QUERYING MULTIPLE DOCUMENT COLLECTIONS ACROSS THE INTERNET

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Luis Gravano
August 1997

Abstract

Information sources are available everywhere, both within the internal networks of organizations and on the Internet. The source contents are often hidden behind search interfaces and models that vary from source to source. Furthermore, these sources are usually numerous, and users cannot evaluate their queries over all of them. Consequently, it is crucial for users to have *metasearchers*, which are services that provide unified query interfaces to multiple information sources. Given a user query, the metasearcher first chooses the best sources to evaluate the query. Second, the metasearcher submits the query to these sources. Finally, the metasearcher merges the query results from the sources. To address the first task, we designed *GlOSS*, a scalable system that chooses the best document sources for a query. The *GlOSS* information about each source is orders of magnitude smaller than the source contents. To address the other two tasks above and to facilitate the extraction of the *GlOSS* information from the sources, we coordinated the design of *STARTS*, an emerging protocol for Internet retrieval and search involving around 11 companies and organizations. Unfortunately, extracting the best objects for a query according to the metasearcher might be an expensive operation, since the sources' ranking algorithms might differ radically from that of the metasearcher's. We studied a result merging condition that characterizes what sources are "good" with respect to result merging. Finally, we also studied the metasearching problem for a novel application: the detection of illegal dissemination of copyrighted material. To address this problem we developed *dSCAM*, an "illegal copy" metasearcher that finds potential copies of a document over distributed text sources.

A mis abuelos Luis y Adelita

Acknowledgements

First things first. Hector García-Molina, my advisor at Stanford, has been the greatest: a truly 7×24 advisor, incredibly busy but always available for discussion and guidance. His famous, sometimes dreaded yellow sheets with hand-written comments about our papers have improved my writing style enormously. He has taught me how to write, communicate, and think clearly. I hope that I can be at least half as patient and caring with my future students as he is with his.

Jorge Sanz, my advisor at IBM Argentina and IBM Almaden in the pre-Stanford years, was the first one to show me what research was all about. I would probably not be in academia now if I had not met him back in 1990 when I was looking for a topic for my undergraduate thesis. The endless discussions, sometimes extending way past three o'clock in the morning, taught me how much fun research could be. Many other people at IBM were incredibly supportive and exciting to work with during my parallel-processing years, especially Shuki Bruck, Bob Cypher, Magda Konstantinidou, and Dragutin Petkovic at IBM Almaden, and Sergio Felperin and Gustavo Pifarré at IBM Argentina. All of these IBMers encouraged me to come to grad school to the USA. I really thank them for one of the best decisions I have ever made.

During the Stanford years, I continued meeting wonderful people at research labs in the Bay Area. I had the pleasure to work again at IBM Almaden, this time with Laura Haas, Peter Schwarz, and Anthony Tomasic, among others. (The results in Section 4.7 are part of the work that we did together.) I also worked at Hewlett-Packard Laboratories in Palo Alto with Surajit Chaudhuri and Umesh Dayal. Special thanks go to Laura and Umesh, two truly outstanding researchers and the warmest people, who did not complain (well, not too much at least) when they had to send millions of copies of their letters of recommendation during my slightly too "thorough" job hunt earlier this year. Working with Surajit has also been great, in spite (or because!) of our loud, very loud discussions about almost anything. I really hope that I will work again with every single one of these incredible people.

At Stanford, the list is long. Initially, I had the privilege to work with Anthony on *GlOSS* (Chapter 3 and the inspiration of many of the others are joint work with him and Héctor). Anthony gave me invaluable advice on grad-student life. Later, I joined the Digital Library project. Its manager, Andreas Paepcke, is an admirable person whose sense of humor has made many tough days much more bearable. (Chapter 2 is joint work with him, Chen-Chuan Kevin Chang, and Héctor.) The other professors in the project and the database group, Daphne Koller, Jeff Ullman, Jennifer Widom, Gio Wiederhold, and Terry Winograd, have always been willing to brainstorm about virtually any topic. The same is true of the other members of the group, although they tend to discuss issues as interesting as why Argentinians pronounce the letter "y" the "wrong" way. My officemates in Margaret Jacks 402 and Gates 432 have engaged in this and other edifying discussions over the years, especially Sergey Brin, Venky Harinarayan, Yannis Papakonstantinou, who prevented me from eating too many of my NutriGrain bars, Narayanan Shivakumar (a.k.a. Shiva), and Yue Zhuge. (Shiva, Héctor, and I wrote Chapter 6 together.) I have had many inspiring conversations with other members of the group like Michelle Baldonado, Scott Hassan, Larry Page, Anand Rajaraman, Vasilis Vassalos, Tak Yan, and Ramana Yerneni, just to mention a few. Some of the future-work ideas in Chapter 8 have come out of these discussions. Finally, Sharon Lambeth and Marianne Siroker have made my life so much easier over these years, dealing with all the Stanford bureaucracy, finances, obscure requirements, and various reimbursements in the most efficient way.

I would never have finished my Ph.D. without the support of all my friends in Argentina and in the USA. I will not list them here: I am extremely lucky to have *many* great friends (and I do not use the word friend lightly). You know who you are and I thank you for the endless listening-understanding-scolding cycles. I would also like to thank my brothers, Matías and Agustín, for always being there. Agustín:

keep bombarding me with your emails and your energy! I need them.

A final word for my parents, María Virginia Fornari and Juan Carlos Gravano. Les debo todo lo que soy: Uds. me hicieron valorar la educación y el estudio desde que tengo uso de razón. A Uds. y a los abuelos, de quienes heredé, entre muchas otras cosas, el amor por la docencia, les dedico esta tesis.

Contents

List of Figures

Chapter 1

Introduction

The Internet has grown dramatically over the past few years. Document sources are available everywhere, both within the internal networks of organizations and on the Internet. The source contents are often hidden behind search interfaces and models that vary from source to source. Even individual organizations use search engines from different vendors to index their internal document collections. Therefore, using the wealth of available resources effectively presents challenging problems. This thesis focuses primarily on how to help users find and use the information that they need.

Increasingly, users want to issue complex queries across Internet sources to obtain the data they require. Because of the size of the Internet, it is not possible anymore to process such queries in naive ways, e.g., by accessing all the available sources. Thus, we must process queries in a way that *scales with the number of sources.* Also, sources vary in the type of information objects they contain and in the interface they present to their users. Some sources contain text documents and support simple query models where a query is just a list of keywords. Other sources contain more structured data and provide query interfaces in the style of relational database interfaces. User queries might require accessing sources supporting radically different interfaces and query models. Thus, we must process queries in a way that *deals with heterogeneous sources.*

Users can benefit from *metasearchers*, which are services that provide unified query interfaces to multiple search engines. Thus, users have the illusion of a single combined document source. A metasearcher (or any end client, in general) would typically issue queries to multiple sources, for which it needs to perform three main tasks. First, the metasearcher chooses the best sources to evaluate a query. Then, it submits the query to these sources. Finally, it merges the results from the sources and presents them to the user that issued the query. Note that sources often rank the documents in the query results from "best" to "worst" for the query by using undisclosed algorithms.

Building metasearchers is a hard task because different search engines are largely incompatible and do not allow for interoperability. Building metasearchers is also hard because in general sources are too numerous. Therefore, finding the best sources for a query is a challenging task. Finally, even if we know the ranking algorithms that sources use, extracting the best objects for a query according to the metasearcher might be an expensive operation, since the sources' ranking algorithms might differ radically from that of the metasearcher's.

In this thesis, we facilitate the construction of metasearchers by developing the following key technologies:

- *STARTS* (Chapter 2): It is hard for a metasearcher to perform the tasks above with no cooperation from the information sources. Thus, a metasearcher needs at least metadata about the sources' contents (for query translation and source discovery), and statistics about the query results that the sources return (for meaningful result merging). This need for cooperation led to the design of *STARTS*, an emerging protocol for Internet retrieval and search that facilitates the three tasks of metasearchers. *STARTS* has been developed in a unique way. It is not a standard, but a group effort coordinated by Stanford's Digital Library project, and involving around 11 companies and organizations.

- *GlOSS* (Chapters 3 and 4): Once sources cooperate with metasearchers by exporting the content summaries specified by the *STARTS* protocol, metasearchers use these summaries to define their source discovery strategy. One such strategy is the one followed by *GlOSS* (Glossary-Of-Servers Server). *GlOSS* is a system that keeps statistics on the available information sources. A metasearcher can then use *GlOSS* to estimate which sources are the potentially most useful for a given query. We define *GlOSS* for sources supporting either the Boolean or the vector-space model of document retrieval [SM83]. We also generalize our approach by showing how to build a hierarchy of *GlOSS* servers. The top level of the hierarchy is so small it could be widely replicated, even at end-user workstations.

- *A result-merging condition* (Chapter 5): Once a metasearcher has chosen what sources to contact for a given query, it also has to decide how much data to retrieve from each of these sources to find the "best" answers for the query. In effect, a crucial problem that a metasearcher faces is extracting from the underlying sources the top objects for a user query according to the metasearcher's ranking function. We present a condition that a source must satisfy so that a metasearcher can extract the top objects for a query from the source without examining its entire contents. Not only is this condition necessary but it is also sufficient, and we show an algorithm to extract the top objects from sources that satisfy the given condition.

- *dSCAM* (Chapter 6): The metasearching paradigm is applicable for novel applications. One such application is to automatically detect when a "new" document is "suspiciously close" to existing ones. This problem has become of crucial importance, because the Internet has made the illegal dissemination of copyrighted material easy. In this scenario, the information sources contain "registered documents," and the "queries" are actually new documents for which we want to find suspicious documents, i.e., we want to find documents that overlap with the queries significantly. To address this problem we developed *dSCAM*, which is an "illegal copy" metasearcher that uses the *GlOSS* approach for finding the "most suspicious" text sources for a query.

We conclude this thesis by discussing related work in Chapter 7, and challenging open problems in Chapter 8.

Chapter 2

STARTS: A Protocol for Metasearching

As mentioned in the previous chapter, it is hard to build metasearchers over text sources because the search engines that these sources use are incompatible, making interoperability difficult. *STARTS*, the *Stanford Protocol Proposal for Internet Retrieval and Search*, is an emerging protocol whose goal is to facilitate the main three metasearching tasks of Chapter 1 [GCGMP97]:

- Choosing the best sources to evaluate a query
- Evaluating the query at these sources
- Merging the query results from these sources

STARTS has been developed in a unique way. It is not a standard, but a group effort involving around 11 companies and organizations. The objective of this chapter is not only to give an overview of the *STARTS* protocol proposal, but also to discuss the process that led to its definition. In particular:

- We will describe the history of the project, including the current status of a reference implementation, and will highlight some of the existing "tensions" between information providers and search engine builders (Sections 2.1 and 2.2).

- We will explain the protocol, together with some of the tradeoffs and compromises that we had to make in its design (Section 2.3).

2.1 History of our Proposal

The Digital Library project at Stanford coordinated search engine vendors and other key players to informally design a protocol that would allow searching and retrieval of information from distributed and heterogeneous sources. We were initially contacted by Steve Kirsch, president of Infoseek (`http://www.infoseek.com`), in June, 1995. His idea was that Stanford should collect the views of the search engine vendors on how to address the problem at hand. Then Stanford, acting as an unbiased party, would design a protocol proposal that would reconcile the vendors' ideas. The key motivation behind this informal procedure was to avoid the long delays usually involved in the definition of formal standards.

In July, 1995, we started our effort with five companies: Fulcrum (`http://www.-fulcrum.com`), Infoseek, PLS (`http://www.pls.com`), Verity (`http://www.verity.-com`), and WAIS. Microsoft Network (`http://www.msn.com`) joined the initial group in November. We circulated a preliminary draft describing the main three problems that we wanted to address (i.e., choosing the best sources for a query, evaluating the query at these sources, and merging the query results from the sources). We scheduled meetings with people from the companies to discuss these problems and get feedback. We met individually with each company between December, 1995, and February, 1996. During each meeting, we would briefly address each of the three problems to agree on their definition, terminology, etc. After this, we would discuss the possible solutions for each problem in detail.

Based on the comments and suggestions that we received, we produced a first draft of our proposal by March, 1996. We then produced two revisions of this draft using feedback from the original companies, plus other organizations that started participating: Excite (`http://www.excite.com`), GILS (`http://info.er.usgs.-gov:80/gils/`), Harvest (`http://harvest.transarc.com`), Hewlett-Packard Laboratories (`http://www.hpl.hp.com`), and Netscape (`http://www.netscape.com`). Finally we held a workshop at Stanford with the major participants on August 1st, 1996. The goal of this one-day workshop was to iron out the controversial aspects of the proposal, and to get feedback for its final draft [GCGMP96].

Defining *STARTS* has been a very interesting experience: we wanted to design a protocol that would be simple, yet powerful enough to allow us to address the three problems at hand. We could have adopted a "least common denominator" approach for our solution. However, many interesting interactions would have been impossible under such a solution. Alternatively, we could have incorporated the sophisticated features that the search engines provide, but that also would have challenged interoperability, and would have driven us away from simplicity. Consequently, we had to walk a very fine line, trying to find a solution that would be expressible enough, but not too complicated or impossible to quickly implement by the search engine vendors.

Another aspect that made the experience challenging was dealing with companies that have undisclosed, proprietary algorithms, as those for ranking documents. (See Section 2.3.2.) Obviously, we could not ask the companies to reveal these algorithms. However, we still needed to have them export enough information so that a metasearcher could do something useful with the query results.

As mentioned above, the *STARTS*-1.0 specification is already completed. A reference implementation of the protocol has been built at Cornell University by Carl Lagoze. (See `http://www-diglib.stanford.edu` for information.) Also, the Z39.50 community is designing a profile of their Z39.50-1995 standard based on *STARTS*. (This profile was originally called *ZSTARTS*, but has since changed its name to *ZDSR*, for Z39.50 Profile for Simple Distributed Search and Ranked Retrieval.)

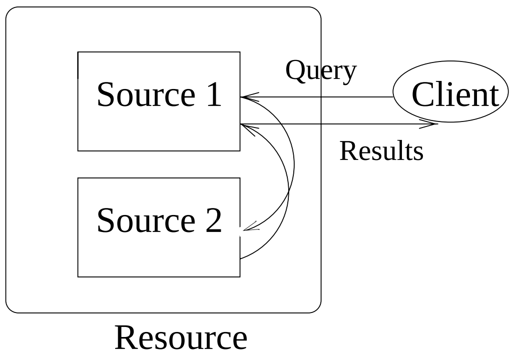

Figure 2.1: A metasearcher queries a source, and may specify that the query be evaluated at several sources at the same resource.

2.2 Our Metasearch Model and its Associated Problems

In this section we expand on the basic metasearch model of Chapter 1, and on the three main problems that a metasearcher faces today. These problems motivated the *STARTS* effort.

For the purpose of the *STARTS* protocol, we view the Internet as a potentially large number of *resources* (e.g., Knight-Ridder's Dialog information service, or the NCSTRL sources [1]). Each resource consists of one or more *sources* (Figure 2.1). A source is a collection of text documents (e.g., Inspec and the Computer Database in the Dialog resource), with an associated search engine that accepts queries from clients and produces results. We assume that documents are "flat," in the sense that we do not, for example, allow any nesting of documents. We do not consider non-textual documents or data either (e.g., geographical data) to keep the protocol simple. Sources may be "small" (e.g., the collection of papers written by some university professor) or "large" (e.g., the collection of World-Wide Web pages indexed by a crawler).

As described in Chapter 1, a metasearcher (or any end client, in general) would typically issue queries to multiple sources. To query multiple sources within the

[1]The NCSTRL sources constitute an emerging library of computer science technical reports (`http://www.ncstrl.org`).

same resource (e.g., as possible in Knight-Ridder's Dialog information service), the metasearcher issues the query to one of the sources at the resource (Source 1 in Figure 2.1), specifying the other "local" sources where to also evaluate the query (Source 2 in Figure 2.1). This way, the resource can eliminate duplicate documents from the query result, for example, which would be difficult for the metasearcher to do if it queried all of the sources independently.

Building metasearchers that query multiple sources is a hard task because different search engines are largely incompatible and do not allow for interoperability. In general, text search engines:

- Use different query languages (*the query-language problem*; Section 2.2.1)
- Rank documents in the query results using secret algorithms (*the rank-merging problem*; Section 2.2.2)
- Do not export information about the sources in a standard form (*the source-metadata problem*; Section 2.2.3)

Below we visit each of these metasearch problems. The discussion will illustrate the need for an agreement between search engine vendors so that metasearchers can work effectively.

2.2.1 The Query-Language Problem

A metasearcher submits queries over multiple sources. But the interfaces and capabilities of these sources may vary dramatically. Even the basic query model that the sources support may vary.

Some search engines (e.g., Glimpse) only support the *Boolean retrieval model* [Sal89]. In this model, a query is a condition that documents either do or do not satisfy. The query result is then a *set* of documents. For example, a query *distributed and systems* returns all documents that contain both the words *distributed* and *systems* in them.

Alternatively, most commercial search engines also support some variation of the *vector-space retrieval model* [Sal89]. In this model, a query is a list of terms, and documents are assigned a score according to how *similar* they are to the query. The query result is then a *rank* of documents. For example, a query *distributed systems* returns a rank of documents that is typically based on the number of occurrences of the words *distributed* and *systems* in them. [2] A document in the query result might contain the word *distributed* but not the word *systems*, for example, or vice versa, unlike in the Boolean-model case above.

Even if two sources support a Boolean retrieval model, their query syntax often differ. A query asking for documents with the words *distributed* and *systems* might be expressed as "`distributed and systems`" in one source, and as "`+distributed +systems`" in another, for example.

More serious problems appear if different fields (e.g., *abstract*) are available for searching at different sources. For example, a source might support queries like `(abstract "databases")` that ask for documents that have the word *databases* in their abstract, whereas some other sources might not support the *abstract* field for querying.

Another complication results from different stemming algorithms or stop-word lists being implicit in the query model of each source. (Stemming is used to make a query on *systems* also retrieve documents on *system*, for example. Stop words are used to not process words like *the* in the queries, for example.) If a user wants documents about the rock group *The Who*, knowing about the stop-word behavior of the sources would allow a metasearcher, for example, to know whether it is possible to disallow the elimination of stop words from queries at each source.

As a result of all this heterogeneity, a metasearcher would have to translate the original query to adjust it to each source's syntax. To do this translation, the metasearcher needs to know the characteristics of each source. (The work in [CGMP96a, CGMP96b] illustrates the complexities involved in query translation.) As we will see in Section 2.3.1, querying multiple sources is much easier if the sources support some common query language. Even if support for most of this language is optional, query translation is much simpler if sources reveal what portions of the language they support.

[2]These ranks also typically depend on other factors, like the number of documents in the source that contain the query words, for example.

2.2.2 The Rank-Merging Problem

A source that supports the vector-space retrieval model ranks its documents according to how "similar" the documents and a given query are. Unfortunately, there are many ways to compute these similarities. To make matters more complicated, the ranking algorithms are usually proprietary to the search engine vendors, and their details are not publicly available.

Merging query results from sources that use different and unknown ranking algorithms is hard. (See Chapter 5.) For example, source S_1 might report that document d_1 has a *score* of 0.3 for some query, while source S_2 might report that document d_2 has a score of 1,000 for the same query. If we want to merge the results from S_1 and S_2 into a single document rank, should we rank d_1 higher than d_2, or vice versa? (Some search engines are designed so that the top document for a query always has a score of, say, 1,000.)

It is even hard to merge query results from sources that use the same ranking algorithm, even if we know this algorithm. The reason is that the algorithm might rank documents differently based on the collection where the document appears. For example, if a source S_1 specializes in computer science, the word *databases* might appear in many of its documents. Then, this word will tend to have a low associated weight in S_1 (e.g., if S_1 uses the *tf·idf* formula for computing weights [Sal89]). The word *databases*, on the other hand, might have a high associated weight in a source S_2 that is totally unrelated to computer science and contains very few documents with that word. Consequently, S_1 might assign its documents a low score for a query containing the word *databases*, while S_2 assigns a few documents a high score for that query. Therefore, it is possible for two very similar documents d_1 and d_2 to receive very different scores for a given query, if d_1 appears in S_1 and d_2 appears in S_2. Thus, even if the sources use the same ranking algorithm, a metasearcher still needs additional information to merge query results in a meaningful way.

2.2.3 The Source-Metadata Problem

A metasearcher might have thousands of sources available for querying. Some of these sources might charge for their use. Some of the sources might have long response times. Therefore, it becomes crucial that the metasearcher just contact sources that might contain useful documents for a given query. (See Chapters 3 and 4.) The metasearcher then needs information about each source's contents.

Some sources freely deliver their entire document collection, whereas others do not. Often, those sources that have for-pay information are of the second type. If a source exports all of its contents (e.g., many World-Wide Web sites), then it is not as critical to have it describe its collection to the metasearchers. After all, the metasearchers can just grab all of the sources' contents and summarize them any way they want. This is what "crawlers" like AltaVista (`http://www.altavista.digital.com`) do. However, for performance reasons, it may still be useful to require that such sources export a more succinct description of themselves. In contrast, if a source "hides" its information (e.g., through a search interface), then it is even more important that the source can describe its contents. Otherwise, if a source does not export any kind of content summary, it becomes hard for a metasearcher to assess what kind of information the source covers.

2.2.4 Metasearch Requirements

In summary, a sophisticated metasearcher will need to perform the following tasks in order to efficiently query multiple resources:

- Extract the list of sources from the resources periodically (to find out what sources are available for querying) (Section 2.3.3)

- Extract metadata and content summaries from the sources periodically (to be able to decide what sources are potentially useful for a given query) (Section 2.3.3)

Also, given a user query:

- Issue the query to one or more sources at one or more resources (Sections 2.3.1 and 2.3.3)
- Get the results from the multiple resources, merge them, and present them to the user (Section 2.3.2)

2.3 Our Protocol Proposal

In this section we define a protocol proposal that addresses the metasearch requirements of Section 2.2.4. This protocol is meant for machine-to-machine communication: users should not have to write queries using the proposed query language, for instance. Also, all communication with the sources is sessionless in our protocol, and the sources are stateless. Finally, we do not deal with any security issues, or with error reporting in our proposal. The main motivation behind these (and many of the other) decisions is to keep the protocol simple and easy to implement.

Our protocol does not describe an architecture for metasearching. However, it does describe the facilities that a source needs to provide in order to help a metasearcher. The facilities provided by a source can range from simple to sophisticated, and one of the key challenges in developing our protocol was in deciding the right level of sophistication. In effect, metasearchers often have to search across simple sources as well as across sophisticated ones. On the one hand, it is important to have some agreed-upon minimal functionality that is simple enough for all sources to comply with. On the other hand, it is important to allow the more sophisticated sources to export their richer features. Therefore, our protocol keeps the requirements to a minimum, while it provides optional features that sophisticated sources can use if they wish.

Our protocol mainly deals with *what information* needs to be exchanged between sources and metasearchers (e.g., a query, a result set), and not so much with *how* that information is formatted (e.g., using Harvest SOIFs [3]) or transported (e.g.,

[3]SOIF objects are typed, ASCII-based encodings for structured objects; see `http://-harvest.transarc.com/afs/transarc.com/public/trg/Harvest/user-manual/`.

using HTTP). Actually, what transport to use generated some heated debate during the *STARTS* workshop. Consequently, we expect the *STARTS* information to be delivered in multiple ways in practice. For concreteness, the *STARTS* specification and examples that we give below use SOIFs just to illustrate how our content can be delivered. However, *STARTS* includes mechanisms to specify other formats for its contents [GCGMP96].

2.3.1 Query Language

In this section we describe the basic features of the query language that a source should support. To cover the functionality offered by most commercial search engines, queries have both a Boolean component: the *filter expression*, and a vector-space component: the *ranking expression*. Also, queries have other associated properties that further specify the query results. For example, a query specifies the maximum number of documents that should be returned, among other things.

Filter and Ranking Expressions

Queries have a filter expression (the Boolean component) and a ranking expression (the vector-space component). The *filter expression* specifies some condition that must be satisfied by every document in the query result (e.g., all documents in the answer must have *Ullman* as one of the authors). The *ranking expression* specifies words that are desired, and imposes an order over the documents in the query result (e.g., the documents in the answer will be ranked according to how many times they contain the words *distributed* and *databases* in their body).

Example 1: Consider the following query with filter expression:

```
((author "Ullman") and (title "databases"))
```

and ranking expression:

```
list((body-of-text "distributed") (body-of-text "databases"))
```

This query returns documents having *Ullman* as one of the authors and the word *databases* in their title. The documents that match the filter expression are then ranked according to how well their text matches the words *distributed* and *databases*. ∎

In principle, a query need not contain a filter expression. If this is the case, we assume that all documents qualify for the answer, and are ranked according to the ranking expression. Similarly, a query need not contain a ranking expression. If this is the case, the result of the query is the set of objects that match the (Boolean) filter expression. Some search engines only support filter or ranking expressions, but not both (e.g., Glimpse only supports filter expressions). Therefore, we allow sources to support just one type of expression. In this case, the sources indicate (Section 2.3.3) what type they support as part of their metadata.

Both the filter and the ranking expressions may contain multiple terms. The filter and ranking expressions combine these terms with operators like "`and`" and "`or`"(e.g., `((author "Ullman") and (title "databases"))`). The ranking expressions also combine terms using the "`list`" operator, which simply groups together a set of terms, as in Example 1. Also, the terms of a ranking expression may have a weight associated with them, indicating their relative importance in the ranking expression.

