Integrated Web Searching: Crawler-Initiated Probe Selection and Content Summary Extraction

Amr El-Helw
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
aelhelw@cs.uwaterloo.ca

Aseem Cheema
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
apscheem@cs.uwaterloo.ca

ABSTRACT

A large percentage of the valuable data on the Internet is stored into hidden databases that cannot be crawled or indexed by traditional search engines. Metasearchers are helpful tools for searching over many such databases at once through a unified query interface. They should be used together with traditional search engines to find data from all over the Web. A critical task for a metasearcher is the selection of the most promising databases for the query, a task that typically relies on statistical summaries of the database contents. These statistics are collected by repeatedly probing a database with queries, retrieving the query results and sampling them. It is very important to use query probes that can return results that would give us the most representative content summaries. In this paper, we present an architecture that integrates traditional search engines and metasearchers, and outline the potential points where the two architectures need to interact. One of these points is the choice of query probes to collect content summaries from hidden databases. We present a new technique for identifying candidate values that can be used in the probing process. Our probe identification technique is the first to utilize traditional web crawlers to locate these values. We also present a new approach to use the extracted probes to compute the content summaries by selecting only databases that would produce useful results when queried with a particular probe. In addition, we propose an algorithm for analyzing the returned result pages, and finding links that lead to actual results. This new algorithm is the first to exploit several characteristics of result pages and utilizex these characteristics to achieve its goal. Finally, we evaluate our techniques thoroughly using a variety of databases, including several real web-accessible databases. Our experiments indicate that our new approach produces content summaries that truly represent the underlying databases, which helps the metasearcher in its database selection task. Also, it is a step on the way of building integrated searching systems that can search both the publicly indexable web, and the hidden web databases.

SIGMOD 2005 June 14-16, 2005 Baltimore, Maryland, USA.

1. INTRODUCTION

The World-Wide Web is growing rapidly, which makes it very difficult for users to locate the available information by simple browsing. Searching the web is today's standard to find information in any field of knowledge. Well-known search engines (e.g. *Google* and *Yahoo!*) crawl and index huge numbers of web pages to make their information available to users. However, the larger portion of the Web is not stored as crawlable Web pages, but hidden inside databases that are accessible only through search interfaces (thus the term *Hidden Web* [5]). Users can only use these interfaces to pose their queries, and their results are returned as dynamically generated Web pages. The contents of these databases cannot be crawled or indexed by search engines.

One way to search into the Hidden Web is using Metasearchers. A metasearcher can return results from multiple databases. The metasearcher's operation can be summarized in three main steps. Given a query, the first step is *"Database Selection"* where it selects the most suitable databases to answer this query. The next step is *"Query Translation"*, where the query is converted into a suitable form for each of the selected databases. The final step is *"Result Merging"*, where the results returned from all databases are merged and returned to the user.

Database selection is the most important step since it determines the sources in which results might be found. If the wrong databases are selected, a result might not be returned, even though it does exist. Database selection algorithms are based on statistics that represent the contents of each database. These statistics are usually called *content summaries*. The content summaries often include the document frequencies of the words that appear in the database, plus perhaps other simple statistics.

Database selection algorithms are given a user's query (usually in the form of one or more keywords). Based on the available content summaries of multiple databases, the algorithm decides which databases are most relevant to this query. Therefore, these content summaries have to be accurate, up-to-date, and truly representative of the underlying data.

Computing these content summaries is not a trivial task. To compute the content summary of a database, the database is probed with a large number of queries, the returned documents (Web pages) are retrieved and sampled, and the doc-

ument frequency of each word in the retrieved documents is stored. A fundamental issue in this process is choosing the probes. Poor probes might result in completely missing documents that address a certain topic, and therefore some words would be missing from the content summaries.

In this paper, we propose an architecture to integrate traditional web searching with metasearching. In the proposed scheme, components from both systems work together to find the most relevant results for a user's query.

We then focus on the problem of content summary extraction. We propose a new approach for finding "good" probes that can be used in content summary extraction. The proposed scheme requires some interaction between the metasearching components and ordinary crawlers, to extract possible probe values during the process of crawling static Web pages.

We also propose a scheme to use the collected probe values to actually probe the databases, identify the result pages and discard irrelevant contents from them, sample them and extract their content summaries, using every term in its proper context, and giving higher priority to most popular terms, thus extracting more accurate summaries especially for popular terms.

The contributions of this paper are (a) a novel technique for identifying probe values, (b) a new idea for selecting which probes to use with which databases, and (c) a new algorithm for analyzing search result pages and identifying the links that lead to actual results.

The rest of the paper is organized as follows. Section 2 gives the necessary background and the related work. Section 3 presents the proposed architecture for an integrated web searching system that combines traditional web searching and metasearching. Section 4 outlines our new technique to find values that can be used as probes. Section 5 presents a novel approach for content summary extraction that uses the probes identified earlier. The experiments in Section 6 show that our method extracts very representative content summaries. Finally, Section 7 concludes the paper.

2. RELATED WORK

In this section we give the required background and report the related efforts. Section 2.1 briefly describes existing techniques for finding candidate probe values. Section 2.2 describes existing methods of probing hidden web databases, and which databases to use with which probes. Then, section 2.3 discusses methods of identifying the results in the returned pages.

