is a daemon called *search* on the server side, monitoring the current_query table; once it detects a new entry in the table, it will fetch the query, start the search process, send the result to the web client, and finally delete this entry from the table when the search process terminates. The schema of table current_query is shown in Table 3.2.

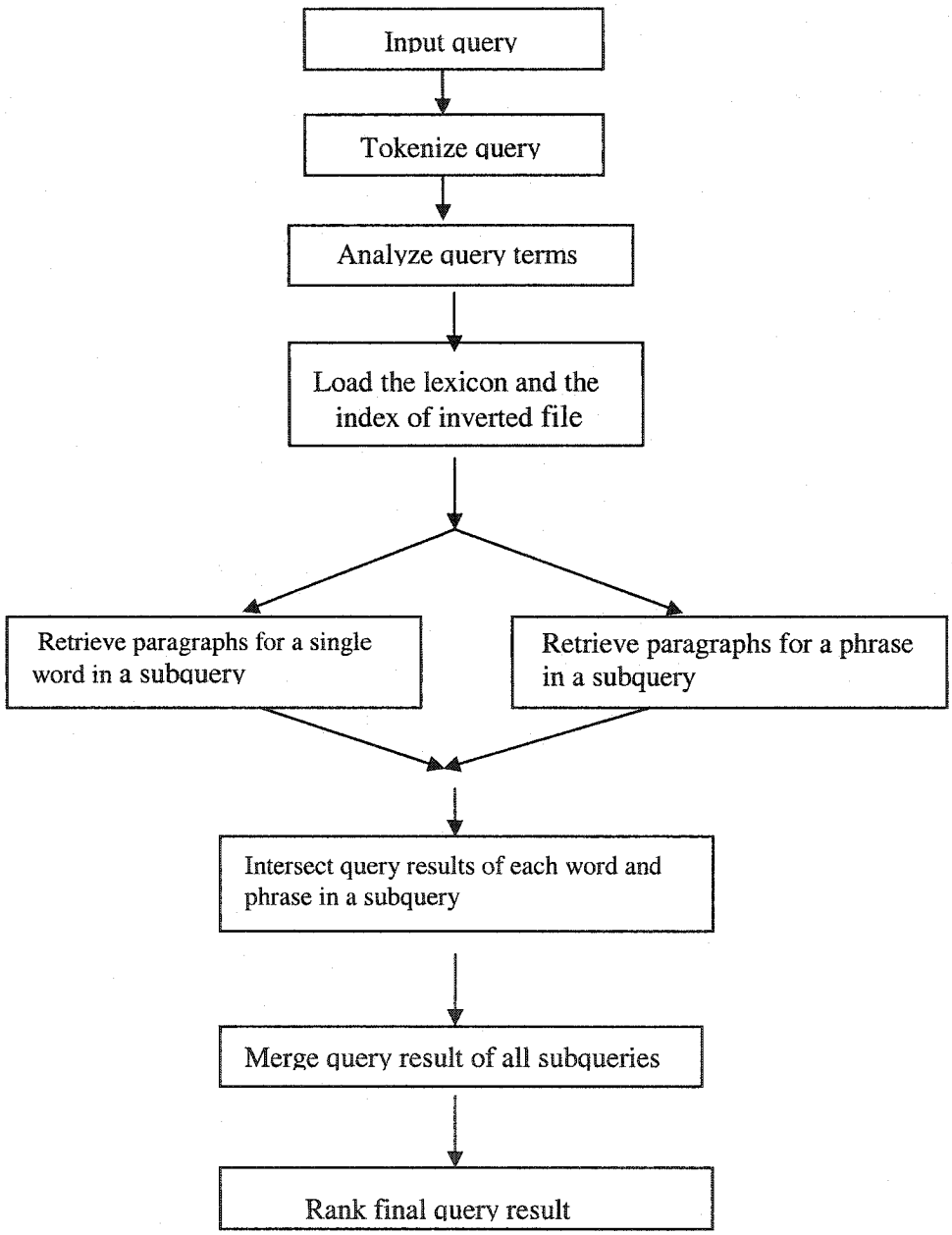


Figure 3.7 Processing step of the search module

Table 3.2 Schema of relation current_query

Attribute Name	Attribute Type	Comments
sessionID	integer	The session id of the user who inputs the query.
query	string	The query string.

3.4.2 Tokenize query

In VQAS, a query is a disjunction of a number of subqueries, and a subquery is a conjunction of keywords and phrases. A query is input as a set of terms separated by | (OR) and + (AND). A term can be a single word or a phrase distinguished by quotation marks. Since the AND operator has a higher precedence, a query is considered to be a set of subqueries separated by the OR operator. The OR operator implies that retrieved paragraphs have to satisfy at least one subquery. Each subquery is a set of keywords, and the AND operator is used to separate keywords. The AND operator implies that the result of a subquery has to contain all the words or phrases in the subquery. For example, the query ***"search engine" + system | "question answering" + system*** contains two subqueries: ***"search engine" + system*** and ***"question answering" + system***, and each subquery contains a phrase and a word. The result of this query are those paragraphs that satisfy the Boolean query *("search engine"* **AND** *system)* **OR** *("question answering"* **AND** *system)*. This step aims to break down a query into subqueries and then divide each subquery into a conjunction of words and phrases. A phrase is defined as a token between left quotation and right quotation marks while a word is defined as a token between spaces. After tokenizing, the query words and phrases in each subquery are stored for further processing.

3.4.3 Analyze query terms

This process includes three steps: removal of punctuation marks, conversion of all characters to lowercase, and elimination of stop words. These three steps share the same algorithms and approaches as those in the document processing module. Since stop word is a part of phrase, for example "theory of database", we do not eliminate stop words in phrases. The detail of the algorithms is given in section 3.2.

3.4.4 Load the lexicons and the index part of inverted index

As mentioned in section 3.2.4, the main lexicon and exc_lexicon used in the document processing module are stored in disk after the inverted index is created. Since the search module is run as a daemon, the main lexicon and exc_lexicon are reloaded into the memory using the same B-tree structures when the daemon is started. They will be used to look up the wordID of query word.

The index part of inverted index is loaded into the memory using the B-tree structure shown in Figure 3.8. In this structure, each node contains a set of index_items and pointers; each index_item contains an identification of a word (denoted as wordId), number of documents that contain this word, and the offset of the inverted list of this word in the inverted file. Given a wordId, this B-tree index is used to get the word occurrence from the inverted file. If this B-tree (the index part of the inverted index) can not be entirely loaded in the memory, we have to create a multi-level inverted index and only load the top level index into the memory.

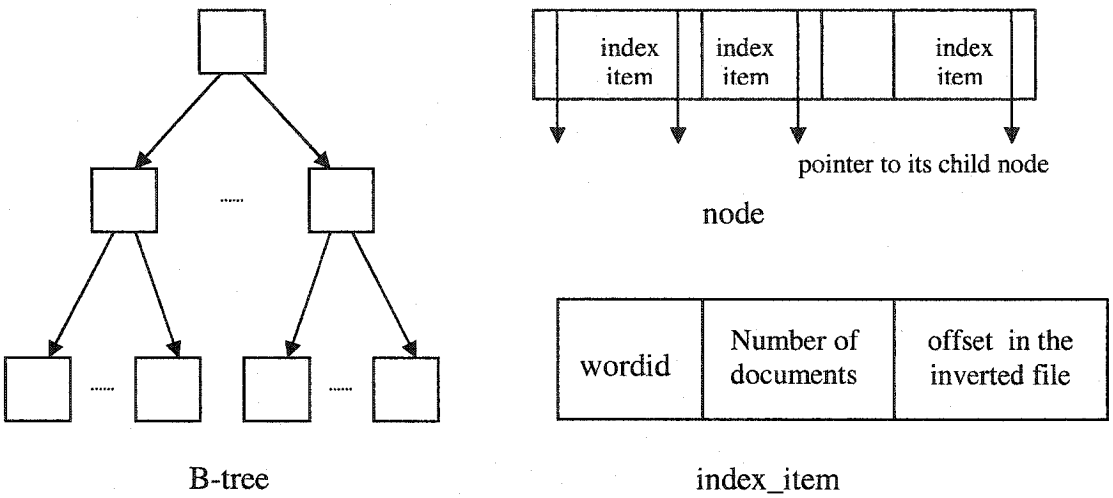


Figure 3.8 B-tree used to store the index of inverted file

3.4.5 Retrieve paragraphs for a single word

VQAS uses a linked list structure called match_list to store the metadata of retrieved paragraphs that includes the identification of the document, the position of the paragraph in the document, the hit list of the query word in the paragraph, and the relevance of the paragraph to the query. The details of this data structure are shown in Figure 3.9.