In defining the expressive power of the filter and ranking expressions we had to balance the needs of search engine builders and metasearchers. On the one hand, builders in general want powerful expressions, so that all the features of their engine can be called upon. On the other hand, metasearchers want simpler filter and ranking expressions, because they know that not all search engines support the same advanced features. The simpler the filter and ranking expressions are, the more likely it is that engines will have common features, and the easier it will be to interoperate. Also, those metasearchers whose main market is Internet searching prefer simple expressions because most of their customers use simple queries. In contrast, search engine builders cater to a broader mix of customers. Some of these customers require sophisticated query capabilities.

Next we define the filter and ranking expressions more precisely. We start by defining the *l-strings*, which are the basic building blocks for queries. Then we show how these strings are adorned with fields and modifiers to build atomic terms. Finally, we describe how to construct complex filter and ranking expressions.

Atomic Terms

One of the most heavily discussed issues in our workshop was how to support multiple languages and character sets. Our initial design had not supported queries using multiple character sets or languages. However, the search engine vendors felt strongly against this limitation. So, we decided early on in our workshop to include multi-lingual/character support, but the question was how far to go. For example, did we want to support a query asking for documents with the Spanish word *taco*? Did we also want to handle queries asking for documents whose abstract was in French, but that also included the English word *weekend*? Another issue was how to handle dialects, e.g., how to specify that a document is written, say, in British English vs. in American English.

During the workshop we also discussed whether we could make the multi-language support invisible to those who just wanted to submit English queries. That is, we do not want to specify English explicitly everywhere if no other language is used. The design we settled on does allow English and ASCII as the defaults, while giving the query writer substantial power to specify languages and character sets used.

A *term* in our query language is an *l-string* modified by an unordered list of *attributes* (e.g., `(author "Ullman")`). To allow queries in languages other than English, an l-string is either a string (e.g., `"Ullman"`), or a string qualified with its associated language and, optionally, with its associated country. For example, `[en-US "behavior"]` is an l-string, meaning that the string "behavior" represents a word in American English. The language-country qualification follows the format described in RFC 1766 (`http://andrew2.andrew.cmu.edu/rfc/rfc1766.html`). (Countries are optional.) To support multiple character sets, the actual string in an l-string is a Unicode sequence encoded using UTF-8. A nice property of this encoding is that the code for a plain English string is the ASCII string itself, unmodified.

An attribute is either a *field* or a *modifier*. The term `(date-last-modified > "1996-08-01")`, for example, has field `date-last-modified` and modifier `>`. This term matches documents that were modified after August 1, 1996.

To make interoperability easier, we decided to define a "recommended" set of attributes that sources should try to support. This set needed to be large enough so that users can express their queries. At the same time, the set needed to be simple enough to not compromise interoperability. The choice of the recommended attribute set was fodder for heated discussion, especially around what attributes we should require the sources to support. In effect, requiring that sources support some attributes would make the protocol more expressive, but harder to adhere to by the sources.

We considered several candidate attribute sets that had already been defined within different standards efforts. (See Section 7.2.) Unfortunately, none of the existing attribute sets contained just the attributes that we needed, as determined from our discussions. Therefore, we decided to pick the GILS [4] attribute set [Chr97], which in turn inherits all of the Z39.50-1995 Bib-1 use attributes [Org95]. The GILS set contained most of the attributes that we needed, and we simplified it to include only those attributes. We also added a few attributes that were not in the GILS set but that were considered necessary in our discussions.

Below is the "Basic-1" set of attributes (i.e., fields and modifiers), which are the attributes that we recommend that sources support. The attributes not marked as new are from the GILS attribute set. In [GCGMP96] we explain how to use other attribute sets for sources covering different domains, for example.

- **Fields**: A field specifies what portion of the document text is associated with the term (e.g., the author portion, the title portion, etc.). At most one should be specified for each term. If no field is specified, `"Any"` is assumed. Those fields marked as required must be supported, meaning that the source must recognize these fields. However, the source may freely interpret them. The

[4]The Government Information Locator Service, GILS, is an effort to facilitate access to governmental information.

rest of the fields are optional. (Our fields correspond to the Z39.50/GILS "use attributes.")

Field	*Required?*	*New?*
`Title`	Yes	No
`Author`	No	No
`Body-of-text`	No	No
`Document-text`	No	Yes
`Date/time-last-modified`	Yes	No
`Any`	Yes	No
`Linkage`	Yes	No
`Linkage-type`	No	No
`Cross-reference-linkage`	No	No
`Languages`	No	No
`Free-form-text`	No	Yes

The `Document-text` field provides a way to pass documents to the sources as part of the queries, which could be useful to do *relevance feedback* [Sal89]. Relevance feedback allows users to request documents that are similar to a document that was found useful.

The value of the `Linkage` field of a document is its URL, and it is returned with the query results so that the document can be retrieved outside of our protocol.

The `Linkage-type` of a document is its MIME type. The list of the URLs that are mentioned in the document is reported in its `Cross-reference-linkage`.

The `Free-form-text` field provides a way to pass to the sources queries that are not expressed in our query language, adding flexibility to our proposal. A search engine vendor asked for this capability so that informed metasearchers could use the sources' richer native query languages, for example.

- **Modifiers**: A modifier specifies what values the term represents (e.g., treat the term as a stem, as its phonetics (soundex), etc.). Zero or more modifiers can be specified for each term. All the modifiers below are optional, i.e., the search engines need not support them. (Our modifiers correspond to the Z39.50 "relation attributes.")

Modifier	*Default*	*New?*
`<, <=, =, >=, >, !=`	`=`	No
`Phonetic`	No soundex	No
`Stem`	No stemming	No
`Thesaurus`	No thesaurus expansion	Yes
`Right-truncation`	No right truncation	No
`Left-truncation`	No left truncation	No
`Case-sensitive`	Case insensitive	Yes

The `<, <=, =, >=, >, !=` modifiers only make sense for fields like "`Date/time--last-modified`," for example.

Example 2: Consider the following filter expression:

```
(title stem "databases")
```

The documents that satisfy this expression have the word *databases* in their title, or some other word with the same stem, like *database*. ▮

Complex Filter Expressions

Our complex filter expressions are based on a simple subset of the type-101 queries of the Z39.50-1995 standard. We use operators to build complex filter expressions from the terms. As with the attributes, we wanted to choose a set of operators that would both be easy to support and be sufficiently expressive. The "Basic-1"-type filter expressions use the following operators. If a source supports filter expressions, it must support all these operators.

- `and`

- `or`
- `and-not`
- `prox`, specifying two terms, the required distance between them, and whether the order of the terms matters.

Example 3: Consider two terms t_1 and t_2 and the following filter expression:

$$(t_1\ \texttt{prox[3,T]}\ t_2)$$

The documents that match this filter expression contain t_1 followed by t_2 with at most three words in between them. `"T"` (for "true") indicates that the word order matters (i.e., that t_1 has to appear before t_2). ∎

Note that `not` is not one of our operators, to prevent users from asking for documents with the sole qualification that they not contain the word *databases* in them, for example. Such a query would be too expensive to evaluate. Instead, we have the `and-not` operator. Thus, all queries always have a "positive" component.

The proximity operator is an interesting example of a compromise that we had to reach: some search engine vendors found our initial proposal, which allowed for unidirectional or bidirectional "paragraph" and "sentence" distance, for example, unacceptably complicated to implement. Later, we simplified the proximity operator to only allow for unidirectional word distance. A search engine vendor still thought that this operator was too complicated, while other participants, especially information providers, found it unreasonably limiting. We finally managed to agree on the current specification.

Complex Ranking Expressions

We also use operators to build complex ranking expressions from terms. The "Basic-1"-type ranking expressions use the operators above ("`and`," "`or`," "`and-not`," and "`prox`") plus a new operator, "`list`," which simply groups together a set of terms.

The "`list`" operator represents the most common way of constructing vector-space queries: these queries are typically just flat lists of terms. Our original design did not allow for any other operators in the ranking expressions. However, some search engine vendors felt that this language was not expressive enough, and asked that the Boolean-like operators be included. If a source supports ranking expressions, it must now support all these operators. But again, a source might choose to simply ignore the Boolean-like operators from ranking expressions, and process a ranking expression like `("distributed" and "databases")` as if it were `list("distributed" "databases")`.

The Boolean-like operators would most likely be interpreted as "fuzzy-logic" operators by the search engines in order to rank the documents, as the following example illustrates.

Example 4: Consider two ranking expressions:

$$R_1 = \texttt{("distributed" and "databases")}$$
$$R_2 = \texttt{list("distributed" "databases")}$$

Consider a source with a document that has a weight (as determined by the local search engine) of 0.3 for the word *distributed* and a weight of 0.8 for the word *databases*. Then, the search engine might assign the document a score of $\min\{0.3, 0.8\} = 0.3$ for ranking expression R_1 (interpreting the `and` operator as the minimum function). The same engine might use a different scoring algorithm for "`list`" queries with the same terms, and assign the document a score of, say, $0.5 \times 0.3 + 0.5 \times 0.8 = 0.55$ for ranking expression R_2.

Thus, by interpreting the Boolean-like operators and the `list` operator for building ranking expressions differently, sources can provide richer semantics for user queries. ∎

Each term in a ranking expression may have an associated *weight* (a number between 0 and 1), indicating the relative importance of the term in the query.

Example 5: Consider the following ranking expression:

```
list(("distributed" 0.7) ("databases" 0.3))
```

The weights in the expression indicate that the search engines should treat the term `"distributed"` as more important than the term `"databases"` when ranking the documents in the query results. ∎

Further Result Specification

To complete the specification of the query results, our queries include the following information in addition to a filter and a ranking expression:

- **Drop stop words**: whether the source should delete the stop words from the query or not. A metasearcher knows if it can turn off the use of stop words at a source from the source's metadata (Section 2.3.3).
- **Default attribute set and language** used in the query. This is optional, just for notational convenience, since queries may include attributes from attribute sets other than "Basic-1," and terms may correspond to languages other than English.
- **Sources** (in the same resource) where to evaluate the query in addition to the source where the query is submitted (Section 2.2).
- **Answer specification**:
  - Fields to be returned in the query answer (Default: `Title`, `Linkage`)
  - Fields to be used to sort the query results, and whether the order is ascending or descending (Default: `Score` of the documents for the query, in descending order)
  - Documents to be returned:
    - Minimum acceptable document score
    - Maximum acceptable number of documents

A complete query is represented as a list of attribute-value pairs, providing the filter and ranking expressions, the answer specification, etc. Below is an example of such a query encoded using Harvest's SOIF. As discussed in Section 2.3, we encode the *STARTS* information using SOIF here just to illustrate how our query content could be delivered, but other encodings are possible. We describe the formal syntax and format for the queries in [GCGMP96].

Example 6: Below is a SOIF object for a query. The number in brackets after each SOIF attribute (e.g., "48" after the `FilterExpression` SOIF attribute) is the number of bytes of the value for that attribute, to facilitate parsing.

```
@SQuery{
Version{10}: STARTS 1.0
FilterExpression{48}: ((author "Ullman") and
                       (title stem "databases"))
RankingExpression{61}: list((body-of-text "distributed")
                            (body-of-text "databases"))
DropStopWords{1}: T
DefaultAttributeSet{7}: basic-1
DefaultLanguage{5}: en-US
AnswerFields{12}: title author
MinDocumentScore{3}: 0.5
MaxNumberDocuments{2}: 10
}
```

This query specifies that the sources should eliminate any stop words from the filter and ranking expressions before processing them, and that the word *databases* in the filter expression should be stemmed. Then, for example, a document having the word *database* in its title will match subexpression `(title stem "databases")`. The query results should contain the `title` and `author` of the documents, in addition to the `linkage` (URL) of the documents, which is always returned. Also, only documents with a score for the ranking expression of at least 0.5 should be in the answer. Furthermore, only the 10 documents with the top score are to be returned. ∎

2.3.2 Merging of Ranks

There are three types of complications that arise in interpreting query results from multiple sources. One is that each source may execute a different query, depending on its local query capabilities. Thus, a source might ignore parts of a query that it does not support, for example. Another complication is that sources may use different algorithms to rank the documents in the query results. Furthermore, the sources do not reveal their ranking algorithms. A third complication is that the ranking information by itself is insufficient for merging multiple query results, even if all the sources execute the same query using the same ranking algorithm. In effect, the actual document ranks might depend on the contents of each source, as described in Section 2.2.2. We will now discuss how our protocol copes with these issues.

As mentioned above, sources are not required to support all of the features of the query language of Section 2.3.1. So, a source might decide to ignore certain parts of a query that it receives, for example. Then, each source returns the query that it actually processed together with the query results, as the following example illustrates. Since we do not include any way of reporting errors in our protocol, this mechanism assists the metasearchers in interpreting the query results.

Example 7: Consider a source that does not support the ranking-expression part of queries. Consider the query with filter expression:

```
((author "Ullman") and (title stem "databases"))
```

and ranking expression:

```
list((body-of-text "distributed") (body-of-text "databases"))
```

If the source simply ignores the ranking expressions, the actual query that the source processes has filter expression:

```
((author "Ullman") and (title stem "databases"))
```

and an empty ranking expression. This actual query is returned with the query results. ∎

To merge the query results from multiple sources into a single, meaningful rank, a source should return the following information for each document in the query result:

- The unnormalized score of the document for the query
- The id of the source(s) where the document appears
- Statistics about each query term in the ranking expression (as modified by the query fields, if possible):
  - `Term-frequency`: the number of times that the query term appears in the document.
  - `Term-weight`: the weight of the query term in the document, as assigned by the search engine associated with the source (e.g., the normalized *tf.idf* weight [Sal89] for the query term in the document, or whatever other weighing of terms in documents the search engine might use).
  - `Document-frequency`: the number of documents in the source that contain the term. This information is also provided as part of the metadata for the source.

Also:

- `Document-size`: the size of the document (in KBytes)
- `Document-count`: the number of tokens (as determined by the source) in the document

The results for a query start with a SOIF object of type "`SQResults`," followed by a series of SOIF objects of template type "`SQRDocument`." Each of the latter objects corresponds to a document in the query result.

Example 8: The result for the query of Example 6 from the Source-1 source may look like the following.

```
@SQResults{
Version{10}: STARTS 1.0
Sources{8}: Source-1
ActualFilterExpression{48}: ((author "Ullman") and
                             (title stem "databases"))
ActualRankingExpression{26}: (body-of-text "databases")
NumDocSOIFs{1}: 1
}

@SQRDocument{
Version{10}: STARTS 1.0
RawScore{4}: 0.82
Sources{8}: Source-1
linkage{47}: http://www-db.stanford.edu/~ullman/pub/dood.ps
title{68}: A Comparison Between Deductive and Object-Oriented
           Database Systems
author{18}: Jeffrey D. Ullman
TermStats{89}: (body-of-text "distributed") 10 0.31 190
               (body-of-text "databases") 15 0.51 232
DocSize{3}: 248
DocCount{5}: 10213
}
```

The first SOIF object reports properties of the entire query result. For example, we learn from the value of `ActualRankingExpression` that Source-1 eliminated the term `(body-of-text "distributed")` from the ranking expression. Presumably, the word *distributed* is a stop word at Source-1. We also find out that there is only one document in the query result. All of the other documents in Source-1 either do not satisfy the filter expression, or have a score lower than 0.5 for the ranking expression.

The second SOIF object corresponds to the only document in the query result. This document, whose URL is given as the value for the `linkage` attribute, has a score of 0.82 for the ranking expression, and satisfies the filter expression. In effect, the word *database* appears in the document's title (*database* shares its stem with *databases*), and *Ullman* is one of the authors of the document.

The document SOIF object also contains statistics about the document, which are crucial for rank merging. For example, we know that the word *distributed* appears 10 times in the document, and the word *databases* 15 times. The size of the document is 248 KBytes, and there are 10,213 words in it. ▮

Using all this information, a metasearcher can then re-rank the documents that it obtained from the various sources, following its own criteria and without actually retrieving the documents, as the following example illustrates.

Example 9: Consider the following SOIF object describing the only document in the result for the query of Example 6 from source Source-2.

```
@SQRDocument{
Version{10}: STARTS 1.0
RawScore{4}: 0.27
Sources{8}: Source-2
linkage{37}: http://elib.stanford.edu/lagunita.ps
title{73}: Database Research: Achievements and Opportunities
           into the 21st. Century
author{48}: Avi Silberschatz, Mike Stonebraker, Jeff Ullman
TermStats{89}: (body-of-text "distributed") 20 0.12 901
               (body-of-text "databases") 34 0.15 788
DocSize{3}: 125
DocCount{4}: 9031
}
```

This document has a lower score than the document from Source-1 of Example 8. However, the Source-2 document might be a better match for the query than the Source-1 document, and the lower score could just be an artifact of the ranking algorithm that the sources use, or be due to the characteristics of the holdings of both sources. A metasearcher could then simply discard the sources' scores, and compute a new score for each document based on, say, the number of times that the words in the ranking expression appear in the documents. Then, such a metasearcher would rank the Source-2 document higher than the Source-1 document, since the former document contains the words *distributed* and *databases* 20 and 34 times, respectively, whereas the latter document only contains these words 10 and 15 times, respectively. ∎

Example 9 shows one simple-minded way in which a metasearcher can re-rank documents from multiple query results. More sophisticated schemes could also use the document frequencies of the query terms, for example. However, there are still unresolved issues when merging document ranks from multiple sources. For example, one possibility is to rank documents as if they all belonged in a single, large document source. Alternatively, we could use information about the originating sources to design the final document rank. The goal of our protocol is not to resolve these issues, but simply to provide the "raw material" so that metasearchers can experiment with a variety of formulas and approaches for combining multiple query results.

From our discussions with the search engine vendors, it became clear that it would be hard for some of them to provide the statistics above with their query results. The reason is that by the time the results are returned to the user, these statistics, which are typically used to compute the document scores, are lost. Since returning just the document scores with the query results is not enough for rank merging, we are asking sources to at least provide the query results for a given *sample document collection* and a given set of queries as part of their metadata. This way, the metasearchers would treat each source as a "black box" that receives queries and produces document ranks. However, the metasearchers would try to approximate how each source ranks documents using their knowledge of what is in the sample collection. So, if the sample queries are carefully designed, the metasearchers might be able to draw some conclusions on how to calibrate the query results in order to produce a single document rank.

2.3.3 Source Metadata

To select the right sources for a query and to query them we need information about the sources' contents and capabilities. In this section we describe two pieces of metadata that every source is required to export: a list of metadata attribute-value pairs, describing properties of the source, and a content summary of the source. Each piece is a separate object, to allow metasearchers to retrieve just the metadata that they need. For simplicity, each of these two pieces is retrieved as a single "blob." We do not ask sources to support more sophisticated interfaces, like a search interface, to export this data.

In this section we also describe the information that a resource exports. This information identifies the metadata for the sources in the resource.

Source Metadata Attributes

Each source exports information about itself by giving values to the metadata attributes below. A metasearcher can use this information to rewrite the queries that it sends to each source, since each source may support different parts of the query language of Section 2.3.1, for example.

As with the attribute sets for documents, several attribute sets have been defined to describe sources. Unfortunately, none of these sets contain exactly the attributes that we need, as determined from our discussions. Therefore, we defined the "MBasic-1" set of metadata attributes, borrowing from two well known attribute sets, the Z39.50-1995 Exp-1 and the GILS attribute sets. We added a few attributes, marked as new below, that are not in these two attribute sets, and that the participating organizations concluded were necessary. Some attributes are marked as required, and the sources must support them.

Field	*Required?*	*New?*
`FieldsSupported`	Yes	Yes
`ModifiersSupported`	Yes	Yes
`FieldModifierCombinations`	Yes	Yes
`QueryPartsSupported`	No	Yes
`ScoreRange`	Yes	Yes
`RankingAlgorithmID`	Yes	Yes
`TokenizerIDList`	No	Yes
`SampleDatabaseResults`	Yes	Yes
`StopWordList`	Yes	Yes
`TurnOffStopWords`	Yes	Yes
`SourceLanguages`	No	No
`SourceName`	No	No
`Linkage`	Yes	No
`ContentSummaryLinkage`	Yes	Yes
`DateChanged`	No	No
`DateExpires`	No	No
`Abstract`	No	No
`AccessConstraints`	No	No
`Contact`	No	No

The `FieldsSupported` attribute for a source lists the optional fields (Section 2.3.1) that are supported at the source for querying, in addition to the required fields like `Linkage` and `Title`. Also, each field is optionally accompanied by a list of the languages that are used in that field in the source. Required fields can also be listed here with their corresponding language list.

Similarly, the `ModifiersSupported` attribute lists the modifiers (Section 2.3.1) that are supported at a source. Each modifier is optionally accompanied by a list of the languages for which it is supported at the source, since modifiers like `Stem` are language dependent.

To keep the metadata objects simple, we do not require sources to indicate what supported fields are actually searchable, as opposed to being only retrievable. For example, a source might report the `Language` of each document in the query results, although it might not accept queries involving that field. However, we do ask sources to report what combinations of fields and modifiers are legal at the source in the `FieldModifierCombinations` attribute. For example, asking that an author name be stemmed might be illegal at a source, even if the `Author` field and the `Stem` modifier are supported in other contexts at the source.

The `QueryPartsSupported` attribute specifies whether the source supports ranking expressions only, filter expressions only, or both.

The `ScoreRange` attribute lists the minimum and maximum score that a document can get for a query at the source (including $\Leftrightarrow\infty$ and $+\infty$); we use this information for merging ranks, to interpret the scores that come from the sources.

The `RankingAlgorithmID` attribute contains some form to identify the ranking algorithm the source uses. Even when we do not know the actual algorithm used it is useful to know that two sources use the same algorithm (e.g., `Acme-1`), for merging ranks.

The `TokenizerIDList` attribute has values like `(Acme-1 en-US) (Acme-2 es)`, for example, meaning that the source uses tokenizer Acme-1 to extract the indexable tokens from strings in American English, and tokenizer Acme-2 for strings in Spanish. The inclusion of this metadata attribute was controversial: our original proposal required that sources export a list of the characters that they used as token separators (e.g., “ ,;.”, meaning that a blank space, a comma, etc., are used to delimit tokens). Obviously this information would not be sufficient to completely specify the tokens at each source, but it could at least help metasearchers decide if a query on *Z39.50* should include this term as is, or should instead contain two terms, namely *Z39* and *50*, for example, as would be the case if “.” were a separator. Alternatively, it was proposed that sources export some regular expression describing what their tokens looked like. Both of these proposals were not general enough to describe tokens for arbitrary languages and character sets, and were deemed too complicated to support. Therefore, we settled on the current proposal, which simply requires that sources name their tokenizers. This way, a metasearcher can learn how a particular tokenizer works by submitting a query to a source that uses it, and examining the actual query that the source processes, as specified in the query results (Section 2.3.2). A metasearcher would need to do this not on a source-by-source basis, but only once per tokenizer.

The `SampleDatabaseResults` attribute provides the URL to get the query results for a sample document collection (Section 2.3.2).

The `Linkage` attribute reports the URL where the source should be queried, while the `ContentSummaryLinkage` attribute gives the URL of the content summary of the source.

Example 10: Consider the following SOIF object with some of the metadata attributes for a source Source-1:

```
@SMetaAttributes{
Version{10}: STARTS 1.0
SourceID{8}: Source-1
FieldsSupported{17}: [basic-1 author]
ModifiersSupported{19}: {basic-1 phonetics}
FieldModifierCombinations{39}: ([basic-1 author] {basic-1 phonetics})
QueryPartsSupported{2}: RF
ScoreRange{7}:  0.0 1.0
RankingAlgorithmID{6}: Acme-1
...
DefaultMetaAttributeSet{8}: mbasic-1
source-languages{8}: en-US es
source-name{17}: Stanford DB Group
linkage{26}: http://www-db.stanford.edu/cgi-bin/query
content-summary-linkage{38}: ftp://www-db.stanford.edu/cont_sum.txt
date-changed{9}: 1996-03-31}
```

This source supports the `Author` field for searching, in addition to the required fields, and the `Phonetics` modifier. It also accepts queries with both filter and ranking expressions, and the document scores it produces range from 0 to 1. Source-1 contains documents in American English (`en-US`) and Spanish (`es`). Queries should be submitted to this source at `http://www-db.stanford.edu/cgi-bin/query`, and its content summary is available at `ftp://www-db.stanford.edu/cont_sum.txt`. ∎

Source Content Summary

Content summaries help the metasearchers in choosing the most promising sources for a given query. These summaries could be manually generated, like the ones associated with the `Abstract` metadata attribute. However, this approach usually yields outdated and incomplete summaries, and is a burden on the source administrators.

On the other end of the spectrum, a source summary could simply be the entire contents of the source. This approach is similar to the one taken by several World-Wide Web "crawlers."

In addition to its `Abstract`, we require that each source export partial data about its contents. This data is automatically generated, is orders of magnitude smaller than the original contents, and has proven useful in distinguishing the more useful from the less useful sources for a given query (Chapters 3 and 4). Our content summaries include:

- List of words that appear in the source. This list is preceded by information indicating whether:

  - The words listed are stemmed or not.
  - The words listed include stop words or not.
  - The words listed are case sensitive or not.
  - The words listed are accompanied by the field corresponding to where in the documents they occurred or not (e.g., `(title "algorithm")`).

  If possible, the words listed should not be stemmed, and should include the stop words. Also, the words should be case sensitive, and be accompanied by their corresponding field information, as shown above.

  In addition, the words might be qualified with their corresponding language (e.g., `[en-US "behavior"]`).

