5.1 INTRODUCTION

Making search engines responsive to human needs require understanding of users’ queries and their navigations within the search results. Understanding the semantics of submitted queries provide an interesting perspective towards building efficient index structures while user behavior characterization provides knowledge about the workloads imposed on the search engine, which can further be used to address crucial points such as content crawling and caching, data distribution and relevant information retrieval. In view of this, a couple of techniques under the Pre-Mining phase have been developed to improve the effectiveness of search engines. With a view to optimize back-end tasks of a search engine and to understand query semantics and user’s usage trends, the various web resources (dictionary base sites, query log etc) have been employed.

A detailed discussion on proposed pre-mining techniques is given in the following sections:

5.2 OPTIMIZATION OF CRAWLING AND INDEXING PROCESS

The proposed optimization technique [122] utilizes the users’ browsing behavior to regulate the crawling and indexing process with a view to direct the crawler to download the important pages, which were not previously crawled and indexed. As the work attempts to index most of important pages based on user feedback, it would result in retrieval of relevant pages.

To support this work, the basic crawler and indexer process have been modified to incorporate some additional modules and data structures as shown in Fig. 5.1. For instance, the URL frontier that contains URLs whose corresponding pages are yet to be downloaded by the downloader module of the crawler. The extracted URLs from the downloaded pages are added to frontier for further crawling. A URL Filter & Duplicate Eliminator module
examines every URL added in the frontier to increase the speed and accuracy of the crawler. *URL filter* is being used to determine whether the extracted URL be excluded from the frontier based on one of several tests. For instance, the filter may seek to exclude certain domains (say, all `.com` URLs) and accordingly, the test would simply filter out the URL pertaining to `.com` domain. A similar test could be inclusive also as many hosts on the Web place certain portions of their websites off-limits to crawling (robot exclusion protocol), the URLs of such portions are also eliminated by this module. Moreover, certain depth-sensitive crawlers crawl up to a specific depth in the websites to speed up their crawling process and such depth violations are also examined by the URL filter.

![Fig. 5.1 Refined Crawling and Indexing Process of a Search engine](image)

Finally, the URLs are checked for duplicate elimination by the *Duplicate Eliminator*. If a URL is already in the frontier or has already been crawled, it is not added to the frontier. When a URL is added to the frontier, it is assigned a priority based on which, it is eventually dispatched from the frontier for downloading.
An effective feedback Mechanism continuously works on query logs maintained by the search engine and finds the user relevance feedback. Crawl-Index Updater (CI.Updater) computes the feedback. This periodically results in a set of URLs that seem relevant and thus, need to be downloaded and indexed. These URLs are considered by the crawler in its next crawl.

The following subsection describes various data structures proposed for the modified crawling and indexing process, while the feedback mechanism is dealt in later subsections.

### 5.2.1 DATA STRUCTURES USED

Data structures play an important role in the task of information accumulation. The existing data structures employed by crawling and indexing process have been updated for the sake of better interpretability, efficiency and effectiveness. Following are the modified data structures, which are used in the current work:

1. URL Frontier
2. Page_Info Repository
3. Inverted Index

The schema for these data structures is shown in Fig. 5.2. It may be noted that Page_Info Repository is basically a combination of two sub-schemas: Link store and Token store.

![Fig. 5.2 Data Structures for Crawling and Indexing Process](image)
The three data structures are described in detail as follows.

5.2.1.1 URL Frontier

URL Frontier (or queue) stores the URLs of the pages in the order in which the URLs need to be downloaded from the web. Initially, it contains the seed URLs i.e. the URLs to start the crawling process. Afterwards, it is populated with the URLs extracted from the downloaded pages and consists of the URLs whose corresponding pages have yet to be fetched. As pages are fetched, the corresponding URLs are removed from the frontier. This data structure has been modified by adding extra fields (such as depth and Crawl_bit) in its previous versions. Description regarding various fields of the modified frontier is shown in Table 5.1.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>The name of the URL to be fetched, e.g., 'en.wikipedia.org/wiki/Domain_Name_System'.</td>
</tr>
<tr>
<td>Depth</td>
<td>This field specifies how deep the URL is on the Web. It is a numerical constant such that $0 &lt; \text{Depth} &lt; c$. Here $c$ is a pre-specified max depth threshold used by depth-sensitive crawlers to crawl to a specific depth in the web graph. The URLs violating the depth threshold are eliminated by the URL Filter module.</td>
</tr>
<tr>
<td>Priority</td>
<td>The URLs are assigned an integer priority $P$, $1 &lt; P &lt; N$ (where $N$ is the highest priority), based on its fetch history, depth and other constraints. For instance, a page that has exhibited frequent change would be assigned a higher priority. Other heuristics could be application dependent.</td>
</tr>
<tr>
<td>Crawl_bit</td>
<td>A bit indicating the status of the URL, i.e., whether the corresponding page should be fetched by the downloader in the current crawl or not. Its value is either 0 or 1, with 1 indicating that it has passed the filtering mechanism and can be crawled, while a 0 indicates that it is yet to be examined by the URL Filter and Duplicate Eliminator.</td>
</tr>
<tr>
<td>Host Name</td>
<td>The host name of the URL, e.g., 'en.wikipedia.org' for the above example URL.</td>
</tr>
</tbody>
</table>

5.2.1.2 Page_Info Repository

This repository contains entire information about all downloaded pages. It employs two data structures to store this information: the link_store and token_store. The link_store contains structural summary of the web graph, while token_store contains information regarding the content of web pages. The description of various fields in these two schemas is described in Table 5.2.
Table 5.2 Description of Page_Info Repository

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>Name of the URL (say $u_i$), which has been fetched from the web.</td>
</tr>
<tr>
<td>Next_URL</td>
<td>Name of the URLs corresponding to out linked pages of the fetched page having URL $u_i$.</td>
</tr>
<tr>
<td>Depth</td>
<td>It specifies how deep the page with URL $u_i$ is in the web graph.</td>
</tr>
<tr>
<td>Doc_ID</td>
<td>Each page is assigned a unique serial number i.e. ID for referencing. The ID can be a sequential number or string e.g. D1, D2, D3 etc.</td>
</tr>
<tr>
<td>Page_address</td>
<td>It gives the location on the server site where the page is stored. This field allows fast accessing of page.</td>
</tr>
<tr>
<td>Token</td>
<td>When parser tokenizes a page, a set of tokens (possible strings of characters) are produced. Punctuations are usually thrown out in this process. A token is stored in this field.</td>
</tr>
<tr>
<td>Freq</td>
<td>It specifies the number of occurrences of the token in the page.</td>
</tr>
<tr>
<td>Position_Info</td>
<td>This field tells at what positions the token appears in the page e.g. a token ‘mining’ at position 23 specifies that it is the 23rd token of the page.</td>
</tr>
</tbody>
</table>

5.2.1.3 Inverted Index

Index contains information about all the terms (keywords) present in the downloaded pages. Here, a term means something different from a token. A normalized token after undergoing linguistic preprocessing (stemming, lemmatization processes etc.) is called as term. Index stores all the page terms alphabetically and is usually split into two partitions: a dictionary and postings. The fields in the index indicate the information as shown in Table 5.3.

Table 5.3 Description of Index Repository

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>A normalized token in the page.</td>
</tr>
<tr>
<td>Doc_ID</td>
<td>The Document ID of the page, in which the specified term appears.</td>
</tr>
<tr>
<td>Depth</td>
<td>The depth of the page with Doc_ID in the Web.</td>
</tr>
<tr>
<td>In-links</td>
<td>The number of back links of the page derived from the Link_Store repository.</td>
</tr>
<tr>
<td>Out_links</td>
<td>The number of forward links of the page derived from the Link_Store repository.</td>
</tr>
<tr>
<td>Rank</td>
<td>It is a score provided to a page which is generally based upon its link information e.g. Google’s PageRank. The rank may also be provided on other parameters of the page such as its content, depth, click count etc.</td>
</tr>
<tr>
<td>Freq</td>
<td>It is the number of occurrences of the specified term in the page.</td>
</tr>
<tr>
<td>Position_Info</td>
<td>At what positions, the term appears in the page.</td>
</tr>
<tr>
<td>Click_count</td>
<td>It is an integer number indicating the number of times users clicked on the page. This information is derived from the search engine logs.</td>
</tr>
</tbody>
</table>
The organization of information in this index structure can be understood by means of some sample terms and related page information as given in Table 5.4, which after actually being placed in the index get converted into a dictionary and associated list of postings as shown in Fig. 5.3.