Data structure of match _list

Match_list is a linked list that stores the hit lists of a word for all paragraphs in the document collection. Its components are described below.

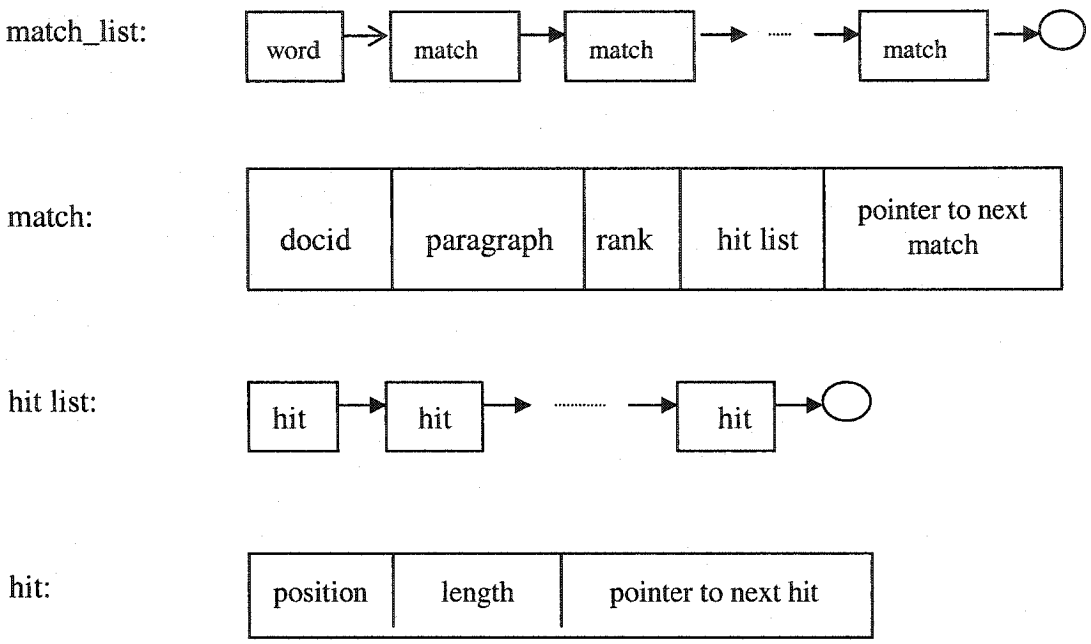


Figure 3.9 Data structure of match_list

Match: A match is used to store the hit list of a word in a specific paragraph. A match contains a docid, the start position of the paragraph in this document, a rank of this paragraph based on the relevance to the query word, and the hit list of the query word in this paragraph. The docid and start position of the paragraph are used together to identify a paragraph.

Hit list: A hit list is a linked list of all the hits of a word in a particular paragraph.

Hit: A hit contains the position and the length of the word in a particular paragraph, and a pointer to the next hit. The position and the length of the word are used to highlight the query word.

Algorithm for single word query

Step1: look up the wordId of the query word in lexicons;

Step2: Search for the wordId in the B-tree index to get the offset of inverted list of the query word in the inverted file;

Step3: retrieve the inverted list of the query word from the inverted file;

Step4: divide the hit list of the word foreach document into the hit lists for each paragraph in the document;

Step5: store the hit lists of this query word in paragraphs to a match_list;

Calculation of rank

How frequently a query word appears in a paragraph is one of the most understandable ways to determine a paragraph's relevance to a query [1]. At the same time, we perceive that the location of a paragraph indicates its significance to the document. Query words occurring at the beginning of a document may be more likely to be relevant than query words occurring later in the document. Based on this idea, we use the formula suggested by Salton and Buckley [34] to calculate the rank of a paragraph for a query word. During the implementation, we tested several variations of this formula while changing the parameters. The experimental results show that this formula can achieve better result.

Let N be the total number of documents in the system and n_i be the number of document in which the index term ki appears. Let $freq_{i,j}$ be the frequency of term k_i in the paragraph p_j (i.e., the number of times the term k_i is mentioned in the text of the paragraph p_j for a given document d_l), and $max_l(freq_{i,l})$ be the maximum

frequency of term k_i in all paragraphs in the document d_l. Let w_j be the weight of the paragraph in the document based on the position of the paragraph. Then, the rank of paragraph j for query word i is given by

$$R_{i,j}=(0.5 + 0.5\ freq_{i,j} / max_l(freq_{i,l})) * logN/n_i * w_j \quad [34]$$

where w_j is defined as

$w_j=1/m$, if paragraph j is mth paragraph in the document.

The motivation for usage of the factor $logN/n_i$ is that words appearing in many documents are not very useful for distinguishing a relevant document from a non-relevant one.

3.4.6 Retrieve paragraphs for phrase queries

Phrase queries only return paragraphs that contain the exact phrase. For each phrase query, all the words in the phrase must occur in a paragraph in the same order without any intervenient word.

For a phrase query, VQAS uses the algorithm for single word query to retrieve paragraphs for each word in the phrase and stores them in a match_list. Then it intersects the retrieved paragraphs for all words in the phrase using a function called phrase_intersection. For the phrase without stop words in it, only those paragraphs that contain all the words in the phrase and in the same order without any intervenient word will be chosen as the result for the phrase query. Since stop words are not indexed, for the phrase with stop words in it, only those paragraphs that contain all the non-stop words in the phrase and the distance between these non-stop words equals to the length of the stop word between them plus 2 (2 for 2

spaces before and after the stop word) will be chosen as the result for the phrase query. This is done by checking the position and the length of the word in the hit list. The detailed algorithm of function phrase_intersection can be found in Appendix C.

Phrase index tree

The processing of a phrase query is much slower than the processing of a single word query. The most time consuming part in processing a phrase query is to intersect the search results of the words into a phrase. In order to speed up a phrase query, an in-memory B-tree index is created to store the search results of most frequently searched phrases. A table is created in the CINDI database to store the frequency of phrases being searched. The number of phrases stored in the B-tree index depends on the size of memory. In VQAS, the search result of the top 1000 frequently searched phrases are stored in the B-tree index. For the frequently searched phrases, VQAS first uses the intersection process to get the search result; afterwards the system inserts this phrase and the search result into the phrase index tree. When this phrase is queried next time, VQAS can search the phrase index tree and get the result directly from the tree, which makes the query process much faster. The data structure of this B-tree index is shown in Figure 3.10.

Calculation of rank for phrase query

In a document, a phrase can be considered just as a single word; a paragraph's relevance to a phrase also depends on the frequency of the phrase and the location of the paragraph. So, we use the same function $R_{i,j}$ used in the rank calculation of a single query to calculate the rank of paragraph j for phrase i.