- Statistics for each word listed, including at least one of the following:
  - Total number of postings for each word (i.e., the number of times that the word appears in the source)
  - Document frequency for each word (i.e., the number of documents that contain the word)
- Total number of documents in source

Example 11: Consider the following SOIF object with part of the content summary for Source-1 from Example 10:

```
@SContentSummary{
Version{10}: STARTS 1.0
Stemming{1}: F
StopWords{1}: F
CaseSensitive{1}: F
Fields{1}: T
NumDocs{3}: 892

Field{5}: title
Language{5}: en-US
TermDocFreq{11023}: "algorithm" 100 53
                    "analysis" 50 23
...

Field{5}: title
Language{2}: es
TermDocFreq{1211}: "algoritmo" 23 11
                   "datos" 59 12
...
}
```

This content summary reports statistics on unstemmed, case insensitive words that are qualified with field information. For example, the English word *algorithm* appears in the title of 53 documents, while the Spanish word *datos* appears in the title of 12 documents in Source-1. The summary also tells us that there are 892 documents in the source. A metasearcher can use this information to decide whether a given query is likely to have good matches in Source-1, as we will see in Chapters 3 and 4. ▮

Resource Definition

So far, we have focused on *sources*. As discussed in Section 2.2, our model allows several sources to be grouped together as a single *resource* (e.g., Knight-Ridder's Dialog information service). Each resource exports contact information about the sources that it contains. More specifically, a resource simply exports its list of sources, together with the URLs where the metadata attributes for the sources can be accessed and the format of this data. Using this information, a metasearcher learns how and where to contact each of the sources in the resource.

Example 12: Consider the following SOIF object with contact information for a resource. This object reports that there are two sources available for querying at the resource, Source-1 and Source-2, and also gives the URLs where to obtain their corresponding metadata-attribute SOIF objects.

```
@SResource{
Version{10}: STARTS 1.0
SourceList{83}: Source-1 ftp://www.stanford.edu/source_1
                Source-2 ftp://www.stanford.edu/source_2
}
```

▮

2.4 Conclusion

Search engines for text sources do not allow for easy interoperability among them, making it hard to build metasearchers. We believe that the *STARTS* protocol provides simple but fundamental facilities for searching and resource discovery across Internet resources. If implemented, *STARTS* can significantly streamline the construction of metasearchers, as well as enhance the functionality they can offer. We also think that our discussion of issues and "tensions" emerging from our *STARTS* experience can provide useful lessons for anyone dealing with Internet data access.

Chapter 3

GlOSS: Boolean Source Discovery

The dramatic growth of the Internet over the past few years has created a new problem: finding the right text databases (sources or collections) to evaluate a given query. There are thousands of sources available to the users on the Internet, and it is practically impossible for a metasearcher to query all of them when processing a user query: not only would such an exhaustive search take a long time to complete, but it could also be expensive, since some of the text databases on the Internet may charge for their use. Consequently, metasearchers need a way to narrow their searches to a few useful text databases. This chapter presents a framework for (and analyzes a solution to) this problem, which we call the *text-source discovery problem*. Our solution assumes that sources export summaries of their contents as specified by the *STARTS* protocol described in the previous chapter.

Many tools have recently appeared on the Internet to help users (in particular, metasearchers) select the (text) databases that might be most useful for their queries (see Section 7.3). However, many of these tools essentially keep a global index of the available documents. This approach does not scale well with the growing number of sources and documents. Furthermore, this approach is problematic for commercial sources that are not willing to export their contents for indexing. Alternatively, many other tools index only a small part of each available document (e.g., its title). This approach fails to identify many useful sources because a significant part of each document is simply discarded. Similarly, other tools just keep succinct summaries of the contents of each database. These summaries are sometimes manually written, are often out of date, and fail to capture the whole content of the databases.

Our solution to the text-source discovery problem is to build a service that can suggest potentially good databases to search. Then, a metasearcher will present a query to our service (dubbed *GlOSS*, for *Glossary-Of-Servers Server*) to select a set of promising databases to search. *GlOSS* keeps only *partial information* on the contents of each database, so it scales with the growing number of available databases. However, this information covers the *full-text* content of the documents, so that the useful sources are identified. This chapter describes *GlOSS* for sources supporting the Boolean model of document retrieval [GGMT94a, GGMT94b, TGL+97], while Chapter 4 describes *gGlOSS*, a generalized version of *GlOSS* that works for sources supporting the vector-space model of document retrieval [GGM95a].

3.1 Text-Source Discovery for Boolean Sources

GlOSS gives a hint of what databases might be useful for the user's query, based on word-frequency information for each database. This information indicates, for each database and each word in the database vocabulary, how many documents at that database actually contain the word. For example, a collection of computer science technical reports could indicate that the word *Knuth* occurs in 180 documents, the word *computer* in 25,548 documents, and so on. This information is orders of magnitude smaller than a full index since for each word we only need to keep its frequency, as opposed to the identities of the documents that contain it.

Example 13: Consider three databases, A, B, and C, and suppose that *GlOSS* has collected the statistics of Figure 3.1. If *GlOSS* receives a query q=*retrieval* $\wedge$ *discovery* (this query searches for documents that contain both words, *retrieval* and *discovery*), *GlOSS* has to estimate the number of matching documents in each of the three databases. Figure 3.1 shows that database C does not contain any documents with the word *discovery*, and so, there cannot be any documents in C matching query q. For the other two databases, *GlOSS* has to "guess" what the number of documents matching query q is. There are different ways in which this can be done. An *estimator* for *GlOSS* uses the *GlOSS* information to make this guess. One of the estimators for *GlOSS* that we study in this chapter, *Ind*, estimates the result size of the given query in each of the databases in the following way. Database A contains 100 documents, 40 of which contain the word *retrieval*. Therefore, the probability that a document in A contains the word *retrieval* is $\frac{40}{100}$. Similarly, the probability that a document in A contains the word *discovery* is $\frac{5}{100}$. Under the assumption that words appear independently in documents, the probability that a document in database A has both the words *retrieval* and *discovery* is $\frac{40}{100} \times \frac{5}{100}$. Consequently, we can estimate the result size of query q in database A as $Goodness(q, A) = \frac{40}{100} \times \frac{5}{100} \times 100 = 2$ documents [1]. Similarly, $Goodness(q, B) = \frac{500}{1000} \times \frac{40}{1000} \times 1000 = 20$, and $Goodness(q, C) = \frac{10}{200} \times \frac{0}{200} \times 200 = 0$.

The *Ind* estimator chooses those databases with the highest estimates as the databases where to direct the given query. So, *Ind* will return $\{B\}$ as the answer to q (see Figure 3.2). This may or may not be a "correct" answer, depending on different factors. Firstly, it is possible that some of the result-size estimates given by *Ind* are wrong. For example, it could be the case that database B did not contain any matching document for q, while *Ind* predicted there would be 20 such documents in B. Furthermore, if database A did contain matching documents, then *Ind* would fail to pick any database with matching documents (since its answer was $\{B\}$).

Secondly, even if the estimates given by *Ind* are accurate, the correctness of the answer produced depends on the user's *semantics* for the query. Assume in what follows that the result-size estimates given above are correct (i.e., there actually are two documents matching query q in database A, 20 in database B, and none in database C). Given a query q and a set of databases, the user may be interested in one out of (at least) two different sets of databases over which to evaluate query q:

- *Matching*, the set of all of the databases containing matching documents for the query. For the sample query, this set is $\{A, B\}$, whereas *Ind* produced $\{B\}$ as its answer. Therefore, if the semantics intended by the query submitter are "recall

[1] We will discuss other ways of determining how *good* a database is for a query in Section 3.4.3.

Database	A	B	C
Number of documents	100	1000	200
Number of documents with the word *retrieval*	40	500	10
Number of documents with the word *discovery*	5	40	0

Figure 3.1: A portion of the database frequency information that *GlOSS* keeps for three databases.

oriented," in the sense that *all* of the databases in *Matching* should be searched, then *Ind*'s answer is not correct. Such a user is interested in getting exhaustive answers to the queries. (Section 3.6.2 presents the *Bin* estimator, aimed at addressing these semantics.) If, on the other hand, the intended semantics are "precision oriented," in the sense that *only* databases in *Matching* should be searched, then *Ind*'s answer is correct. In this case, the user is in "sampling" mode, and simply wants to obtain *some* matching documents, without searching useless databases.

- *Best*, the set of all of the databases containing more matching documents than any other database. Searching these databases yields the highest payoff (i.e., the largest number of documents). For the sample query, this set is $\{B\}$, which is also the answer produced by *Ind*. Again, users might be interested in emphasizing "precision" or "recall," in the sense described for the *Matching* case.

∎

To evaluate the set of databases that *GlOSS* returns for a given query, we present a framework based on the precision and recall metrics of information-retrieval theory. In that theory, for a given query q and a given set S of relevant documents for q, *precision* is the fraction of documents in the answer to q that are in S, and *recall* is the fraction of S in the answer to q. We borrow these notions to define metrics for the text-source discovery problem: for a given query q and a given set of "relevant databases" S, P is the fraction of databases in the answer to q that are in S, and R is the fraction of S in the answer to q. We further extend our framework by offering different definitions for

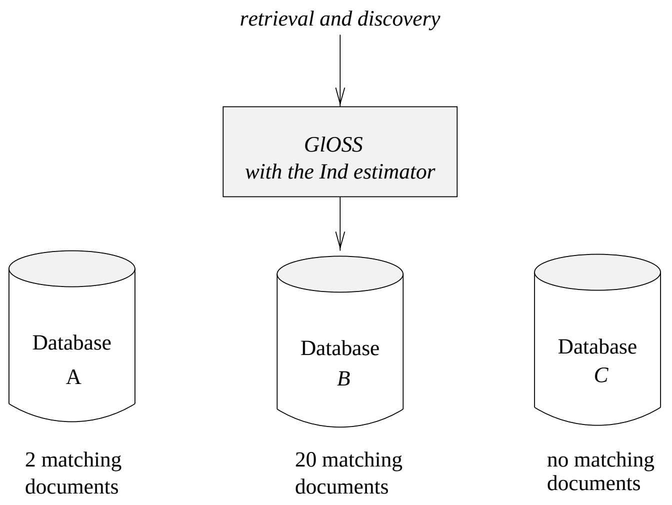

Figure 3.2: The *Ind* estimator for *GlOSS* chooses the most promising databases for a given query. In the example, database B, which is actually the database containing the highest number of matching documents, is chosen.

a "relevant database." We have performed experiments using query traces from the FOLIO library information-retrieval system at Stanford University, and involving six databases available through FOLIO. As we will see, the results obtained for *GlOSS* and several estimators are very promising. In Section 4.7 we also report experiments involving 500 text sources. Even though *GlOSS* keeps a small amount of information about the contents of the available databases, this information proved to be sufficient to produce very useful hints on where to search.

Another advantage of *GlOSS* is that its frequency information can be updated mechanically. Other approaches (see Section 7.3) require human-generated summaries of the contents of a database, and are prone to errors or very out-of-date information. Also, *GlOSS*'s storage requirements are low: a rough estimate suggested that 22.29 MBytes were enough to keep all of the data needed by *GlOSS* for the six databases we studied, or only 2.15% of the estimated size of a full index of the six databases. Therefore, it is straightforward to replicate the service at many sites. Thus, a user may be able to consult *GlOSS* at the local machine or cluster, and immediately determine the candidate databases for a given query.

Of course, *GlOSS* is not the only solution to the text-source discovery problem, and in practice we may wish to combine it with other complementary strategies. These strategies are described in Section 7.3. Incidentally we note that, to the best of our knowledge, experimental evaluations of these other strategies *for the text-source discovery problem* are rare: in most cases, strategies are presented with no statistical evidence as to how good they are at locating sites with documents of interest *for actual user queries*. Thus, we view our experimental methodology and results (even though they still have limitations) as an important contribution to this emerging research area.

Section 3.2 introduces *GlOSS* and the concept of an *estimator*. In particular, Section 3.2.4 describes *Ind*, the first estimator for *GlOSS* that we will evaluate in this chapter. Section 3.3 defines our first evaluation metrics, based on the precision and recall parameters [SM83]. Section 3.4 describes the experiments performed to assess the effectiveness of *GlOSS*. Section 3.4.3 identifies three different "right" sets of databases where users might want to evaluate their queries. Section 3.5 reports the experimental results, including experiments on two query traces to assess how dependent our results are on a specific query trace (Section 3.5.4). Section 3.6.1 introduces variants to *Ind* and to our evaluation metrics. Section 3.6.2 presents *Min* and *Bin*, two new estimators for *GlOSS*. Finally, Section 3.7 estimates *GlOSS*'s storage requirements, using the sources in our effectiveness experiments.

3.2 *GlOSS*: Glossary-Of-Servers Server

Consider a query q (permissible queries are defined in Section 3.2.1) that we want to evaluate over a set of databases DB. *GlOSS* selects a subset of DB consisting of "good candidate" databases for actually submitting q. To make this selection, *GlOSS* uses an *estimator* (Section 3.2.3), that assesses how "good" each database in DB is with respect to the given query, based on the word-frequency information on each database (Section 3.2.2).

3.2.1 Query Representation

In this chapter, we will only consider *Boolean "and"* queries, that is, queries that consist of positive atomic subqueries connected by the Boolean "and" operator (denoted as "$\wedge$" in what follows). (We consider other kinds of queries in Chapter 4.) An atomic subquery is a *keyword field-designation* pair. An example of a query is:

$$\textit{author Knuth} \wedge \textit{subject computer}$$

This query has two atomic subqueries: *author Knuth* and *subject computer*. In *author Knuth*, *author* is the field designation, and *Knuth* the corresponding keyword [2]. Although we restrict our study to "and" queries, we can extend our approach to include "or" queries in a variety of different ways. For example, in Section 3.7.2 we analyze a limited form of "or" queries, showing how *GlOSS* can handle this type of queries.

3.2.2 Database Word-Frequency Information

GlOSS keeps the following information about every database db_i:

- $|db_i|$, the total number of documents in database db_i, and
- f_{ij}, the number of documents in db_i that contain t_j, for all keyword field-designation pairs t_j. Note that *GlOSS* does not have available the actual "inverted lists" corresponding to each keyword-field pair and each database, but just the length of these inverted lists.

  If $f_{ij} = 0$, *GlOSS* does not need to store this explicitly, of course. Therefore, if *GlOSS* finds no information about f_{ij}, then f_{ij} will be assumed to be 0.

A real implementation of *GlOSS* requires that each database cooperate and periodically export these frequencies to the *GlOSS* server following some predefined protocol, like the *STARTS* protocol of Chapter 2.

[2]Uniform field designators for all the databases we considered (see Section 3.4.1) were available for our experiments. However, *GlOSS* does not rely completely on this, and could be adapted to the case where the field designators are not uniform across the databases, for example.

3.2.3 Estimators

Given the frequencies and sizes for a set of databases DB, *GlOSS* uses an *estimator* EST to select the set of databases to which to submit the given query. An estimator consists of a function $Estimate_{EST}$ that estimates the result size of a query in each of the databases, and a "matching" function (the max function below) that uses these estimates to select the set of databases ($Chosen_{EST}$ below) to which to submit the query. Once $Estimate_{EST}(q, db)$ has been defined, we can determine $Chosen_{EST}(q, DB)$ in the following way:

$$\begin{aligned} Chosen_{EST}(q, DB) \;=\; & \{db \in DB \mid Estimate_{EST}(q, db) > 0 \wedge \\ & Estimate_{EST}(q, db) = \max_{db' \in DB} Estimate_{EST}(q, db')\} \end{aligned} \tag{3.1}$$

Equation 3.1 may seem targeted to identifying the databases containing the *highest* number of matching documents. However, Section 3.6.2 shows how we can define $Estimate_{EST}(q, db)$ so that $Chosen_{EST}(q, db)$ becomes the set of *all* of the databases potentially containing matching documents, when we present the *Bin* estimator. Instances of $Estimate_{EST}$ are given in Sections 3.2.4 and 3.6.2, while a different "matching" function is used in Section 3.6.1.

3.2.4 The *Ind* Estimator

This section describes *Ind*, the estimator that we will use for most of our experiments. *Ind* (for "independence") is an estimator built upon the (possibly unrealistic) assumption that keywords appear in the different documents of a database following independent and uniform probability distributions. Under this assumption, given a database db_i, any n keyword field-designation pairs $t_1, \ldots, t_n$, and any document $d \in db_i$, the probability that d contains all of $t_1, \ldots, t_n$ is:

$$\frac{f_{i1}}{|db_i|} \times \ldots \times \frac{f_{in}}{|db_i|}$$

	INSPEC	PSYCINFO
Number of documents	1,416,823	323,952
Number of documents with *author Knuth*	13	0
Number of documents with *title computer*	24,086	2704

Figure 3.3: Information *Ind* needs for $DB = \{\text{INSPEC, PSYCINFO}\}$ and $q=$ *author Knuth* $\wedge$ *title computer*.

So, according to *Ind*, the estimated number of documents in db_i that will satisfy the query $t_1 \wedge \ldots \wedge t_n$ is [SFV83]:

$$Estimate_{Ind}(\ t_1 \wedge \ldots \wedge t_n, db_i) \;=\; \frac{\prod_{j=1}^{n} f_{ij}}{|db_i|^{n-1}} \tag{3.2}$$

The $Chosen_{Ind}$ set is then computed with Equation 3.1. Thus, *Ind* chooses those databases with the *highest* estimates (as given by $Estimate_{Ind}$).

To illustrate these definitions, let $DB = \{\text{INSPEC, PSYCINFO}\}$ (INSPEC and PSYCINFO are databases that we will use in our experiments, see Section 3.4). Also, let:

$$q = author\ Knuth \wedge title\ computer$$

Figure 3.3 shows the statistics available to *Ind*. From this, *Ind* computes:

$$Estimate_{Ind}(q, \text{INSPEC}) = \frac{13 \times 24,086}{1,416,823} \simeq 0.22$$

Incidentally, the actual result size of the query q in INSPEC, $Goodness(q, \text{INSPEC})$, is one document.

Since *Knuth* is not an author in the PSYCINFO database, and due to the Boolean semantics of the query representation, the result size of query q in the PSYCINFO database must be zero. This agrees with what Equation 3.2 predicts: $Estimate_{Ind}(q, \text{PSYCINFO}) = \frac{0 \times 2704}{323,952} = 0$. This holds in general for Boolean "and" queries: if $f_{ij} = 0$ for some $1 \leq j \leq n$, then

$$Estimate_{Ind}(\ t_1 \wedge \ldots \wedge t_n, db_i) = Goodness(\ t_1 \wedge \ldots \wedge t_n, db_i) = 0$$

As we have seen, when all frequencies are non-zero, $Estimate_{Ind}$ can differ from *Goodness*. Section 3.5.1 analyzes how well $Estimate_{Ind}$ approximates *Goodness*.

To continue with our example, since DB ={INSPEC, PSYCINFO}, and INSPEC is the only database with a non-zero result-size estimate, as given by $Estimate_{Ind}$, it follows that $Chosen_{Ind}(q, DB) = \{\text{INSPEC}\}$. So, *Ind* chooses the only database in the pair that might contain some matching document for q. In fact, since *Goodness*(q, INSPEC) = 1, *Ind* succeeds in selecting the only database that actually contains a document matching query q.

3.3 Evaluation Parameters

Let DB be a set of databases and q a query. In order to evaluate an estimator *EST*, we need to compare its prediction against what actually is $Right(q, DB)$, the "right subset" of DB to query. There are several notions of what the right subset means, depending on the semantics the query submitter has in mind. Section 3.4.3 examines some of these options. For example, $Right(q, DB)$ can be defined as the set of all the databases in DB that contain documents that match query q. Once we have defined the *Right* set for a query q and a database set DB, we evaluate how well $Chosen_{EST}(q, DB)$ approximates $Right(q, DB)$. (In general, we will drop the parameters of functions when this will not lead to confusion. For example, we refer to $Right(q, DB)$ as *Right*, whenever q and DB are clear from the context.)

To evaluate $Chosen_{EST}$, we adapt the well-known *precision* and *recall* parameters from information-retrieval theory [SM83] to the text-source discovery framework. If we regard *Right* as the set of "items" (databases in this context) that are relevant to a given query q, and $Chosen_{EST}$ as the set of items that is actually retrieved, we can define the following functions P^{EST}_{Right} and R^{EST}_{Right}, based upon the precision and recall parameters:

$$P^{EST}_{Right}(q, DB) = \begin{cases} \frac{|Chosen_{EST}(q,DB) \cap Right(q,DB)|}{|Chosen_{EST}(q,DB)|} & \text{if } |Chosen_{EST}(q, DB)| > 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.3}$$

$$R^{EST}_{Right}(q, DB) = \begin{cases} \frac{|Chosen_{EST}(q,DB) \cap Right(q,DB)|}{|Right(q,DB)|} & \text{if } |Right(q, DB)| > 0 \\ 1 & \text{otherwise} \end{cases} \quad (3.4)$$

Intuitively, P is the fraction of selected databases that are *Right* ones, and R is the fraction of the *Right* databases that are selected. For example, suppose that the set of databases is $DB = \{A, B, C\}$, and that for a given query q, $Right(q, DB)$ is defined to be $\{A, B\}$. (This could be the case if only A and B contained documents matching query q, as in Example 3.1.) Furthermore, if $Chosen_{EST}(q, DB) = \{B\}$, then $P^{EST}_{Right}(q, DB) = 1$, since the only chosen database, B, is in the *Right* set. On the other hand, $R^{EST}_{Right}(q, DB) = 0.5$, since only half of the databases in *Right* are included in $Chosen_{EST}$.

Note that $P^{EST}_{Right}(q, DB) = 1$ whenever $Chosen_{EST} = \emptyset$, to capture the fact that no database in $Chosen_{EST}$ is not in *Right*. Similarly, $R^{EST}_{Right}(q, DB) = 1$ if $Right = \emptyset$, since all of the *Right* databases are included in $Chosen_{EST}$.

Different users will be interested in different semantics for the queries. One way to define different semantics is through the definition of *Right* (see Section 3.4.3). Even for a fixed *Right* set of databases, some users may be interested in emphasizing "precision" (databases not in *Right* should be avoided, even if this implies missing some of the "right" databases), while some others may want to emphasize "recall" (at least all of the databases in *Right* should be included in the answer to query q). Therefore, high values of P^{EST}_{Right} should be the target in the former case, and high values of R^{EST}_{Right} in the latter.

In the remainder of this chapter, we evaluate different estimators in terms of the average value, over a set of user queries, of the P and R parameters defined above, for different *Right* sets of databases. In Section 3.7 we introduce alternative metrics for our experiments, and show their relationship with the P and R metrics above.

3.4 Experimental Framework

To evaluate the performance of different *GlOSS* estimators according to the P and R parameters of Section 3.3, we performed experiments using query traces from the

Database	*Number of documents*	*Area*
INSPEC	1,416,823	Physics, Elect. Eng., Computer Sc.
COMPENDEX	1,086,289	Engineering
ABI	454,251	Business Periodical Literature
GEOREF	1,748,996	Geology and Geophysics
ERIC	803,022	Educational Materials
PSYCINFO	323,952	Psychology

Figure 3.4: Summary of the characteristics of the six databases considered.

FOLIO library information-retrieval system at Stanford University.

3.4.1 Databases and the INSPEC Query Trace

Stanford University provides on-campus access to its information-retrieval system FOLIO. FOLIO gives access to several databases. Figure 3.4 summarizes some characteristics of the six databases we chose for our experiments. Six is a relatively small number, given our interest in exploring hundreds of databases. However, we were limited to a small number of databases by their accessibility and by the high cost of our experiments. Thus, our results will have to be taken with caution, indicative of the *potential* benefits of this type of estimators. Section 4.7 shows experimental results for *GlOSS* that involve 500 text sources.

A trace of all user commands for the INSPEC database was collected from 4/12 to 4/25 in 1993. This set of commands contained 8392 queries. As discussed in Section 3.2.1, we only considered correctly formed "and" queries. Also, we did not consider the so-called "phrase" queries (e.g., *titlephrase knowledge bases*). The final set of queries, $TRACE_{INSPEC}$, has 6897 queries, or 82.19% of the original set.

3.4.2 Constructing the Database Frequency Information

To perform our experiments, we evaluated each of the $TRACE_{INSPEC}$ queries in the six databases described in Figure 3.4. This is the data we need to build the different *Right* sets (see Section 3.4.3) for each of the queries.

Also, to build the database word-frequency information needed by *GlOSS* (Section 3.2.2) we evaluated, for each query of the form $t_1 \wedge \ldots \wedge t_n$, the n queries $t_1, \ldots, t_n$ in each of the six databases. Note that the result size of the execution of t_j in database db_i is equal to f_{ij} as defined in Section 3.2. This is exactly the information an estimator *EST* needs to define $Chosen_{EST}$, for each query in $TRACE_{INSPEC}$ [3]. It should be noted that this is just the way we gathered the data in order to perform our experiments. An actual implementation of such a system requires that each database export the number of postings for each word to *GlOSS*.

3.4.3 Different "Right" Sets of Databases

Section 3.3 introduced the notion of the *Right* set of databases for a given query. Different definitions for the *Right* set determine different instantiations of the P and R parameters defined by Equations 3.3 and 3.4. To illustrate the issues involved in determining *Right*, consider the following example:

Example 14: Figure 3.5 shows three databases: A, B, and C. Consider a query q issued by a user. Each database produces a set of matching documents as the answer to q. Figure 3.5 shows that database A gives document 4 as the answer to q, database B, documents 5, 6, and 7, and database C, documents 8 and 9. Also, each database contains a set of documents that are relevant to the user that issued query q, that is, are actually of interest to the user. These documents may or may not match the answer to q. Thus, database A has three relevant documents: documents 1, 2, and 3, database B has one relevant document: document 5, and database C has two relevant documents: documents 8 and 9. Furthermore, assume that the user is interested in evaluating the query in one database only. The question is how to define the *Right* set given this scenario. There are three alternatives:

- $Right = \{A\}$, since A is the database with the highest number of documents (three) relevant to the user's information need. However, the answer produced

[3] In fact, we are not retrieving all of the word frequencies, but only those that are needed for the queries in $TRACE_{INSPEC}$.

by database A when presented with query q consists of document 4 only, which is not a relevant document. Therefore, the user would not benefit from the fact that A contains the highest number of relevant documents among the three available databases, making this definition for *Right* not very useful.