Table 5.4 Sample Index Information

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc ID/URL</th>
<th>Depth</th>
<th>In-links</th>
<th>Out-links</th>
<th>Rank</th>
<th>Freq</th>
<th>Click count</th>
<th>Position Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>D5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>7, 35</td>
</tr>
<tr>
<td>abstract</td>
<td>D7</td>
<td>5</td>
<td>4</td>
<td>25</td>
<td>6</td>
<td>5</td>
<td>21</td>
<td>4, 6, 26, 99, 328</td>
</tr>
<tr>
<td>balloon</td>
<td>D4</td>
<td>2</td>
<td>13</td>
<td>22</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td>33, 59, 300, 552</td>
</tr>
</tbody>
</table>

Fig. 5.3 Organization of Information in Proposed Index Structure

*Dictionary* consists of terms and their respective document frequencies i.e. the number of documents containing the term, while *Postings* provide the related document information. The last field in the *document postings* is the term frequency which gives the number of occurrences of the term in the document. The *position postings* give the positions in a document at which term appears. So, the schema can also be described as:

*Dictionary* → *Document Postings* → *Position Postings*

where

*Dictionary*: `<term, doc_freq>`

*Document Postings*: `<Doc_ID, depth, inlinks, outlinks, rank, click_count, term_freq>`

*Position Postings*: `<pos_1, pos_2, ... pos_n>`
5.2.2 SIGNIFICANCE OF THE IMPROVED DATA STRUCTURES

The refined data structures are efficient due to the following reasons:

- Placing *depth* information with the associated URLs help improves the crawling efficiency for depth-sensitive crawlers. The URLs above a specific depth need not to be crawled unless they seem important. It also helps in conveying user feedback back to the crawlers (Section 5.2.3).

- *Crawl_bit* directs the crawlers to crawl the required URLs. If a page seems important, it can be made to be fetched even in the cases of depth threshold violations by setting its crawl bit and raising its *priority*.

- The *URL* and *next_URL* field in the *link_store* helps indexer to easily and efficiently gather the in-link and out-link information about a page in an incremental manner. For example, if it is required to assimilate the in-links of a page (say B), B is searched for in the *next_URL* field and the number of occurrences of B specifies the number of in-links as shown in Table 5.5. The associated URLs in the URL field represent the in-linked URLs. In the sample, there are three in-links of B which are A, C, D and one out-link of B which is E.

<table>
<thead>
<tr>
<th>URL</th>
<th>Next URL</th>
<th>Depth</th>
<th>Doc ID</th>
<th>Page address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C</td>
<td>3</td>
<td>D1</td>
<td>P1</td>
</tr>
<tr>
<td>C</td>
<td>B, D</td>
<td>4</td>
<td>D2</td>
<td>P2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>4</td>
<td>D3</td>
<td>P3</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>5</td>
<td>D4</td>
<td>P4</td>
</tr>
<tr>
<td>E</td>
<td>A, F, G</td>
<td>5</td>
<td>D5</td>
<td>P5</td>
</tr>
</tbody>
</table>

- *Page_address* helps to identify the server where page is stored. It helps in fast retrieval of the required web page.

- The *freq* and *Position_Info* are very important from the perspective of the user query. They are used to correlate the document content with the user queries.

- *Token* and *Term* both are preserved so as to response maximum matched result pages to the user. In cases, where linguistic preprocessing is supposed to miss the precise meaning of a query (e.g. *operation, operate, operational* get converted to *opera*), the results can be compiled by considering both tokens and related terms.
• By storing in-links, out-links, depth and rank, new ranking methods can be devised and the existing ones can be optimized.

• Click count can be considered as important measure for determining the importance of a page and in turn the crawl bit of a URL. (Section 5.2.3).

The data structures described here serve as a platform for the proposed feedback mechanism employed in the crawling and indexing process, which is illustrated in the next section.

5.2.3 USER FEEDBACK MECHANISM

The purpose of proposing a feedback mechanism in search engines is to make crawlers adaptive in accordance to users’ accessing histories, which previously were ignored by the crawling and indexing processes of search engines. Most of the present day search engines’ administrators extract usage trends from their server logs, but use them towards building recommender systems, web personalization systems and prefetching systems etc. None of them employs this important knowledge towards improving the crawler efficiency so as to fetch important pages, which in turn will help satisfy the user information needs. The proposed mechanism does the same by communicating usage trends extracted from query logs to the crawler and indexer modules.

5.2.3.1 Problem Statement

To better identify the problem and to understand the proposed mechanism, consider an example scenario which is described below in terms of three different states:

State 1:
A user (say \( U_i \)) submits a query \( Q \) on the interface of a search engine and gets the resultant URL set \( R = \{ R_1, R_2, \ldots, R_n \} \), where \( n \) is the number of result pages (or URLs) arranged in the form of an ordered list.

State 2:
\( U_i \) randomly accesses some URL \( R_i, 1 < i < n \) (see Fig. 5.4). He browses the page \( R_i \) and finds a link \( L_j \) of some another page on \( R_i \), \( L_j/R_i, 1 < j < n \) and the page corresponding to \( L_j \) has not been previously indexed by the search engine. If link \( L_j \) seems interesting to him, he browses it. Afterwards, he may continue browsing other result pages or stop the
process. In general, the accessing information i.e. query $Q$ and corresponding clicked URLs get stored in query logs.

Web Usage Mining Architecture

We have developed a general architecture for Web usage mining which is presented in [MJHS96] and [CMS97]. The WEBMINER is a system that implements parts of this general architecture. The architecture divides the Web usage mining process into two main parts. The first part includes the domain-dependent processes of transforming the Web data into suitable transaction form. This

Fig. 5.4 A Sample Page $R_i$ and embedded Hyperlinks $L_i$

**State 3:**

After a period of time, when another user $U_2$ (or same user $U_1$) gives the same query $Q$ or some similar query $Q'$, gets the same result set $R$ (assuming the same page collection). But $R$ does not directly contain URL $L$ (previously accessed page) and other such URLs, which were not indexed by the search engine due to depth constraints but may convey relevance to the user query.

The above three states illustrate that, in general, such non-indexed URLs remain hidden in the depths of web and thus, get unvisited by the web crawlers. Such URLs are difficult to find and can only be traced by the user accessing patterns. For instance, consider the following pattern of URLs accessed by a user:

$$R_i: R_3: \{R_7 \rightarrow L_1 \rightarrow L_2 \rightarrow L_3\}; \{R_9 \rightarrow \ldots\}; \ldots$$

The user accesses three random pages $R_3, R_5, R_7$ and then page $L_1$ which is out-linked to $R_7$: $L_2$ linked to $L_1$ and then $L_3$ linked to $L_2$. At last, he accesses page $R_9$ and some forward links in it. The example scenario assumes that any user may or may not browse the forward links of any result page. Now, it may be that any or all of $L_1, L_2$ and $L_3$ pages are not indexed by the search engine due to their depth constraints. But, user may find any of these links relevant to his search query and obviously would like such relevant pages to be directly present in the search results. Therefore, search engines must be capable to index such type of pages.
The proposed mechanism finds out such non-indexed links from the query logs and optimizes the crawling and indexing process based upon these links. The next subsection describes the proposed component \textit{CI\_Updater} of search engine to carry out this work.

5.2.3.2 The Crawl-Index Updater

The Crawl-Index Updater (CI\_Updater) has been designed with a view to evaluate user feedbacks and optimize the crawling-indexing process by using them. It may be noted that it works offline in the proposed architecture as shown in Fig. 5.1 and thus, does not affect the search and retrieval time.

The detailed feedback mechanism is shown in Fig. 5.5, from where it can be observed that CI\_Updater works on the query logs maintained by search engines. It extracts click-through information from the logs, based on which it applies necessary modifications in the index and also sends the collected user feedbacks to the crawling phase. The CI\_Updater performs its functioning with the help of two sub-modules:

1. Index Updater & Feedback Calculator Module (IUFM)
2. Crawl\_Updater Module (CUM)

![Fig. 5.5 Feedback Mechanism in a Search Engine](image-url)
The detail functioning of these two sub-modules is shown in Fig. 5.6 and is explained below:

1. Updating the Index: Index Updater & Feedback Calculator Module

The Index Updater & Feedback Calculator Module (IUFCM) as shown in Fig. 5.6 continuously picks a clicked URL from the query log, searches it in the index files and if found, updates its click-count in the index, so that next time, current information is available. After doing this, the URL is remembered by IUFCM so as to avoid searching it again. After some arbitrary time, when same URL is accessed by some user, a new log entry would be created and IUFCM visits it again. The same procedure is repeated for every accessed URL.