2.1 Probe Identification

Many ways have been recommended by researchers for finding probes. Callan et al. [1, 2] present an algorithm for content summary collection, in which there are two variants for probe identification. The first variant, called *RandomSampling-OtherResource* (*RS-Ord* for short) picks a random word from the dictionary. The second variant, called *RandomSampling-LearnedResource* (*RS-Lrd*) selects the next probe from among the words that have been already discovered during sampling. However, there are some problems associated with these approaches. Since *RS-Ord* selects a random word from the dictionary, there would be a lot of times where a probe would return no results (which is wasted effort). On the other side, since *RS-Lrd* selects a word from the already sampled documents, this means that all probe values are dependant on the first probe, so a poorly chosen first probe might affect the whole process.

Raghavan and Garcia-Molina [11] used a task-specific database, populated with label-value pairs for different form fields. This database is initialized manually by filling in labels and associated values. It can also collect more values from finite-domain form elements (e.g. lists and comboboxes) that it encounters. However, this approach was not intended for content summary extraction, but rather for crawling.

Ipeirotis and Gravano [8] proposed a focused probing technique that can actually classify the databases into a hierarchical categorization. The idea is to pre-classify a set of training documents under some categories, and then extract different terms from these documents and use them as query probes for the databases. Naturally, in this approach, a probe cannot belong to more than one category, and the documents returned for this probe give more weight to its associated category.

Another approach is proposed in [7]. In this approach, they start by randomly selecting a term from the search interface itself. They claim that most probably this term will be related to the contents of the database. The database is queried with this term, and the top k documents are retrieved. A subsequent term is then randomly selected from terms extracted from the retrieved documents. The process is repeated until a pre-defined number of documents are retrieved. This technique eliminates the need to have a predefined set of categories, and therefore does not have the problem of a probe that can belong to multiple categories. However, this technique has the same problem as the *RS-Lrd* described earlier. The retrieved documents are heavily dependent on the first probe (which is randomly chosen), because all subsequent probes are selected from the documents retrieved from the first one. The fact that the first probe is randomly selected from the search interface means that it could be any term in the page, which might not be very relevant to the contents of the database. And as a result of that, all (or most of) the subsequent documents will not be representative of the database.

In our approach, we present a new method of identifying values that can be used as candidate probes for Hidden Web databases. Our approach involves some integration with traditional web crawlers to identify these potential probes while crawling. This is explained in more detail in section 4.

2.2 Database Probing

This problem deals with selecting which probe(s) to use on which database(s), in order to get results that can be representative of the database, i.e., covers most of the terms in that database, and with an indicative sample frequency for each term.

Most of the researches in this field did not address this question, or just decided to use the available probes on all the databases. This, of course, is a time-consuming process, without any considerable gain, because an unsuitable probe would not return any results that can be sampled (the only information gained here is the frequency of the probe itself, which would be zero in this case).

However, the focused probing approach proposed in [8] tackles this problem. The goal of this approach is to place a database in an appropriate category as well as to collect its content summaries. In the first step, the database is probed with all probes belonging to general categories. The database is assumed to belong to the category whose probes return the greatest number of results. Subsequently, the database is further investigated by using probes from the subcategories of the current category, and so on.

As mentioned in section 2.1, this approach has the disadvantage that it does not handle the fact that a database might belong to more than one category (for example, a database about sport injuries should be in both the sports and the health categories). In that case, the database is only categorized under one of them, and queries related to the other category would not consider this database.

Our technique discussed in section 5.1 uses a heuristic to choose which probes to use with which databases, which would produce more representative content summaries.

2.3 Result Set Identification

When a database is queried (either with a probe or with an actual user query), the metasearcher should be able to analyze the returned page (which contains a set of links that represent the results of the query). These pages usually contain a considerable amount of irrelevant data (or *noise*), in terms of advertisements, navigational items (menus, previous/next links, etc.), or even other links that might be pointing to totally irrelevant data. The system should be able to identify those links that represent the actual results of the submitted query. Moreover, in case of content summary collection, the metasearcher should follow these links, retrieve the corresponding pages, and again, identify the portions of these pages that represent the actual data that would be sampled.

Most of the result pages follow some template that has a considerable amount of text used only for presentation purposes. Hedley et al. [6] proposed a method to identify Web page templates by analyzing the textual contents and the adjacent tag structures of a document in order to extract query-related data. In this paper, a Web page is represented as a sequence of text segments, where a text segment is of the form (*text; tag; tag*). The mechanism to detect templates is described as follows:

1. Text segments of documents are analyzed based on textual contents and their adjacent tag segments.
2. An initial template is identified by examining the first two sample documents.
3. The template is then generated if matched text segments along with their adjacent tag segments are found from both documents.
4. Subsequent documents retrieved are compared with the template generated. Text segments that are not found in the template are extracted for each document to be further processed.
5. When no matches are found from the existing template, document contents are extracted for the generation of future templates.

A similar approach is presented in [3] to analyze the result page and extract relevant information.

In [7], they go further by representing the document content by text segments and their neighboring tag segments, which they refer to as *Text with Neighboring Adjacent Tag Segments (TNATS)*. The neighboring adjacent tag segments of a text segment are defined as the list of tags that are located immediately before and after the text segment until another text segment is reached. The neighboring tag segments of a text segment describe how the text segment is structured and its relation to the nearest text segments. A text segment, txs, is defined as: $txs = (tx_i, tg{-}lst_j, tg{-}lst_k)$, where tx_i is the textual content of the i^{th} text segment, $tg - lst_j$ and $tg{-}lst_k$ represent tags located before and after tx_i until another text segment is reached. This gives better representation of the document, and then the same algorithm as in [6] is used to identify the template.