- *Right* $= \{C\}$, since C is the database that produces the highest number of relevant documents in the answer to query q. This is an interesting definition. However, we believe that it is unreasonable to expect a service like *GlOSS* to guess this type of *Right* set of sites. Since the information kept by *GlOSS* about each database is necessarily much less detailed than that kept by the search engine at each database, it would be very hard for *GlOSS* to accurately guess the number of relevant documents in the answer to a query given by a database.

- *Right* $= \{B\}$, since B is the database that produces the largest number of matching documents for q. Presumably, if the individual databases retrieve a reasonable approximation of the set of documents relevant to the given query, the *Right* database according to this definition would yield the highest number of useful documents. Also, the semantics of this definition are easily understood by the users, since they do not depend on relevance judgments, for example.

∎

In our first two definitions of the *Right* set, we will take the third approach illustrated in the example. That is, the goodness of a database db with respect to a query q will be determined by the number of documents that db returns when presented with q (i.e., the number of documents matching q in db). Our first definition for $Right(q, DB)$ is $Matching(q, DB)$, the set of all databases in DB containing at least one document that matches query q. More formally,

$$Right(q, DB) = Matching(q, DB) = \{db \in DB | Goodness(q, db) > 0\} \tag{3.5}$$

There are (at least) two types of users that may specify $Matching(q, DB)$ as their right set of databases. One is users that want an exhaustive answer to their query.

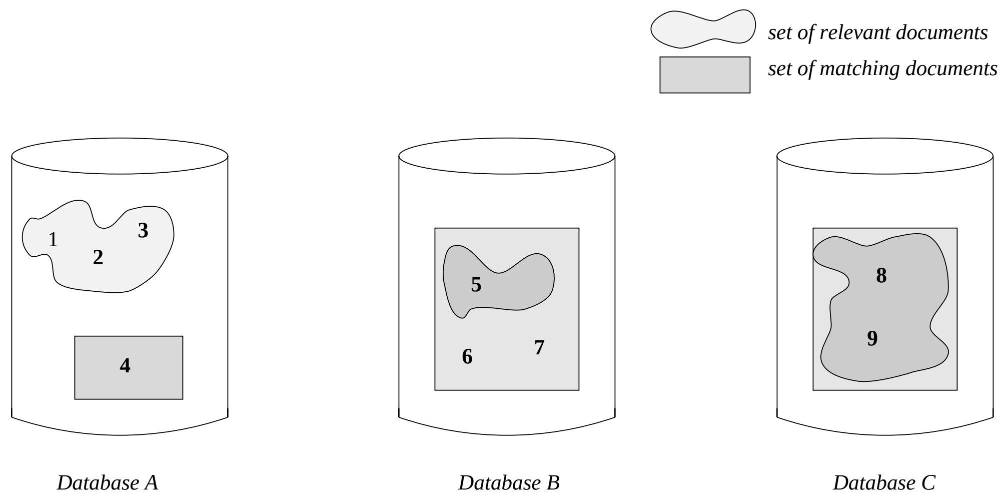

Figure 3.5: The documents relevant to a given query vs. the documents actually given as the answer to the query, for three different databases. Documents are represented by numbers in this figure.

They are not willing to miss any of the matching documents. We will refer to these users as "recall-oriented" users. On the other hand, "precision-oriented" users may be in "sampling" mode: they simply want to obtain *some* matching documents without searching useless databases.

Our second definition for $Right(q, DB)$ is $Best(q, DB)$, the set of those databases that contain more matching documents than any other database. More formally,

$$\begin{aligned} Right(q, DB) &= Best(q, DB) \\ &= \{db \in DB \mid Goodness(q, db) > 0 \land \\ &\quad Goodness(q, db) = \max_{db' \in DB} Goodness(q, db')\} \end{aligned} \tag{3.6}$$

Again, users that define $Best(q, DB)$ as their right set of databases for query q might be classified as being "recall oriented" or "precision oriented." "Recall-oriented" users want all of the best databases for their query. These users are willing to miss some databases, as long as they are not the best ones. That is, the users recognize that there are more databases that could be examined, but want to ensure that at least those having the highest payoff (i.e., the largest number of documents) are searched.

On the other hand, "precision-oriented" users want to examine (some) best databases. Due to limited resources (e.g., time, money) the users only want to submit their query at databases that will yield the highest payoff.

Our third definition for $\mathit{Right}(q, DB)$, $\mathit{Matching}_I(q, DB)$, is specific for the case INSPEC $\in DB$, and for queries $q \in \mathit{TRACE}_{\mathit{INSPEC}}$. (This definition will be useful in the experiments we describe starting in Section 3.5.2.) In this case, we assume that INSPEC is the right database to search, regardless of the number of matching documents in the other databases, because the users issued the $\mathit{TRACE}_{\mathit{INSPEC}}$ queries to the INSPEC database, and perhaps they knew what the right database to search was. This is somewhat equivalent to regarding each query $q \in \mathit{TRACE}_{\mathit{INSPEC}}$ as augmented with the extra conjunct $\wedge$ *database* INSPEC. So, our third definition for *Right* is:

$$\begin{aligned}\mathit{Right}(q, DB) &= \mathit{Matching}_I(q, DB) \\ &= \begin{cases} \{\text{INSPEC}\} & \text{if INSPEC} \in DB \wedge \\ & \quad \mathit{Goodness}(q, \text{INSPEC}) > 0 \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \tag{3.7}$$

3.4.4 Configuration of the Experiments

There are a number of parameters to our experiments. Figure 3.6 shows an assignment of values to these parameters that will determine the *basic configuration*. In later sections, some of these parameters will be changed, to produce alternative results. The parameters ϵ_C and ϵ_B will be defined in Section 3.6.1.

3.5 *Ind* Results

In this section, we evaluate *Ind* by first studying how well it can predict the result size of a query and a database (Section 3.5.1). After this, we analyze *Ind*'s ability to distinguish between two databases (Section 3.5.2) and then we generalize the experiments to include six databases (Section 3.5.3). Finally, we repeat some of the experiments for a different set of queries to see how dependent our results are on the

Database set (DB)	{INSPEC, COMPENDEX, ABI, GEOREF, ERIC, PSYCINFO}
Estimator	*Ind*
Query set	$TRACE_{INSPEC}$
Query sizes considered	All
ϵ_C	0
ϵ_B	0

Figure 3.6: Basic configuration of the experiments.

query trace used (Section 3.5.4).

3.5.1 *Ind* as a Predictor of the Result Size of the Queries

The key to *Ind* is its estimation function $Estimate_{Ind}(q, db)$, which predicts how many documents matching query q database db has. Before seeing how accurate *Ind* is at selecting a good subset of databases, let us study its estimation function $Estimate_{Ind}$. An important question is whether $Estimate_{Ind}$ is a good predictor of the result size of a query in absolute terms, that is, whether the following holds:

$$Estimate_{Ind}(q, db) \approx Goodness(q, db)$$

If we analyze the data we collected, as explained in Section 3.4, the answer is no, unfortunately. In general, *Ind* tends to underestimate the result size of the queries. The more conjuncts in a query, the worse this problem becomes. Figure 3.7 shows a plot of the pairs:

$$< Goodness(q, \text{INSPEC}), Estimate_{Ind}(q, \text{INSPEC}) >$$

for the queries in $TRACE_{INSPEC}$. (See Section 3.4.) The accumulation of points on the $y = x$ axis corresponds to the one-atomic-subquery queries (e.g., *author Knuth*), for which $Estimate_{Ind} = Goodness$. (This follows from Equation 3.2.)

Nevertheless, *Ind* will prove to be good at discriminating between useful and less useful databases according to the P and R parameters of Section 3.3. The reason for

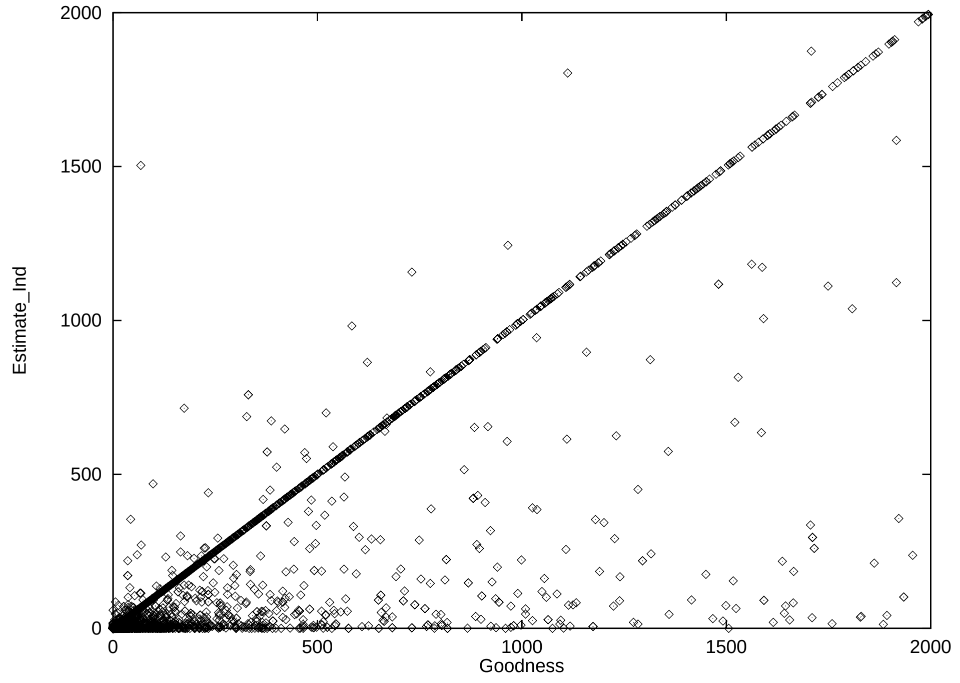

Figure 3.7: *Ind* as an estimator of the result size of the queries.

this is that even though $Estimate_{Ind}(q, db)$ will in general not be a good approximation of $Goodness(q, db)$, it is usually the case that $Estimate_{Ind}(q, db') < Estimate_{Ind}(q, db)$ if database db contains more documents matching query q than database db' does.

3.5.2 Evaluating *Ind* over Pairs of Databases

In this section, we report some results for the basic configuration (Figure 3.6), but with DB, the set of available databases, set to just two databases. Figures 3.8 and 3.9 show two matrices classifying the 6897 queries in $TRACE_{INSPEC}$ for the cases $DB = \{$INSPEC, PSYCINFO$\}$ and $DB = \{$INSPEC, COMPENDEX$\}$. The sum of all of the entries of each matrix equals 6897. Consider for example Figure 3.8, for $DB = \{$INSPEC, PSYCINFO$\}$. Each row of the matrix represents an outcome for *Matching* and *Best*. The first row, for instance, represents queries where both INSPEC and PSYCINFO had matching documents (*Matching*$=\{$INSPEC, PSYCINFO$\}$) but where INSPEC had the most matching documents (*Best* $=\{$INSPEC$\}$). On the other hand, each column represents the prediction made by *Ind*. For example, the number 2678 means that for 2678 of the queries in $TRACE_{INSPEC}$, $Best$ ={INSPEC}, $Matching$ ={INSPEC, PSYCINFO}, and *Ind* selected INSPEC as its prediction ($Chosen_{Ind}$ ={INSPEC}). In the same row, there were 26 other queries where *Ind* picked a matching database (PSYCINFO) but not the best one. In the first two rows, we see that for most of the queries (5614 out of 6897), INSPEC was the best database. This is not surprising, since the queries used in the experiments were originally issued by users to the INSPEC database.

The two matrices of Figures 3.8 and 3.9 show that $Chosen_{Ind} = \emptyset$ only if $Matching = \emptyset$. From Equations 3.1 and 3.2 it follows that this relationship holds in general, that is, as long as there is at least one database that contains matching documents, $Chosen_{Ind}$ will be non-empty. Also, note that very few times (15 for {INSPEC, PSYCINFO} and 92 for {INSPEC, COMPENDEX}) does *Ind* determine a tie between the two databases (and so, $Chosen_{Ind}$ consists of both databases). This is so since it is unlikely that $Estimate_{Ind}(q, db_1)$ will be exactly equal to $Estimate_{Ind}(q, db_2)$ if $db_1 \neq db_2$. With the current definition of $Chosen_{Ind}$, if for some query q and databases db_1 and db_2 it is the case that, say, $Estimate_{Ind}(q, db_1) = 9$ and $Estimate_{Ind}(q, db_2) = 8.9$, then $Chosen_{Ind}(q, \{db_1, db_2\}) = \{db_1\}$. We might want in such a case to include db_2 also in $Chosen_{Ind}$. We address this issue in Section 3.6.1, where we relax the definition of $Chosen_{Ind}$ and *Best*.

Figures 3.10 and 3.11 report the values of the P and R parameters for the three different target sets defined in Section 3.4.3. For example, in the second row of Figure 3.10, R^{Ind}_{Best}= 0.9910. This means that for the average query, $Chosen_{Ind}$ includes 99.10% of the *Best* databases when DB ={INSPEC, PSYCINFO}. Therefore, for most of the $TRACE_{INSPEC}$ queries, $Best \subseteq Chosen_{Ind}$: from Figure 3.8, $Best \subseteq Chosen_{Ind}$ for 6831 queries. Also, for 6328 queries, $Chosen_{Ind}$ was exactly equal to *Best*. The reason for such high values is that INSPEC and PSYCINFO cover very different topics (see Figure 3.4). Therefore, for each query there is likely to be a clear "winner" (generally INSPEC for the queries in $TRACE_{INSPEC}$). On the other hand, INSPEC and COMPENDEX cover somewhat overlapping areas, thus yielding a lower (0.9216) value for R^{Ind}_{Best} (see Figure 3.11), for example.

		$Chosen_{Ind}$			
Best	*Matching*	{I}	{P}	{I, P}	∅
{I}	{I, P}	2678	26	0	0
{I}	{I}	2894	16	0	0
{P}	{I, P}	11	224	0	0
{P}	{P}	5	34	0	0
{I, P}	{I, P}	3	5	15	0
∅	∅	462	41	0	483

Figure 3.8: Results corresponding to DB = {INSPEC (I), PSYCINFO (P)} and *Ind* as the estimator.

The values for $R^{Ind}_{Matching}$ are lower in both the PSYCINFO and COMPENDEX cases: this is not surprising since *Ind* chooses the *most* promising databases, not all of the ones potentially containing matching documents. Therefore, some matching databases may be missed. Section 3.6.2 introduces a different estimator for *GlOSS*, *Bin*, aimed at optimizing the case *Right* = *Matching*. Notice that $R^{Ind}_{Matching}$ is particularly low (0.6022) for the pair {INSPEC, COMPENDEX}, since for most of the queries, there are matching documents in both databases (see the rows of Figure 3.9 corresponding to *Matching* ={INSPEC, COMPENDEX}), and very rarely does *Ind* choose more than one database, as explained above.

From Figure 3.10, P^{Ind}_{Best}= 0.9187, showing that for each query, an average of 91.87% of the databases in $Chosen_{Ind}$ are among the *Best* databases. So, for most of the queries, $Chosen_{Ind} \subseteq Best$: from Figure 3.8, $Chosen_{Ind} \subseteq Best$ for 6336 queries. In general, the values for P^{Ind}_{Best} and $P^{Ind}_{Matching}$ are relatively high for both pairs of databases, showing that in most cases $Chosen_{Ind}$ consists only of matching databases (high $P^{Ind}_{Matching}$) and in many of these cases, $Chosen_{Ind}$ consists only of "best" databases (high P^{Ind}_{Best}). Furthermore, it is always the case that $P^{Ind}_{Best}(q, DB) \leq P^{Ind}_{Matching}(q, DB)$, since $Best(q, DB) \subseteq Matching(q, DB)$.

Finally, note that the values of $P^{Ind}_{Matching_I}$ and $R^{Ind}_{Matching_I}$ are higher for the {INSPEC, PSYCINFO} pair than for the {INSPEC, COMPENDEX} pair: for the {INSPEC, PSYCINFO} pair, INSPEC is almost always clearly the best database (see Figure 3.8), whereas this is true to a lesser extent for the {INSPEC, COMPENDEX} pair (see Figure 3.9).

		$Chosen_{Ind}$			
Best	*Matching*	{I}	{C}	{I, C}	$\emptyset$
{I}	{I, C}	4053	247	0	0
{I}	{I}	382	43	0	0
{C}	{I, C}	144	743	0	0
{C}	{C}	23	100	0	0
{I, C}	{I, C}	125	43	92	0
$\emptyset$	$\emptyset$	319	173	0	410

Figure 3.9: Results corresponding to DB = {INSPEC (I), COMPENDEX (C)} and *Ind* as the estimator.

Right	P^{Ind}_{Right}	R^{Ind}_{Right}
Matching	0.9240	0.7833
Best	0.9187	0.9910
$Matching_I$	0.8810	0.9607

Figure 3.10: Parameters P and R for DB ={INSPEC, PSYCINFO} and *Ind* as the estimator.

Right	P^{Ind}_{Right}	R^{Ind}_{Right}
Matching	0.9191	0.6022
Best	0.8624	0.9216
$Matching_I$	0.7482	0.8440

Figure 3.11: Parameters P and R for DB ={INSPEC, COMPENDEX} and *Ind* as the estimator.

Reference [GGMT93] reports experimental results for all the pairs of databases from {INSPEC, COMPENDEX, ABI, GEOREF, ERIC, PSYCINFO}. The two pairs of databases analyzed in this section, {INSPEC, PSYCINFO} and {INSPEC, COMPENDEX}, are among the best and the worst, respectively, for *Ind*, among all possible pairs: in general, the more unrelated the subject domains of the two databases considered were, the better *Ind* behaved in distinguishing the databases.

3.5.3 Evaluating *Ind* over Six Databases

In this section we report some results for the basic configuration, as defined in Figure 3.6. Figure 3.12 summarizes the results corresponding to the three definitions of the *Right* set of Section 3.4.3. This figure shows that the same phenomena described in Section 3.5.2 prevail, although in general the values are lower. For example, $R^{Ind}_{Matching}$ is much lower (0.4044), since *Ind* chooses only the most promising databases, not all of the ones that might contain matching documents (see Section 3.6.2). Still, R^{Ind}_{Best} is high (0.9010), showing *Ind*'s ability to predict what the best databases are. Also, $P^{Ind}_{Matching}$ and P^{Ind}_{Best} are high (0.9126 and 0.8438, respectively), making *Ind* useful for exploring *some* of the matching/best databases. This is particularly significant for *Ind*: $Chosen_{Ind}(q, DB)$ will be non-empty as long as there is some database in DB that contains some document matching query q.

Another interesting piece of information that we gathered in our experiments is that for only 96 out of the 6897 $TRACE_{INSPEC}$ queries does $Chosen_{Ind}$ consist of more than one database. Furthermore, 95 out of these 96 queries are one-atomic-subquery queries, for which $Chosen_{Ind} = Best$ necessarily (this follows from Equations 3.1 and 3.2). So, revisiting the results of Figure 3.12, since R^{Ind}_{Best}=0.9010, for most of the $TRACE_{INSPEC}$ queries not only does *Ind* narrow down the search space to one database (out of the six available ones), but it also manages to select the best database when there is one.

Right	P^{Ind}_{Right}	R^{Ind}_{Right}
Matching	0.9126	0.4044
Best	0.8438	0.9010
$Matching_I$	0.5966	0.7012

Figure 3.12: Parameters P and R for the basic configuration of the experiments.

Right	P^{Ind}_{Right}	R^{Ind}_{Right}
Matching	0.8960	0.4621
Best	0.8498	0.9384
$Matching_E$	0.5485	0.6876

Figure 3.13: Parameters P and R for the basic configuration, but using the queries in $TRACE_{ERIC}$.

3.5.4 Impact of Using Other Traces

So far, all of our experiments were based on the set of 6897 $TRACE_{INSPEC}$ queries. To analyze how dependent the results are on the trace used, we ran our experiments using a different set of queries. Real users issued these queries to the ERIC database from 3/28 to 4/10 in 1993. We processed the trace in the same way as the INSPEC trace (see Section 3.4). The final set of queries, $TRACE_{ERIC}$, has 2404 queries, or 78.82% of the original 3050 query set.

Figure 3.13 shows the results for the different instances of the P and R parameters, for the basic configuration (Figure 3.6) but using $TRACE_{ERIC}$. The definition of the $Matching_E$ set of databases is analogous to that of $Matching_I$ (see Equation 3.7), using ERIC instead of INSPEC. The results obtained differ only slightly from the ones in Figure 3.12 for $TRACE_{INSPEC}$. This suggests that our results are not sensitive to the type of trace used.

3.6 Improving *GlOSS*

In this section we introduce variations to the definition of the $Chosen_{EST}$ and *Best* sets in order to make them more flexible (Section 3.6.1), and present two new estimators, *Min* and *Bin* (Section 3.6.2).

3.6.1 Making Chosen_{EST} and Best More Flexible

The definitions of Chosen_{EST} and *Best* given by Equations 3.1 and 3.6 are sometimes too "rigid." Consider the following example. Suppose $\{db_1, db_2\}$ is our set of databases, and let q be a query with $\mathit{Goodness}(q, db_1) = 1,000$, and $\mathit{Goodness}(q, db_2) = 1,001$. According to Equation 3.6, $\mathit{Best}(q, DB) = \{db_2\}$. But this is probably too arbitrary, since both databases are almost identical regarding the number of matching documents they have for query q. Also, if an estimator *EST* predicts that the two databases contain a very similar number of documents satisfying a query, though not exactly equal, it might be preferable to choose both databases as the answer instead of picking the one with absolute highest estimated size.

In this section, we extend the definitions of Chosen_{EST} and *Best*, through the introduction of two parameters, ϵ_B and ϵ_C. Parameter ϵ_B will make the definition of *Best* looser, by letting databases with a number of documents close but not exactly equal to the maximum be considered as "best" databases also. Parameter ϵ_C changes the "matching" function (Section 3.2.3) of an estimator *EST* by making it able to choose databases that are close to the predicted optimal ones. The new definitions for Chosen_{EST} and *Best* are, for given $\epsilon_B, \epsilon_C \geq 0$:

$$\begin{aligned} \mathit{Chosen}_{EST}(q, DB) &= \{db \in DB | \mathit{Estimate}_{EST}(q, db) > 0 \wedge \\ &\quad \left| \frac{\mathit{Estimate}_{EST}(q, db) - m_e}{m_e} \right| \leq \epsilon_C\} && (3.8) \\ \mathit{Best}(q, DB) &= \{db \in DB | \mathit{Goodness}(q, db) > 0 \wedge \\ &\quad \left| \frac{\mathit{Goodness}(q, db) - m_g}{m_g} \right| \leq \epsilon_B\} && (3.9) \end{aligned}$$

where

$$m_e = \max_{db \in DB} \mathit{Estimate}_{EST}(q, db) \text{ and } m_g = \max_{db \in DB} \mathit{Goodness}(q, db).$$

Therefore, the larger ϵ_B and ϵ_C, the more databases will be included in *Best* and Chosen_{EST}, respectively. Note that Equations 3.1 and 3.6 coincide with Equations 3.8 and 3.9 for $\epsilon_B = \epsilon_C = 0$. Also, if $\epsilon_C = 1$, *Ind* becomes the *Bin* estimator described in Section 3.6.2: $Chosen_{Ind}(q, DB)$ thus consists of all of the databases in DB that *might* contain some matching documents for query q.

Figures 3.14 and 3.15 show the average values of the P and R parameters, respectively, for the basic configuration of the experiments ($\epsilon_C = 0$), but for different values of ϵ_B. Thus, our *Ind* estimator remains fixed (since $\epsilon_C = 0$) and so do *Matching* and $Matching_I$, since they do not depend on the parameter ϵ_B. This is why the curves corresponding to $P^{Ind}_{Matching}$, $R^{Ind}_{Matching}$, $P^{Ind}_{Matching_I}$, and $R^{Ind}_{Matching_I}$ are flat. On the other hand, the set of best databases, *Best*, varies as ϵ_B does. By varying ϵ_B alone, we are leaving the estimator fixed, and we change the semantics of our evaluation criteria, because we are modifying (i.e., making more flexible) our *Best* "target" set.

In Figure 3.15 we see that parameter R^{Ind}_{Best} worsens as ϵ_B grows, since *Best* tends to contain more databases, while $Chosen_{Ind}$ remains fixed. This is exactly why P^{Ind}_{Best} (Figure 3.14) improves with higher values of ϵ_B. Note that for $\epsilon_B = 1$, $Best = Matching$, and so, $P^{Ind}_{Matching}$ and $R^{Ind}_{Matching}$ coincide with P^{Ind}_{Best} and R^{Ind}_{Best}, respectively.

As mentioned above, parameter ϵ_B is not a parameter of our estimator, but of the semantics of the queries. The submitter of a query does not give an ϵ_B value to *GlOSS*. Instead, higher values for ϵ_B yield more comprehensive *Best* sets. Therefore, parameter ϵ_B should be fixed according to the desired "meaning" for *Best*. For example, suppose that we are evaluating *Ind* for a user that wants to locate *Best* databases, but is willing to search at sites that have 90% or more of the number of matching documents than the overall *Best* sites have. Then, the experimental results that are relevant to this user are those obtained for $\epsilon_B = 0.1$.