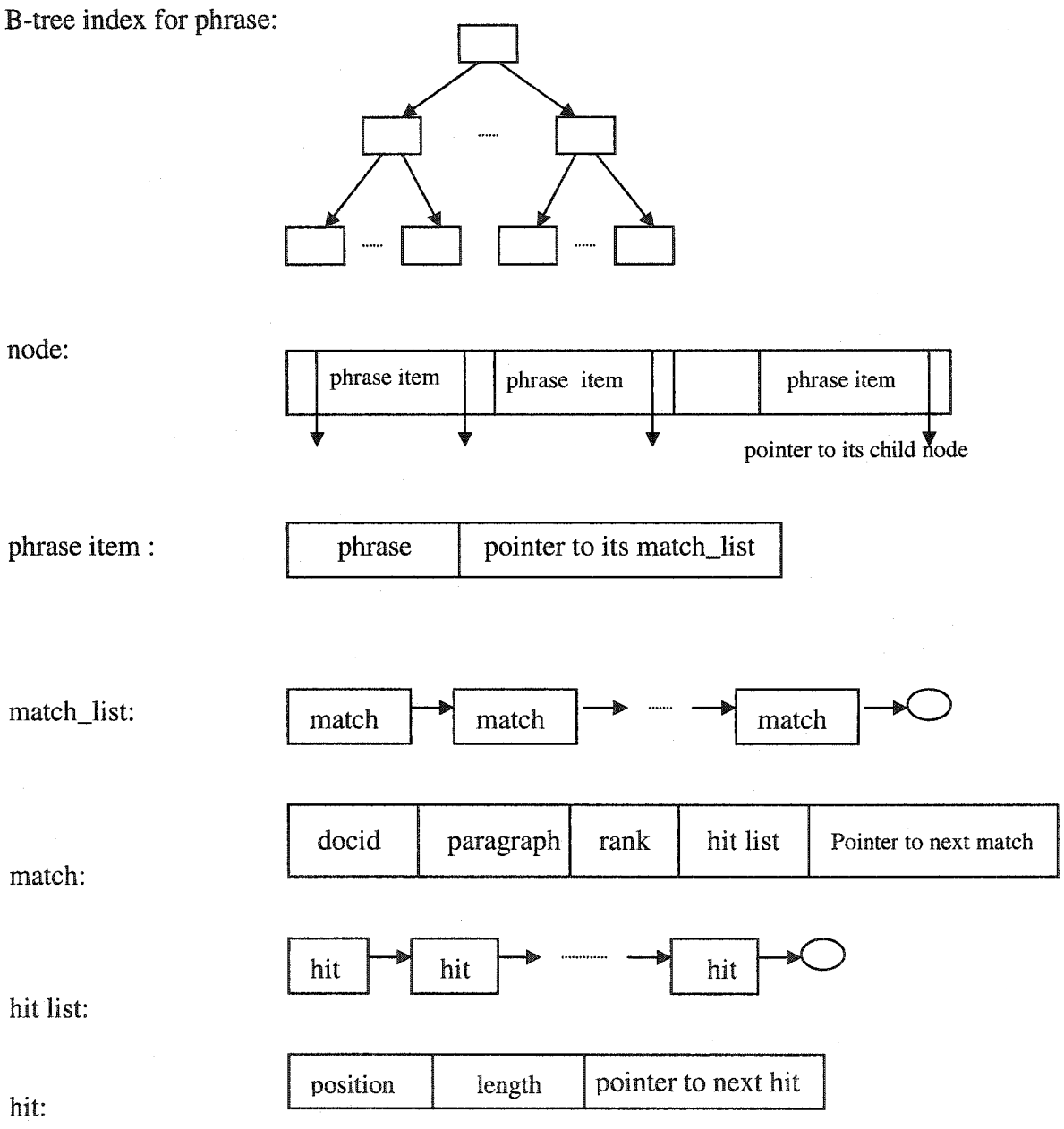


Figure 3.10 Data structure for phrase index

3.4.7 Intersect query results of single words and phrases in a subquery

Once we get the query results for all single words and phrases in a subquery, we have to intersect them to get the result for the whole subquery. The retrieved paragraphs for a subquery should contain all the words and phrases in the subquery, which means that all the words and phrases in the subquery are connected by "logical and". So only those paragraphs that contain all the words and phrases in a subquery will be stored in the match_list as the final result of the subquery. The hit list of a retrieved paragraph will contain all the hits of the words and phrases in the subquery in that paragraph. A function called intersection is used to intersect the query results of the words and phrases in a subquery. The detailed algorithm of function intersection can be found in Appendix E.

Calculation of rank

Each retrieved paragraph has its rank in the match_list that is used to store the meta data of this paragraph. When the query results of single words and phrases in a subquery are intersected, only those paragraphs that contain all the words and phrases in a subquery will be stored in the match_list as the final result of the subquery, and the average of the ranks of the retrieved paragraph for all the words and phrases in the subquery is taken as the rank of this paragraph for the subquery.

3.4.8 Merge query results of all subqueries

In the previous step, the search result of each subquery is stored in a match_list. In order to generate the result for the combined query of these subqueries, we need to merge these results for the subqueries. The retrieved paragraphs for a query should satisfy at least one of its subqueries. Since the connection of the subqueries is

"logical or", the set union of the retrieved paragraphs for the subqueries are in the result of the query. If some subqueries have the same paragraph in their result, the union of the hit lists of these subqueries is generated. A function called union is used to merge the query results of subqueries in the query. The detailed algorithm of function union can be found in Appendix D.

Calculation of rank

Each retrieved paragraph has its rank in the match_list that is used to store the meta data of this paragraph. When the query results of subqueries are merged, those paragraphs that satisfy one of the subqueries will be stored in the match_list as the final result of the query. If some subqueries have the same paragraph in their result, the maximum rank of the paragraph for all subqueries is taken as the rank of this paragraph for the whole query; for those paragraphs that satisfy only one of subqueries, we just keep the rank of this subquery as the rank for the whole query.

3.4.9 Rank the final query result

In this step, the meta data of retrieved paragraphs stored in a match_list is sorted by the rank of paragraphs respect to the query. Afterwards, the meta data of paragraphs in the match_list are exported and stored in a disk file, which will be used in the user interface module to generate the web page as the response to the user's query. In order to communicate with the user interface module, the name of this file is the same as the user's session id that is stored along with user's query in table current_query shown in Table 3.2.

3.5 User interface module

The user interface module provides users a graphical web interface, passes users' queries to the search module, and displays the search result. The input of this module are the user's query and the disk file created at the end of the search module that stores the meta data of retrieved paragraphs, and the output of this module is the web page that displays the query result to users.

The main page of the interface is shown in Figure 3.11. In this page, the user can input his/her query in the search "box"; once he/she clicks the search button, the query will be passed to the server and inserted into the table current_query shown in Table 3.2. Then the search module fetches this query as input, processes the search, and stores meta data of retrieved paragraphs in a disk file, in which the identification of the document and the start position of the paragraph are used to locate the paragraph, and the positions and length of the words are used to highlight the query words in the displayed paragraphs. This disk file will be removed after the result page is displayed.

The page of a query result is shown in Figure 3.12. Each page displays ten paragraphs in the order of the relevance to the query. Users can click the NEXT button in the bottom of the page to go to the next ten paragraphs or the PREVIOUS button to go to the previous ten paragraphs. Users can also click the number in between to go directly to the Nth page. Users may go to the next page after waiting for a long time, but if we keep the result file in the disk, it will use a lot of disk resource when the number of users is large. In VQAS, the disk file that stores meta data of retrieved paragraphs will be removed after the result page is displayed. The request for the next page will be considered as a new query; the

same result file will be created, but different parts will be displayed. A link to the original document in PDF format is available at the bottom of each paragraph.