However, these techniques might still miss links that do not belong to the template, and that would be mistaken as actual results in the page. Our technique described in section 5.2 uses this algorithm as a first step, and then further processes the remaining contents taking into consideration some observed characteristics to precisely identify the results.

3. PROPOSED ARCHITECTURE

Figure 1 depicts an architecture that integrates traditional searching components and metasearchers. The dotted rectangles represent each of the two individual systems, and the thick arrows represent potential points of interaction. The first possible interaction is that the crawler can have a secondary task (besides crawling and storing web pages), which is to populate the probe repository. Other possible interactions between the two systems can include exploiting feedback from the search engine (in terms of most frequently queried keywords) that can also be used as potential probes for hidden web databases. Moreover, both systems must interact when returning the results to the client, since results from both sources should undergo some sort of global ranking.

Section 4 discusses the first interaction, i.e. using the crawler to populate the probe repository. The other interactions can be further investigated in future research.

4. IDENTIFYING CANDIDATE PROBES

As mentioned in section 2.1, it is important to choose good probes that, when used to query the database, would return results that would result in very representative content summaries.

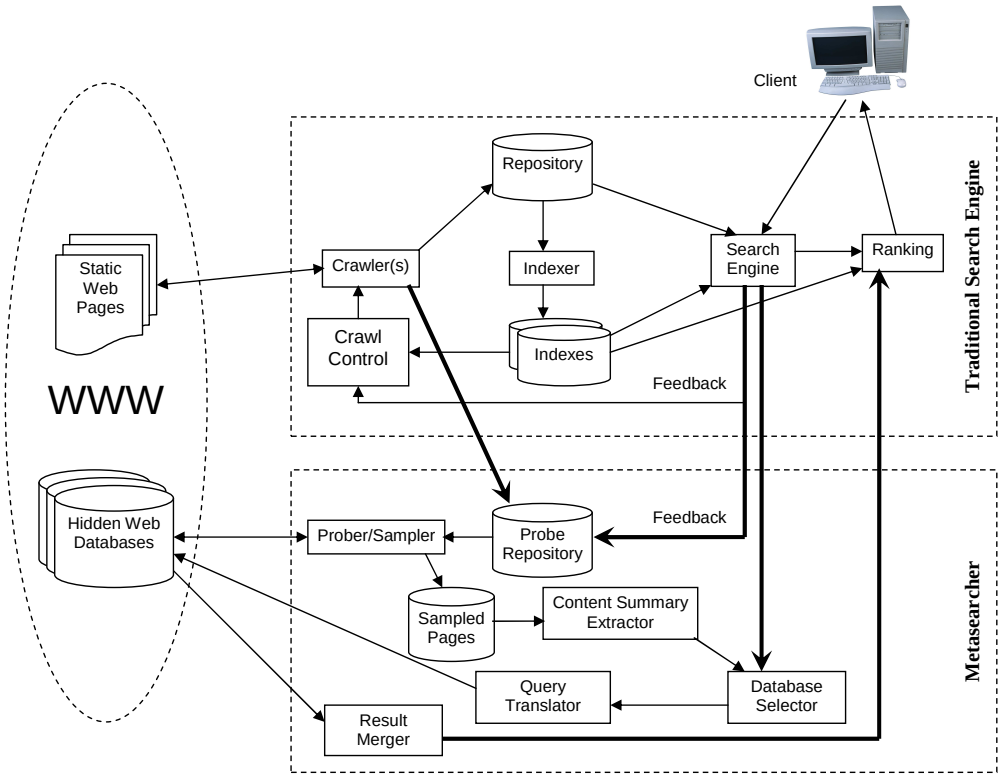

Figure 1: Integrated Web Searching Architecture

To accomplish this, we present an approach that depends on traditional web crawlers. A traditional web crawler is supplied with a set of starting URLs. For each of these URLs, the crawler retrieves and parses the page with that URL, extracts any hyperlinks in it, and adds these hyperlinks to a queue of URLs. In the next cycle, the crawler extracts a URL from the queue (based on some order) and repeats the same process.

However, most crawlers avoid links that have query parameters in them. These links are usually of the form:

```
http://x.y.z?param1=value1&param2=value2&...
```

When a crawler encounters a link of this form, it simply ignores that link, and does not add it to its queue. This is mainly for the following reasons:

1. These links do not point to web pages, but rather to scripts that dynamically generate web pages. The generated web pages can be countless and change very frequently, depending on the passed parameters, therefore they are not meant to be crawled or indexed.
2. These links might lead to potential *Spider Traps*. A spider trap is a script that is designed specifically so that when a crawler tries to crawl it, it will keep crawling it over and over again indefinitely. Spider traps are usually implemented by providing links that have different parameters but point to the same script, so the crawler would *think* they are different links. Some approaches are used to handle this problem, like limiting the links collected from the same domain to a certain limit.

Our approach makes use of these parameterized links, instead of completely ignoring them. In our proposed system, when a crawler encounters a link of this form, it should extract the *parameter-value* pairs, and store them in what is called a *Probe Repository*. These values can be good candidates for probing the hidden web databases.

In the following subsections, we discuss the details of this approach. Section 4.1 discusses the Probe Repository, and how values are stored in it, and section 4.2 describes some heuristic rules that we used to identify values that can act as good probes.

4.1 The Probe Repository

The probe repository is the core of the probing module. It is used to store the candidate probes that are extracted by the crawler, so they can later be used to probe the hidden databases.

The structure of the probe repository is shown in Figure 2. For each candidate probe, we store the parameter name and the value that have been encountered while crawling. Section 4.2 describes which *parameter-value* tuples are considered as candidate probes, and which are not.