Figures 3.16 and 3.17 show the average values of the P and R parameters, respectively, for the basic configuration of the experiments ($\epsilon_B = 0$), but for different values of ϵ_C. Here, the *Matching* and $Matching_I$ sets do not change (they do not depend on ϵ_C), and neither does *Best* (since $\epsilon_B = 0$). *Ind* is affected, since ϵ_C is variable. Since $Chosen_{Ind}$ tends to cover more databases as ϵ_C grows, $R^{Ind}_{Matching}$, R^{Ind}_{Best}, and $R^{Ind}_{Matching_I}$ improve for higher values of ϵ_C. For $\epsilon_C = 1$, $R^{Ind}_{Matching} = R^{Ind}_{Best} = R^{Ind}_{Matching_I} = 1$, since $Chosen_{Ind}$ contains all of the potentially matching databases: as mentioned above, *Ind* becomes the *Bin* estimator (Section 3.6.2) for $\epsilon_C = 1$. This is also why P^{Ind}_{Best} and $P^{Ind}_{Matching_I}$ worsen as ϵ_C grows. Parameter $P^{Ind}_{Matching}$ remains basically unchanged for

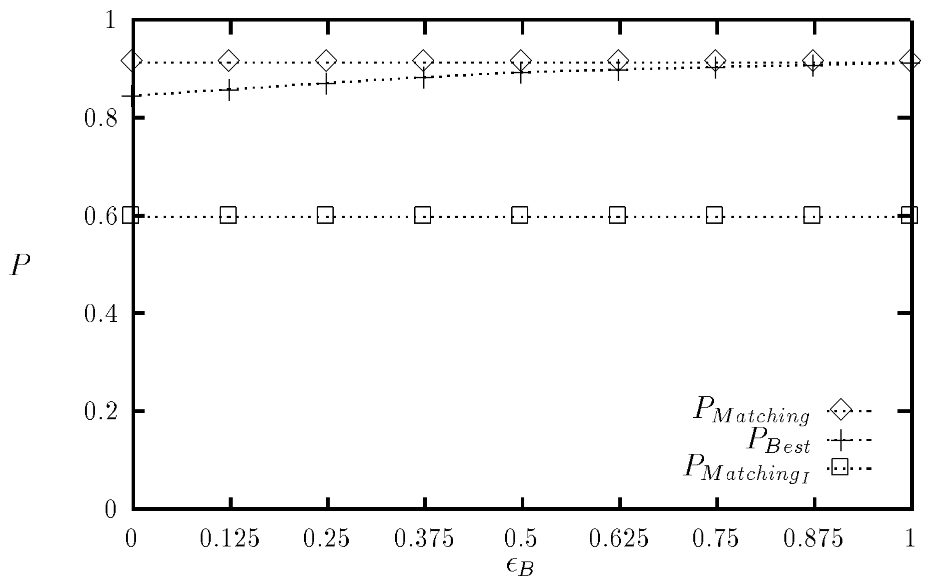

Figure 3.14: The average P parameters as a function of ϵ_B for the *Ind* estimator ($\epsilon_C = 0$).

higher values of ϵ_C, but worsens for ϵ_C close to one, for the same reasons P^{Ind}_{Best} and $P^{Ind}_{Matching_I}$ get lower. Note that for $\epsilon_C = 1$, $P^{Ind}_{Best} \neq P^{Ind}_{Matching}$, since *Best* and *Matching* differ ($\epsilon_B = 0$).

From Figures 3.16 and 3.17 we can conclude that the value for ϵ_C should be set according to whether precision or recall should be emphasized (in the sense of Section 3.3). Users can set the value for ϵ_C to be used by *Ind* according to the query semantics they are interested in: in general, higher values for ϵ_C make the R parameters improve, while the P parameters worsen. However, when the *Right* set of databases is equal to the *Best* set, $\epsilon_C = 0$ is a good compromise to obtain both high P and high R values, since R^{Ind}_{Best} is already high for $\epsilon_C = 0$ (and so is P^{Ind}_{Best}).

3.6.2 Other Estimators

So far, all of our experiments involved *Ind* as the estimator for *GlOSS*. In this section, we consider two other estimators, and compare their performance with that of *Ind*.

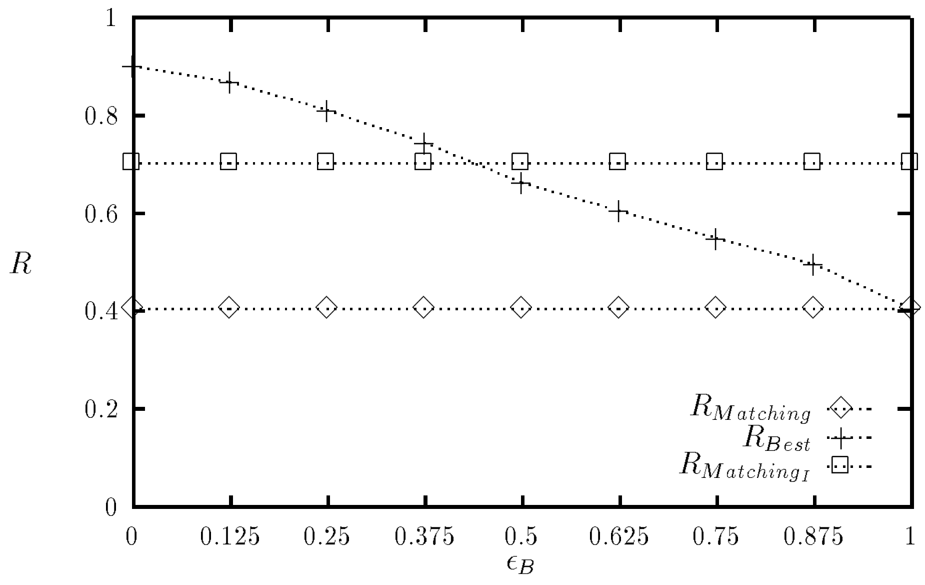

Figure 3.15: The average R parameters as a function of ϵ_B for the *Ind* estimator ($\epsilon_C = 0$).

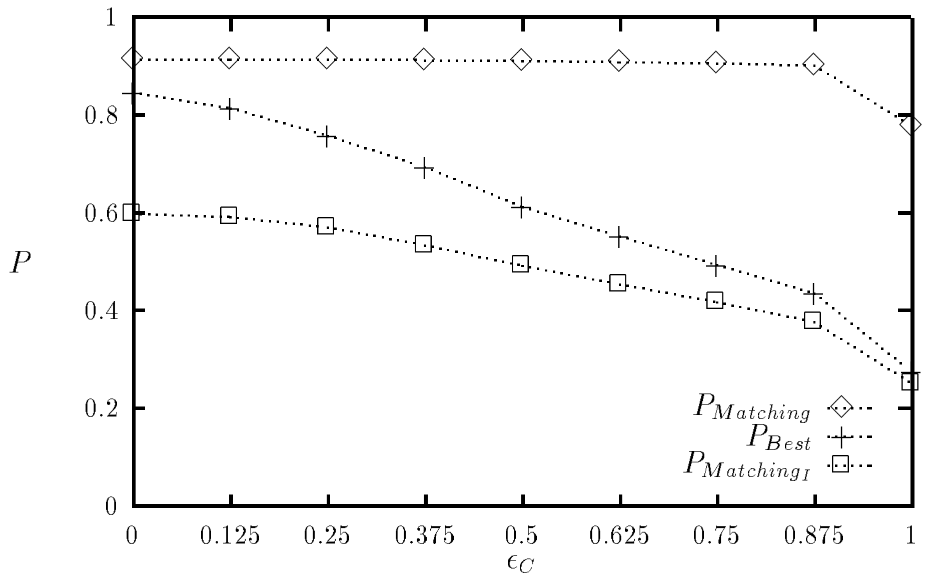

Figure 3.16: The average P parameters as a function of ϵ_C for the *Ind* estimator ($\epsilon_B = 0$).

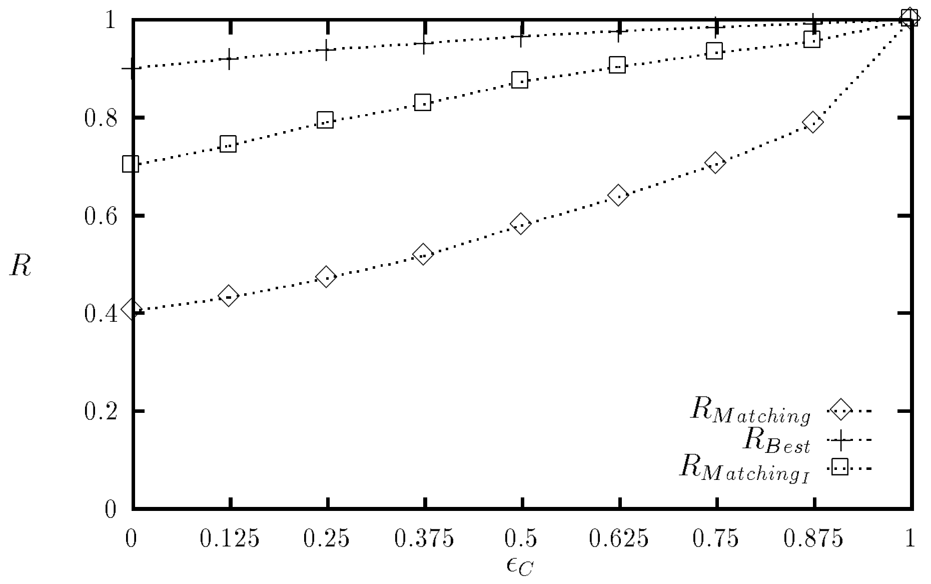

Figure 3.17: The average R parameters as a function of ϵ_C for the *Ind* estimator ($\epsilon_B = 0$).

Ind is based upon the assumption that the occurrence of query keywords in documents follows independent and uniform probability distributions. We can build alternative estimators by departing from this assumption. For example, we can adopt the "opposite" assumption, and assume that the keywords that appear together in a user query are strongly correlated. So, we define another estimator for *GlOSS*, *Min* (for "minimum"), by letting:

$$Estimate_{Min}(\ t_1 \wedge \ldots \wedge t_n, db_i) \ = \ \min_{j=1}^{n} f_{ij} \tag{3.10}$$

$Estimate_{Min}(q, db_i)$ is an upper bound of the actual result size of query q:

$$Goodness(q, db_i) \leq Estimate_{Min}(q, db_i)$$

$Chosen_{Min}$ follows from the definition of $Estimate_{Min}$, using Equation 3.1.

If our goal is to maximize $R^{EST}_{Matching}$, then we should be very conservative in dropping databases from the $Chosen_{EST}$ set. With this motivation we define another estimator for *GlOSS*, *Bin* (for "binary"):

$$Estimate_{Bin}(\ t_1 \wedge \ldots \wedge t_n, db_i) \ = \begin{cases} 0 & \text{if } \exists j,\, 1 \leq j \leq n \mid f_{ij} = 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.11}$$

Again, $Chosen_{Bin}$ follows from the definition of $Estimate_{Bin}$, using Equation 3.1. So, we are guaranteed that $R^{Bin}_{Matching} = R^{Bin}_{Best} = R^{Bin}_{Matching_I} = 1$ (at the expense of likely low values for the P parameters).

Figures 3.18 and 3.19 show the results obtained for the basic configuration (Figure 3.6) using the *Min* and *Bin* estimators, respectively. The results for *Min* are very similar to the corresponding results for *Ind*, with no significant differences. Note that the definition of $Estimate_{Min}(q, db)$ does not depend on the size of the *db* database, unlike the definition of $Estimate_{Ind}(q, db)$. This does not seem to have played an important role for the queries and databases we considered in the experiments, since the results we obtained for *Ind* and *Min* are very similar.

As expected, although *Bin* gets much higher values for the R parameters (in fact, $R^{Bin}_{Matching} = R^{Bin}_{Best} = R^{Bin}_{Matching_I} = 1$), it performs much worse for the P parameters than *Ind* and *Min*. For example, P^{Bin}_{Best} is very low: 0.2739. Note that $P^{Bin}_{Matching_I}$ is also low (0.2494), since *Bin* tends to produce overly conservative $Chosen_{Bin}$ sets, so as not to miss any of the databases with matching documents.

Consequently, a user might indicate what the query semantics are to *GlOSS*. *GlOSS* would then choose one of the estimators to answer the user query accordingly. Thus, if the user is interested in high values of the P parameters, then the *Ind* estimator would be used, whereas *Bin* would be the choice if high values of R are of interest. If, on the other hand, the user wants both high values of P and R, then *Ind* would be chosen for *Right* = *Best*, and *Bin* for *Right* = *Matching*.

3.7 *GlOSS*'s Storage Requirements

In this section we study the space requirements of *GlOSS* and compare them with those of a full index of the databases. (See [TGL+97] for a study of data structures to maintain the *GlOSS* information efficiently both for queries and for updates.) We

Right	P^{Min}_{Right}	R^{Min}_{Right}	P^{Ind}_{Right}	R^{Ind}_{Right}
Matching	0.9077	0.4031	0.9126	0.4044
Best	0.8356	0.8938	0.8438	0.9010
$Matching_I$	0.6261	0.7316	0.5966	0.7012

Figure 3.18: The average P and R parameters for the basic configuration with *Min* as the estimator. The last two columns show the corresponding values for the basic configuration, using *Ind* as the estimator.

Right	P^{Bin}_{Right}	R^{Bin}_{Right}	P^{Ind}_{Right}	R^{Ind}_{Right}
Matching	0.7757	1	0.9126	0.4044
Best	0.2739	1	0.8438	0.9010
$Matching_I$	0.2494	1	0.5966	0.7012

Figure 3.19: The average P and R parameters for the basic configuration with *Bin* as the estimator. The last two columns show the corresponding values for the basic configuration, using *Ind* as the estimator.

base our study on real index information about the INSPEC database. To keep the *GlOSS* storage requirements low, we would like *GlOSS* not to store frequency information for field designators like "subject," which are not "primitive" field designators in INSPEC, but are instead derived from other field designators. However, the experiments of Section 3.5 assume that *GlOSS* has frequency information corresponding to such non-primitive indices. Hence, we start our analysis by studying experimentally if the effectiveness of the *Ind* estimator for *GlOSS* is affected by not keeping non-primitive frequency information (Section 3.7.2). After doing this, we are ready to estimate the storage requirements of *GlOSS* (Section 3.7.4). To reduce the *GlOSS* information further, we analyze the impact on the effectiveness of *Ind* of eliminating information on low frequency words (Section 3.7.5). The experiments in this section preceded those in Section 3.5, and used different effectiveness metrics. We start our storage discussion by describing these new metrics first (Section 3.7.1).

3.7.1 New Evaluation Parameters

This section describes the effectiveness metrics for the experiments of Sections 3.7.2 and 3.7.5. The motivation behind these metrics is that even if the *GlOSS* estimates are accurate, the correctness of an answer depends on the query *semantics*, as intended by the user that issued the query. In particular, if the *Right* set of databases for a query q is defined to be the *Best* set of databases for q (Section 3.4.3), then we can define the following two "right" ways of answering q:

- *All-Best Search:* We are interested in searching all of the *Best* databases for q. By searching these databases we seek a compromise between two potentially conflicting goals: obtaining an exhaustive answer to q (this would be guaranteed if we searched all of the databases containing matching documents, not only those containing the highest number of matching documents) and searching databases that would deliver a significant number of answers, to compensate for access costs, for example. Thus, we say that $Chosen_{Ind}$ satisfies criterion C_{AB} if:

  $$C_{AB} : Best \subseteq Chosen_{Ind}$$

  So, we ensure that at least those databases having the highest payoff (i.e., the largest number of matching documents) are searched.

- *Only-Best Search:* We are less demanding than with C_{AB}: we are just interested in searching (some of) the best databases for q. Our goal is to get a sample of the documents that match the query q (we might be missing some of these best databases), but we do not want to waste any time and resources by searching a non-optimal database. So, we say that $Chosen_{Ind}$ satisfies criterion C_{OB} if:

  $$C_{OB} : Chosen_{Ind} \subseteq Best$$

The set $Chosen_{Ind}$ will be said to *strictly satisfy* both criteria C_{AB} and C_{OB} if $Chosen_{Ind} = Best$.

Now, let C be either of the criteria above and Q be a fixed set of queries. Then,

$$Success(C, Ind) = 100 \times \frac{|\{q \in Q | Chosen_{Ind} \text{ satisfies } C\}|}{|Q|} \qquad (3.12)$$

In other words, $Success(C, Ind)$ is the percentage of Q queries for which Ind produced the "right answer" under criterion C.

Following notions analogous to those used in statistics, we define the *Alpha* and the *Beta* errors of *Ind* for an evaluation criterion C as follows:

$$\begin{aligned} Alpha(C, Ind) &= 100 \Leftrightarrow Success(C, Ind) & (3.13) \\ Beta(C, Ind) &= Success(C, Ind) \Leftrightarrow \\ & \quad 100 \times \frac{|\{q \in Q | Chosen_{Ind} \text{ strictly satisfies } C\}|}{|Q|} & (3.14) \end{aligned}$$

So, $Alpha(C, Ind)$ is the percentage of queries in Q for which the estimator gives the "wrong answer," that is, the $Chosen_{Ind}$ set does not satisfy criterion C at all. $Beta(C, Ind)$ measures the percentage of queries for which the estimator satisfies the criterion, but not strictly. For the *Beta* queries, the estimator yields a correct but "overly conservative" (for C_{AB}) or "overly narrow" (for C_{OB}) answer. For example, consider an estimator, *TRIV*, that would always produce $\emptyset$ as the value for $Chosen_{TRIV}$. *TRIV* would have $Success(C_{OB}, TRIV) = 100$ (and $Alpha(C_{OB}, TRIV) = 0$). However, *Beta* has a high value for conservative estimators: $Beta(C_{OB}, TRIV)$ would be quite high.

We now relate *Success*, *Alpha*, and *Beta* to the P and R parameters of Section 3.3. Consider, for example, criterion $C_{AB} : Best \subseteq Chosen_{Ind}$. Having $R^{Ind}_{Best}(q, DB) = 1$ is equivalent to having $Best(q, DB) \subseteq Chosen_{Ind}(q, DB)$. Therefore, *Ind* satisfies criterion C_{AB} for query q and set of databases DB if and only if $R^{Ind}_{Best}(q, DB) = 1$. This is shown in Figure 3.20, in the "*Success*" row, under C_{AB}. That is, $Success(C_{AB}, Ind)$ gives the fraction of the queries for which the condition shown ($R^{Ind}_{Best} = 1$) is true.

Assume now that $R^{Ind}_{Best}(q, DB) = 1$ for some query q. Then, it is also the case that $P^{Ind}_{Best}(q, DB) < 1$ if and only if $Best(q, DB) \subset Chosen_{Ind}(q, DB)$. Therefore, given that q satisfies criterion C_{AB} (or equivalently, $R^{Ind}_{Best}(q, DB) = 1$), q will add to $Beta(q, DB)$ if and only if $P^{Ind}_{Best}(q, DB) < 1$.

Now, consider criterion $C_{OB} : Chosen_{Ind} \subseteq Best$. It follows from the definition of P^{Ind}_{Best} that a query q will "contribute" to $Success(C_{OB}, Ind)$ if and only if $P^{Ind}_{Best}(q, DB) = 1$. The conditions on P^{Ind}_{Best} and R^{Ind}_{Best} for *Beta* are analogous to those

	C_{AB}	C_{OB}
Success	$R^{Ind}_{Best} = 1$	$P^{Ind}_{Best} = 1$
Alpha	$R^{Ind}_{Best} < 1$	$P^{Ind}_{Best} < 1$
Beta	$R^{Ind}_{Best} = 1$ and $P^{Ind}_{Best} < 1$	$P^{Ind}_{Best} = 1$ and $R^{Ind}_{Best} < 1$
Success $\Leftrightarrow$ *Beta*	$R^{Ind}_{Best} = P^{Ind}_{Best} = 1$	$P^{Ind}_{Best} = R^{Ind}_{Best} = 1$

Figure 3.20: Summary of the relationship between the *Success*, *Alpha*, and *Beta* functions and P^{Ind}_{Best} and R^{Ind}_{Best}, for criteria C_{AB} and C_{OB}.

for C_{AB} with the roles of P^{Ind}_{Best} and R^{Ind}_{Best} interchanged.

As a final comment, notice that criterion C_{AB} can be regarded as emphasizing recall over precision: this criterion is satisfied whenever *Best* is included in $Chosen_{Ind}$. On the other hand, C_{OB} can be thought of as emphasizing precision over recall: even when $Chosen_{Ind}$ is not a "complete" answer, success is achieved if no "useless" databases are included in $Chosen_{Ind}$.

3.7.2 Eliminating the "Subject" Index

To keep the *GlOSS* storage requirements low, we would like *GlOSS* not to store frequency information for field designators like "subject," which are not "primitive" field designators in the databases that we considered, but are instead derived from other field designators. Therefore, before we compute the frequency information size, we will analyze the way the "subject" index is treated in the six databases we considered. In all of these databases, "subject" is a compound index, built from other "primitive" indexes. For example, in the INSPEC database, the "subject" index is constructed from the "title," "abstract," "thesaurus," "organization," and "other subjects" indexes: a query *subject computers* is equivalent to the "or" query: *title computers* $\vee$ *abstract computers* $\vee$ *thesaurus computers* $\vee$ *organization computers* $\vee$ *other subjects computers*.

All of the experiments we reported so far treated "subject" as a primitive index, as though *GlOSS* kept the entries corresponding to the "subject" field designation as part of the database frequency information. However, given that *GlOSS* has the entries for the constituent indexes from which the "subject" index is formed, we could attempt to *estimate* the entries corresponding to the "subject" index using the entries for the primitive indexes. This way, we can save space by not having to store entries for the "subject" index.

There are different ways to estimate f_{ij} where $t_j = \mathit{subject}\ w$ for some keyword w, given the primitive indexes $l_1, l_2, \ldots, l_n$ that compose the "subject" index in database db_i. One such way takes the maximum of the individual frequencies for the primitive indexes:

$$f_{ij} \approx \max_{k=1,\ldots,n} f_{ij_k} \tag{3.15}$$

where t_{j_k} has field designation l_k and keyword w. Note that this estimate constitutes a lower bound for the actual value of f_{ij}.

Figure 3.21 shows the results obtained for the basic configuration (Figure 3.6) but estimating the "subject" frequencies as in Equation 3.15, with one difference: only those indexes that actually appeared in $\mathit{TRACE}_{\mathit{INSPEC}}$ queries were considered. The other indexes are seldom used so it does not make sense for *GlOSS* to keep statistics on them. The indexes considered are the ones that are listed in Figure 3.22. For example, we simply ignored the "other subjects" index for the INSPEC database. The last column in Figure 3.21 shows the *Success* figures for the basic configuration, using the exact frequencies for the "subject" index: there is very little change in performance if we estimate the "subject" frequencies as in Equation 3.15 [4]. Therefore, when we compute the size of the *GlOSS* frequency information in the next section, we will assume that *GlOSS* does not store "subject" entries. Thus, we will consider only primitive indexes that appear in $\mathit{TRACE}_{\mathit{INSPEC}}$ queries.

Criteria	*Success*	*Alpha*	*Beta*	*Success* $\Leftrightarrow$*Beta*	*Success*
C_{AB}	88.23	11.77	6.93	81.30	88.95
C_{OB}	83.82	16.18	2.52	81.30	84.38

Figure 3.21: Evaluation criteria for the basic configuration, but estimating the "subject" frequencies as the *maximum* of the frequencies of the primitive indexes. The last column shows the *Success* values for the basic configuration, using the exact "subject" frequencies.

Field Designator	*Full Index*	*GlOSS (threshold=0)*
	# of postings	*# of entries*
Author	4108027	311632
Title	10292321	171537
Publication	6794557	18411
Abstract	74477422	487247
Thesaurus	11382655	3695
Conference	7246145	11934
Organization	9374199	62051
Class	4211136	2962
Numbers (ISBN, ...)	2445828	12637
Report Numbers	7833	7508
Totals	130,340,123	1,089,614

Figure 3.22: Characteristics of the database frequency information kept by *GlOSS* vs. those of a full index, for the INSPEC database.

3.7.3 Characteristics of the Database Frequency Information and Full Indexes

As explained in Section 3.2.2, *GlOSS* needs to keep, for each database, the number of documents that satisfy each possible keyword field-designation pair. Figure 3.22 was generated using information of the corresponding INSPEC indexes obtained from Stanford's FOLIO library information retrieval system. The "# of entries" column reports the number of entries required for each of the INSPEC indexes appearing in the $TRACE_{INSPEC}$ queries. For example, there are $311,632$ different author last names appearing in INSPEC (field designation "author"), and each will have an associated entry in the INSPEC frequency information. A total of $1,089,614$ entries will be required for the INSPEC database. Each of these entries will correspond to a keyword field-designation pair and its associated frequency (e.g., <*author Knuth, 47*>, meaning that there are 47 documents in INSPEC with *Knuth* as the *author*). In contrast, if we were to keep the complete inverted lists associated with the different indexes we considered, $130,340,123$ postings would have to be stored in the full index.

3.7.4 Storage Cost Estimates

In the following, we will roughly estimate the space requirements of a full index vs. those of the frequency information kept by *GlOSS*. We start our analysis with the INSPEC database, and then consider all six databases. The figures we will produce should be taken just as an indication of the relative order of magnitude of the corresponding requirements.

Each of the *postings* of a full index will typically contain a field designation and a document identifier. If we dedicate one byte for the field designation and three bytes for the document identifier, we end up with four bytes per posting. Let us assume that, after compression, two bytes suffice per posting (compression of 50% is typical for inverted lists).

[4]In [GGMT93] we explore an alternative estimate for the "subject" frequencies whose corresponding experimental results were very similar to those for the Equation 3.15 estimate.

Size of	*Full Index*	*GlOSS/threshold=0*
Vocabulary	3.13 MBytes	3.13 MBytes
Index	248.60 MBytes	2.60 MBytes
Total	251.73 MBytes	5.73 MBytes
% of Full Index	100	2.28

Figure 3.23: Estimated storage costs of a full index vs. the *GlOSS* frequency information for the INSPEC database.