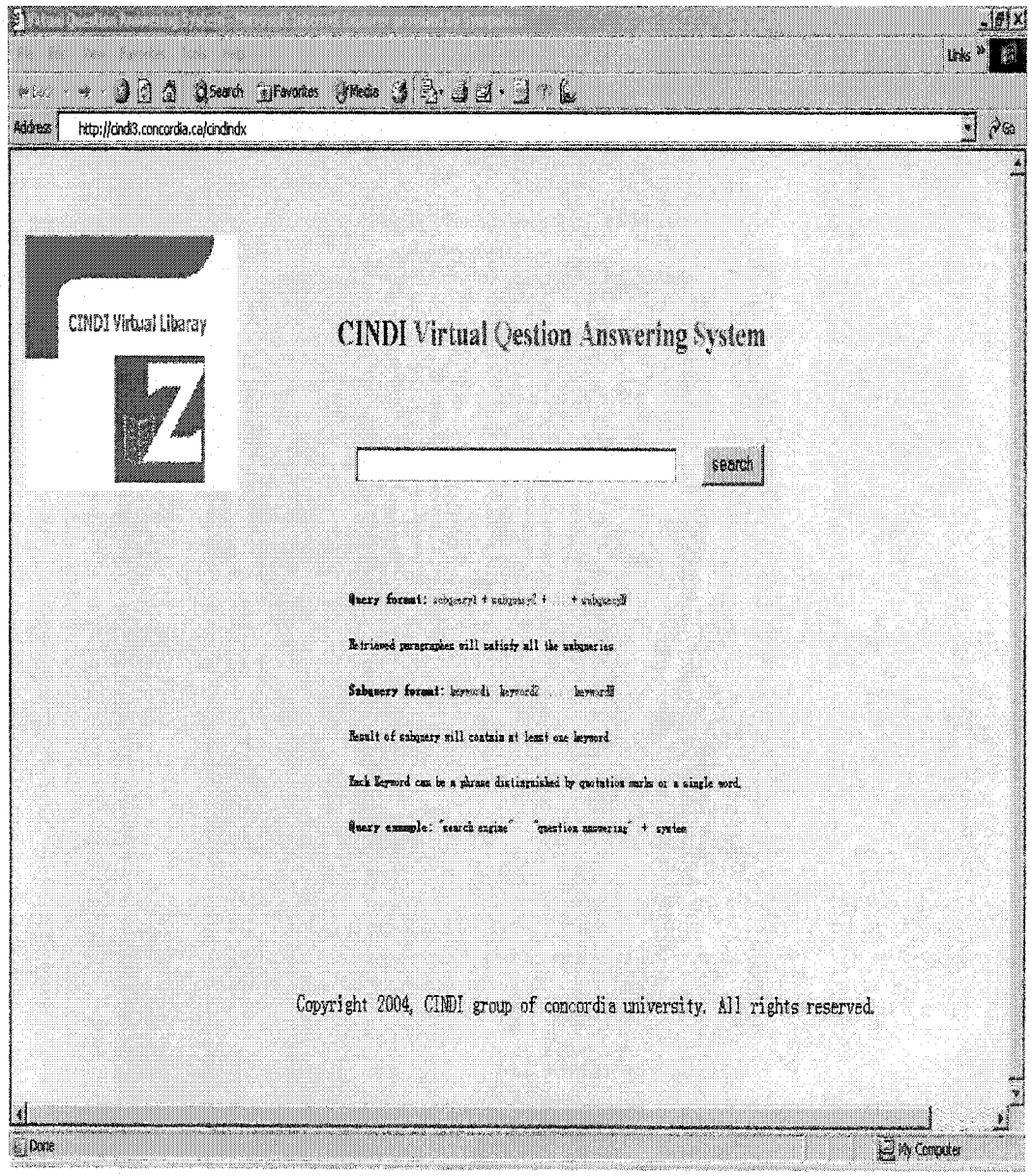


Figure 3.11 Main page of VQAS

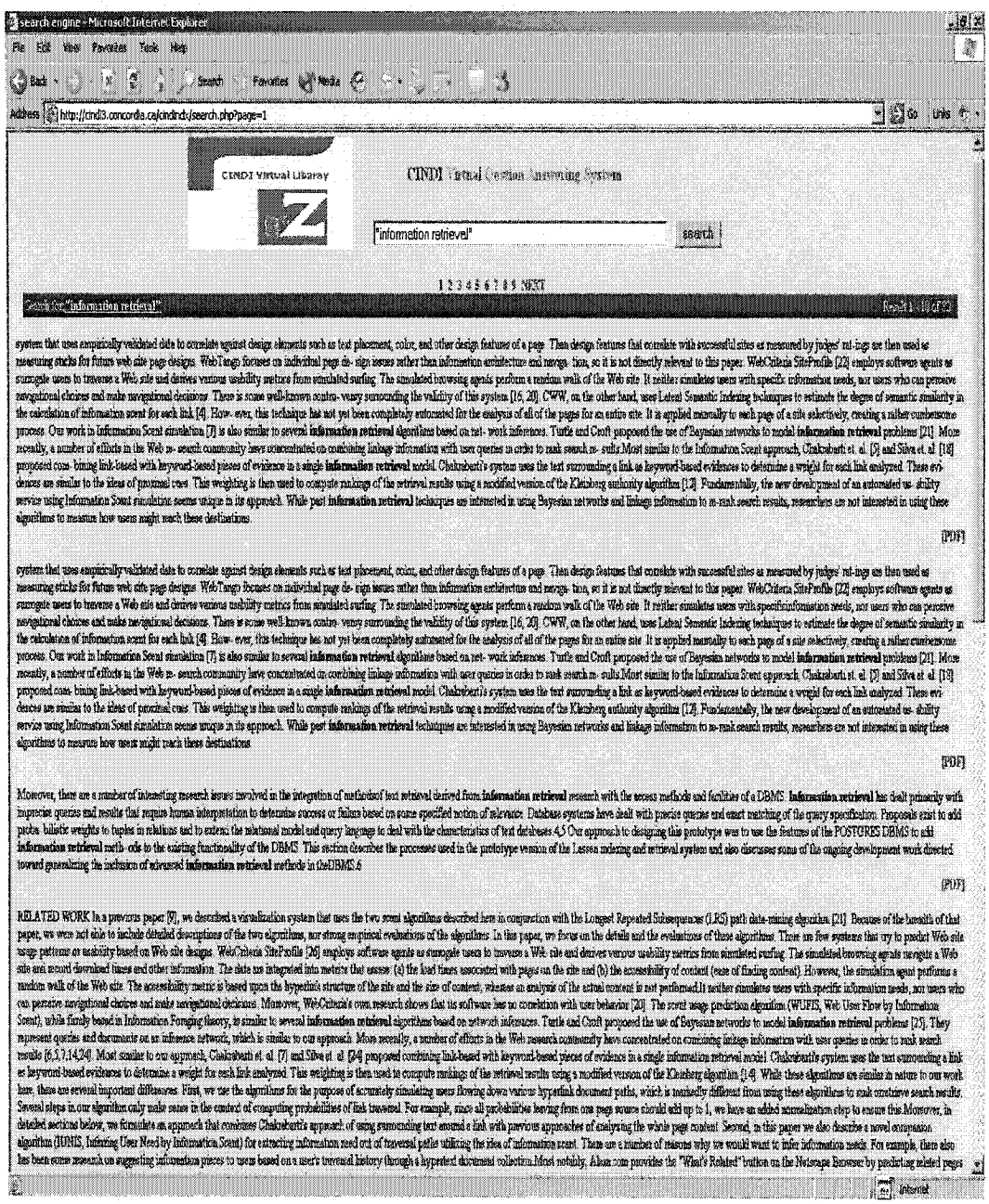


Figure 3.12 Query result

After passing a user's query to the search module, the user interface module will wait for the search module to create a disk file that contains the search result. Once the file is created, the user interface module continues its processing. Therefore, system synchronization must be included in this module. The synchronization strategy used in VQAS is shown in Figure 3.13.

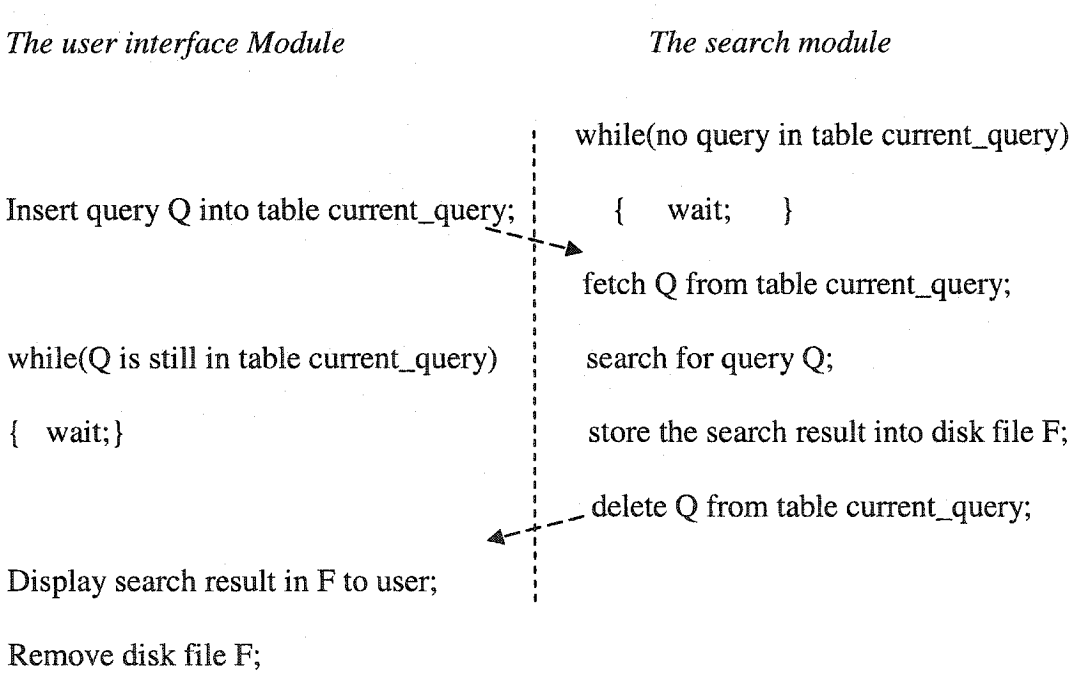


Figure 3.13 Synchronization between the interface module and the search module

Chapter 4

Experimental Results

VQAS is implemented in C++ on Linux, and the user interface is written in PHP. Also Mysql is used as database management system. Two kinds of test have been done on VQAS; one is to test the system performance, and the other is to test the performance of the in-memory phrase index. Both tests have been performed for a collection of 1000 documents, a collection of 5000 documents, and a collection of 10000 documents. The highlights of test results are given below.