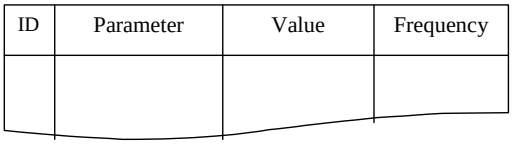

Figure 2: Probe Repository

```
url_queue.add(start_URLs);
while(not url_queue.empty()){
  url = url_queue.get();
  page = crawlpage(url);
  crawled_queue.add(url);
  url_list = extract_urls(page);
  for each u in url_list{
    if (u contains parameters){
      param_list = extract_params(u);
      refine(param_list, heuristics);
      for each p in param_list{
        if((p.name, p.value) in probe_repository)
          increment frequency for (p.name, p.value);
        else
          probe_repository.add(p.name, p.value, 1);
      }
    }
    else{
      if (u not in url_queue) and (u not in crawled_queue)
        url_queue.add(u);
    }
  }
}
```

Figure 3: Crawling Algorithm

The *Frequency* field is used as an indicator of this particular probe's *popularity*. If a certain probe is found many times during crawling, it means that many pages have the same links with the same parameter value. Thus, this value can be considered as a popular item.

As a result of that, the crawling algorithm should be modified to include this functionality. A simple crawling algorithm that incorporates probe extraction is shown in Figure 3. In this algorithm, if a link does not contain any parameters, the crawler proceeds normally (as a traditional crawler). However, if a link with parameters is encountered, the parameter names and values are extracted and added to the repository. The step $refine(param\_list, heuristics)$ is explained in the next section.

4.2 Candidate Probe Heuristics

When collecting parameter names and values from hyperlinks to use them as probes, we noticed that not all of these parameters can be useful for this purpose. This is mainly because the values of these parameters can by proper English words, names, item codes, dates, sometimes even HTTP session IDs (which are usually long sequences of letters and numbers). To overcome this problem, we had to filter the extracted parameters using some heuristic rules.

In our proposed scheme, a parameter name-value pair is considered a valid candidate probe if *all* of the following conditions hold:

1. *Neither the name nor the value are numbers. Also, the value should not contain any numeric digits* (other than those used to represent ASCII characters). This is to filter out item codes.
2. *The value is not a URL.* This is because some web sites pass page URLs as parameters to be used in redirection.
3. *The value is not a date.*
4. *The value is not a file path* (same reason as rule 2).
5. *The parameter name does not contain the words "**id**" or "**code**"* (for the same reason as rule 1).
6. *The parameter name and value are less than or equal to l characters in length* (in our implementation, we used $l = 50$ which produced very good results). This is also to eliminate meaningless sequences of letters (mostly used to represent session information and browser cookies).
7. *The parameter value is not a stop word.*

The last rule states that the parameter value should not be a *stop word*. Stop words are common words that appear in many different contexts without having a meaning of their own. they are used either for grammatical purposes, or to emphasize the meaning of a neighboring word (or group of words). Examples of common stop words are $\{a, an, the, of, more, etc.\}$. These words cannot be used as probes basically because most databases do not allow querying using stop words (because they can be found in every possible result in the database). Therefore, using them as probes is useless.

5. CONTENT SUMMARY EXTRACTION

After collecting terms that can be used as probes, the next step is to use these probes to query the hidden web databases, in order to form an idea about the contents of these databases. In this phase we assume that the metasearcher has a pre-defined list of databases that it uses for searching. A lot of work has been done discussing how to actually find hidden web databases so that this list can be constructed, however, this is an independent problem and is not in the scope of this paper. The interested reader is directed to [11, 9, 4].

In the next subsections, we explain some main issues that affect the content summary extraction process. Section 5.1 answers the question of which databases should be used with which probes to get representative results. Section 5.2 explains how to identify the results in the returned pages. Finally section 5.3 discusses the content summary extraction itself.

5.1 Probing the Databases

As we already mentioned, for this part we assume that we have a list of all the databases that we want to search. This is shown in Figure 4a. For each database D, we store the URL of the search interface of that database. Also, we store a list of all the search fields in each of these interfaces, as shown in Figure 4b. For each field, we store the *field ID* and the *field label*. The field ID is used to query this database,

while the field ID and label together are used to match it with suitable probes, as will be explained.

DBID	DB URL
1	http://.....
2	http://....

(a) Databases

DBID	Field ID	Field Label
1	kw	Keywords
1	auth	Author

(b) Database Fields

Figure 4: List of Databases and their Fields

Given the probe repository that was populated by the crawler, the *Prober* module gets probes from this repository, one at a time, and uses them to query the database. The Prober always selects the probe that has the following characteristics:

1. It should not have been used for probing before.
2. Among all the unused probes, it should have the highest frequency. If more than one probe have the same frequency, an arbitrary probe is chosen.

The frequency of a probe is an indication of how *popular* this probe is, and therefore, how probable it is that a user's query might contain this probe's value. As a result, it is most desirable to use this value to probe the databases to get information about their contents.

Now that the probe has been chosen , it is important to choose the database(s) on which to use this probe. A naive approach would use this probe on all available databases, but this would incur extra overhead, in terms of both processing power and network traffic.

To find the databases to probe, we select from the database list, all those databases that satisfy *any* of the following conditions:

1. The database has a field whose ID equals the probe name.
2. The database has a field whose label equals the probe name.
3. The database has only one search field.