Each of the *frequencies* kept by *GlOSS* will typically contain a field designation, a database identifier, and the frequency itself. Regarding the size of the frequencies themselves, only 1417 keyword field-designation pairs in INSPEC have more than 2^{16} documents containing them. Therefore, in the vast majority of the cases, two bytes suffice to store these frequencies, according to the INSPEC data we have available. We will thus assume that we dedicate two bytes per frequency. So, using one byte for the field designation and two bytes for the database identifier, we end up with five bytes per frequency. Again, after compression we will assume that 2.5 bytes are required per frequency. Using the data from Figure 3.22 and our estimates for the size of each posting and frequency information entry, we obtain the index sizes shown in Figure 3.23 ("Index" row).

The vocabulary for INSPEC [5], including only indexes that appear in $TRACE_{INSPEC}$ queries, consists of 819,437 words. If we dedicate four bytes to store each keyword (see [GGMT93]), around $4 \times 819{,}437$ bytes, or 3.13 MBytes are needed to store the INSPEC vocabulary. This is shown in the "Vocabulary" row of Figure 3.23.

After adding the vocabulary and index sizes ("Total" row of Figure 3.23), the size of the frequency information that *GlOSS* needs is only around 2.28% the size of the corresponding full index, for the INSPEC database.

So far, we have only focused on the space requirements of a single database, namely INSPEC. We will base the space requirement estimates for the six databases on the figures for the INSPEC database, for which we have reliable index information. To do this, we multiply the different values we calculated for INSPEC by a growth factor

[5]The field designators are stored with each posting and frequency, as described above.

G (see Figure 3.4):

$$G = \frac{\sum_{db \in DB} |db|}{|\text{INSPEC}|} \approx 4.12$$

where $DB = \{\text{INSPEC, COMPENDEX, ABI, GEOREF, ERIC, PSYCINFO}\}$. Therefore, the number of postings required by a full index of the six databases is estimated as $G \times$ INSPEC number of postings $= 537,001,307$ postings, or around 1024.25 MBytes. The number of frequencies required by *GlOSS* for the six databases is estimated as $G \times$ INSPEC number of frequencies $= 4,489,210$ frequencies, or around 10.70 MBytes (see the "Index" row of Figure 3.24).

The space occupied by the index keywords of the six databases considered will be proportional to the size of their *merged* vocabularies. Using index information from Stanford's FOLIO system, we can determine that the size of the merged vocabulary of the six databases we considered is approximately 90% of the sum of the six individual vocabulary sizes. Therefore, we estimate the size of the merged vocabulary for the six databases as $G \times 0.90 \times$ INSPEC vocabulary size = 3,038,472 words, or around 11.59 MBytes (see the "Vocabulary" row of Figure 3.24).

Figure 3.24 summarizes the storage estimates for *GlOSS* and a full index. Note that the *GlOSS* frequency information is only 2.15% the size of the full index. This is even less than the corresponding figure we obtained above just for the INSPEC database (2.28%). The reason for this is the fact that the merged vocabulary size is only 90% of the sum of the individual vocabulary sizes. Although this 10% reduction "benefits" both *GlOSS* and the full index case, the impact on *GlOSS* is higher, since the vocabulary size is a much larger fraction of the total storage needed by *GlOSS* than it is for the full index.

We have obtained the numbers of Figure 3.24 using some very rough estimates and approximations, so they should be taken cautiously. However, we think they are useful to illustrate the low space requirements of *GlOSS*: around 22.29 MBytes would suffice to keep the word frequencies for the six databases that we studied.

Size of	*Full index*	*GlOSS/threshold=0*
Vocabulary	11.59 MBytes	11.59 MBytes
Index	1024.25 MBytes	10.70 MBytes
Total	1035.84 MBytes	22.29 MBytes
% of Full index	100	2.15
$Success(C_{AB}, \_)$	100	88.23
$Success(C_{OB}, \_)$	100	83.82
$Success(C_{OB}, \_) - Beta(C_{OB}, \_)$	100	81.30

Figure 3.24: Storage estimates for *GlOSS* and a full index for the six databases. The entries for *GlOSS* in the last three rows correspond to the basic configuration, but estimating the "subject" frequencies as the maximum of the frequencies of the primitive indexes.

3.7.5 Pruning the Word-Frequency Information

To further reduce the amount of information that we keep about each database, we introduce the notion of a *threshold*. If a database db_i has fewer than *threshold* documents with a given keyword-field pair t_j, then *GlOSS* will not keep this information. Therefore, *GlOSS* will assume that f_{ij} is zero whenever this data is needed.

As a result of the introduction of *threshold*, the estimator may now conclude that some database db_i does not contain any documents matching a query of the form $t_1 \wedge \ldots \wedge t_n$ if f_{ij} is missing, for some j, while in fact db_i does contain documents matching the query. This situation was not possible before: if f_{ij} was missing from the information set of the estimator, then $f_{ij} = 0$, and so, there could be no documents in db_i satisfying such a query.

To see if *Ind*'s performance deteriorates by the use of this *threshold*, Figures 3.25 and 3.26 show some results for different values of *threshold*, for the basic configuration, but estimating the "subject" index entries as in Equation 3.15. These figures show that the performance for the different criteria is only slightly sensitive to (small) increases in *threshold*. Ironically, the *Success* values for criterion C_{OB} tend to improve for higher values of *threshold*. The reason for this is that $Chosen_{Ind}$ does not include databases with $Estimate_{Ind} = 0$. By increasing *threshold*, the number of such databases will presumably increase, thus making $Chosen_{Ind}$ smaller, and more likely

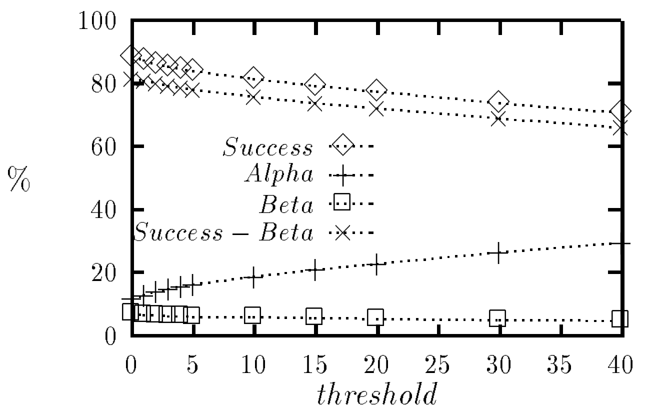

Figure 3.25: Criterion C_{AB}, for different values of *threshold*. The "subject" entries are estimated as the maximum of the entries corresponding to the primitive indexes.

to satisfy C_{OB} : $Chosen_{Ind} \subseteq Best$.

The reason for introducing *threshold*s is to have to store less information for the estimator. Figure 3.27 reports the number of entries that would be left, for different field designators, in the frequency information for the INSPEC database. Some field designators (e.g., "thesaurus") are not affected much by this pruning of the smallest entries, whereas the space requirements for some others (e.g., "author," "title," and "abstract") are reduced drastically. Adding together all of the indexes, the number of entries in the INSPEC frequency information kept by *GlOSS* decreases very fast as *threshold* increases: for *threshold*=1, for instance, 508,978 entries, or 46.71% of the original number of entries, are eliminated. Therefore, the size of the *GlOSS* frequency information can be substantially reduced beyond the already small size estimated in Figure 3.24.

3.8 Conclusions

In this chapter we presented several estimators for *GlOSS*, a solution to the text-source discovery problem. We also developed a formal framework for this problem and defined different "right sets" of databases for evaluating a user's query. We used

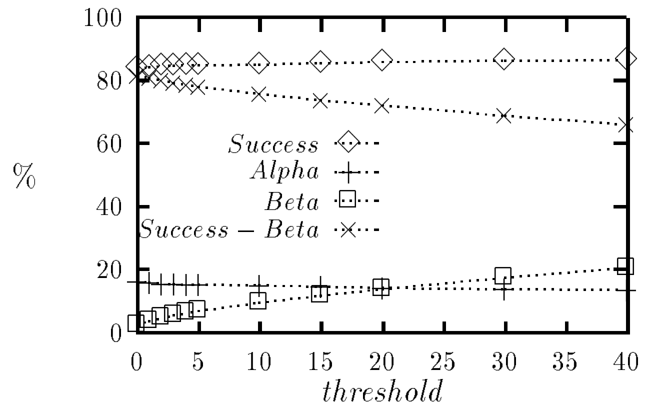

Figure 3.26: Criterion C_{OB}, for different values of *threshold*. The "subject" entries are estimated as the maximum of the entries corresponding to the primitive indexes.

Field Designator	*threshold*					
	0	1	2	3	4	5
Author	311632	194769	150968	125220	107432	94248
Title	171537	85448	62759	51664	44687	40007
Publication	18411	11666	10042	9281	8832	8535
Abstract	487247	227526	163644	133323	115237	102761
Thesaurus	3695	3682	3666	3653	3641	3637
Conference	11934	10138	9887	9789	9702	9653
Organization	62051	34153	26518	22612	20121	18382
Class	2962	2953	2946	2937	2931	2926
Numbers (ISBN, ...)	12637	10199	10067	9946	9865	9779
Report Numbers	7508	102	37	22	14	12
Totals	1089614	580636	440534	368447	322462	289940
%	100	53.29	40.43	33.81	29.59	26.61
Success(C_{AB}, *Ind*)	88.23	87.12	86.07	85.28	84.44	83.82
Success(C_{OB}, *Ind*)	83.82	84.24	84.53	84.65	84.76	84.79
Success(C_{OB}, *Ind*) − *Beta*(C_{OB}, *Ind*)	81.30	80.64	79.85	79.15	78.44	77.85

Figure 3.27: Number of entries left for the different thresholds and field designators in the INSPEC database. The last three rows correspond to the basic configuration, but estimating the "subject" frequencies as the maximum of the frequencies of the primitive indexes.

this framework to evaluate the effectiveness of the *GlOSS* estimators using real-user query traces. The experimental results we obtained are encouraging. Furthermore, we believe that our results are independent of the query traces we used, since we obtained very similar results using two different query traces.

The storage cost of *GlOSS* is relatively low: a rough estimate suggested that 22.29 MBytes would be enough to keep all the data needed for the six databases we studied. In contrast, a full index of the six databases was estimated to require 1035.84 MBytes. Given its low space requirement, we can replicate *GlOSS* to increase its availability.

Our approach to solving the text-source discovery problem could also deal with information servers that charge for their use. Since we are selecting what databases to search according to a quantitative measure of their "goodness" for a query (given by $Estimate_{EST}$), we could easily incorporate this cost factor into the computation of $Estimate_{EST}$ so that, for example, given two equally promising databases, a higher value would be assigned to the less expensive of the two.

We have implemented a *GlOSS* server that keeps information on the 40+ collections of computer science technical reports that are part of the NCSTRL project (`http://www.ncstrl.org`). The *GlOSS* server is available on the World-Wide Web at `http://gloss.stanford.edu`.

Chapter 4

gGlOSS: Vector-Space Source Discovery

In Chapter 3 we described *GlOSS* , a centralized server that keeps meta-information about databases supporting the *Boolean* model of document retrieval. Although the Boolean model of document retrieval is widely used, it is a rather primitive one. One of the most popular alternative models is the *vector-space retrieval model* [Sal89, SM83]. This model represents both the documents in a database and the queries themselves as weight vectors. Given a query, the documents are ranked according to how "similar" their corresponding vectors are to the given query vector.

In this chapter we present *gGlOSS*, a generalized and more powerful version of *GlOSS* that also deals well with vector-space databases and queries. Like *GlOSS*, *gGlOSS* periodically collects statistics on the underlying sources (this time including summary word-weight information). We first determine the *goodness* of a database for a query, and the *ideal database rank* for a query (i.e., the rank that *gGlOSS* should try to produce for the query). Then, given a query and a desired goodness metric, *gGlOSS* can rank the available sources.

Since *gGlOSS* produces estimates of the ideal database ranks, we need to compare these estimates against the ideal ranks. For this, we evaluate the performance of *gGlOSS* using real-user queries and 53 vector-space databases, in terms of how close the *gGlOSS* ranks are to the ideal ones. Although we can estimate the size of the *gGlOSS* information to be only around 2% of the size of a full index of the databases, as we discussed in Chapter 3, its performance is good (Section 4.4), showing that *gGlOSS* can closely approximate the ideal database ranks for the given queries.

We also present facilities for building hierarchies of *gGlOSS* servers. In this case, *hGlOSS*, a high-level server, summarizes the contents of lower-level *gGlOSS* servers, in much the same way as the *gGlOSS* servers summarize the contents of the underlying databases. Given a query, the *hGlOSS* server suggests *gGlOSS* servers that might index useful databases for the query. Because the storage requirements of the *hGlOSS* server are much smaller than those of the *gGlOSS* servers, we can easily replicate the *hGlOSS* server so that it does not become a performance bottleneck, thus distributing the load for searching the system.

In what follows, Section 4.1 defines one "ideal" database rank for a query. Section 4.2 shows how *gGlOSS* approximates the ideal database rank using partial information. Section 4.3 introduces the methodology for the experimental results of Section 4.4. Section 4.5 discusses alternative definitions of the ideal database rank. Section 4.6 shows how to build the higher-level *hGlOSS* servers. Finally, Section 4.7 reports experimental results for the Boolean *GlOSS* (Chapter 3) using the metrics presented in this chapter and 500 text sources.

4.1 Ranking Databases

Given a query, we would like to rank the available vector-space databases according to their usefulness, as in Section 3.4.3 for Boolean databases. This ranking should capture the ideal order for searching the databases: we should first search the most useful database(s), then the second most useful database(s), and so on, until we either exhaust the rank, or become satisfied with whatever documents we got up to that point. This section presents one definition for the *ideal database rank* for vector-space sources. The next section explores how *gGlOSS* will try to rank the databases as closely as possible to this ideal rank.

Determining the ideal database rank for a query is a hard problem. As in Section 3.4.3, the definition of this section is based solely on the answers (i.e., the document ranks and their scores) that each database produces when presented with the query in question. This definition does not use the relevance [SM83] of the documents to the end user who submitted the query. Using relevance would be appropriate for evaluating the search engines at each database; instead, we are evaluating how well *gGlOSS* can predict the answers that the databases return. In Section 4.5 we discuss our choice further, and analyze some of the possible alternatives that we could have used.

To define the ideal database rank for a query q, we need to determine how good each database db is for q. In this chapter we assume that all databases use the same algorithms to compute weights and similarities. (See Chapter 2.) We consider that the only documents in db that are useful for q are those with a similarity to q greater than a given threshold l, as determined by db. Documents with lower similarity are unlikely to be useful, and therefore we ignore them. Thus, we define:

$$Goodness(l, q, db) = \sum_{d \in Rank(l,q,db)} sim(q, d) \tag{4.1}$$

where $sim(q, d)$ is the similarity between query q and document d, and $Rank(l, q, db) = \{d \in db | sim(q, d) > l\}$. The ideal rank of databases $Ideal(l)$ is then determined by sorting the databases according to their goodness for the query q.

Example 15: Consider two databases, db_1 and db_2, a query q, and the answers that the two databases give when presented with query q:

$$\begin{aligned} db_1 &: (d^1_1, 0.9), (d^1_2, 0.9), (d^1_3, 0.1) \\ db_2 &: (d^2_1, 0.8), (d^2_2, 0.4), (d^2_3, 0.3), (d^2_4, 0.1) \end{aligned}$$

In the example, db_1 returns documents d^1_1, d^1_2, and d^1_3 as its answer to q. Documents d^1_1 and d^1_2 are ranked the highest in the answer, because they are the "closest" to query q in database db_1 (similarity 0.9). To determine how good each of these databases is for q, we use Equation 4.1. If threshold l is 0.2 (i.e., the user is willing to examine every document with similarity to q higher than 0.2), the goodness of db_1 is $Goodness(0.2, q, db_1) = 0.9 + 0.9 = 1.8$, because db_1 has two documents, d_1^1 and d_2^1, with similarity higher than 0.2. Similarly, $Goodness(0.2, q, db_2) = 0.8+0.4+0.3 = 1.5$. Therefore, $Ideal(0.2)$ is db_1, db_2. ∎

The goodness of a database tries to quantify how useful the database is for the user that issued the query. It does so by examining the document-query similarities as computed by each local source. A problem with this definition is that these similarities can depend on the characteristics of the collection that contains the document. Therefore, these similarities are not "globally valid." For example, if a database db_1 specializes in computer science, the word *databases* might appear in many of its documents. Then, this word will tend to have a low associated weight in db_1 (e.g., if db_1 uses the *tf·idf* formula for computing weights [Sal89]). The word *databases*, on the other hand, might have a high associated weight in a database db_2 that is totally unrelated to computer science and contains very few documents with that word. Consequently, db_1 might assign its documents a low score for a query containing the word *databases*, while db_2 assigns a few documents a high score for that query. The *Goodness* definition of Equation 4.1 might then determine that db_2 is better than db_1, while db_1 is the best database for the query. In Section 4.5 we further discuss this problem, together with alternative ways of defining *Goodness*.

4.2 Choosing Databases

gGlOSS helps users (and in particular, metasearchers) determine what databases might be most helpful for a query. *gGlOSS* ranks the databases according to their potential usefulness for a given query. To perform this task, *gGlOSS* keeps information on the available databases, to estimate their goodness for the query. As in the Boolean case, one option would be for *gGlOSS* to keep complete information on each database: for each database db and word t, *gGlOSS* would know what documents in db contain t, what weight t has in each of them, and so on. Although *gGlOSS*'s answers would always be accurate (if this information is kept up to date), the storage requirements of such an approach would be too high: *gGlOSS* needs to index many databases, and keeping so much information on each of them does not scale. Furthermore, this information might not be available for commercial databases, for example.

More reasonable solutions keep incomplete yet useful information on the databases. In this chapter we explore some options for *gGlOSS* that require one or both of the following matrices:

- $F = (f_{ij})$: f_{ij} is the number of documents in database db_i that contain word t_j
- $W = (w_{ij})$: w_{ij} is the sum of the weight of word t_j over all documents in database db_i

In other words, for each word t_j and each vector-space database db_i, *gGlOSS* needs (at most) two numbers. The second of these numbers is the sum of the weight of t_j over all documents in db_i, as determined by the vector-space retrieval algorithm that db_i uses. Typically, the weight of a word t_j in a document d is a function of the number of times that t_j appears in d and the number of documents in the database that contain t_j [Sal89]. Although the information that *gGlOSS* stores about each database is incomplete, it will prove useful to generate database ranks that resemble the ideal database rank of Section 4.1, as we will see in Section 4.4.2. Furthermore, this information is orders of magnitude smaller than that required by a full-text index of the databases (Section 3.7).

To obtain the data that *gGlOSS* keeps about a database db_i, namely rows f_{i*} and w_{i*} of the F and W matrices above, database db_i will have to periodically run a *collector* program that extracts this information from the local indexes and sends it to the *gGlOSS* server, or export this information using the *STARTS* protocol of Chapter 2, for example.

Example 16: Consider a database *db* and the word *computer*. Suppose that the following are the documents in *db* having the word *computer* in them, together with the associated weights:

$$computer: (d_1, 0.8), (d_2, 0.7), (d_3, 0.9), (d_8, 0.9)$$

That is, document d_1 contains the word *computer* with weight 0.8 (for some weight-computation algorithm [SM83]), document d_2, with weight 0.7, and so on. Database *db* will not export all this information to *gGlOSS*: it will only tell *gGlOSS* that the word *computer* appears in four documents in database *db*, and that the sum of the weights with which the word appears in the documents is $0.8+0.7+0.9+0.9=3.3$. ∎

In our definitions below, we assume that a query q is expressed as a weight vector $Q=(q_1,\ldots,q_j,\ldots,q_t)$ [SM83], where q_j is the weight of word t_j in query q. For example, this weight can simply be the number of times that word t_j appears in the query. We also assume throughout this chapter that the vector-space databases compute the similarity between a document and a query by taking the inner product of the corresponding document and query weight vectors.

Since *gGlOSS* represents both the databases and the queries as vectors, *gGlOSS* could compute similarities between these vectors analogously to how *documents* and queries are compared. *gGlOSS* could use these similarities to rank the databases for the given query. For example, *gGlOSS* could estimate the goodness of database db_i for query q as the inner product $w'_{i*} \cdot Q$, where $w'_{i*}=(w'_{i1},\ldots,w'_{it})$ is the (normalized) row of W that corresponds to db_i. However, we are interested in finding the databases that contain useful documents for the queries, not those databases that are "similar" to the given queries. The definitions of the *gGlOSS* ranks below reflect this fact. Also, note that the vectors with which *gGlOSS* represents each database can be viewed as *cluster centroids* [Sal89], where each database is considered as a single document cluster [1].

Because the information that *gGlOSS* keeps about each database is incomplete, it has to make assumptions regarding the distribution of query keywords and weights across the documents of each database. These assumptions allow *gGlOSS* to compute better estimates. The following sections present two sets of assumptions that *gGlOSS* will use to derive different database ranks for a given query. These assumptions are

[1] An interesting direction to explore is to represent each database *db* as a set of (very few) cluster centroids. Each of these centroids would summarize a set of closely related documents of *db*.

artificial: very rarely would a set of databases and queries conform to them. However, we use them because these type of assumptions proved themselves useful for Boolean *GlOSS* to choose the "right" databases for a query (Chapter 3).

4.2.1 High-Correlation Scenario

To derive $Max(l)$, the first database rank with which *gGlOSS* tries to match the $Ideal(l)$ database rank of Section 4.1, *gGlOSS* assumes that if two words appear together in a user query, then these words will appear in the database documents with the highest possible correlation:

Assumption 1: *If query keywords t_1 and t_2 appear in f_{i1} and f_{i2} documents in database db_i, respectively, and $f_{i1} \leq f_{i2}$, then every db_i document that contains t_1 also contains t_2.*

Example 17: Consider a database db_i and the query q=*computer science department*. For simplicity, let t_1= *computer*, t_2= *science*, and t_3= *department*. Suppose that $f_{i1} = 2$, $f_{i2} = 9$, and $f_{i3} = 10$: there are 2 documents in db_i with the word *computer*, 9 with the word *science*, and 10 with the word *department*.

gGlOSS assumes that the 2 documents with the word *computer* also contain the words *science* and *department*. Furthermore, all of the $9 - 2 = 7$ documents with word *science* but not with word *computer* also contain the word *department*. Finally, there is exactly $10 - 9 = 1$ document with just the word *department*. ∎

gGlOSS also needs to make assumptions on the weight distribution of the words across the documents of a database:

Assumption 2: *The weight of a word is distributed uniformly over all documents that contain the word.*

Thus, word t_j has weight $\frac{w_{ij}}{f_{ij}}$ in every db_i document that contains t_j. This assumption simplifies the computations that *gGlOSS* has to make to rank the databases. We will see in Section 4.4 that this unrealistic assumption is surprisingly effective.

Example 17: (cont.) Suppose that the total weights for the query words in database db_i are $w_{i1} = 0.45$, $w_{i2} = 0.2$, and $w_{i3} = 0.9$. According to Assumption 2, each of the two documents that contain word *computer* will do so with weight $\frac{0.45}{2} = 0.225$, each of the 9 documents that contain word *science* will do so with weight $\frac{0.2}{9} = 0.022$, and so on. ▮

gGlOSS uses the assumptions above to estimate how many documents in a database have similarity greater than some *threshold* l to a given query, and what the added similarity of these documents is. These estimates determine the $Max(l)$ database rank.

Consider database db_i with its two associated vectors f_{i*} and w_{i*}, and query q, with its associated vector Q. Suppose that the words in q are $t_1, \ldots, t_n$, with $f_{ia} \leq f_{ib}$ for all $1 \leq a \leq b \leq n$. Assume that $f_{i1} > 0$. From Assumption 1, the f_{i1} documents in db_i that contain word t_1 also contain all of the other $n \Leftrightarrow 1$ query words. From Assumption 2, the similarity of any of these f_{i1} documents to the query q is:

$$sim_1 = \sum_{j=1,\ldots,n} q_j \times \frac{w_{ij}}{f_{ij}}$$

Furthermore, these f_{i1} documents have the highest similarity to q among the documents in db_i. Therefore, if $sim_1 \leq l$, then there are no documents in db_i with similarity greater than threshold l. If, on the other hand, $sim_1 > l$, then *gGlOSS* should explore the $f_{i2} \Leftrightarrow f_{i1}$ documents (Assumption 1) that contain words $t_2, \ldots, t_n$, but not word t_1. Thus, *gGlOSS* finds p such that:

$$sim_p = \sum_{j=p,\ldots,n} q_j \times \frac{w_{ij}}{f_{ij}} \quad > \quad l, \text{ but} \tag{4.2}$$

$$sim_{p+1} = \sum_{j=p+1,\ldots,n} q_j \times \frac{w_{ij}}{f_{ij}} \quad \leq \quad l \tag{4.3}$$

Then, the f_{ip} documents having (at least) query words $t_p, \ldots, t_n$ have an estimated similarity to q greater than threshold l (Condition 4.2), whereas the documents having only query words $t_{p+1}, \ldots, t_n$ do not.

Using this definition of p and the assumptions above, we give the first definition for $Estimate(l, q, db_i)$, the estimated goodness of database db_i for query q, that determines the $Max(l)$ database rank:

$$\begin{aligned} Estimate(l, q, db_i) &= \sum_{j=1,\ldots,p} (f_{ij} \Leftrightarrow f_{i(j-1)}) \times sim_j \\ &= (\sum_{j=1,\ldots,p} q_j \times w_{ij}) + f_{ip} \times \sum_{j=p+1,\ldots,n} q_j \times \frac{w_{ij}}{f_{ij}} \end{aligned} \tag{4.4}$$

where we define $f_{i0} = 0$, and sim_j is the similarity between q and any document having words $t_j, \ldots, t_n$, but not words $t_1, \ldots, t_{j-1}$. There are $f_{ij} \Leftrightarrow f_{i(j-1)}$ such documents in db_i. This definition computes the added similarity of the f_{ip} documents estimated to have similarity to q greater than threshold l. (See Conditions 4.2 and 4.3, and Assumptions 1 and 2.)