4.1 System performance

The system performance is evaluated by the relevance of search results and the response time. Twenty queries of various types are chosen as sample queries. To evaluate the relevance of the search result, the first ten paragraphs in the result of each query are evaluated by colleagues who determine if each paragraph actually does contain relevant information. Next, we calculated the percentage of paragraphs judged as relevant to the query. To evaluate the response time, we record the search time for each query and calculate the average. Table 4.1, 4.2, and 4.3 shows the detailed results of system performance test, and Table 4.4 shows the statistics.

Table 4.1 The test of system performance for a collection of 1000 documents

Query input	Number of relevant paragraphs out of 10	Response time(second)
OLAP	9	0.03
Multimedia	7	0.03
Clustering	9	0.05
Animation	8	0.01
Graphics | animation	9	0.02
OLE | OLAP	10	0.02
security | network	10	0.08
"data mining"	9	0.54
"query processing"	9	0.61
"information retrieval"	9	0.17
"logic programming"	9	0.02
"query optimization"	10	0.57
"data mining" | "data warehousing"	10	1.06
OLAP + benchmark	9	0.03
Network + security	9	0.03
"query optimization" + database	9	1.57
text image + "information retrieval"	10	0.21
"logic programming" + application	9	0.3
Text + "information retrieval" + web	10	0.24
"database system" + "query processing"	10	1.56

Table 4.2 The test of system performance for a collection of 5000 documents

Query input	Number of relevant paragraphs out of 10	Response time(second)
OLAP	10	0.02
Multimedia	8	0.04
Clustering	9	0.08
Animation	9	0.02
Graphics | animation	10	0.03
OLE | OLAP	10	0.02
security | network	10	0.21
"data mining"	9	0.54
"query processing"	9	0.69
"information retrieval"	9	0.54
"logic programming"	10	0.05
"query optimization"	10	0.58
"data mining" | "data warehousing"	10	1.05
Network + security	9	0.07
OLAP + benchmark	10	0.03
"query optimization" + database	9	1.58
text image + "information retrieval"	10	0.61
"logic programming" + application	9	0.65
Text + "information retrieval" + web	10	0.63
"database system" + "query processing"	10	1.7

Table 4.3 The test of system performance for a collection of 10000 documents

Query input	Number of relevant paragraphs out of 10	Response time(second)
OLAP	10	0.03
Multimedia	9	0.11
Clustering	9	0.21
Animation	10	0.04
Graphics | animation	10	0.05
OLE | OLAP	10	0.04
security | network	10	0.97
"data mining"	9	0.58
"query processing"	10	1.08
"information retrieval"	9	1.51
"logic programming"	10	0.54
"query optimization"	10	0.86
"data mining" | "data warehousing"	10	1.06
Network + security	9	0.31
OLAP + benchmark	10	0.07
"query optimization" + database	10	1.84
text image + "information retrieval"	10	0.97
"logic programming" + application	9	1.05
Text + "information retrieval" + web	10	0.94
"database system" + "query processing"	10	2.06

Table 4.4 System performance

	Percentage of retrieved paragraphs judged as relevant	Average response time (second)	Size of the index (M)
For 1000 documents	92%	0.3575	15
For 5000 documents	95%	0.457	26
For 10000 documents	97%	0.716	64

From the experimental result, we can see that VQAS can achieve a high accuracy and a short response time. The time and space complexity in an inverted index is liner with a variation due to the size of the documents [1]. Therefore our experimental result clearly matches the theory and it proves to be a good implementation. However, VQAS still retrieves some irrelevant paragraphs, although those paragraphs contain all the query words. The reason is that some paragraphs in which the query words occur frequently may not under the topic of those query words. Table 4.5 shows an example of relevant paragraph and an example of irrelevant paragraph.

4.2 Performance of the in-memory phrase index

As mentioned in section 3.4.6, an in-memory phrase index is used to speed up the phrase query for top 1000 frequently searched phrases. To evaluate the performance of the in-memory phrase index, the response time of phrase search that use the phrase index is compared to the response time of phrase search that do not use the phrase index. Ten phrase queries are chosen as the sample queries.

In the beginning, since all the sample phrases are not in the top 1000 frequently searched phrases, the system will not use phrase index to process the search. We

Table 4.5 Examples of retrieved paragraphs

query	"information retrieval"
relevant paragraph	Moreover, there are a number of interesting research issues involved in the integration of methods of text retrieval derived from **information retrieval** research with the access methods and facilities of a DBMS. **Information retrieval** has dealt primarily with imprecise queries and results that require human interpretation to determine success or failure based on some specified notion of relevance. Database systems have dealt with precise queries and exact matching of the query specification. Proposals exist to add probabilistic weights to tuples in relations and to extend the relational model and query language to deal with the characteristics of text databases. Our approach to designing this prototype was to use the features of the POSTGRES DBMS to add **information retrieval** methods to the existing functionality of the DBMS. This section describes the processes used in the prototype version of the Lassen indexing and retrieval system and also discusses some of the ongoing development work directed toward generalizing the inclusion of advanced **information retrieval** methods in the DBMS.
Irrelevant paragraph	The main focus of this research group is the development of methods to improve today's **information retrieval** systems. Besides the design of the user interface itself, the focus is on the development of methods to adapt a retrieval system dynamically to the needs and interests of the user.

record the search time for all sample queries and take the average. Then we repeat querying the sample queries. Finally, all the sample queries will be in the top 1000 frequently searched phrases, and the system will use phrase index to process the search. Then we record the search time for all sample queries and take the average. Table 4.6, 4.7, and 4.8 shows the detailed results of performance test for phrase index, and Table 4.9 shows the average response time for both cases.

Table 4.6 The performance test of phrase index for 1000 documents

Phrase query	Response time without using phrase index(second)	Response time using phrase index(second)
“data mining”	0.54	0.02
“query processing”	0.62	0.02
“information retrieval”	0.17	0.02
“logic programming”	0.02	0.01
“query optimization”	0.57	0.02
“information system”	0.61	0.01
"data warehousing"	0.52	0.01
“file system”	0.52	0.02
"database system"	1.01	0.04
“question answering”	0.03	0.01

Table 4.7 The performance test of phrase index for 5000 documents documents

Phrase query	Response time without using phrase index(second)	Response time using phrase index(second)
“data mining”	0.54	0.01
“query processing”	0.69	0.02
“information retrieval”	0.54	0.01
“logic programming”	0.05	0.03
“query optimization”	0.58	0.02
“information system”	1.04	0.02
"data warehousing"	0.52	0.01
“file system”	0.62	0.02
"database system"	1.05	0.04
“question answering”	0.03	0.01

Table 4.8 The performance test of phrase index for 10000 documents

Phrase query	Response time without using phrase index(second)	Response time using phrase index(second)
“data mining”	0.55	0.02
“query processing”	1.04	0.02
“information retrieval”	0.68	0.02
“logic programming”	0.54	0.04
“query optimization”	0.84	0.02
“information system”	1.05	0.02
"data warehousing"	0.53	0.02
“file system”	1.04	0.02
"database system"	1.07	0.05
“question answering”	0.12	0.02

Table 4.9 Performance of the in-memory phrase index

	Average response time for phase queries that do not use phrase index	Average response time for phase queries that use phrase index
For a collection of 1000 documents	0.461	0.018
For a collection of 5000 documents	0.566	0.019
For a collection of 10000 documents	0.746	0.025

From the experimental result, we can see that the phrase index can raise the speed of the phrase search up to 30 times. As the number of documents in the collection increases, the phrase index plays a more important role for phrase search.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The VQAS system presented in this thesis aims to provide a method to retrieve full-text information from a digital library by targeting the need between a true question answering system and current search engines. It retrieves as result the paragraphs where the possible response corresponding to the user's query appear, not just a few words before and after the search terms. In our experiments, the user would be able to find the answer directly in the paragraphs of the query result.