The first two conditions are used for databases that have an *Advanced Search* interface. These interfaces usually have multiple fields, and these fields have IDs or labels that indicate what kind of input is expected in them. This way, we target only databases that expect the same domain of the probe on hand. However, some search interfaces have only a single field that accepts any keywords. It is not possible to know what kind of inputs this database expects by analyzing its interface. For this purpose, we included the third condition that basically allows the Prober to use a probe on all databases that have a single field in their search interface.

In order to probe a database, a parameter *name-value* pair is formed from the matching field ID and the probe value, and this pair is appended to the database interface's URL (as retrieved from the stored database list in Figure 4a). The Prober module has a time-out condition that prevents it from waiting indefinitely for a database response, in order to handle network problems and problems in the database server itself.

5.2 Result Set Identification

Once the database is probed, it returns a page that contains a set of links to the results of the submitted query. However, this page usually contains more than just these links. If the page contained only those links, it would be simple to extract them in a manner similar to that of the crawler, and just retrieve the target pages. However, as mentioned in section 2.3, it is usually the case that this page might contain a lot of other irrelevant links. These links might be advertisements that point to other commercial web sites, navigational links which can be used by the used to navigate through the current site itself (for example, menu items or links to more result pages), or links associated with each result, but do not provide useful information in the case of content summary extraction (for example, in some online markets, each item might have a link that would lead to a page with possible payment methods, or terms of agreement that the user has to read before purchasing). Therefore, an important step in collecting content summaries, is to identify those links that represent the actual results of the posed query.

As mentioned in section 2.3, most result pages, follow some kind of a template that describes the presentation of the page. This template is usually the same for all result pages that belong to the same database, regardless of the submitted query, except for some minor differences that can be encountered.

So, as a first step to eliminate irrelevant links from the result page, we used the approach presented in [7], in which the page contents are represented as *Text Segments And their neighboring Tag Segments (TNATS)*. Each TNATS entry is a tuple of the form $(tx_i, tg - lst_j, tg - lst_k)$. For example, consider the HTML source code in Figure 5. The TNATS representation of this fragment would be as shown in Table 1.

The algorithm in [7] detects the template at run time by continuously comparing pages retrieved from the same source (after converting them to the TNATS representation), and identifying the common parts between these pages. These parts constitute the template of this specific database. Then the algorithm makes another pass on the pages removing this common part from them, so that only the non-template contents are left.

Table 1: TNATS representation of Figure 5

tx_i	$tg - lst_j$	$tg - lst_k$
Results	<html><head><title>	</title></head><body><hr><h3>
1. 21 inch TV set	</title></head><body><hr><h3>	</h3>
Color:	</h3>	
Black		
...	...	...

```
…
<HTML><HEAD><TITLE>Results</TITLE></HEAD>
<BODY>
<HR><H3><B><I> 1. 21 inch TV set</I></B></H3>
<I><B>Color: </B></I>
Black<BR>
<I><B>Weight (KG): </B></I>
5<BR>
...
```

Figure 5: HTML source

However, in our approach, we chose to execute the template identification phase offline, i.e. before the actual operation of the system. We used an automated process to issue multiple queries at each database, and retrieve the returned pages. These pages are compared, and the common parts are identified. These parts form the template of this database, and is stored in the database list that is accessible to the Prober.

At runtime, the Prober retrieves this stored template, matches it with the results returned from the database being queried, and removes all common items from the returned result page, thus the remaining data do not contain any of the template items.

However, this does not solve the problem completely. In most of the cases there are still some links that do not belong to the template but that do not lead to actual results. In order to remove these links, we introduced a new algorithm based on some observations about the characteristics of the *result links*, and the *non-result links*.

Observation 1 All result links point to pages in the same domain of the database interface.

Observation 2 All result links have very similar URLs that differ only in the last part (which is usually in the form of some parameter-values).

Observation 3 In most (if not all) of the results, the keyword that initiated the search is present either in the link text itself, or in the snippet directly following the link.

The algorithm to extract the result links is shown in Figure 6. The algorithm first starts by constructing a tree with all the links in the page (after removing the template data). Using observation 1, the root of this tree is the domain of the search interface, and only links that belong to this domain

```
Algorithm GetResultSet(){
  // Construct the link_tree
  url_list = extract_urls(page);
  link_tree = null;
  link_tree.addnode(db_domain);
  for each url in url_list{
    if (url.domain = db_domain){
      level_list = url.split();
      current_node = link_tree.root;
      for each level in level_list
        if (level not in current_node.children)
          current_node.addchild(level)
        current_node = level;
      }
    }
  }
  // Get the subtree with maximum number of leaves
  leaf_list = link_tree.get_leaves();
  parent_list = null;
  for each leaf in leaf_list{
    if (leaf.parent in parent_list)
      parent_list.increment(leaf.parent);
    else
      parent_list.add(leaf.parent, 1);
  }
  max_parent = parent_list.getmax();
  return max_parent.children;
}
```

Figure 6: Result Identification Algorithm

are added to this tree. A link is represented in the tree as follows. Consider this link `http://x.y.z/a?id=1`. The tree branch that represents this link would be a root node (`http://x.y.z`) that has a child (`a`) that has a child (`id=1`). The depth of the branch is the number of levels in the link.

To construct the tree, the algorithm collects all links in the page. Each link is split into its individual levels. Any link that does not belong to the domain of the database is discarded (observation 1). A branch representing each link is added to the tree, adding any nodes that do not already exist.

Once the tree is constructed, and according to observation 2, all result links should be found as leaf nodes that have the same parent. So, the algorithm collects all the leaf nodes of the tree, with their parents, then finds the parent that has the maximum number of children (this should be the subtree containing the result links).