For every visited URL, IUFCM also checks its possibility for feedback calculation and if it satisfies the decided criterion, it is considered further for feedback calculation. For this purpose, forward links of the page corresponding to the URL under examination are extracted from the index, reason being that the forward links may or may not be indexed by the search engine and IUFCM tries to send the information of such non-indexed links of frequently visited pages to the crawler. Below are given the various steps taken in the feedback calculation process of this module.

**Step 1:**
For each clicked URL, it checks, if the URL is at least at the maximum depth constraint set by the crawler (note that a general user is not aware of the URL depth on the Web) and has undergone at least a fixed number of user clicks. If yes, it is considered a valid target URL. Importance of the URL is examined by checking its click_count with some pre-specified threshold C. If URL’s click_count is below C, then for the present visit, it is rejected for the feedback calculation.

**Step 2:**
It retrieves the forward linked URLs of the target URL from the index repository and sends an update signal to the another module of C1_updater which is Crawl Updater to collect the feedback in terms of list containing forward linked URLs, which must be fetched by the crawler.

IUFCM repeats steps 1 and 2 in an unremitting manner, always sending user feedbacks in terms of important URLs to the crawlers, which in turn results in indexing of relevant pages in the index of the search engine.
2. Modifying the Crawling Process: Crawl Updater

After receiving feedbacks from IUFCM, the Crawl Updater Module (CUM) starts functioning as described in Fig. 5.6. The information sent to CUM represents the feedback in terms of following:

- A list of URLs, which were not previously crawled and indexed
- Parent URLs of the sent URLs
- Depth information of URLs

The depth of a non-indexed URL can be found from its parent indexed URL as given below:

\[
\text{Depth (URL)} = \text{Depth (Parent (URL))} + 1
\]  

(C.1)

CUM places these URLs in the URL frontier, setting their crawl bit to 1, and writes their depth and host name as extracted from the parent URLs. Setting the crawl bit forces the crawler to crawl the new URLs even if they violate the depth constraints set by the crawler. CUM also assigns initial priorities to these URLs. The priority can be calculated on the basis of many factors such as importance of parent URLs, visit frequency and depth etc.

When crawler visits such a URL, it resets its crawl bit to 0 and downloads its corresponding page. Parser parses the page, but its further links are not entered into the frontier due to depth
constraints but they are kept in the link_store, so that they can be crawled next time if they happen to be some relevant URLs. After parsing, the page gets indexed by the indexer and next time, may appear in the search results. The new indexed URL also changes the out link information of its parent URL in the index and other repositories.

5.2.3.3 Algorithm of Cl_Updater

The algorithm of Cl_Updater is shown in Fig. 5.7. The algorithm operates offline and thus, does not affect the online query processing performed by the search engine. As an input, it takes a clicked URL, related information from the query log, depth threshold $d$ and Click count threshold $C$. It outputs in the modified index and URL frontier according to user needs.

Algorithm: Cl_Updater()

I/P:
- Depth threshold $d$
- Click count threshold $C$
- clicked URL $u$
- click_count $c$ of URL $u$ extracted from log.

O/P:
- Updated Index & Updated URL Frontier

While (true)
{
    Pick a clicked URL $u$ from the query log;  // start of IUFCM
    Search $u$ in Index repository;
    click_count(u) = click_count(u) + c;  //update the click_count(u) in Index
    If ((depth(u) > d) && (c > C))
    {
        Retrieve the forward linked URLs $F$ of $u$ from index;  // $F$ is the set of forward links of $u$
        If ($F \neq \emptyset$)
        {
            For (all $f \in F$)
            {
                Add the URL $f$ in the URL frontier;  // update URL Frontier
                depth (f) = depth (u) + 1;
                Initialize the Priority of $f$;  //prioritizer assigns the priority
                Crawl_bit (f) = 1;
            }
        } else continue;
    }
} //end While

Fig. 5.7 The Algorithm for Crawl_Index Updater

It can be observed from the algorithm that the Cl_Updater continuously picks up a user clicked URL with respect to a particular query. The clicked_count of the URL is supposed to
be a numerical value $c$ equal to or greater than 1, reason being that the URL may be clicked by multiple users by the time it is considered by the CI Updater. The value of $c$ for a particular URL is computed by counting its number of occurrences in the query log for distinct user IDs. The picked URL is searched for in the index for making the necessary modifications in its click_count. It undergoes an examination on its depth and click_count information based on which it may participate in the feedback calculation. If it does not comply with the examination process, it is rejected by the CI Updater for the present visit; otherwise the forward links of the examined URL are retrieved and communicated to the crawler through URL frontier for downloading.

5.2.3.4 Example of Feedback Communication
Consider the following instances depicted in Fig. 5.8 to have an understanding of the complete process carried out by the proposed feedback mechanism. In this scenario, the process of the CI Updater has been shown in 6 steps (see Fig 5.8). After the sixth step, updater starts working again from the first step and continuously updates the index and the URL frontier. The crawler, when accesses the URL frontier, considers the new inserted URLs based on their priorities and in turn fetches their pages and indexes them in the index. The indexed URLs may then appear in the results according to their content matching with the user query.

5.2.3.5 Advantages of Feedback Communication
Following are the benefits of the proposed mechanism:

1. By communicating the user feedback from query logs, crawlers can be made adaptive in nature for traversing the web in a normal fashion as well as according to user information needs that improves their scalability.

2. The pages, which are considered relevant according to user needs and were previously ignored by the search engine for indexing, can be made to appear in the search results, thus increasing the relevancy and quality of search results.

3. The mechanism works in an offline mode, thus does not affect the online query processing time of the search engine. Rather, it improves the search engine efficiency.
4. The index always contains the current and up to date information according to user information needs, which better serves the user queries resulting in more relevant pages in the search results.

5. The pages deep inside a website may also appear in the search results depending on its past tracking history stored in the index itself.

![Fig. 5.8 Example Illustration of Crawl_Index Updater](image)

The next section describes another pre-mining technique, developed in this work, towards relevant page retrieval by considering the semantics of user queries.

### 5.3 QUESEM: A QUERY SEMANTIC SEARCH SERVICE

The outcome of a web search typically depends on the submitted queries, but the effectiveness of queries cannot be guaranteed as they vary from user to user. Even, the same
user query may convey different interpretations or different information needs. So, most of the users get irrelevant results and the effort of reaching needed information becomes very high due to vague or ambiguous formation of their queries. Consider a user, who intends to search any one term out of “mouse”, “apple”, “cluster” or “tree” on a search engine. Most commonly, the returned results concern computer field only or the one which is most popular. In case, user did not find the required information, the modified query may be resubmitted and even then, it is not guaranteed that he will find the exact required pages. This process is very time consuming and irritating. Query refinement becomes handy in these situations.

The proposed work [123, 124] in this section addresses the following issues:

1. A term may have several synonyms, but due to the lack of their availability they are not considered while returning the search results to the user. The meanings are sometimes totally unrelated, for instance, how can “lead” be a verb meaning to go first and also the name of a heavy metal?
2. A term may have several different meanings in different contexts. One may be interested in a particular field while other contexts, which are not needed, may increase only the volume of search results for nothing good and just become a hurdle in finding the required pages. Thus, the problem of “Information Overkill” arises.
3. General search engines are unable to provide different descriptions of query terms to users so as to assist them in searching in the right direction.

Most of the information about the synonyms, contexts and descriptions (or definitions) exists in definition based or dictionary based sites, which can be utilized by the search engines to resolve the above said issues.

This work proposes the concept of QUESEM ('Query Semantic Search Service' pronounced /'Qu-sem/) [123, 124], a Meta search service layer over the keyword based indexing. It utilizes the online web resources to provide semantic and context-oriented web search to the users. The next section explains in detail of the proposed approach.
5.3.1 PROPOSED APPROACH TO SEMANTIC SEARCH

In an abstract form, the approach to be followed by the system can be shown diagrammatically as shown in Fig. 5.9. Given a topical query, QUESEM [123] first analyses it and thereafter consults the query terms with its local database (called Definition Repository). The relevant definitions also called sub-terms related to query are located wherein the definition describes a semantic description of the query in question. For every distinct definition of the query, it uses an existing keyword-based search engine to search for the pages on the WWW. In this sense, QUESEM acts as a Meta layer over the keyword based indexes maintained by the search engines. QUESEM displays the results in the form of clusters corresponding to each extracted definition.