The VQAS system consists of four modules: document processing module, index creation module, search module, and user interface module. The document processing module scans the documents and extracts index entries from them. The index creation module creates an inverted index for all the documents in the collection. The search module matches the query terms to the inverted file generated by the index creation module to get search result. The user interface module provides users a graphical web interface of the system.

The implementation of VQAS shows promising results from the statistical tests performed. This system can be incorporated into various application areas, especially in digital library.

5.2 Future work

Future work for VQAS leads to the following directions:

- Introduce spelling correction for query input.
- Introduce format checking for query input.
- Introduce new features to capture the notion of relevance of a paragraph to a query.
- When the size of document collection goes to 10^6 or 10^7, the response time of VQAS will become very slow. Then we can store the hit lists of some frequently used query words in memory to speed up the search. The number of query words whose hit lists will be stored in memory will depend on the size of free memory. The proposed algorithm can be found in Appendix F.

Reference:

[1] Ricardo Baeza-Yates, Berthier Bibeiro-Neto, "Modern Information Retrieval", ACM press, New York, 1999.

[2] Ellen M. Voorhees, "Overview of theTREC 2003 Question Answering Track", Proceeding of TREC-10 QA track, 2001.

[3] Berners-Lee T., Caillian R., Luotonen A., Frystyk Nielsen H., Secret A, "The World Wide Web", In Communication of the ACM, vol 37-8, p76-82, August 1994.

[4] B. Kahle, "Archiving the internet". Scientific American, Mar. 1997.

[5] N. Shivakumar, H. Garcia-Molina, "Finding near-replicas of documents on the Web", In Workshop on Web Databases, p204-212, Valencia, Spain, March 1998.

[6] Bar-Yossef Z., Berg A., Chien S., Weitz, J. F. D, "Approximating aggregate queries about web pages via random walks", Proceeding of the Twenty-sixth International Conference on Very Large Database, p535-544, 2000.

[7] Lawrence S., Giles C. L, "Accessibility of information on the web", Nature 400, 1999.

[8] Bharat K., Broder A, "Mirror, mirror on the web: A study of host pairs with replicated content", Proceedings of the Eighth International World-Wide Web Conference, 1999.

[9] G. Navarro, "Approximate Text Searching", PhD thesis, Dept. of Computer Science, Univ. of Chile, December 1998.

[10] A. Emtage, P. Deutsch, "Archie -- an electronic directory service for the internet", In USENIX Association Winter Conference Proceedings, p93-110, San Francisco, 1992.

[11] Matthew Gray, "Internet growth summary". http://www.mit.edu/people/ mkgray/net/internetgrowth -raw-data.html, 1997.

[12] M. Koster, "Aliweb - Archie-Like Indexing in the Web", In Proceedings of the First International World Wide Web Conference, p175-182, Amsterdam, 1994.

[13] Eichmann, D., "The RBSE Spider - Balanching Effective Search against Web Load", In Proceedings of the First International Conference on the World Wide Web, P113-120, Geneva, Switzerland, May 1994.

[14] Brian Pinkerton, "WebCrawler: Finding *what people want"*, PhD thesis, University of Washington, November 2000.

[15] Mauldin, M. L, "Lycos: Design Choices in an Internet Search Service", IEEE Expert, 12(1): p8-11, 1997.

[16] "History of Search Engines & Web History", http://www.search-marketing.info/search-engine-history/index.htm, January 21,2004.

[17] B. Morrissey, "Overture to Buy AltaVista", vol. 2003: Internet Advertising Report, 2003.

[18] Bipin C. Desai, Rajabihan Shayan Nader, R. Shinghal, Youquan Zhou, "CINDI: A System for Cataloging Searching and Annotating Documents in Digital Libraries", Library-trend, Vol. 48-1, Summer 1999.

[19] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine". Computer Networks and ISDN Systems, vol. 30, no. 1.7, p107-117, 1998.

[20] Ellen M. Voorhees, "The TREC-8 Question Answering Track Report", Proceeding of TREC-8 QA track, 1999.

[21] Cody C. T. Kwok, Oren Etzioni, Daniel S. Weld, "Scaling Question Answering to Web", Tenth World Web Conference, p150-161, Hong Kong, China. May1-5, 2001.

[22] Sanda Harabagiu, Marius Pasca, Steven Maiorano, "Experments with open-domain textual question answering", COLING-2000, p292-298, Association for Computational Linguistics/Morgan Kaufmann, Aug 2000.

[23] Boris Katz, "From sentence Processing to Information Access on the World Wide Web", AAAI Spring Symposium on Natural Language Processing for the World Wide Web, Stanford, California. 1997.

[24] Sanda Harabagiu, Dan Moldovan, Razvan Bunescu, "Answering Complex, List and Context Questions with LCC's Question-Answering Server", Tenth Text Retrieval Conference (TREC-10), Gaithersberg, MD. November 13-16, 2001.

[25] Eduard Hovy, Laurie Gerber, Chin-Yew Lin, "Question Answering in Webclopedia", Ninth Text Retrieval Conference (TREC-9), Gaithersberg, MD. November 13-16, 2000.

[26] Bipin C. Desai, Sami S. Haddad Abdelbaset Ali, "Automatic Semantic Header Generator", ISMIS2000, Springer-Verlag, Charlotte, NC, 2000.

[27] Google, http://www.google.com/help/basics.html, February 03, 2004.

[28] J. Zobel, A. Moffat, R. Sacks-Davis, "An efficient indexing technique for full-text database systems", Proceedings of the International Conference on Very Large Database, p352-362, 1996.

[29] P. O'Neil, "Model 204 Architecture and Performance", in Proceedings of the 2nd International Workshop on High Performance Transactions Systems, p40-59, 1987.

[30] C. Faloutsos, S. Christodoulakis, "Signature files: an access method for documents and its analytical performance evaluation", ACM Transaction on Office Information System, p267-288, 1984.

[31] Steve Jones, Sally Jo Cunningham, Rodger McNab, Stefan Boddie, "A transaction log analysis of a digital library", International Journal on Digital Libraries, Vol. 3, p152-153, 2000.

[32] J. B. Lovins, "Development of a stemming algorithm". Mechanical Translation and Computational Linguistics 11, p22-31,1968.

[33] WordNet online lexical reference system, http://www.cogsci.princeton.edu/~wn/, July 16, 2004.

[34] G. Salton, C. Buckley, "Term-weighting approaches in automatic retrieval". Information Processing and Management, p513-523, 1988.

[35] Tong Zhang, "Gleaning subsystem for CINDI". Master's thesis, Dept. of Computer Science, Concordia University, August 2004.

Appendix A: Stop word list

A	about	abs	accordingly
after	again	against	all
almost	already	also	although
always	am	among	an
and	any	anyone	apparently
are	as	aside	at
away	be	because	been
between	both	briefly	but
by	can	cannot	could
do	does	during	e.g
each	either	etc	for
from	further	had	has
have	having	he	her
here	his	how	however
if	in	into	is
it	its	itself	just
may	me	mine	more
moreover	must	my	need
no	now	of	often
on	only	or	other
our	out	refs	shall
she	should	since	so
such	than	that	the
their	them	then	there
therefore	these	they	this
those	though	through	thus
to	too	under	until
upon	us	was	we
were	what	whatever	when
where	whether	which	while
who	whose	will	with
without	within	would	yet
you	your□		