Example 17: (cont.) Assume that query q has weight 1 for each of its three words. According to Assumption 1, the two documents with the word *computer* also have the words *science* and *department* in them. The similarity of any of these two documents to q is, using Assumption 2, $\frac{0.45}{2} + \frac{0.2}{9} + \frac{0.9}{10} = 0.337$. If our threshold l is 0.2, then all of these documents are acceptable, because their similarity to q is higher than 0.2. Also, there are $9 \Leftrightarrow 2 = 7$ documents with the words *science* and *department* but not *computer*. The similarity of any of these 7 documents to q is $\frac{0.2}{9} + \frac{0.9}{10} = 0.112$. Then these documents are not acceptable for threshold $l = 0.2$. There is $10 \Leftrightarrow 9 = 1$ document with only the word *department*, but this document's similarity to q is even lower. Consequently, $p = 1$. (See Conditions 4.2 and 4.3.) Then, according to the $Max(0.2)$ definition of *Estimate*, $Estimate(0.2, q, db_i) = f_{i1} \times (q_1 \times \frac{w_{i1}}{f_{i1}} + q_2 \times \frac{w_{i2}}{f_{i2}} + q_3 \times \frac{w_{i3}}{f_{i3}}) = 2 \times (1 \times \frac{0.45}{2} + 1 \times \frac{0.2}{9} + 1 \times \frac{0.9}{10}) = 0.674$. ∎

4.2.2 Disjoint Scenario

To derive $Sum(l)$, another rank that *gGlOSS* uses to approximate $Ideal(l)$, *gGlOSS* assumes that if two words appear together in a user query, then these words do not appear together in any database document (if possible):

Assumption 3: *The set of db_i documents with word t_1 is disjoint with the set of db_i documents with word t_2, for all t_1 and t_2, $t_1 \neq t_2$, that appear in query q.*

Therefore, the words that appear in a user query are assumed to be negatively correlated in the database documents. *gGlOSS* also needs to make Assumption 2, that is, the assumption that weights are uniformly distributed.

Consider database db_i with its two associated vectors f_{i*} and w_{i*}, and query q, with its associated vector Q. Suppose that the words in q are $t_1, \ldots, t_n$. For any query word t_j ($1 \leq j \leq n$), then the f_{ij} documents containing t_j do not contain query word t_p, for all $1 \leq p \leq n$, $p \neq j$ (Assumption 3). Furthermore, the similarity of each of these f_{ij} documents to q is exactly $q_j \times \frac{w_{ij}}{f_{ij}}$, if $f_{ij} > 0$ (from Assumption 2).

For rank $Sum(l)$ we then define $Estimate(l, q, db_i)$, the estimated goodness of database db_i for query q, as:

$$\begin{aligned} Estimate(l, q, db_i) &= \sum_{j=1,\ldots,n \mid (f_{ij}>0) \wedge (q_j \times \frac{w_{ij}}{f_{ij}} > l)} f_{ij} \times (q_j \times \frac{w_{ij}}{f_{ij}}) \\ &= \sum_{j=1,\ldots,n \mid (f_{ij}>0) \wedge (q_j \times \frac{w_{ij}}{f_{ij}} > l)} q_j \times w_{ij} \qquad (4.5) \end{aligned}$$

Example 18: Consider the data of Example 17. According to Assumption 3, there are 2 documents containing the word *computer* and none of the other query words, 9 documents containing the word *science* and none of the other query words, and 10 documents containing the word *department* and none of the other query words. The documents in the first group have similarity $\frac{0.45}{2} = 0.225$ (from Assumption 2), and are thus acceptable, because our threshold l is 0.2. The documents in the second and third groups have similarity $\frac{0.2}{9} = 0.022$ and $\frac{0.9}{10} = 0.09$, respectively, and are thus not acceptable for our threshold. So, the only documents close enough to query q are the two documents that contain word *computer*. Then, according to the $Sum(0.2)$ definition of *Estimate*, $Estimate(0.2, q, db_i) = f_{i1} \times \frac{w_{i1}}{f_{i1}} = 0.45$. ∎

Notice the special case when the threshold l is zero. In this case, the $Max(0)$ and $Sum(0)$ definitions of *Estimate* (Equations 4.4 and 4.5) become:

$$Estimate(0, q, db_i) = \sum_{j=1,\ldots,n} q_j \times w_{ij}$$

assuming that if $f_{ij} = 0$, then $w_{ij} = 0$. Then, $Estimate(0, q, db_i)$ becomes the inner product $Q \cdot w_{i*}$. To compute the $Max(0)$ and $Sum(0)$ ranks, *gGlOSS* does not need the matrix F of document frequencies of the words; it only needs the matrix W of added weights. [2] Therefore, the storage requirements for *gGlOSS* to compute the database ranks may be much lower if $l = 0$. We pay special attention to these ranks in our experiments of Section 4.4.2.

4.3 Comparing Database Ranks

In this section we analyze how we can compare *gGlOSS*'s ranks (Section 4.2) to the ideal one (Section 4.1). In the following section we report experimental results using the comparison methodology of this section.

Let q be a query, and $DB = \{db_1, \ldots, db_s\}$ be the set of available databases. Let $G = (db_{g_1}, \ldots, db_{g_{s'}})$ be the database rank that *gGlOSS* generated for q, using one of the schemes of Section 4.2. We only include in G those databases with estimated goodness greater than zero: we assume that users ignore databases with zero estimated goodness. Thus, in general, $s' \leq s$. Finally, let $I = (db_{i_1}, \ldots, db_{i_{s''}})$ be the ideal database rank. We only include in I those databases with actual goodness greater than zero. Our goal is to compare G against I, and quantify how close the two ranks are.

One way to compare the G and I ranks is by using the *Goodness* metric that we used to build I. We consider the top n databases in rank I, and compute i_n, the accumulated goodness of these n databases for query q. Because rank I was generated using this metric, the top n databases in rank I have the maximum accumulated

[2]We might need F, though, to compute the weight vector for the queries, depending on the algorithm used for this.

goodness for q that any subset of n databases of DB can have. We then consider the top n databases in rank G, and compute g_n, the accumulated goodness of these n databases for q. Because *gGlOSS* generated rank G using only partial information about the databases, in general $g_n \leq i_n$. (If $n > s'$ (resp. $n > s''$), we compute g_n (i_n) by just taking the s' (s'') databases in G (I).) We then compute:

$$\mathcal{R}_n = \begin{cases} \frac{g_n}{i_n} & \text{if } i_n > 0 \\ 1 & \text{otherwise} \end{cases}$$

This number gives us the fraction of the optimum goodness (i_n) that *gGlOSS* captured in the top n databases in G, and models what the user that searches the top n databases that *gGlOSS* suggests would get, compared to what the user would have gotten by searching the top n databases in the ideal rank.

Example 19: Consider a query q, and five databases db_i, $1 \leq i \leq 5$. Figure 4.1 shows I, the ideal database rank, and G and H, two different *gGlOSS* database ranks for q, for some definition of these ranks. For example, db_1 is the top database in the ideal rank, with $Goodness(l, q, db_1) = 0.9$. Database db_5 does not appear in rank I, because $Goodness(l, q, db_5) = 0$. *gGlOSS* correctly predicted this for rank G ($Estimate(l, q, db_5) = 0$ for G), and so db_5 does not appear in G. However, db_5 does appear in H, because $Estimate(l, q, db_5) = 0.2$ for H.

Let us focus on the G rank: db_2 is the top database in G, with $Estimate(l, q, db_2) = 0.8$. The real goodness of db_2 for q is $Goodness(l, q, db_2) = 0.4$. From the ranks of Figure 4.1, $\mathcal{R}_1 = \frac{0.4}{0.9}$: if we access db_2, the top database from the G rank, we obtain $Goodness(l, q, db_2) = 0.4$, whereas the best database for q is db_1, with $Goodness(l, q, db_1) = 0.9$. Similarly, $\mathcal{R}_3 = \frac{0.4+0.9+0.3}{0.9+0.4+0.3} = 1$. In this case, by accessing the top three databases in the G rank we access exactly the top three databases in the ideal rank, and thus $\mathcal{R}_3 = 1$. However, $\mathcal{R}_4 = \frac{0.4+0.9+0.3}{0.9+0.4+0.3+0.2} = 0.89$, since the G rank does not include db_4 ($Estimate(l, q, db_4) = 0$), which is actually useful for q ($Goodness(l, q, db_4) = 0.2$).

Now consider the H rank. H includes all the databases that have $Goodness > 0$ in exactly the same order as G. Therefore, the $\mathcal{R}_n$ metric for H coincides with that for

I		*G*		*H*	
db	*Goodness*	*db*	*Estimate*	*db*	*Estimate*
db_1	0.9	db_2	0.8	db_2	0.9
db_2	0.4	db_1	0.6	db_1	0.8
db_3	0.3	db_3	0.3	db_3	0.4
db_4	0.2			db_5	0.2

Figure 4.1: The ideal and *gGlOSS* database ranks for Example 19.

G, for all n. However, rank G is in some sense better than rank H, since it predicted that db_5 has zero goodness, as we mentioned above. H failed to predict this. The $\mathcal{R}_n$ metric does not distinguish between the two ranks. This is why we introduce our following metric. ∎

As the previous example motivated, we need another metric, $\mathcal{P}_n$, to distinguish between *gGlOSS* ranks that include useless databases and those that do not. Given a *gGlOSS* rank G for query q, $\mathcal{P}_n$ is the fraction of $Top_n(G)$, the top n databases of G (which have a non-zero *Estimate* for being in G), that actually have non-zero goodness for query q:

$$\mathcal{P}_n = \frac{|\{db \in Top_n(G) | Goodness(l, q, db) > 0\}|}{|Top_n(G)|}$$

(Actually, $\mathcal{P}_n = 1$ if for all db, $Estimate(l, q, db) = 0$.) Note that $\mathcal{P}_n$ is independent of the ideal database rank I: it just depends on how many databases that *gGlOSS* estimated as potentially useful turned out to actually be useful for the query. A ranking with higher $\mathcal{P}_n$ is better because it leads to fewer fruitless database searches.

Example 19: (cont.) In the previous example, $\mathcal{P}_4 = \frac{3}{3} = 1$ for G, because all of the databases in G have actual non-zero goodness. However, $\mathcal{P}_4 = \frac{3}{4} = 0.75$ for H: of the four databases in H, only three have non-zero goodness. ∎

4.4 Evaluating *gGlOSS*

In this section we evaluate different *gGlOSS* ranking algorithms experimentally. We first describe the real-user queries and databases that we used in the experiments. Then, we report results for $Max(l)$ and $Sum(l)$, the two *gGlOSS* ranks of Section 4.2.

4.4.1 Queries and Databases

To evaluate *gGlOSS* experimentally, we used real-user queries and databases. The queries that we used where profiles that real users submitted to the SIFT Netnews server developed at Stanford [YGM95b]. Users send profiles in the form of Boolean or vector-space queries to the SIFT server, which in turn filters Netnews articles every day and sends the articles matching the profiles to the corresponding users. We used the 6800 vector-space profiles that were active on the server in December 1994.

To evaluate the *gGlOSS* performance using these 6800 queries, we used 53 newsgroups as 53 databases: we took a snapshot of the articles that were active at the Stanford Computer-Science-Department news host on one arbitrary day, and used these articles to populate the 53 databases. We selected all the newsgroups in the `comp.databases`, `comp.graphics`, `comp.infosystems`, `comp.security`, `rec.-arts.books`, `rec.arts.cinema`, `rec.arts.comics`, and `rec.arts.theatre` hierarchies that had active documents in them when we took the snapshot.

We indexed the 53 databases and evaluated the 6800 queries on them using the SMART system (version 11.0) developed at Cornell University. To keep our experiments simple, we chose the same weighting algorithms for the queries and the documents across all of the databases. We indexed the documents using the SMART *ntc* formula, which generates document weight vectors using the cosine-normalized *tf·idf* product [Sal89]. We indexed the queries using the SMART *nnn* formula, which generates query weight vectors using the word frequencies in the queries. The similarity coefficient between a document vector and a query vector is computed by taking the inner product of the two vectors.

For each query and *gGlOSS* ranking algorithm we compared the ideal rank against the *gGlOSS* rank using the methodology of Section 4.3. We evaluated each query at each of the 53 databases to generate its ideal database rank. For a fixed *gGlOSS* ranking definition and a query, we computed the rank of databases that *gGlOSS* would produce for that query: we extracted the (partial) information that *gGlOSS* needs from each of the 53 databases. For each query word, *gGlOSS* needs the number of documents in each database that include the word, and the sum of the weight of the word in each of these documents. To extract all this information, we queried the 53 databases using each query word individually, which totaled an extra 18,213 queries. We should stress that this is just the way we performed the experiments, not the way a *gGlOSS* server will obtain the information it needs about each database: in a real system, each database will periodically scan its indexes, generate the information that *gGlOSS* needs, and export it to the *gGlOSS* server. (See Section 4.2.)

4.4.2 Experimental Results

In this section we experimentally compare the *gGlOSS* database ranks against the ideal ranks in terms of the $\mathcal{R}_n$ and $\mathcal{P}_n$ metrics. We study which of the $Max(l)$ and $Sum(l)$ database ranks is better at predicting ideal rank $Ideal(l)$, and what impact the threshold l has on the performance of *gGlOSS*. We also investigate whether keeping both the F and W matrices of Section 4.2 is really necessary, since *gGlOSS* needs only one of these matrices to compute ranks $Max(0)$ and $Sum(0)$ (Section 4.2.2).

Ideal database rank $Ideal(0)$ considers any document with a non-zero similarity to the query as useful. Ranks $Max(0)$ and $Sum(0)$ are identical to $Ideal(0)$, and so they have $\mathcal{R}_n = \mathcal{P}_n = 1$ for all n. Consequently, if a user wishes to locate databases where the overall similarity between documents and the given query is highest and any document with non-zero similarity is interesting, *gGlOSS* should use the $Max(0)$ (or, identically, $Sum(0)$) ranks and get perfect results.

To study the impact of higher rank thresholds, Figures 4.2 and 4.3 show results for the $Ideal(0.2)$ ideal rank. We show $\mathcal{R}_n$ and $\mathcal{P}_n$ for values of n ranging from 1 to 15. We do not report data for higher n's because most of the queries have fewer than 15 useful databases according to $Ideal(0.2)$ and hence, the results for high values of n are not that significant. Figure 4.3 shows that rank $Sum(0.2)$ has perfect $\mathcal{P}_n$

($\mathcal{P}_n = 1$) for all n, because if a database db has $Estimate(0.2, q, db) > 0$ according to the $Sum(0.2)$ rank, then $Goodness(0.2, q, db) > 0$ according to $Ideal(0.2)$. In other words, rank $Sum(0.2)$ only includes databases that are guaranteed to be useful. Rank $Max(0.2)$ may include databases not guaranteed to be useful, yielding higher $\mathcal{R}_n$ values (Figure 4.2), but lower $\mathcal{P}_n$ values (Figure 4.3).

To decide whether *gGlOSS* really needs to keep both matrices F and W (Section 4.2), we also use ranks $Max(0)$ and $Sum(0)$ to approximate rank $Ideal(0.2)$. *gGlOSS* needs only one of the two matrices to compute these ranks (Section 4.2.2). Since ranks $Max(0)$ and $Sum(0)$ are always identical, we just present their data once labeled $Max(0)/Sum(0)$. Figure 4.2 shows that the $Max(0)$ rank has the highest values of $\mathcal{R}_n$. This rank assumes a threshold $l = 0$, and thus it tends to include more databases than its counterparts with threshold 0.2. This is also why $Max(0)$ has much lower $\mathcal{P}_n$ values (Figure 4.3) than $Max(0.2)$ and $Sum(0.2)$: it includes more databases that have zero goodness according to $Ideal(0.2)$.

In summary, if the users are interested in not missing any useful database, but are willing to search some useless ones, then $Max(0)$ is the best choice for *gGlOSS*, and *gGlOSS* can do without matrix F. If the users wish to avoid searching useless databases, then $Sum(0.2)$ is the best choice. Unfortunately, $Sum(0.2)$ also has low $\mathcal{R}_n$ values, which means it can also miss some useful sources. As a compromise, a user can have $Max(0.2)$, which has much better $\mathcal{P}_n$ values than $Max(0)$ and generally better $\mathcal{R}_n$ values than $Sum(0.2)$. Also, note that in the special case where users are interested in accessing only one or two databases ($n = 1, 2$) then $Max(0.2)$ is the best choice for the $\mathcal{R}_n$ metric. In this case, it is worthwhile for *gGlOSS* to keep both matrices F and W.

To show the impact of the rank thresholds, Figures 4.4 and 4.5 show the $\mathcal{R}_n$ and $\mathcal{P}_n$ values for the different ranks and a fixed $n = 3$, and for values of the threshold l from 0 to 0.4. For larger values of l, most of the queries have no database with goodness greater than zero. For example, for ideal rank $Ideal(0.6)$ each query has on average only 0.29 useful databases. Therefore, we only show the data for threshold 0.4 and lower. At first glance one might expect the $\mathcal{R}_n$ and $\mathcal{P}_n$ performance of $Max(0)$ not to change as the threshold l varies, since the ranking it computes is independent

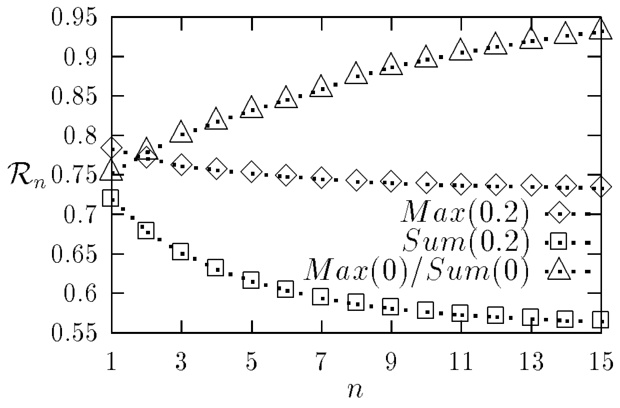

Figure 4.2: Parameter $\mathcal{R}_n$ as a function of n, the number of databases examined from the ranks, for the *Ideal*(0.2) ideal database ranking and the different *gGlOSS* rankings.

of the desired l. However, as l increases, the ideal rank *Ideal*(l) changes, and the static estimate provided by *Max*(0) performs worse and worse for $\mathcal{P}_n$. The *Max*(l) and *Sum*(l) ranks do take into account the target l values, and hence do substantially better. Our earlier conclusion still holds: strategy *Sum*(l) is best at avoiding useless databases, while *Max*(0) provides the best $\mathcal{R}_n$ values (at the cost of low $\mathcal{P}_n$ values).

In summary, *gGlOSS* generally predicts fairly well the best databases for a given query. Actually, the more *gGlOSS* knows about the users' expectations, the better *gGlOSS* can rank the databases for the query. If high values of both $\mathcal{R}_n$ and $\mathcal{P}_n$ are of interest, then *gGlOSS* should produce ranks based on the high-correlation assumption of Section 4.2.1: rank *Max*(l) is the best candidate for rank *Ideal*(l) with $l > 0$. If only high values of $\mathcal{R}_n$ are of interest, then *gGlOSS* can do without matrix F, and produce ranks *Max*(0) or *Sum*(0). If only high values of $\mathcal{P}_n$ are of interest, then *gGlOSS* should produce ranks based on the disjoint-scenario assumption of Section 4.2.2: rank *Sum*(l) is the best candidate. For rank *Ideal*(0), ranks *Max*(0) and *Sum*(0) give perfect answers.

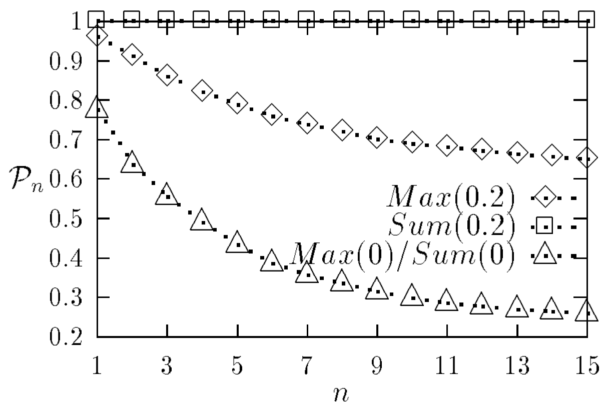

Figure 4.3: Parameter $\mathcal{P}_n$ as a function of n, the number of databases examined from the ranks, for the *Ideal*(0.2) ideal database ranking and the different *gGlOSS* rankings.

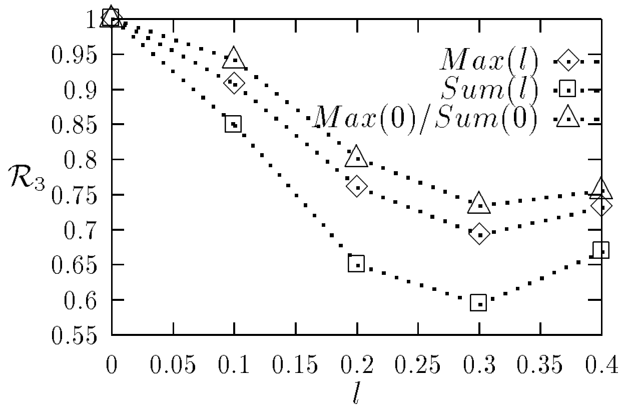

Figure 4.4: Parameter $\mathcal{R}_3$ as a function of the threshold l, for ideal rank *Ideal*(l).

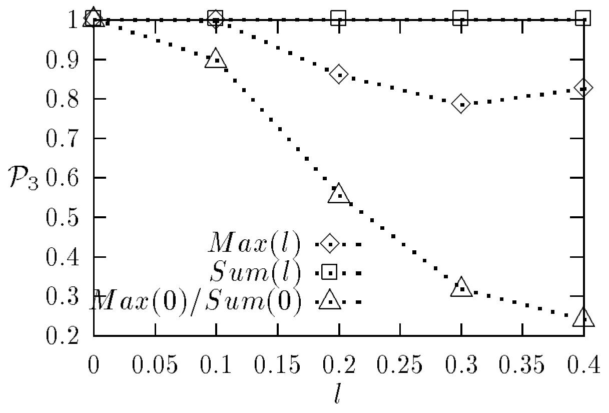

Figure 4.5: Parameter $\mathcal{P}_3$ as a function of the threshold l, for ideal rank *Ideal*(l).

4.5 Alternative Ideal Ranks

Section 4.1 presented a way of defining the goodness of a database for a query, and also showed a problem with its associated ideal database rank. In this section we explore alternative ideal database ranks for a query, similarly as in Section 3.4.3 for Boolean databases. (Even other possibilities are discussed in [GGM95b].)

We can organize the different database ranks for a query into two classes, according to whether the ranks depend on the number of relevant documents for the query in each database or not (Section 3.4.3). The first two alternative ranks belong to the first class.

The first rank, *Rel_All*, simply orders the databases based on the number of relevant documents they contain for the given query. By relevant we mean that the user who submits q will judge these documents to be of interest. To see a problem with this rank, consider a database db that contains, say, three relevant documents for some query q. Unfortunately, it turns out that the search engine at db does not include any of these documents in the answer to q. So, the user will not benefit from these three relevant documents. Thus, we believe it is best to evaluate the ideal goodness of a database by what its search engine might retrieve, not by what potentially relevant documents it might contain. Notice that a user might eventually obtain these relevant documents by successively modifying the query. Our model would treat each of these queries separately, and decide which databases are the best for each individual query.

Our second rank, *Rel_Rank*(l), improves on *Rel_All* by considering only the relevant documents in each database that have a similarity to q greater than a threshold l, as computed by the individual databases. The underlying assumption is that users will not examine documents with lower similarity in the answers to the queries, because these documents are unlikely to be useful. This definition does not suffer from the problem of the *Rel_All* rank: we simply ignore relevant documents that db does not include in the answer to q with sufficiently high similarity. However, in general we believe that ranks based on end-user relevance are not appropriate for evaluating schemes like *gGlOSS*. That is, the best we can hope for any tool like *gGlOSS* is that it predicts the answers that the databases will give when presented with a query. If the databases cannot rank the relevant documents high and the non-relevant ones low with complete index information, it is asking too much that *gGlOSS* derive relevance judgments with only partial information. Consequently, the database rankings that are not based on document relevance seem a more useful frame of reference to evaluate the effectiveness of *gGlOSS*. Hence, the remaining ranks that we consider do not use relevance information.

The *Global*(l) rank is based on considering the contents of all the databases as a single collection. The documents are then ranked according to their "global" similarity to query q. We consider only those documents having similarity to q greater than a threshold l. The *Goodness* metric associated with rank *Global*(l) would add the similarities of the acceptable documents. The problem with this rank is related to the problem with the *Rel_All* rank: a database db may get high goodness values for documents that do not appear (high) in the answer that the database produces for q. Therefore, db is not as useful to q as the *Goodness* metric predicted. To avoid this problem, the goodness of a database for a query should be based on the document rank that the database generates for the given query.

The definition of *Goodness* of Section 4.1 does not rely on relevance judgments, and is based on the document ranks that the databases produce for the queries. Therefore, that definition does not suffer from the problems of the alternative ranks that we considered so far in this section. However, as we mentioned in Section 4.1, a problem is that the similarities computed at the local databases can depend on the characteristics of the collections, and thus they might not be valid globally. The next definition attempts to compensate for this collection-dependent computations.