![Fig. 5.9 Approach taken by QUESEM](image)

A Topical Crawler has been developed in this work. It downloads those Web sites, which specialize in definition-related or dictionary-related content. A Definition-Generator/Annotator with machine learning techniques has also been designed to automatically extract the relevant definitions from the crawled Web pages.

An example scenario of a common search is given below to better illustrate the proposed approach:

A User ‘U’ wants to gain knowledge about the term called “cluster”. He is totally new to this term and has no idea about this.
As shown in Fig. 5.10, instead of displaying the URLs matching the query keywords or their combinations as in traditional search engines, QUESEM displays the definitions (matched terms having distinct meanings) related to the term 'cluster'. It may also display the related term descriptions.

\[ \text{User Query} \]

\[ \begin{array}{c}
\text{Def. 1} \\
\text{Def. 2} \\
\text{Def. 3} \\
\text{Def. n}
\end{array} \]

\[ \begin{array}{c}
\text{A set of URLs related to Def. 1} \\
\text{A set of URLs related to Def. 2} \\
\text{A set of URLs related to Def. 3} \\
\text{A set of URLs related to Def. n}
\end{array} \]

\[ \text{Fig. 5.10 The Expected output of QUESEM} \]

'U' chooses the matched term which is of his interest, reads the description and if found interesting, further explores the related URLs.

The aim here is to solve the information overkill problem with the help of dictionary based sites. The terms that are found closer to the given search query on the sites such as yourdictionary.com, thesaurus.com and wordnetonline.org etc. are extracted by the definition generator module, annotated by annotator and stored in the Definition Repository for further use by the system. The next section describes the detailed system architecture and the functioning of various components involved over there.

5.3.2 SYSTEM ARCHITECTURE

The detailed system architecture [123] of QUESEM is shown in Fig. 5.11, where the dashed line represents the proposed meta-search service. In order to achieve the required task, architecture is divided into two major sub-systems as given below:

1. Definition Repository Generation
2. Definition based Search
The basic definitions and the working of these two subsystems are explained as follows:

**DEFINITION BASED SEARCH**
1. Analyze Query
2. Search Definition Repository
3. Consult Indexes of Keyword based Search engine

**DEFINITION REPOSITORY GENERATION**
1. Topical Crawler
2. Definition Generator
3. Term Annotator

---

**Fig. 5.11 High-Level System Architecture of QUESEM**

5.3.2.1 Basic Definitions
Some definitions are formulated here, which are related to the proposed search system.

- **Definition of “Definition”:**
  A *definition* is a phrase or set of symbols that define the meaning of a term or similar kind of things. A term may have many different senses or meanings in different contexts. For each such specific sense, a definition is a set of words that defines it. Its existence in a particular field defines a terminology of that field.

- **Definition of “Definition Repository”:**
  A database for storing terms, their related definitions and a group of programs, which provide means to collect and access the data. Its schema must contain at least two relations as shown in Fig. 5.12, and may further contain optional fields or relations to facilitate term hierarchy. It collects and manages definitions, which are used by
Definition based search sub-system to expand the initial query into multiple sub-queries or definitions.

**TERM**

<table>
<thead>
<tr>
<th>Term_ID</th>
<th>Term_Title</th>
<th>Term_Description</th>
</tr>
</thead>
</table>

**DEFINITION ANNOTATION**

<table>
<thead>
<tr>
<th>Def_ID</th>
<th>Def_Title</th>
<th>Def_Description</th>
<th>Term_ID</th>
</tr>
</thead>
</table>

Fig. 5.12 Schema of Definition Repository

The schema is very simple and is made to handle only single level of hierarchy. The fields *Term_ID* and *Def_ID* with a solid underline represent the primary key of the relation. Table 5.6 gives description of various fields in the Definition Repository.

**Table 5.6 Description of Definition Repository**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term_ID</td>
<td>An attribute that gives a unique identity to the term under consideration.</td>
</tr>
<tr>
<td>Term_Title</td>
<td>It contains the actual query terms that are entered by the user.</td>
</tr>
<tr>
<td>Term_Description</td>
<td>It contains a limited length snippet to have a small description of the term.</td>
</tr>
<tr>
<td>Def_ID</td>
<td>It represents the unique identifiers of the semantic Definitions (sub-queries) extracted for each query term.</td>
</tr>
<tr>
<td>Def_Title</td>
<td>This column specifies the definitions for query terms that are found relevant in the dictionary-based sites after parsing.</td>
</tr>
<tr>
<td>Def_Description</td>
<td>Small description of each definition.</td>
</tr>
</tbody>
</table>

Fig. 5.13 gives the state of the repository for the term “Cluster” and its various definitions viz. “Cluster Headache”, “Cluster Bomb” etc.

**TERM**

| 1 | Cluster | A group of similar objects.......... |

**DEFINITION ANNOTATION**

| 1.1 | Cluster headache | It is less common than migraine headache..... | 1 |
| 1.2 | Cluster bomb | Cool crust band..... | 1 |
| 1.3 | Cluster analysis | Cluster analysis classifies a set of observations......... | 1 |
| 1.4 | Cluster bean | Drought tolerant herb..... | 1 |
| 1.5 | Cluster (computer) | It's a technique to categorize ... | 1 |

Fig. 5.13 Example Illustration of Definition Repository
- **Definition of “Definition based Search”:**

It is an extension to the traditional Web search as it first discovers the implied definitions \( \{d_1, d_2, ..., d_n\} \) (if these exist) of the initial query (say \( q \)) using definition repository and then performs traditional Web search on each definition \( d_i \), \( i = 1, 2, ..., n \). Let \( r_i \) denote the obtained URL list by searching on \( d_i \), then \( \langle r_1, r_2, ..., r_n \rangle \) represent the result of the Query Semantic search on the initial query \( q \).

5.3.2.2 Definition Repository Generation

Definition Repository is maintained by a series of steps as illustrated in Fig. 5.14. Various inputs, outputs and the components involved in the process with their mode of working are explained below in detail.

1. **Information Resource: Dictionary Based Sites**

For automatically populating the repository with high-quality definitions, the input resource is the dictionary based sites, which are a rich store of semantic definitions of terms explicitly published on the Web. In general, the web pages, which possibly contain this type of information, are very sparsely distributed on the Web. In order to efficiently extract more sub terms related to the query terms, while downloading relatively fewer pages, a topical crawler is needed to crawl only those Web sites, which are specialized in dictionary-related content.

![Fig. 5.14 Definition Repository Generation Process](image-url)
2. **Topical Crawler**

A typical crawler for a generic web search engine recursively traverses through hyperlinks to explore the undiscovered portion of the Web. The basic idea behind topical crawling is to estimate the relevance of undiscovered documents by the relevance of fetched documents, which directly or indirectly link to the undiscovered ones [116]. To start the crawl process, some topic of interest such as “dictionary” or related content is needed. Topical crawlers maintain priority queues, where related documents have the highest priority to be downloaded under the constraints of limited computing resources. Thus, topical crawlers traverse the topic-related portion of the Web.

A publicly available Web search service (Google SOAP API) can be used to filter dictionary-related Web sites by searching dictionary-related keywords (e.g. dictionary, thesaurus, synonyms, answers, definition etc.) in the title of the home page of a web site. For the present work, only one site yourdictionary.com is considered, but a number of sites can be statically used to do this task.

The topical crawler outputs a set $S$ of sites related to dictionary based content, which is stored in a local repository to be referred further by definition generator.

3. **Local-Site Search by Definition Generator**

When the user first time submits a query to QUESEM, its keywords would be passed to definition generator module. This component consults the set $S$ of dictionary-based sites and passes these query terms over their interfaces to perform a local-site search. After that, the resultant set of pages related to query terms are retrieved by this component and placed in a set $D$. These pages are further parsed to extract the linked pages related to query terms. The extracted pages are also kept in $D$.

The algorithm for local-site searching is shown in Fig. 5.15. The set $D$ of documents $d_i$ for $i=1 \ldots n$ is the “returned set of pages” downloaded from dictionary-based sites. These sites contain the relevant information regarding the semantics or context of query terms. The set $D$ is used for definition generation and annotation.
Algorithm: Local Site Searching (Initial query, S )

\( I/P = \) Initial query \( q \) and Set \( S \) of Sites stored in a repository

\( O/P = \) Unstructured document set \( D \) containing documents that are response pages against the initial query \( q \).