Appendix B: The Algorithm for inverted index creation

```
void create_ib()
{
    int i=0,j=0,k,tem1,tem0;
    long l;
    unsigned long doc0;
    char filename1[128]="/cndhm/cindindx/fb";
     char filename2[128]="/cndhm/cindindx/ib";
     char filename3[128]="/cndhm/cindindx/ind";
     fstream fpt0("/cndhm/cindindx/temp1",ios::out|ios::binary);//open a temp inverted file
     fstream fromfb1fp0(filename1, ios::in|ios::binary);//open the forward index file
     ofstream outin(filename3,ios::out); //open the file to store the index of inverted file
     if(fromfb1fp0.eof()) //if forward index is empty
          return;
     fromfb1fp0.read((char *)&doc0,4);//read a docid from forward index
     fromfb1fp0.read((char *)&wh0,8);//read a wordid and its number of hits
     i=0;
     // create the inverted file and its index array for the first document
     while(!((wh0.wordid==0)&&(wh0.hits==0))) //while it not the end of forward list
     {
              //store the wordid, number of documents and the offset in inverted file
       lex1[i].wordid=wh0.wordid;
       lex1[i].docs=1;
       lex1[i].offset=fpt0.tellp();
       i++;
       dh0.hits=wh0.hits;
       dh0.docid=doc0;
       fpt0.write((char *)&dh0,8);//write the docid and number of hits into inverted file
       //write the hit list into inverted file
       for(int k1=0;k1<int(dh0.hits);k1++)
       {
         fromfb1fp0.read((char *)&oh,8);
         fpt0.write((char *)&oh,8);
       }
       fromfb1fp0.read((char *)&wh0,8);//read next wordid and its number of hits
   }
     fpt0.close();
     lex1[i].wordid=0;
     lex1[i].docs=0;
     //copy the index array to a temporary array
     for(int k1=0;k1<=i;k1++)
```

```
 {
  lex[k1].wordid=lex1[k1].wordid;
  lex[k1].docs=lex1[k1].docs;
  lex[k1].offset=lex1[k1].offset;
 }
j=i;
fromfb1fp0.read((char *)&doc0,4);//read the next docid
fromfb1fp0.read((char *)&wh0,8);//read wordid and number of hits
 // merge the forward list of next document into the inverted index
while(!fromfb1fp0.eof()) //while it is not the end of forward index file
{
  fstream fpt1("/cndhm/cindindx/temp1",ios::in|ios::binary);
  fstream fpt2("/cndhm/cindindx/temp2",ios::out|ios::binary);
  i=0;  j=0;
  tem1=lex1[i].wordid;
  tem0=wh0.wordid;
  while(!((tem1==0) && (tem0==0))) //while it is not the end of forward list
  {
    if(tem0==tem1)// if wordid is already in the index array
     {
      lex[j].wordid=wh0.wordid;
      l=fpt2.tellp();
      lex[j].docs=lex1[i].docs+1; //increase the number of document by 1
      lex[j].offset=l;
      k=lex1[i].docs;
     bool flag=1;
     // merge the hit lists of the word in the two documents
       while(k>0)
       {
        if(dh1.docid>doc0 && flag)
         {
          dh0.docid=doc0;
          dh0.hits=wh0.hits;
          fpt2.write((char *)&dh0,8);
          for(int k1=0;k1<int(wh0.hits);k1++)
           {
            fromfb1fp0.read((char *)&oh,8);
            fpt2.write((char *)&oh,8);
           }
          fromfb1fp0.read((char *)&wh0,8);
          flag=0;
         }
        fpt1.read((char *)&dh1,8);
```

```
      fpt2.write((char *)&dh1,8);
      for(int k1=0;k1<int(dh1.hits);k1++)
        {
          fpt1.read((char *)&oh,8);
          fpt2.write((char *)&oh,8);
        }
      k--;
    }
    if(flag)
    {
        dh0.docid=doc0;
        dh0.hits=wh0.hits;
        fpt2.write((char *)&dh0,8);
      for(int k1=0;k1<int(wh0.hits);k1++)
       {
         fromfb1fp0.read((char *)&oh,8);
         fpt2.write((char *)&oh,8);
       }
      fromfb1fp0.read((char *)&wh0,8);
      flag=0;
    }
    i++;  j++;  //go to next word
    tem1=lex1[i].wordid;
    tem0=wh0.wordid;
}
// if the wordid in the forward list is less than the current wordid in the index
   array, or the end of index array has been reached.
else if((tem0<tem1 && tem0!=0) || tem1==0)
{
    lex[j].wordid=wh0.wordid;
    l=fpt2.tellp();
    lex[j].docs=1;
    lex[j].offset=l;
    j++;
    dh1.docid=doc0;
    dh1.hits=wh0.hits;
    fpt2.write((char *)&dh1,8);
    // insert a new inverted list to the inverted file
  for(int k1=0;k1<int(wh0.hits);k1++)
  {
      fromfb1fp0.read((char *)&oh,8);
      fpt2.write((char *)&oh,8);
      for(int k1=0;k1<int(wh0.hits);k1++)
```

```
        {
          fromfb1fp0.read((char *)&oh,8);
          fpt2.write((char *)&oh,8);
         }
        fromfb1fp0.read((char *)&wh0,8);
        tem0=wh0.wordid;
   }
  // if the wordid in the forward list is greater than the current wordid in the
     index array, or the end of forward list has been reached.
  else if((tem0>tem1 && tem1!=0)|| tem0==0)
  {
     lex[j].wordid=lex1[i].wordid;
     l=fpt2.tellp();
     lex[j].docs=lex1[i].docs;
     lex[j].offset=l;
     k=lex1[i].docs;
    //copy the inverted list to the new inverted file
     while(k>0)
       {
        fpt1.read((char *)&dh1,8);
        fpt2.write((char *)&dh1,8);
        for(int k1=0;k1<int(dh1.hits);k1++)
          {
           fpt1.read((char *)&oh,8);
           fpt2.write((char *)&oh,8);
          }
        k--;
       }
     i++; j++;
     tem1=lex1[i].wordid;
    }
  }
lex[j].wordid=0;
lex[j].docs=0;
fpt1.close();
fpt2.close();
// replace old inverted file by the new one
remove("/cndhm/cindindx/temp1");
rename("/cndhm/cindindx/temp2","/cndhm/cindindx/temp1");
//copy the new index array to lex1
for(int k1=0;k1<=j;k1++)
{
  lex1[k1].wordid=lex[k1].wordid;
```

```
        lex1[k1].docs=lex[k1].docs;
        lex1[k1].offset=lex[k1].offset;
      }
      fromfb1fp0.read((char *)&doc0,4); //read the next docid
      fromfb1fp0.read((char *)&wh0,8); //read a wordid and its number of hits
    }
    // replace old inverted file by the new one
    remove(filename2);
    rename("/cndhm/cindindx/temp1",filename2);
    // store the index array to a disk file
    for(int k1=0;k1<=j;k1++)
     {
      outin<<lex[k1].wordid<<" "<<lex[k1].docs<<" "<<lex[k1].offset<<endl;
     }
    outin.close();
    fromfb1fp0.close();
    return;
}
```

Appendix C: The Algorithm for the phrase_intersection function

The phrase_intersection function is used to intersect the match_lists of two words in the same phrase.

```
phrase_intersection(t1, t2, t3)
{
define two pointer p1 and p2 with type of match;
let p1 point to the first match object in t1, p2 point to the first object match in t2;
while (p1!= NULL  || p2 !=NULL)
{
    if (p1->docid == p2->docid &&  p1->paragraph == p2->paragraph)
    {
        create a match object m3;
        define two pointer h1 and h2 with type of hit;
        let h1 point to the first hit in t1, h2 point to the first hit in t2;
        while(h1!=NULL && h2!=NULL)
        {
            if (h1->position < h2->position)
            {
                if (h2->position – h1->position == (h1->length +1))
                {
                    m3.docid=p1->docid;
                    m3.paragraph=p1->paragraph;
                    calculate a new rank using the ranks of p1 and p2;
                    assign the new rank to m3.rank;
                    create a hit object h;
                    h.position=h1->position;
                    h.length= h1->length + h2->length +1;
                    add h to the hit list of m3;
                     h2=h2->next;
                }
                h1=h1->next;
            }
```

```
                else
                    h2=h2->next;
                }
                add m3 to t3;
                move p1 to the next match in t1;
            move p2 to the next match in t2;
        }
        else if(p1->docid>p2->docid || (p1->docid==p2->docid && p1->paragraph>p2->paragraph))
            move p2 to the next match in t2;
        else
            move p1 to the next match in t1;
    }
}
```

Appdix D: The algorithm for the union function

The union function is used to merge the match_lists of two words in a subquery.

```
union(t1, t2, t3)
{
    define two pointer p1 and p2 with type of match;
    let p1 point to the first match in t1, p2 point to the first match in t2;
    while (p1!= NULL || p2 !=NULL)
    {
        create a match object m3;
        if (p1->docid == p2->docid && p1->paragraph == p2->paragraph)
        {
            copy the docid and the paragraph of p1 to m3;
            calculate a new rank using the ranks of p1 and p2;
            assign the new rank to m3.rank;
            merge the hit list of p1 and the hit list of p2 to the hit list of m3 in the
             order of position;
            move p1 to the next match in t1;
            move p2 to the next match in t2;
        }
        else if(p1->docid>p2->docid || (p1->docid==p2->docid && p1->paragraph>p2-
>paragraph))
        {
            copy the match object pointed by p2 to m3;
            move p2 to the next match in t2;
        }
        else
        {
            copy the match object pointed by p1 to m3;
            move p1 to the next match in t1;
        }
        add m3 to t3;
    }
    while ( p1!= NULL)
```

```
        {
            create a match object m3;
            copy the match object pointed by p1 to m3;
            move p1 to the next match in t1;
                add m3 to t3;
        }
        while ( p2!= NULL)
        {
            create a match object m3;
            copy the match object pointed by p2 to m3;
            move p2 to the next match in t2;
                add m3 to t3;
        }
}
```