Note that, in Figure 6, the line

```
max_parent = parent_list.getmax();
```

finds the parent that has the maximum number of children that have the search keyword in the link text or in the

snippet (observation 3). This observation is very important in fact, since sometimes there are other subtrees that have more children but do not contain the keywords in their text.

To illustrate the technique, consider the following example. Assume that a query to the database whose domain is (`http://x.y.z`) returned a page that has the following hyperlinks:

```
http://x.y.z/a?id=1
http://a.b.c/a?id=1
http://x.y.z/a?id=5
http://x.y.z/b?id=6
http://x.y.z/c
http://a.b.c/
http://x.y.z/a?id=23
```

According to our algorithm, all the links that are pointing outside the domain (`http://x.y.z`) will be discarded, and the remaining links will be arranged in a tree structure, as shown in Figure 7. In order to extract the result links, the algorithm searches for the largest number of leaf nodes that have the same parent, which in this case would be the leaf nodes whose parent is the node "a", assuming that the link text of these nodes contains the keywords that initiated the query.

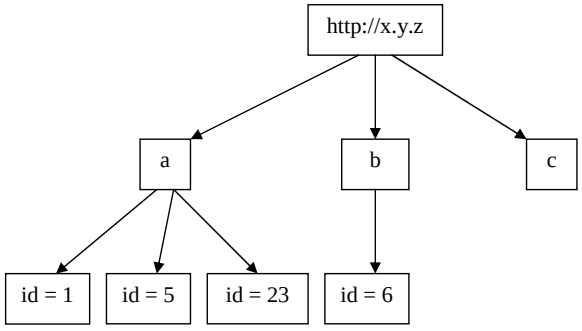

Figure 7: Tree Representation of Hyperlinks

5.3 Extracting summaries

Once the result links are identified, the Prober retrieves the corresponding result pages. To make sure that each result page is never retrieved more than once, the URLs of these pages are stored, and only pages with new URLs are retrieved.

For each retrieved page, once again template detection and removal take place. The remaining text is filtered to remove stop words, and then the *inverse document frequency (IDF)* for each of the remaining words is incremented.

6. EXPERIMENTAL RESULTS

This section presents the results of the various experiments carried out throughout this research.

6.1 Experimental Settings

For these experiments, we had to implement two modules. The first module is the web crawler, based on the algorithm shown in Figure 3, including the probe extraction functionality. The second module is the Prober that uses the collected probes to extract content summaries from hidden web databases.

The crawler and the prober were implemented in Java, and ran on a Pentium IV (2.8 GHz) machine with 1 GB of main memory. For the probe repository, we used an Oracle 10g database server, running on another Pentium IV (3.0 GHz) machine, with 1 GB of main memory.

We carried out our probing and content summary extraction on the following hidden web databases:

ACM Digital Library (*http://portal.acm.org/dl.cfm*) a collection of scientific research papers and articles.

IEEE Xplore (*http://ieeexplore.ieee.org*) another library of scientific research material.

CanLearn (*http://www.canlearn.ca*) learning and education resources in Canada.

Walmart (*http://www.walmart.com*) a popular online shopping portal.

Gately's (*http://www.gatelys.com*) a furniture retail store online interface.

Barnes & Noble (*http://www.barnesandnoble.com*) an online bookstore.

CiteSeer (*http://citeseer.ist.psu.edu/cs*) a scientific literature digital library.

All probing or querying was done through the search interfaces of these databases.

When carrying out our experiments, we were interested in the following: (a) examining the effectiveness of the probe extraction technique by determining the rate by which "popular" probe values are found (section 6.2), (b) evaluating the accuracy of the result identification algorithm (section 6.3), and (c) evaluating the extracted content summaries and determining whether or not they truly represent the data in the databases (section 6.4).

6.2 Probe Extraction

In this experiment we tested the effectiveness of our probe extraction scheme. We let the crawler crawl web pages and collect probes during the crawling process. The crawler module crawled about 4,000 web pages from different web-sites. In order to guarantee that the crawler crawls different web-sites, we started the crawler with some initial URLs of web directories like *Yahoo! directory*[1], and *Google directory*[2].

We measured the rate of probe extraction, i.e. the number of new probes found per page. Figure 8 shows at the beginning of the crawling process, the number of new probes