\{ 
For (every site \( si \in S \)) // perform the local-site search

\{ 
    Step1: fetch the homepage of \( si \)
    Step2: find the local-site search form
    Step3: Submit the query terms \( (q) \)
    Step4: Fetch the response pages in local Repository \( D \)
    Step5: For (every response Page)
    \{ 
        \{ 
            Parse the Page;
            Fetch all result links;
            If synonyms link exist
            \{ 
                Fetch synonyms page and place in \( D \)
            \}
        \}
    \}
\}
Return the fetched page set \( D \)
\}

Fig. 5.15 Algorithm for Local Site Search for Query Terms

4. Definition Generation & Annotation

As the response pages are the result of the dictionary based sites, it is assumed that the pages will contain the direct thesaurus and synonyms of the query terms. The approach to be followed to find semantic definitions is given below:

"Extract the prefix and suffix tokens of query terms from pages \( d \), annotate them with query terms and store the result in Definition Repository."

The Definition Generator simply extracts the prefix and postfix present consecutively in combination with the query terms from the pages belonging to \( D \), while Definition Annotator combines them suitably with the query terms to result in proper annotations of definitions. The algorithm for definition generation and annotation is shown in Fig. 5.16.

The prefixes and postfixes of the query terms play an important role in semantic definition generation. It is assumed that most of the definition based sites manage the data in the thesaurus and synonyms form. This assumption may restrict the number of definitions that could be found, but for a precise search, this assumption may be much more relevant.
because, in dictionary based sites, the most relevant sub terms can be found nearby to the basic term. Therefore, prefix and suffix represent the synonyms/contexts, which have some how a meaning equivalent to query term.

Algorithm: Definition Generator Annotator ($D$)

**I/P** Unstructured document set $D$ of pages containing semantics of query terms.

**O/P** Term and their associated Annotated Definitions

1. $term_{title} = q$
2. For (every $d_i$ in $D$)
   1. Set before = Consecutive tokens occurring before $q$ in $d_i$
   2. Set after = Consecutive tokens occurring after $q$ in $d_i$
3. For (every token $t_i$ in Set before)
   1. $t_i.Title = t_i + q$  // $t_i.Title$ is the title of definition
4. For (every $t_i$ in Set after)
   1. $t_i.Title = q + t_i$
5. Place term titles, definition titles $t_i.Title$ in the Definition Repository

**Fig. 5.16 Algorithm for Definition Generation/Annotation**

To extract query semantics from the pages $d_i$, which are in the form of prefix and suffix, parsing is required wherein the parser tokenizes the response pages. The query terms are kept in track to find their relevant prefixes or suffixes, which are in turn are annotated with the query terms by the Definition Annotator to generate the definition titles. For example, as was shown in Fig. 5.13. “bean”, “bomb”, “headache”, “analysis”, “computer”, “controller” etc. all represent the prefixes or suffixes related to the term “cluster”. The figure also shows the definition titles after annotations e.g. “cluster headache” is the annotated definition.

5. Populating the Definition Repository

Initially definition repository is empty and it is populated as soon as the user queries are submitted to QUESEM. The definitions generated by the Definition Generator & Annotator with respect to each new query are populated into the Definition Repository. It may be possible that the definition repository may contain some inadequate definitions, but as user explores a cluster of URLs relative to his interest, some abrupt clusters may not affect the search.
5.3.2.4 The Definition based Search

When a user submits a keyword based query to QUESEM, the keywords are passed to “Definition Repository Generation”, the subsystem that builds the definition Repository. The keywords are also passed to the “Definition based Search” subsystem that responds to the user in the form of cluster of URLs. Contrary to the traditional keyword based search, semantic or definition based search requires slightly complex query processing. Various modules which are used in definition based query processing are given below and are outlined in Fig. 5.17.

1. Query Analyzer
2. Definition Searcher
3. Query Transformer and Processor

![Definition Based Search Subsystem](image)

Fig. 5.17 Definition Based Search Subsystem

Query processing involves analysis of user query to perform extended search for it. Given a query, a Query Analyzer first decides whether to perform definition based search or a
traditional search for the query? The QUESEM works only for topical queries. The topical user queries, which may be solved by this system, can observe two characteristics:

- They can have multiple meanings.
- They can have a number of synonyms.

After the analysis, the next step is to find the related definitions by searching the definition repository. This is performed by the Definition Searcher module. Finally, the initial query is transformed into a sequence of sub-queries or definitions by the Query Transformer & Processor. Obtained sub-queries are then sent individually to the traditional keyword-based search engine (like Google) to find the matched pages. Now the Query Processor represents the results obtained by different sub-queries in the form of clusters to the user.

The functioning of different components is described briefly as follows:

1. **Query Analyzer**
   Query analyzer is responsible to check whether definition based search is applicable to the query or not? Analyzer first examines the query to decide whether a query is topical or not? A topical is the query, which is generally framed by simple keywords. For example “mouse”, “waiter”, “data mining”, “HCL laptop” etc. are topical queries, while “how to drive a car”, “when monsoon will come” etc. are not topical queries.

   For analyzing the queries, openNLP functions [137] are used by the system. A query which contains a combination of <predicate, object> is considered a goal based query, otherwise queries containing either <predicate> or <object> is considered a topical query, for which QUESEM gives better results. It is assumed that topical search query either have a verb phrase or a noun phrase. The queries, which are not topical, are made to be searched by using the traditional search system.

2. **Definition Searcher**
   The job of Definition Searcher is to check whether the topical query already exists in the system’s Definition Repository. If it is there, that means some user has already queried it and hence the definitions are present in the database. Now, these definitions are simply
extracted from the definition repository. On the contrary, if definition repository doesn’t return any set or say return NULL then it is the functional responsibility of the topical crawler of Definition Generation Subsystem to become active and perform all the tasks necessary to get the new definitions matching the new query and expand the definition repository.

In searching the definitions, literal term matching is done. It is simple keyword-to-keyword matching which return a set of definitions corresponding to the term. As the user queries tend to contain common words, punctuation marks, stop words, case sensitivity etc. handling all these aspects comes in preprocessing the query which is sent to the definition searcher. The related definitions are passed to Query Transformer and Processor to process the queries.

3. Query Transformer & Processor
Query transformer is responsible for making the retrieved definitions as a sequence of well-defined queries so that they could be searched individually as independent queries. The original query and the matched set of definitions are combined to form a new set of sub-queries. Query Processor searches these sub-queries individually with the help of a traditional search engine and represents the resultant URLs in the form of clusters with each cluster label being the corresponding definition Title.

5.3.3 PERFORMANCE EVALUATION

QUESEM was implemented in asp.net 2.0, C# and HTML with MS SQL Server 2005 at the back end to support definition repository. For the experimental purposes, only yourdictionary.com has been used for the current scenario. A group of 25 users from different domains were asked to search on QUESEM and other keyword based search engines like Google, Yahoo etc. Assisted by the information of definitions related to semantics of query terms, QUESEM was able to understand users’ queries in a better way to perform more meaningful searches. The net performance of QUESEM in terms of quality of search results and reduced navigation time is observed (see Chapter 7) to be higher than normal traditional keyword based web search.

123
5.4 ICADG: AN APPROACH FOR OPTIMIZATION OF XML INDEX

Due to the absence of structured data on the Web, it becomes very difficult for information retrieval tools to fully utilize the Web information. As a solution to this problem, XML (Extensible Markup Language) comes into play, which provides structural information in web pages. XML [117] goes one step beyond HTML and describes the document content in terms of what data is being described. It is “extensible” because the tags are unlimited and self-defining. XML is really a Meta language for describing mark-up languages. It provides facility to define tags and the structural relationships between them. Since there is no predefined tag set, there cannot be any preconceived semantics. All of the semantics of an XML document will either be defined by the applications that process them or by style sheets. A sample XML file representing the catalog of a Bookstore is given in Fig. 5.18, from where it can be observed that the page conveys some defined tree like structure instead of being unstructured.

```xml
<?xml version="1.0"?>
<catalog>
  <book id="bk01">
    <author>G.a.-;.bardella</author>
    <title>KL Developer's Guice</title>
    <genre>API</genre>
    <publish_date>2000-11-01</publish_date>
    <price>4.55</price>
    <description>The former architect battles corporate zombies, an evil sorceress, and her own childhood to become queen of the world.</description>
  </book>
  <book id="bk02">
    <author>Kirki</author>
    <title>Ghost Rider</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-11-16</publish_date>
    <description>A former architect battles corporate zombies, an evil sorceress, and her own childhood to become queen of the world.</description>
  </book>
</catalog>
```

Fig. 5.18 Sample XML File for Bookstore

Due to the flexibility of XML, it is rapidly emerging as a standard for exchanging and querying documents on the web required for the next generation web applications including electronic commerce and intelligent web searching. It is now common for XML to be used in interchanging data over the Internet. But without efficient indexes, query processing can be quite inefficient due to an exhaustive traversal on XML data.
In this section, Incremental Content-Aware DataGuide (ICADG), an indexing approach for XML pages is being proposed that uses “frequently used query paths” extracted from historical query logs to improve performance in querying XML pages. ICADG is an improved content-centric approach over the Content-Aware DataGuide (CADG) [119], which is an existing indexing technique for XML databases. Here, the index can also be updated incrementally according to the changes in query workload and thus, the overhead of reconstruction can be minimized. This indexing technique proves to be efficient as partial matching queries can be executed efficiently and users can get more relevant documents in the results.