Appdix E: The algorithm for the intersection function

The intersection function is used to intersect the match_lists of two subqueries in a query.

```
void intersection(match* m1,match* m2,match* m3)
{
  hit_list *p1,*p2,*p3;
  int i=0,j=0,k=0;
  clean_table(m3, 5000);
  while(m1[i].docid!=0 && m2[j].docid!=0)
  {
    if(m1[i].docid==m2[j].docid && m1[i].paragraph==m2[j].paragraph)
    {
        p1=m1[i].hits;
        p2=m2[j].hits;
        while(p1!=NULL && p2!=NULL)
        {
            if(p1->position < p2->position)
            {
              if(m3[k].docid==0)
              {
                    m3[k].docid=m1[i].docid;
                    m3[k].paragraph=m1[i].paragraph;
                    m3[k].rank=m1[i].rank+m2[j].rank;
                    m3[k].hits=(hit_list*)malloc(sizeof(struct hit_list));
                    p3=m3[k].hits;
                    p3->position=p1->position;
                    p3->length=p1->length;
                    p3->next=NULL;
              }
              else
              {
                    p3->next=(hit_list*)malloc(sizeof(struct hit_list));
                    p3=p3->next;
```

```
                p3->position=p1->position;
                p3->length=p1->length;
                p3->next=NULL;
            }
          p1=p1->next;
        }
        else
        {
          if(m3[k].docid==0)
          {
                m3[k].docid=m1[i].docid;
                m3[k].paragraph=m1[i].paragraph;
                m3[k].rank=m1[i].rank+m2[j].rank;
                m3[k].hits=(hit_list*)malloc(sizeof(struct hit_list));
                p3=m3[k].hits;
                p3->position=p2->position;
                p3->length=p2->length;
                p3->next=NULL;
          }
          else
          {
                p3->next=(hit_list*)malloc(sizeof(struct hit_list));
                 p3=p3->next;
                p3->position=p2->position;
                 p3->length=p2->length;
                 p3->next=NULL;
          }
           p2=p2->next;
        }
 }//end of while
 while(p1!=NULL)
{
    if(m3[k].docid==0)
     {
                m3[k].docid=m1[i].docid;
```

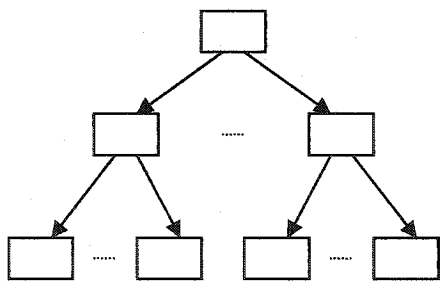
```
                m3[k].paragraph=m1[i].paragraph;
                m3[k].rank=m1[i].rank+m2[j].rank;
                m3[k].hits=(hit_list*)malloc(sizeof(struct hit_list));
                p3=m3[k].hits;
                p3->position=p1->position;
                p3->length=p1->length;
                p3->next=NULL;
        }
        else
        {
                p3->next=(hit_list*)malloc(sizeof(struct hit_list));
                p3=p3->next;
                p3->position=p1->position;
                p3->length=p1->length;
                p3->next=NULL;
        }
        p1=p1->next;
}
while(p2!=NULL)
{
      if(m3[k].docid==0)
      {
                m3[k].docid=m1[i].docid;
                m3[k].paragraph=m1[i].paragraph;
                m3[k].rank=m1[i].rank+m2[j].rank;
                m3[k].hits=(hit_list*)malloc(sizeof(struct hit_list));
                p3=m3[k].hits;
                p3->position=p2->position;
                p3->length=p2->length;
                p3->next=NULL;
        }
        else
        {
                p3->next=(hit_list*)malloc(sizeof(struct hit_list));
                p3=p3->next;
```

```
                    p3->position=p2->position;
                    p3->length=p2->length;
                    p3->next=NULL;
                }
                p2=p2->next;
            }
            i++;
            j++;
            k++;
        } //end of if
        else if((m1[i].docid < m2[j].docid) || (m1[i].docid==m2[j].docid &&
(m1[i].paragraph < m2[j].paragraph)))
        {       i++;}
        else if(m1[i].docid>m2[j].docid || (m1[i].docid==m2[j].docid &&
m1[i].paragraph>m2[j].paragraph))
        {       j++; }
    }//end of while loop
    clean_table2(m1, 5000);
    clean_table2(m2, 5000);
    return;
}
```

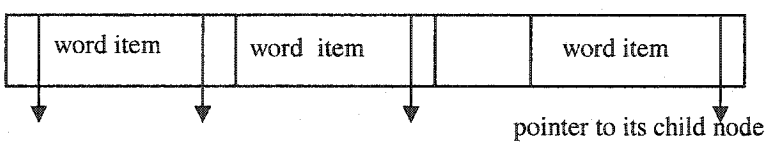
Appendix F: The proposed algorithm for scale up

In order to speed up the search, an in-memory B-tree is created to store the search results of most frequently searched words. A sorted array is created in memory to store the search frequency of all words. The number of words stored in the B-tree index depends on the size of memory. Assume the search result of the top N frequently searched words are stored in the B-tree, when a query word becomes one of the top N frequently searched words, we insert this word and its search result into the B-tree. When this word is queried next time, VQAS can search the B-tree and get the result directly from the tree, which makes the query process much faster. The data structure of this B-tree is shown below.

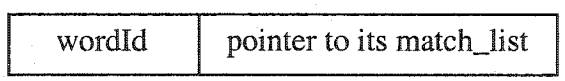
B-tree index for the top N frequently searched words:

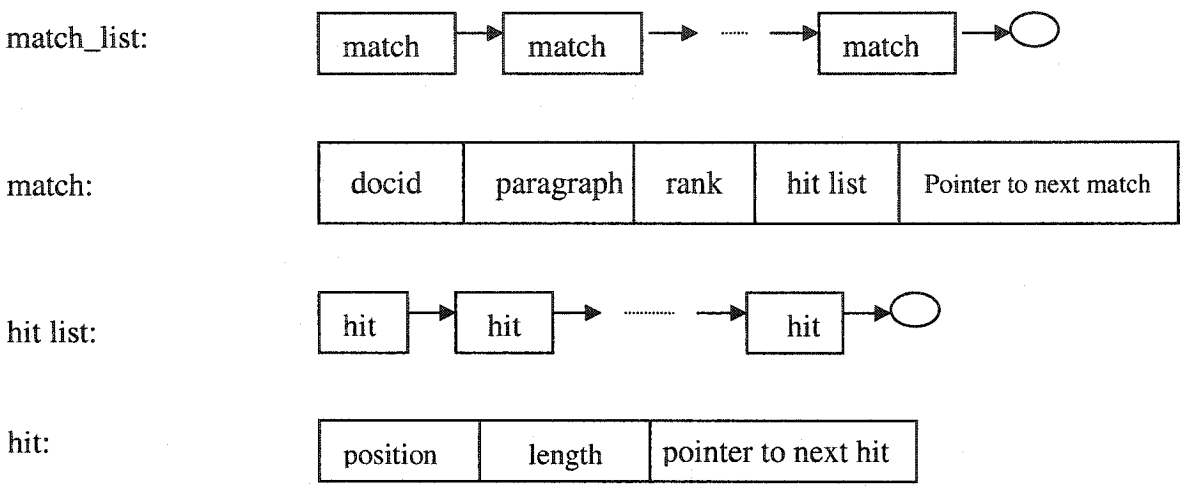


node:



word item :




match_list:
match
match
match
match:
docid
paragraph
rank
hit list
Pointer to next match
hit list:
hit
hit
hit
hit:
position
length
pointer to next hit