[1]http://dir.yahoo.com
[2]http://directory.google.com

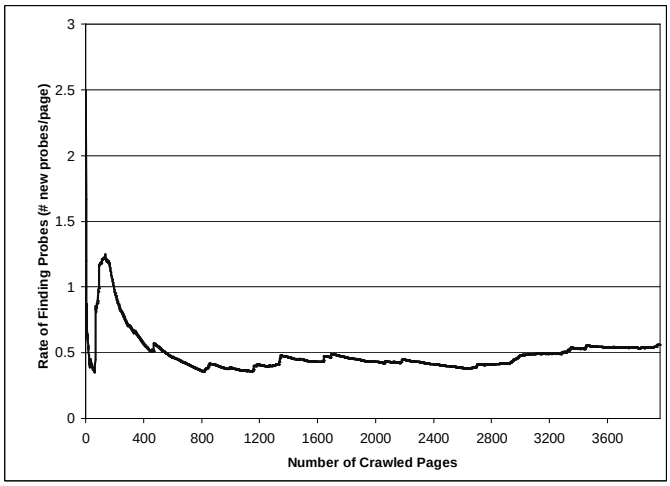

Figure 8: Probe Extraction Rate

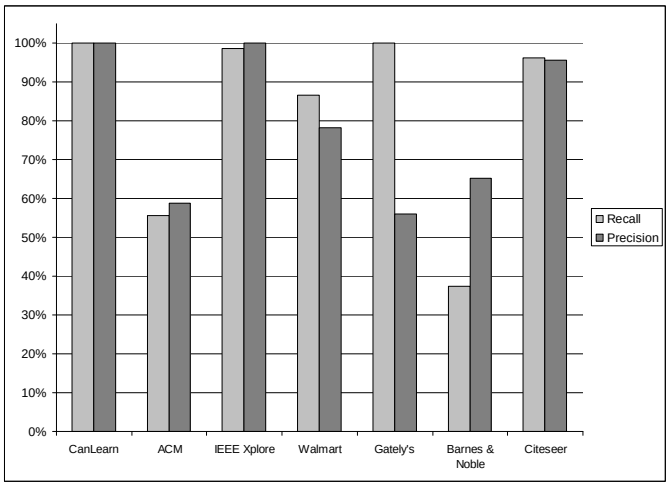

Figure 9: Result Set Identification

Table 2: Most Popular Probes

Probe	Frequency
$page = p1$	3500
$hl = en$	2288
$tmpl = story$	2117
$src = ym$	2108
$lineup = us\_pacific$	910

Table 3: Number of Sampled Pages

Database	#Pages
CanLearn	281
ACM	819
IEEE Xplore	619
Walmart	1003
Gately's	85
Barnes & Noble	106

found is relatively high, and then it decreases until it becomes fairly stable at the rate of about 0.5 *probes/page*. This is mainly because, after the initial transient stage in the crawling process, the probe repository will contain a considerable amount of probes, so the chance that a probe encountered later already exists in the repository increases, and thus the rate of finding a new probe decreases. This shows that a considerable amount of *"popular"* probe values are identified early in the crawling process.

However, since we extract probes from hyperlinks, popular probes do not have to be proper English words, as long as they are popular (i.e. the same probe is found multiple times), because they can generate a number of results from the databases, and that is what we look for in a probe. Table 2 shows the most popular probes extracted in our experiments while crawling 4,000 pages.

6.3 Result Set Identification

In this experiment, we used the probes collected earlier to query the databases, in order to observe the effectiveness of the result identification algorithm. We applied the algorithm on the returned pages to identify the result links, and then we compared these results with actual results that we have manually identified for the same queries. This experiment was carried out using 10 query probes over the 7 databases mentioned in section 6.1.

For each query on each database, we measured the following three values:

N_{actual}**:** the number of results returned by the database that we have manually identified.

$N_{identified}$**:** the number of results that are identified by the algorithm.

$N_{correct}$**:** the number of results that have been identified by the algorithm and which are in the manually identified results as well.

These three values are then accumulated (summed) for each database over the whole set of queries, and that provides us with total N_{actual}, $N_{identified}$ and $N_{correct}$ values for every database. The reasoning behind this is that, as we noticed, there is a maximum limit on the number of results being returned by a query, so taking a summation over a number of queries guarantees that the contribution of each query is the same in the total.

Given these 3 parameters, we calculate the following 2 metrics for each database DB_i:

$$Precision(DB_i) = N_{correct}(DB_i)/N_{identified}(DB_i)$$

$$Recall(DB_i) = N_{correct}(DB_i)/N_{actual}(DB_i)$$

Precision is the percentage of correct identified results with respect to the total identified results. Recall is the per-

Table 4: A Sample of the Extracted Content Summaries

Database	CanLearn	ACM	IEEE Xplore	Walmart	Gately's	Barnes & Noble
#Pages	281	819	619	1003	85	106
Sample Words (word, freq)	canlearn, 281 education, 281 tuition, 233 credentials, 207 degree, 153	computer, 762 publish, 747 information, 722 library, 714 survey, 705	paper, 617 cite, 617 article, 519 volume, 432 issue, 360	product, 1002 price, 975 gift, 969 department, 917 seller, 916	quantity, 82 delivery, 70 sale, 65 room, 50 bedroom, 27	bestseller, 104 recommended, 104 textbooks, 91 biography, 89 document, 89

centage of correct identified results over all actual results returned by the database. Figure 9 shows the results of the comparison. It shows the precision and recall of the algorithm in case of each database.

Five of the seven databases show considerably good recall. Note that the recall represents the percentage of actual results that were captured by the algorithm. One reason for low recall in some databases (for example, Barnes & Noble) is due to the "not so meaningful" query probes. If the query probe is not a meaningful word, it does not usually appear in the link text, and thus the link is not considered as a valid result. One other reason for low recall is that some of the search interfaces use *stemming*, so again the submitted probe value would not necessarily exist in the link text (although a word similar to it might exist, but our algorithm does not handle this case), and therefore the link would not be recognized. However, there are two possible solutions for this problem, which we have not implemented due to time limitations. These possible solutions involve modifying the part of the algorithm that looks for the probe value in the link text. It can be modified to do either of the following:

- Perform *approximate string matching* instead of exact string matching, so any word that is similar to the probe value would be considered as a match.
- Enumerate all words that have the same stem as the probe value, and search for all of them in the link text, instead of searching only for the probe value itself.