The subsequent sections describe the existing approach of indexing i.e. CADG and proposed approach ICADG.

5.4.1 DATAGUIDE AND CONTENT-AWARE DATAGUIDE

Path traversal [118] plays an important role in XML query processing. There are three approaches of path traversal, namely top-down, bottom-up and hybrid approach. The top-down approach starts traversal from the root of the tree down level by level to find the matching nodes, while the bottom-up approach starts traversal from the bottom of the tree up level by level to find matching nodes. The hybrid approach, however, combines both the top-down and bottom-up approaches and stops when a convergence is found. Consider the following query:

\[ q1: \text{book}[\text{title} = "\text{Java Programming}\)]\]

Using the top-down approach to process query \( q1 \), all downward paths starting from any \( \text{book} \) element should be traversed to find out whether there exists any immediate \( \text{title} \) element. If there exists, the downward path will be traversed further to the leaf node to find the book with the title ‘Java programming’. This process will be repeated for the rest of the \( \text{book} \) elements in the XML database to determine all possible paths.

The worst case is: if the element \( \text{book} \) resides at the root of the tree, the entire tree needs to be traversed. Furthermore, these traversal methods are inefficient to answer queries in ancestor-descendant relationships such as \( \text{book//section} \). For instance, using the top-down approach, all downward paths starting from any \( \text{book} \) element is traversed down to its child
one by one to check whether there exists any \textit{section} element. This is done until every child reaches the leaf node. For the next set of matches, it needs to backtrack to its previously visited \textit{book} element node and start the search again from the next child. Thus, this is certainly very exhaustive and inefficient. To overcome performance degradation due to excessive traversal limitations, index structures have been introduced, which certainly reduce the portion of the XML tree to be scanned during query processing. The following are two important indexing techniques:

1. DataGuide [118]
2. Content-aware DataGuide (CADG) [119]

\textbf{5.4.1.1 DataGuide}

A DataGuide [118] is a compact representation of the document tree, in which all distinct label paths appear exactly once. Consider the example of XML document tree of book database as shown in Fig. 5.19(a). Document tree contains labels related to book database along with corresponding keywords to the document nodes are shown. The corresponding DataGuide is shown in Fig. 5.19(b). Multiple instances of the same document label path, like book/chapter/section in Fig. 5.19(a) collapse to form a single index label path as shown in Fig. 5.19(b). So the resulting index tree, which serves as a path index, is usually much smaller than the document tree. Hence it is supposed to be held in main memory even for large document collections.

\begin{figure}[h]
\centering
\begin{tabular}{c c}
\hspace{0.5cm} (a) & \hspace{0.5cm} (b) \\
\end{tabular}
\caption{Fig. 5.19 (a) Document Tree of Book.xml (b) Corresponding Index Tree}
\end{figure}
Index tree only allows to find out the existence of a given label path, but not its position in the document tree. So, every index node needs to be annotated with the ids of those document nodes it represents. For example, the index node 
 in Fig. 5.19(b) with the label path book/chapter/section points to the document nodes &4 and &7, as they are accessible through this path in the document tree. The annotations of all index nodes are stored on disk in an annotation table shown in Table 5.7. The annotation table represents mapping
\[ d_{ga}: i \rightarrow D_i \]
where \( i \) is an index node and \( D_i \) is the set of document nodes reached by \( i \)'s label path.

The index tree and the annotation table encode nearly all-structural information, which is present in the document collection. Only parent/child relations between document nodes cannot be reconstructed from the DataGuide, due to the merging of multiple document paths in a single index path. For example, from Fig. 5.19(b) and Table 5.7, one cannot tell whether &8 is a child of &4 or &7, which are both referenced by the parent of &8's index node, &5.

<table>
<thead>
<tr>
<th>#0</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;2</td>
<td>&amp;1</td>
<td>&amp;2</td>
<td>&amp;3</td>
<td>&amp;4, &amp;7</td>
<td>&amp;5, &amp;8</td>
<td>&amp;6</td>
</tr>
</tbody>
</table>

Table 5.7 Annotation Table

<table>
<thead>
<tr>
<th>“index”</th>
<th>“survey”</th>
<th>“XML”</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;2, &amp;5, &amp;6, &amp;8</td>
<td>&amp;2</td>
<td>&amp;8</td>
</tr>
</tbody>
</table>

Table 5.8 Content Table

One more data structure known as content table is created in this technique, in which keywords occurring in the document tree are indexed. It is stored on disk like the annotation table. The content table is an inverted list mapping a keyword to the set of document nodes in which it occurs as shown in Table 5.8. It implements the mapping:
\[ d_{gc}: k \rightarrow D_k \]
where \( k \) is a keyword and \( D_k \) is the set of document nodes, which contain an occurrence of \( k \).

Query processing with DataGuide is divided into the following four retrieval phases.

1. **Path matching**: The query paths are matched separately against the index tree.
2. **Occurrence fetching:** Annotations of the index nodes found in phase 1 are fetched from the annotation table. Query keywords are accessed from the content table.

3. **Content/structure join:** For each query path, the sets of annotations and keyword occurrences are intersected.

4. **Path join:** The results of all query paths are joined to form hits matching the entire query tree.

Consider an example, suppose the query is `/book/* ["XML"]`. It selects all document nodes below tree root labeled `book`, which contain an occurrence of the keyword "XML". In phase 1, the query path `/book/*` is searched in the index tree shown in Fig. 5.19(b). All nodes in the index tree except the root node `/0` will be selected. In phase 2, the annotations of all index nodes from the previous retrieval phase are fetched from the annotation table. According to the index node ids #1 to #6 in the table, six annotation sets `{&1}`, `{&2}`, `{&3}`, `{&4; &7}`, `{&5; &8}` and `{&6}` respectively are retrieved. By searching for the query keyword "XML" in the content table, document node `{&8}` is retrieved.

In phase 3, the content/structure join, each annotation set retrieved during the previous phase is intersected with the occurrence set for "XML" to find out which of the document nodes with a matching label path contain an occurrence of the query keyword. Only node #5 references a document node which meets both the structural and textual selection criteria of the given query path. The document node in the intersection `{&5; &8} \cap \{&8\} = \{&8\}` is returned as the query output. If there were more paths in the query tree, &8’s ancestors would need to be retrieved in order to join the hits of all query paths (in phase 4) as described above.

The following disadvantages have been found in the above indexing technique:

- Each annotation set retrieved during the phase 2 has to be intersected with the occurrence set. This will make the path-matching step unnecessarily complex.
- It also causes needless disk accesses in phase 2.

The reason for above disadvantages is that structural and textual selection criteria are handled separately in DataGuide. Extension of the DataGuide, the **Content-Aware DataGuide**, resolves the mentioned disadvantages.
5.4.1.2 Content-Aware DataGuide (CADG)

The Content-Aware DataGuide (CADG) [119] combines structure and content matching from the very beginning of the retrieval process. The objective of this technique is to integrate content matching with both path matching and annotation fetching. It saves an explicit content/structure join phase as in DataGuide. This content-aware navigation reduces the number of paths to be visited during phase 1 in DataGuide. In CADG, two table look-ups in phase 2 of DataGuide i.e. annotation and keyword occurrence fetching are integrated within a single content-aware annotation-fetching step, which again reduces the number of disk accesses by up to 50%. This also avoids the intersection of possibly large sets of document node IDs at query time.

In this approach, multiple keyword-specific index sub-trees are created. Consider again the example XML document tree shown in Fig. 5.19(a). Fig. 5.20 shows four index sub-trees, each of which indexes only the paths in the document tree where a specific keyword occurs. Document nodes without textual content are associated with the empty symbol $\varepsilon$.

![Fig. 5.20 Content-centric Index Sub-trees](image-url)
For example, the "XML" index subtree contains a single document path 
/book/chapter/section/para, which is the only one leading to an occurrence of the keyword "XML" in Fig. 5.19(a). The annotation-fetching step becomes content-aware when annotation table is partitioned into keyword-specific subtables, which is equivalent to precomputing the content/structure join from retrieval phase 3 in DataGuide.

In this approach, Content/annotation table is constructed which represents a mapping:

\[ \text{cadg}_{cc}: (k; i) \leftrightarrow D_{k,i} \]

where \( k \) is a keyword, \( i \) is an index node ID, and \( D_{k,i} \) is the set of document nodes where \( k \) occurs and which are referenced by \( i \) as shown in Table 5.9.

<table>
<thead>
<tr>
<th>&quot;index&quot;</th>
<th>&quot;survey&quot;</th>
<th>&quot;xml&quot;</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2</td>
<td>#5</td>
<td>#6</td>
<td>#2</td>
</tr>
<tr>
<td>&amp;2</td>
<td>&amp;5;&amp;8</td>
<td>&amp;6</td>
<td>&amp;2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;4;&amp;7</td>
</tr>
</tbody>
</table>

Consider a user query /book// * ["XML"] in CADG, the "XML" subtree is selected and used during path matching. This narrows down the search space to the path reaching index node /#5. Occurrence and annotation fetching is accomplished by a single look-up in the content/annotation table, where the entry for /#5 and "XML" is selected and returned as query result. Note that no explicit content/structure join is required at query time.

This technique also has following drawbacks given below:

- The content/annotation table takes up more space on disk than the original DataGuide's content and annotation tables together. In Table 5.9, there are multiple columns referring to the index nodes #2 or #5. This redundancy is due to the Cartesian product in the join of the content and annotation tables, which increases with the number keywords occurring under a variety of different label paths.

Another important drawback of the content-centric approach is that, content/annotation table and index subtrees reside on secondary storage. The right index subtree for a query keyword has to be loaded from secondary storage at the time of query; so, an additional
disk access is needed. At the time of processing multiple keyword queries, multiple index subtrees need to be fetched from disk, which is clearly prohibitive at query time.

In order to resolve the drawbacks of existing CADG, a modified approach to CADG i.e. *Incremental CADG (ICADG)* is developed in the next section.

### 5.4.2 INCREMENTAL CONTENT-AWARE DATAGUIDE (ICADG)

In this work, an improved content-centric indexing approach to Content-Aware DataGuide [119] called *Incremental Content-Aware DataGuide (ICADG)* [125] is being proposed that has the property that it can be updated incrementally according to the changes in query workloads. It uses frequently used paths to improve the query performance. It also processes partial matching queries efficiently by using the *hash tree* data structure. Hash tree is used here to represent incoming label paths to nodes of index subtree. In this approach, like in CADG, index subtree and hash tree are stored in secondary storage for each keyword present in the user queries. When a user enters a query for a keyword then the respective index subtree and hash tree are loaded in main memory. The modified indexing approach automatically updates index subtrees and the corresponding tables with respect to the frequently used queries.

In the offline mode, a *Path Extraction* algorithm is used for extracting the frequently used paths according to the query workload. Algorithm is basically divided into two phases:

- First phase counts frequency of paths.
- Second phase is the pruning phase, which removes the hash entry whose frequency is less than a specified threshold.

After pruning of hash tree with frequently used paths, the index subtree and some fields in the nodes of hash tree that locate the corresponding nodes in index tree will be updated, that improves query performance and subsequently reduces the processing time with respect to each query. The detail of its construction and query processing is given in the next subsections.
5.4.2.1 Architecture of ICADG

In the proposed architecture of Incremental CADG, as the query workload is collected, the frequently used paths are computed and used to update the current index subtree dynamically. Steps are repeated whenever query workload changes as shown in Fig. 5.21.

Fig. 5.21 Architectural Flow in ICADG

Steps involved in Fig. 5.21 are given below:

1. User query is collected.
2. Appropriate index subtree and hash tree are selected from secondary storage according to the keywords present in user query.
3. Corresponding Content/annotation table is selected from secondary storage.
4. Document nodes are returned from content/annotation table as output.
5. Query input is stored as query workload.
6. Query workload is used in Frequently used Path Extraction Module.
7. Index subtree and hash tree are collected and used as input for Frequently used Path Extraction Module.
8. Frequently used paths are extracted according to query workload using hash tree.
9. The update module updates the index subtree and hash tree according to initial document tree.
10. Content/annotation table is updated according to updated index subtree.
11. Updated index subtree and hash tree become new index subtree and hash tree for next query input.
12. Updated content/annotation table becomes new content/annotation table.

5.4.2.2 Approach of ICADG

In the proposed approach, one more data structure, the hash tree is build over a keyword specific view of data. The appropriate index subtree and hash tree can be chosen only at the time of query after the desired keywords have been specified. Hash tree is then updated using the query workload. The approach constitutes the following three steps:

1. Building the hash tree.
2. Extraction of frequently used paths according to query workload.
3. Updating the data structures with respect to frequently used path.

These steps are explained below in detail:

1. **Building the Hash Tree**

A Hash tree is built for every keyword in the document tree. It consists of five fields: label, count, xnode, new and next. The *label* field keeps the key value for the entry. The *count* field describes the frequency of label path represented by the entry. The *xnode* field points to a node in index subtree whose incoming label path is represented by the entry. The *new* field checks a newly created entry in a node of hash tree. The *next* field points to another node in
hash tree. Consider an example Index subtree in which there are four labels as shown in Fig. 5.22(a). The hash tree corresponding to the index subtree is shown in Fig. 5.22(b).

In Fig. 5.22(b), remainder D in hash tree is the set of edges whose end nodes are reachable by traversing a label path $D$ but not $B.D$. Hash tree is very efficient for partial path queries because if the user submits a query for the path $B.D$, the query processor looks up the hash tree with $B.D$ in the reverse order.

2. **Extraction of frequently used paths**

Sequential pattern mining algorithms are used to extract frequent paths from the path expressions appearing in the query workload stored in logs. An algorithm is used here which simply counts all sequential subsequences that appear in the query workload by one scan and in the second phase hash tree is used to keep the change of the workload. Figure 5.23(a) and (b) show how the frequently used paths are extracted.

Suppose $Q_{workload} = \{C.D, B, C.D\}$ and according to the query workload the state of hash tree after frequency count is shown in Fig. 5.23(a) and Suppose frequency threshold called $\text{minsup} = 2$, so the labels in the hash tree whose count are below than the $\text{minsup}$ are pruned as shown in Fig. 5.23(b). The Frequently used path Extraction Mechanism is described in the following steps:

- Suppose that the required path set was $\{A, B, C, D, B.D\}$ then the current state of hash tree is represented in Fig. 5.22(b). Let the query workload is $\{C.D, B, C.D\}$. 

![Image of hash tree and table](image-url)
- Frequency of each label path is counted which appeared in query workload and the count is stored in hash tree. Remainder entry is not used for counting. Fig. 5.23(a) shows the status of hash tree after the frequency count.

- In the pruning phase, pruning of the label paths is done whose frequency is less than minsup. Assume minsup is 2, the label paths whose frequency is less than 2 are removed. The status of hash tree after pruning is shown in Fig. 5.23(b). The label path B.D is pruned. A label path of size 1 is always in the required path set. So label paths A and B still remain in the required set. The xnode fields, which are not valid any more by the change of frequently used paths, are set to NULL. Remainder entry is also set to NULL to update it later.