The precision metric can be considered as the reverse measure of *noise*. Noise is the number of extra links that are mistakenly identified by the algorithm as being result links, although they are not part of actual result set returned by the database. As depicted in Figure 9, in most cases, precision is quite similar to the recall. Databases with good recall have good precision as well. The reason for low precision in some cases is the inability to identify the cases where no actual results are returned. When databases return no results, our algorithm still tries to find the result set. Hence, it recognizes the wrong result set and precision decreases. The precision can be improved considerably by implementing heuristics to identify zero results being returned. This can be achiever for example by storing HTML templates for the "zero results" cases for each database, and trying to match these tamplates with the returned pages at runtime. However, this was not implemented in our algorithm. Some research efforts have already been done to identify the cases where no results are returned. The reader is referred to [11, 3].

Overall, the algorithm shows convincing results and is able to identify the result sets. The study covers only seven database due to some manual efforts involved in querying the databases. But we strongly believe that the algorithm will work well on a wider range of databases as well. We believe that with minor changes and implementation of heuristics to identify zero result pages, the effectiveness can be improved to a large extent

6.4 Content Summary Extraction

In this experiment, we used the probes collected earlier to extract content summaries from the databases listed in section 6.1. It is not possible to know the actual number of pages that can be generated from each database. These are practically infinite, therefore, our measure was the document frequency of every sampled word relative to the number of documents sampled from the database. The total number of pages sampled during our experiment from each database, is shown in Table 3.

After we retrieved pages from the database, we used the Porter stemming algorithm [10] to find the stem of each encountered word in these documents, after removing stop words. We used a stemming algorithm because it makes more sense to collect statistics of word stems as opposed to words themselves, since there can be different forms of the same word in different documents, which should contribute to the frequency of a single word stem.

The extracted summaries were very representative of the underlying databases. Table 4 shows some of the words sampled from each database, with the sample document frequency of each word.

7. CONCLUSION AND FUTURE WORK

In this paper we presented an architecture for integrating current search engines with hidden web metasearchers, and outlined some of the potential points where the two architectures can interact.

We then focused on one of these points, which is probe identification for content summary extraction. In section 4, we presented a novel approach for collecting these probes. The proposed approach depends on incorporating web crawlers into the probe identification process by extracting candidate values from hyperlinks during crawling, taking into consideration some heuristic rules, to identify values that can be used for probing databases.

In section 5.1, we described a method to use the collected probes for computing content summaries, by using these probes in their context, in order to get content summaries

that best represent the underlying databases.

Finally, in section 5.2, we presented a new technique for identifying the hyperlinks that lead to user query results among other irrelevant links that exist in the result page. Our technique first attempts to discard the template portions of the page that are used only for presentation purposes, and then it exploits some observations about the characteristics of the result links and their relationship with their domain and with each other, by arranging them into a tree structure, in order to identify only those links that point to actual result pages.

The experiments carried out in section 6 demonstrate that our approach succeeds in computing content summaries that are truly representative of the actual data in the hidden database, and therefore can be relied upon in the database selection phase of metasearching.

The next immediate step in this work would be to improve the proposed techniques. For example, in section 6.3 we outlined some ideas that can be investigated to improve the precision and recall of the result identification algorithm. Furthermore, more heuristics can be incorporated in the probe extraction technique to filter out more useless probe values.

As long term future work, it would be useful to further investigate the necessary measures that have to be taken in order to fully integrate traditional web search engines and metasearchers. As already outlined in section 3, there are manu points that can be explored. For instance, feedback from the search engine (in terms of most popular user queries) is currently used to dynamically control the crawling process, in order to give high priority to pages that might be an answer to their queries. The same idea can be applied in metasearching by using the user query feedback as potential probing candidates to aid in the content summary extraction process. This approach was never tackled before and therefore needs further investigation to determine its usefulness. In addition, if the two searching architectures are to be integrated, the results returned from both have to be merger before returning them to the user. This introduced a new problem, that is, assigning an overall rank to the results returned from both sources taking into consideration the individual ranks that the results from each source have.

8. REFERENCES

[1] J. Callan and M. Connell. Query-based sampling of text databases. *ACM TOIS*, 19(2):97–130, 2001.

[2] J. Callan, M. Connell, and A. Du. Automatic discovery of language models for text databases. In *Proc. of ACM SIGMOD conf.*, pages 479–490, 1999.

[3] J. Caverlee, L. Liu, and D. Buttler. Probe, cluster, and discover: Focused extraction of qa-pagelets from the deep web. In *Proc. of the 20th ICDE conf.*, 2004.

[4] A. de Carvalho Fontes and F. S. Silva. Smartcrawl: A new strategy for the exploration of the hidden web. In *Proc of the 6th ACM WIDM workshop*, pages 9–15, 2004.

[5] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Rec.*, 27(3):59–74, 1998.

[6] Y. Hedley, M. Younas, A. James, and M. Sanderson. Query-related data extraction of hidden web documents. In *Proc. of the 27th ACM SIGIR conf.*, pages 558–559, 2004.

[7] Y. Hedley, M. Younas, A. James, and M. Sanderson. A two-phase sampling technique for information extraction from hidden web databases. In *Proc. of the 6th ACM WIDM workshop*, pages 1–8, 2004.

[8] P. G. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *Proc. of the 28th VLDB conf.*, pages 394–405, 2002.

[9] J. P. Lage, A. S. da Silva, P. B. Golgher, and A. H. F. Laender. Collecting hidden web pages for data extraction. In *Proc. of the 4th ACM WIDM workshop*, pages 69–75, 2002.

[10] M. F. Porter. An algorithm for suffix stripping. pages 313–316, 1997.

[11] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proc. of the 27th VLDB conf.*, pages 129–138, 2001.