![Figure 5.23 State of Hash Tree (a) After Frequency Count (b) After Pruning](image-url)
3. Updating with frequently used paths

The idea of update is to first traverse the nodes in index subtree and then update not only the structure of index subtree with frequent paths but also the \textit{xnode} field of entries in hash tree. While visiting a node in index subtree, the entry of the maximum suffix path in hash tree from the root to the currently visiting node is checked in index subtree. An algorithm for update module is described in Fig. 5.24.

\begin{verbatim}
Algorithm: Index Updation()
Input: Document tree, Initial index subtree and pruned hash tree.
Output: Updated index subtree and hash tree
1. For ( \forall nodes \in Document tree)
   xnode.visited=FALSE
2. For first label \textsl{l} in Eset
   xnode= hash(l);
   xnode.visited= TRUE ;
   newpath= l;
3. For (each \textsl{e} that is an outgoing edge of \textsl{xnode})
   newpath= concatenate (newpath, e.label);
   xchild= hash(newpath);
   // For each ending vertex in the outgoing edges of the visiting node in index subtree, \textsl{xchild} is a pointer that represents index subtree with the maximal suffix in hash tree of the label path to the visiting node by calling hash function.
4. If (xchild = NULL)
   xchild = newXNode(); // newXNode function will create a new node and assign to \textsl{xchild}.
   xchild.visited=TRUE;
   Repeat steps 4 and 5 for \textsl{xchild} as \textsl{xnode};
Else
   xchild.visited=TRUE;
   Repeat steps 4 and 5 for \textsl{xchild} as \textsl{xnode};
5. Update content/annotation table according to updated index subtree
\end{verbatim}

\textbf{Fig. 5.24 Algorithm for Update Module}

When the algorithm given above is invoked, then it checks for all labels in the document tree. For each label in the document tree, entries in the updated hash tree and index tree are matched. Index tree is updated according to the updated hash tree. First edge is with label A and its ending node is #0 in the index tree. Now the entry in hash tree for the path A is checked. The xnode field of the entry returned by hash tree points the node #0. This is correct and nothing should be changed. Then again update algorithm is called recursively.

Consider the original document tree shown in Fig. 5.25. The index tree and pruned hash tree are shown in Fig. 5.26(a) and (b) respectively.
Now, the pruned Hash tree and Index subtree are updated using the algorithm shown in Fig. 5.23. Snapshots of Hash tree and Index subtree, while updating, are shown in Fig. 5.27(a) and (b) respectively.
After updating the Hash tree the *xnode* for the label path *C.D* is #6 and for remainder entry, the *xnode* is #5 as shown in Fig. 5.28(a). The updated Index subtree is shown in Fig. 5.28(b).

<table>
<thead>
<tr>
<th>label</th>
<th>count</th>
<th>xnode</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Hash Tree Table](image)

![Index Tree](image)

**Fig. 5.28 (a) Hash Tree after Update (b) Index Tree after Update**

There are three outgoing edges from the node #0. First consider the node #1 in the index tree and check in the hash tree for the path *A.B*. The entry points to the node #1. This entry points to the correct node. Outgoing edge from node #1 is node #2 with the label *D* in the index tree. For the input of *A.B.D*, hash tree returns NULL which is the *xnode* field of the entry for *remainder.D* and should point to the node in index tree representing all label paths ending with *D* except *C.D*. Thus, a new node #5 is created for *remainder.D* and change an outgoing edge of #1 with label *D* to point out node #5 as shown in Fig. 5.27(b).

Now there is no outgoing edge from #2. Another outgoing edge from node #0 is node #4, after updating this also points to node #5 that represents all label paths ending with *D* except *C.D*. There is no outgoing edge from node #4. Next outgoing edge from node #0 points to node #3 for the label path *A.C*. hash tree also returns entry for node #3. This is correct entry. Outgoing edge from node #3 is node #4 but hash tree returns NULL for the label path *A.C.D*. So, a new node #6 is created for the label path *A.C.D* as shown in Fig. 5.27(b). Finally the updated index tree and hash tree are shown in Fig. 5.28(a) and (b).
5.4.2.3 Example illustrating the outcome of Approach

Let us take the example of the document tree as was shown in Fig. 5.19(a) to illustrate the results after including frequently used paths updating algorithm in the existing CADG. Index subtrees and content/annotation table were shown in Fig. 5.20 and Table 5.9 respectively.

Hash tree for each index subtree is also stored in secondary storage. Suppose the user gives a query /book[/ Fig[index] ] and /section/para[“index”]]. So index subtree and its respective hash tree are selected from the secondary storage. Hash tree for the index subtree with keyword “index” is shown below in Fig. 5.29.

<table>
<thead>
<tr>
<th>label</th>
<th>count</th>
<th>xmode</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td></td>
<td></td>
<td>n0</td>
</tr>
<tr>
<td>preface</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chapter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>section</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>figure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5.29 Hash Tree for “index” keyword

Hash tree of Fig. 5.29 is pruned according to the Qworkload = /book[/ Fig[index] ] and /section/para[“index”]]. The hash tree after counting the frequencies of each label path is shown in Fig. 5.30. Suppose minsup=2, therefore, label paths with count less than minSup are pruned as shown in Fig. 5.31. This dictates the status of the hash tree before update.

<table>
<thead>
<tr>
<th>label</th>
<th>count</th>
<th>xmode</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td>1</td>
<td>2</td>
<td>n0</td>
</tr>
<tr>
<td>preface</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>chapter</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>section</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>figure</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5.30 Hash Tree after frequency count
Now the index subtree and hash tree will be updated according to frequently used paths in the query workload. Fig. 5.32 shows the status of index subtree and hash tree before the update process. Fig. 5.33 shows status while updating, finally updated trees are shown in Fig. 5.34.
According to the updated index subtree, content/annotation table is also updated as shown in Table 5.10.

Earlier, in CADG, for a user query for keyword “index”, three nodes #2, #5 and #6 were selected from the content/annotation table of Table 5.9. But after updating the index subtree, only two nodes #7 and #6 will be selected and returned as query results. In the Updated Index subtree for “index” keyword, now only two paths are available for the keyword “index”, which are frequently used paths. User gets documents according to frequently used paths in the query workload. For the next query workload, the index subtrees is updated again. This updation is done at periodic intervals.

![Index Tree](image)

**Table 5.10 Updated Content-Centric Content/annotation Table**

<table>
<thead>
<tr>
<th>“index”</th>
<th>“survey”</th>
<th>“xml”</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>#7 &amp; 2</td>
<td>#6 &amp; 6</td>
<td>#2 &amp; 2</td>
<td>#5 &amp; 8</td>
</tr>
</tbody>
</table>
| #0 & 0  | #1 & 1   | #3 & 3 | #4 & 4,

**5.4.2.4 Advantages of proposed ICADG**

Proposed approach of XML Indexing has the following advantages:

1. **Efficient processing of Partial Matching queries**: In this proposed approach, with the use of hash tree data structure, partial-matching queries can be processed efficiently.

2. **Relevant Results**: Cost of query processing is improved by taking into account the frequently used paths from query workload. Thus, only relevant results/paths are provided to the user.
3. **Incremental update**: Index subtree and hash tree are updated incrementally according to the frequently used paths in query workload and thus, the overhead of construction is minimized.

However, the proposed approach has the disadvantage of **Increased Space complexity**. Faster query processing is provided but storage consumption increases due to hash trees.

### 5.5 SUMMARY

The proposed Pre-Mining Techniques are summarized in Table 5.11. It can be observed from the table that each pre-mining technique optimizes the work of different tasks performed by a search engine. The expert analyst can adopt any one or a combination of techniques to optimize the working of search engine so as to return relevant results to the user.

<table>
<thead>
<tr>
<th>Techniques Parameters</th>
<th>CI Updater</th>
<th>QUESEM</th>
<th>ICADG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Module Optimization</strong></td>
<td>Crawler &amp; Indexer</td>
<td>Indexer</td>
<td>XML Indexer</td>
</tr>
<tr>
<td><strong>Metric</strong></td>
<td>Crawling and indexing important pages as observed by the users</td>
<td>Providing page retrieval according to semantics of queries.</td>
<td>Retrieval of relevant XML pages.</td>
</tr>
<tr>
<td><strong>Mined web Resource</strong></td>
<td>Query Logs and various data structures</td>
<td>Online sites like Dictionary based Sites</td>
<td>Query Logs</td>
</tr>
<tr>
<td><strong>Type of Mining</strong></td>
<td>Web Structure and Web Usage Mining</td>
<td>Web Structure and Web Content Mining</td>
<td>Web Usage Mining</td>
</tr>
</tbody>
</table>
| **Advantages** | • Crawlers can be made adaptive according to user accessing patterns of pages.  
• The relevant pages which previously were not appearing in the results are made to appear.  
• Query expansion is provided based on query semantics.  
• Search space is reduced as the results are presented different semantic categories.  
• XML pages are indexed in an efficient way, incremental updation is carried out.  
• Partial matching queries can be executed.  
• Index requires less storage space. | |

The next chapter describes proposed post-mining techniques i.e. the techniques developed for the front end of the search engine, in detail.