The next rank, *Local*(l), considers only the set of documents in *db* having *scaled* similarity to q greater than a threshold l. We scale the similarities coming from different databases differently, to compensate for the collection-dependent way in which these similarities are computed. Also, we should base the goodness of each database on its answer to the query, to avoid the anomalies we mentioned above for the *Rel_All* and the *Global* ranks. One way to achieve these two goals is to multiply the similarities computed by database *db* by a positive constant $scale(q, db)$:

$$Goodness(l, q, db) = scale(q, db) \times \sum_{d \in Scaled\_Rank(l,q,db)} sim(q, d)$$

where $scale(q, db)$ is the scaling factor associated with query q and database db, and $Scaled\_Rank(l, q, db) = \{d \in db | sim(q, d) \times scale(q, db) > l\}$.

The problem of how to modify the locally computed similarities to compensate for collection-dependent factors in their computation has received attention recently in the context of the collection-fusion problem. (See Chapter 5.) In general, determining what scaling factor to use to define the *Local*(l) ideal database rank is an interesting problem. If we incorporated scaling into the *Goodness* definition, we should modify *gGlOSS*'s ranks to imitate this scaling.

In summary, none of the database ranking schemes that we have discussed is perfect, including the ones we used for our experiments. Each scheme has its limitations, and hence, should be used with care.

4.6 Decentralizing *gGlOSS*

So far, we described *gGlOSS* as a centralized server that users query to select the most promising sources for their queries. In this section we show how we can build a more distributed version of *gGlOSS* using essentially the same methodology that we developed in the previous sections.

Suppose that we have a number of *gGlOSS* servers $G_1, \ldots, G_s$, indexing each a set of databases as we described in the previous sections. (Each of these servers can index the databases at one university or company, for example.) We will now build a *higher-level gGlOSS* server, *hGlOSS*, that summarizes the contents of the *gGlOSS* servers in much the same way as the *gGlOSS* servers summarize the contents of the underlying databases. [3] The users will then query the *hGlOSS* server first, and obtain a rank of the *gGlOSS* servers according to how likely they are to have indexed useful databases. Later, the *gGlOSS* servers will produce the final database ranks. Although the *hGlOSS* server is still a single entry point for users to search for documents, the size of this server will be so small that it will be inexpensive to massively replicate it, distributing the access load among the replicas. In this way, organizations will be able to manage their own "traditional" *gGlOSS* servers, and will let replicas of a logically unique higher-level *gGlOSS*, *hGlOSS*, concisely summarize the contents of their *gGlOSS* servers.

The key point is to notice that *hGlOSS* can treat the information about a database at a traditional *gGlOSS* server in the same way as the traditional *gGlOSS* servers treat the information about a document at the underlying databases. The "documents" for *hGlOSS* will be the *database summaries* at the *gGlOSS* servers.

To keep the size of the *hGlOSS* server small, the information that the *hGlOSS* server keeps about a *gGlOSS* server G_r is limited. For example, *hGlOSS* keeps one or both of the following matrices (see Section 4.2):

- $H = (h_{rj})$: h_{rj} is the number of *databases* in *gGlOSS* G_r that contain word t_j
- $D = (d_{rj})$: d_{rj} is the sum of the number of documents that contain word t_j in each database in *gGlOSS* G_r

In other words, for each word t_j and each *gGlOSS* server G_r, *hGlOSS* needs (at most) two numbers, in much the same way as the *gGlOSS* servers summarize the contents of the document databases (Section 4.2).

Example 20: Consider a *gGlOSS* server G_r and the word *computer*. Suppose that the following are the databases in G_r having documents with the word *computer* in

[3]Although our discussion focuses on a 2-level hierarchy of servers, we can use the same principles to construct deeper hierarchies.

them, together with their corresponding *gGlOSS* weights and frequencies:

$$computer : (db_1, 5, 3.4), (db_2, 2, 1.8), (db_3, 1, 0.3)$$

That is, database db_1 has five documents with the word *computer* in them, and their added weight is 3.4 for that word, database db_2 has two documents with the word *computer* in them, and so on. *hGlOSS* will only know that the word *computer* appears in three databases in G_r, and that the sum of the number of documents for the word and the databases is $5 + 2 + 1 = 8$. *hGlOSS* will not know the identities of these databases, or the individual document counts associated with the word and each database. ∎

We can now use the same methodology we used for *gGlOSS* in the previous sections: given a query q, we define the goodness of each *gGlOSS* server G_r for the query: for example, we can take the database rank that G_r produces for q, together with the goodness estimate for each of these databases according to G_r, and define the goodness of G_r for q as a function of this rank. This computation is analogous to how we computed the goodness of the *databases* in Section 4.1.

After defining what the goodness of each *gGlOSS* server is for query q, we define how *hGlOSS* is going to estimate this goodness given only partial information about each *gGlOSS* server. *hGlOSS* will determine the *Estimate* for a *gGlOSS* server G_r using the vectors h_{r*} and d_{r*} for G_r in a way analogous to how the *gGlOSS* servers determine the *Estimate* for a database db_i using the f_{i*} and w_{i*} vectors. After defining the *Estimate* for each *gGlOSS* server, *hGlOSS* ranks the *gGlOSS* servers so that the users can access the most promising servers first, i.e., those most likely to index useful databases.

To illustrate *hGlOSS*'s potential, we briefly describe one experiment. For this, we divide the 53 databases of Section 4.4 into five randomly-chosen groups of around ten databases each. Each of these groups corresponds to a different *gGlOSS* server.

We assume that the *gGlOSS* servers approximate ideal rank $Ideal(0)$ with the $Max(0)$ database rank. Next, we define the goodness of a *gGlOSS* server G_r for a query q as the number of databases indexed by G_r having a goodness *Estimate* for

n	$\mathcal{R}_n$	$\mathcal{P}_n$
1	0.985	1
2	0.991	1
3	0.994	1
4	0.998	1
5	1	1

Figure 4.6: The $\mathcal{R}_n$ and $\mathcal{P}_n$ metrics for *hGlOSS* and our sample experiment.

q greater than zero. This definition determines the ideal rank of *gGlOSS* servers. To approximate this ideal rank, *hGlOSS* periodically receives the H matrix defined above from the underlying *gGlOSS* servers. For query q with words $t_1, \ldots, t_n$ and *gGlOSS* server G_r, $h_{r1}, \ldots, h_{rn}$ are the database counts for G_r associated with the query words. (Word t_1 appears in h_{r1} *databases* in *gGlOSS* server G_r, and so on.) Assume that $h_{r1} \leq \ldots \leq h_{rn}$. Then, *hGlOSS* estimates the goodness of G_r for q as h_{rn}. In other words, *hGlOSS* estimates that there are h_{rn} databases in G_r that have a non-zero goodness estimate for q.

Figure 4.6 shows the different values of the (adapted) $\mathcal{R}_n$ and $\mathcal{P}_n$ metrics for the 6,800 queries of Section 4.4. Note that $\mathcal{P}_n = 1$ for all n, because every time *hGlOSS* chooses a *gGlOSS* server using the ranking described above, this server actually has databases with non-zero estimates. From the high values for $\mathcal{R}_n$ it follows that *hGlOSS* is extremely good at ranking "useful" *gGlOSS* servers.

Our single experiment used a particular ideal ranking and evaluation strategy. We can also use the other rankings and strategies we have presented adapted to the *hGlOSS* level, and tuned to the actual user requirements. Also, the *hGlOSS* server will be very small in size and easily replicated, thus eliminating the potential bottleneck that the centralized *gGlOSS* architecture can suffer.

4.7 Larger Scale Effectiveness Experiments

The database ranks produced by *gGlOSS* are incremental "plans" for evaluating a query. In effect, we first contact the top database in the rank. If we are not satisfied with the answers retrieved, we contact the second database, and so on. The effectiveness metrics that we introduced in this chapter (Section 4.3) provide a way of evaluating how good these incremental plans for a query are. In this section, we revisit the Boolean version of *GlOSS* of Chapter 3 and evaluate it with these new metrics. Also, to be sure that Boolean *GlOSS* would be useful as a large scale text source discovery system, we scaled up the number of databases by about two orders of magnitude from the six databases used in the experiments of Chapter 3. Thus, this section uses the metrics of this chapter to demonstrate that Boolean *GlOSS* can select relevant databases effectively from among a large set of candidates [TGL+97].

For our new Boolean *GlOSS* experiments, we used as data the complete set of United States patents for 1991. Each patent issued is described by an entry that includes various attributes (e.g., names of the patent owners, issuing date) as well as a text description of the patent. The total size of the patent data is 3.4 gigabytes. We divided the patents into 500 databases by first partitioning them into fifty groups based on date of issue, and then dividing each of these groups into ten subgroups, based on the high order digit of a subject-related patent classification code. This partitioning scheme gave databases that ranged in size by an order of magnitude, and were at least somewhat differentiated by subject. Both properties are ones we would expect to see in a real distributed environment.

For test queries, we used the real-user INSPEC queries of Chapter 3, excluding all queries with field designators not applicable to the patent data. Although INSPEC is not a patent database, it covers a similar range of technical subjects, so we expected a fair number of hits against our patent data. Each query is a Boolean conjunction of one or more words, e.g., *microwave* $\wedge$ *interferometer*. A document is considered to match a query if it contains all the words in the conjunction.

To test *GlOSS*'s ability to locate the databases with the greatest number of matching documents, we compared the recommendations of its *Ind* estimator to those of an "omniscient" database selection mechanism implemented using a full-text index of the contents of our 500 patent databases, as in Chapter 3. For each query, we found the exact number of matching documents in each database, using the full-text index, and ranked the databases accordingly. We compared this ranking with the ranking suggested by *GlOSS* by calculating, for various values of n, the $\mathcal{R}_n$ metric

n	$\mathcal{R}_n$
1	0.712
2	0.725
3	0.730
4	0.736
5	0.744
6	0.750
7	0.755
8	0.758
9	0.764
10	0.769

Figure 4.7: The average $\mathcal{R}_n$ metric for 500 text databases and the $TRACE_{INSPEC}$ queries of Chapter 3.

of Section 4.3. The *Goodness* of a database for a query is the number of matching documents for the query that the database contains.

Figure 4.7 shows the results of this experiment. Compared to an omniscient selector, *GlOSS* does a reasonable job of selecting relevant databases, on average finding over seventy percent of the documents that could be found by examining an equal number of databases under ideal circumstances, with gradual improvement as the number of databases examined increases. Using *GlOSS* gives a dramatic improvement over randomly selecting databases to search, for a fraction of the storage cost of a full-text index.

4.8 Conclusion

We have shown how to construct source-discovery servers for both vector-space text databases and hierarchies of source-discovery servers. Based on compact collected statistics, these servers can provide very good hints for finding the relevant databases, or finding relevant lower-level servers with more information for a given query. An important feature of our approach is that the same machinery can be used for both types of servers, either the lower-level or the higher-level ones. Our experimental results show that *gGlOSS* and *hGlOSS* are quite promising and could provide useful services in large, distributed information systems.

Chapter 5

The Result Merging Problem

Increasingly, sources on the Internet and elsewhere rank the objects in the results of selection queries according to how well these objects match the original condition. For such sources, query results are not flat sets of objects that match a given condition. Instead, query results are sorted starting from the top (best) object for the query at hand. As we have seen in Chapter 4, a typical example of this kind of sources is a source that indexes text documents and answers queries using some variation of the *vector-space* model of document retrieval [Sal89].

Example 21: Consider a World-Wide Web search engine like Excite (`http://www.-excite.com`). Given a query consisting of a series of words, like *distributed databases*, Excite returns the matching documents sorted according to how well they match the query. This way, Excite might return a given WWW page as the top match for the query with a *score* of 82%, some other page as the second top match with a score of 80%, and so on. ▮

Although text sources are probably the best known example, sources with multimedia objects like images are also becoming common. Matches between query values and objects in such sources are inherently "fuzzy" [NBE$^+$93].

Example 22: Consider a World-Wide Web search engine for images like Image Surfer (`http://isurf.interpix.com/`). Given an image of interest, Image Surfer returns a rank of the images that are closest to the given one in terms of their color distribution. The query results for such a source are inherently ranked. In effect, most users would want to find images with a color distribution that is *close*, not identical, to that of a given image. ∎

Even sources with more "traditional" and structured data that rank their query results are appearing on the Internet. These sources rank the highest those objects that match the user's specification the best.

Example 23: Consider a real-estate agent that accepts queries on the *Location* and *Price* attributes of the available houses. This agent could treat query conditions as if they were regular Boolean conditions. This way, the agent (or the user) could determine an acceptable radius around the preferred location, and an acceptable price range, and simply return all the houses with a location and price within these limits. However, there could be too many matching houses, making the user's task of going over them tedious. Also, houses with, say, a very good price but slightly outside of the acceptable location area might be missed. Therefore, some on-line real-estate agents already rank their query results (e.g., CyberHomes, at `http://www.cyberhomes.-com/`). Thus, the top house returned to the user would be one that is closest to the specified location and is relatively inexpensive. As we will see, sources might choose to weigh these two criteria for their rankings in different ways. ∎

As the popularity of this type of sources increases, so does the number of metasearchers. As we saw in Chapters 1 and 2, a key problem that a metasearcher has to address is how to extract the top matches for a query from sources that might use widely different ranking algorithms:

Example 24: A service like SavvySearch (`http://guaraldi.cs.colostate.edu:-2000/`) queries multiple WWW search engines at once, including Excite. It then combines the results into a single ranked result. If a page p is returned only by Excite with a score of 82%, and a page p' is returned only by HotBot (`http://www.hotbot.com/`) with the same score, then both pages would be judged by SavvySearch as equally good for the query at hand. However, Excite and HotBot may use radically different scoring algorithms, so it is really not meaningful to merge the results based on the source scores. ∎

The solution is to have the metasearcher have its own scoring function that it uses to rank and merge the retrieved objects. With this scheme, each page or object retrieved is given a new *target* score, regardless of its source score, and these target scores are used to merge the results. For this to work, the metasearcher needs to retrieve enough information about the source objects to evaluate its target function on them. As we discuss in Section 5.3, in some cases it is not possible to retrieve all the necessary target scoring attributes, thus making it simply impossible to merge the results in a reasonable way. However, even if the metasearcher can retrieve the necessary attributes for each object, there is still the very important problem of extracting the right source objects, i.e., of extracting the source objects that will yield the highest target scores, without having to examine *all* of the source objects.

Example 25: Suppose that the score that the real-estate agent of Example 23 assigns a house for a query is $0.1 \cdot l + 0.9 \cdot p$, where l is a number between 0 and 1 that indicates how close the house is to the target location (higher values of l are better), and p is a number between 0 and 1 that indicates how close the price of the house is to the target price (higher values of p are better). Now, suppose that a metasearcher would like to weigh location and price equally, and it does so by assigning houses a score of $0.5 \cdot l + 0.5 \cdot p$.

Suppose that a user is looking for houses with preferred location in Palo Alto and a target price of $\$100K$. Furthermore, suppose that the agent has only one house in Palo Alto, with $l = 1$ (perfect location) and $p = 0.2$ (high price). *All* the remaining houses available to the agent are located in Mountain View, with $l = 0.6$ (not as good a location) and $p = 0.4$ (moderate price).

Using the definitions above, the real-estate agent would assign a score of $0.1 \cdot 1 + 0.9 \cdot 0.2 = 0.28$ to the Palo Alto house, whereas the metasearcher would assign such a house a higher score of $0.5 \cdot 1 + 0.5 \cdot 0.2 = 0.6$, since the metasearcher weighs location and price equally. Also, the agent would assign a score of $0.1 \cdot 0.6 + 0.9 \cdot 0.4 = 0.42$ to any Mountain View house, whereas the metasearcher would assign any such house a score of $0.5 \cdot 0.6 + 0.5 \cdot 0.4 = 0.5$. Consequently, the answer to the user's query from the metasearcher should be the Palo Alto house, because it has the highest score for the query according to the metasearcher's scoring algorithm. However, the real-estate agent, where the record of the Palo Alto house resides, ranks all of the other houses, which are all Mountain View houses, higher than the Palo Alto house, so the metasearcher would have to retrieve all of the agent's contents before extracting the top house, i.e., the Palo Alto house. ∎

Example 25 illustrates that it may be hard for a metasearcher to extract the best objects from autonomous sources when they use scoring functions that are different, or even slightly different, from the target function used by the metasearcher. This raises some important questions. For example, for what types of source and target scoring functions is it possible to retrieve results "efficiently," without having to retrieve full source contents? In these cases, what is the right strategy for obtaining and ranking results? For instance, given an end-user query, what types of queries, and in what order, should we submit to the sources? Also, how much does the metasearcher need to know about the source scoring function? Turning to a negative scenario, are there "uncooperative" source scoring functions for which there is no strategy whatsoever that avoids an exhaustive full retrieval of the source contents?

In this chapter we address these and other related questions [GGM97]. We start by proposing a searching and ranking model for sources with structured data (Section 5.1). Within this model, we then precisely characterize the classes of source and target functions that make retrieval "efficient" or "exhaustive" (Section 5.4). In the former case, we present an algorithm for searching sources and finding the top-ranking objects according to the metasearcher's target function (Section 5.2). We also describe variations to our model, and their impact on search and ranking (Section 5.3).

Our goal in this chapter is to explore the fundamental complexity and limitations of metasearchers. We believe that our results can guide implementors of search engines, making it clear what scoring functions may make it hard for a client metasearcher to merge information properly, and making it clear how much the metasearcher needs to know about the scoring function. This last point is important since typically search engine builders wish to keep their scoring function secret because it is one of the things that differentiates them from other sources. At the metasearcher end, we believe that our results can also be helpful in the design of the target scoring function, and in distinguishing cases where merging results is meaningful and cases where it is not.

5.1 Our Search Model for Structured Sources

The previous section presented examples of sources and metasearchers, and illustrated some of the problems that metasearchers face when querying autonomous sources. In this section we define our searching model more precisely, and revisit the real-estate agent example in light of the new definitions.

A source S contains a single relation R_S with attributes $A_1, \ldots, A_n$. S accepts queries over R_S. A query over S simply specifies target values for some of the attributes of R_S. Thus, a query Q is an assignment of values $v_1, \ldots, v_n$ to the attributes $A_1, \ldots, A_n$ of R_S. Some of the v_i values might be *don't care* values (noted "*"). The rest of the v_i values are the *significant* values in the query.

Given a query, source S responds with the objects (i.e., tuples) of R_S that "best match" the query values. The query results contain the values for $A_1, \ldots, A_n$ for every object returned. (In Section 5.3 we discuss sources for which this property does not hold.)

Property 1: Information in query results: *The record for an object t in the query results returned by a source S contains all the values $t[1], \ldots, t[n]$ for the attributes $A_1, \ldots, A_n$ that can be used to formulate queries over S.*

Each object t in the result for query Q is ranked according to the *source score* *Source*(S, Q, t) that source S computes for Q and t. These scores range from 0 to 1. Since sources are autonomous, these scores could be computed in a completely arbitrary way. However, we expect them to be a function of the significant values of Q, as discussed below.

Example 26: Consider the real-estate agent S of Example 25. This agent hosts relation R_S*(Location, Price)*. As mentioned above, a query to this agent may specify a target location $L =$ *Palo Alto* and some target price $P = \$100K$, for example. In other words, such a query $Q = (L, P)$ asks for houses located close to Palo Alto, and with a price not too much higher or lower than $\$100K$.

The answers that the agent gives the user are the objects of R_S ranked according to S's source score for Q [1]. This source score is arbitrary, as mentioned above. For example,

$$\mathit{Source}(S, (L, P), t) \;=\; \begin{cases} l & \text{if } P = * \\ p & \text{if } L = * \\ 0.1 \cdot l + 0.9 \cdot p & \text{otherwise} \end{cases}$$

where l is some number between 0 and 1 that is inversely proportional to the distance between t and the preferred location L, and p is some number between 0 and 1 that is inversely proportional to the distance between the price of t and P, as mentioned above. ▮

A metasearcher receives a user query Q and returns the top objects for Q that appear in any of the available sources, according to the *target score*. The *target score* $\mathit{Target}(Q, t)$ for query Q and object t is some known function of the significant values in Q. The values of *Target* range from 0 to 1.

Example 26: (cont.) Continuing with the example above, we can define:

$$\mathit{Target}((L, P), t) \;=\; \begin{cases} l & \text{if } P = * \\ p & \text{if } L = * \\ 0.5 \cdot l + 0.5 \cdot p & \text{otherwise} \end{cases}$$

[1] In the remainder of the chapter, we refer to both source S and its relation R_S as source S, for simplicity.

Consequently, *Target* is quite similar to *Source*: these two functions just differ in the weight that they assign to each of the two query attributes when they are both significant. ∎

To extract the objects for a query Q with the highest *Target* scores (i.e., *the top Target objects*), a metasearcher queries multiple sources that hold different instances of the same relation R and that use different source score functions. The metasearcher extracts from each source S all of the objects t with $Source(S, Q, t) \geq g$, for some score $0 \leq g \leq 1$. (We will discuss how to find g in Section 5.2.) The metasearcher then computes the *Target* score of these objects without accessing the objects themselves, using the attribute values returned in the query results (Property 1). Finally, the metasearcher returns the top *Target* objects for the query.

Example 26: (cont.) Consider the top result that source S returns for the query Q above:

Location: Mountain View; *Price*: \$150K; *Source* score: 0.42

The metasearcher can then simply discard the *Source* score for this house, and compute the *Target* score using its own algorithm. The metasearcher does this for all of the objects extracted from the sources, and returns the objects with the highest *Target* scores. ∎

The *Source* and *Target* scores for a query may vary widely, as we have seen. The following definition captures those *Source* scores that are reasonably close to a given *Target* score. This definition will be useful later to characterize the sources for which we can extract the top *Target* objects efficiently.

Definition 1: *A query Q is* manageable *at source S if there is a constant $0 \leq \epsilon < 1$ such that*

$$Source(S, Q, t) \geq Target(Q, t) \Leftrightarrow \epsilon$$

for all possible objects t. In other words, a query is manageable at a source if the Source scores for this query are not too much lower than the corresponding Target scores.

Example 27: A query Q for the real-estate agent specifying both a *Location* and a *Price* is manageable at S for the *Target* and *Source* scores defined in Example 26. In effect, we can take $\epsilon = 0.4$:

$$\begin{aligned}
\mathit{Target}(Q,t) \Leftrightarrow \epsilon &= 0.5 \cdot l + 0.5 \cdot p \Leftrightarrow 0.4 \\
&= 0.1 \cdot l + 0.4 \cdot (l \Leftrightarrow 1) + 0.5 \cdot p \\
&\leq 0.1 \cdot l + 0.9 \cdot p \\
&= \mathit{Source}(S,Q,t)
\end{aligned}$$

∎

Example 28: Consider the following *Target* score for the real-estate scenario:

$$\mathit{Target}((L,P),t) = \begin{cases} l & \text{if } P = * \\ p & \text{if } L = * \\ \max\{l,p\} & \text{otherwise} \end{cases}$$

and the following *Source* score:

$$\mathit{Source}(S,(L,P),t) = \begin{cases} l & \text{if } P = * \\ p & \text{if } L = * \\ \min\{l,p\} & \text{otherwise} \end{cases}$$

Then, a query Q specifying both a *Location* and a *Price* is not manageable at S, if l and p can assume arbitrary values between 0 and 1. In effect, consider an object t with $l = 1$ and $p = 0$. (Such a house has a perfect location according to the user's specification, but an exorbitant price.) Then, $\mathit{Source}(S,Q,t) = \min\{1,0\} = 0 < \mathit{Target}(Q,t) \Leftrightarrow \epsilon = \max\{1,0\} \Leftrightarrow \epsilon = 1 \Leftrightarrow \epsilon$, $\forall 0 \leq \epsilon < 1$. Consequently, there is no value of ϵ that will satisfy the condition in Definition 1.

Intuitively, Q is not manageable at S because top objects for *Target* can have arbitrarily low scores for *Source*. Therefore, we would have to retrieve all of the objects in S to find the top objects for *Target*, and this is exactly what we are trying to avoid. ∎

Source S is autonomous, and the metasearcher might not know S's *Source* function. However, in this section we assume that the metasearcher knows whether a query Q is manageable at S. (Section 5.3 relaxes this property and considers sources where it does not hold.)

Property 2: Information about source manageability: *Given a query Q and a source S, the metasearcher knows whether Q is manageable at S. Furthermore, in case it is, the metasearcher knows a value for ϵ as in the definition of manageability (Definition 1).*

Definition 2: *Let Q be a query with a significant value v_j for attribute A_j. Then, the* single-attribute query *Q_j for Q and A_j is the query that results from Q by setting the value for v_i to "*" ("don't care") for all $i \neq j$.*

To deal with sources like the one in Example 28, we introduce the notion of a *cover* for a query [2]:

Definition 3: *A set of single-attribute queries over different attributes $C = \{Q_1, \ldots, Q_m\}$ is a* cover *for a query Q if $\exists 0 \leq g_1, \ldots, g_m, G < 1$ such that $\forall$ object t:*

$$\mathit{Target}(Q_i, t) \leq g_i, i = 1, \ldots, m \Rightarrow \mathit{Target}(Q, t) \leq G$$

Intuitively, we will later use the single-attribute queries in a cover to extract a set of objects from a source that includes the top *Target* objects. This way, we will be able to work with sources at which a given query is not manageable (Example 28), or that would otherwise require potentially inefficient executions (Example 26).

Example 29: Let Q_1 be the single-attribute query for Q and the *Location* attribute, and Q_2 be the single-attribute query for Q and the *Price* attribute. Consider the

[2]The notion of cover is related to that of a *complete set of atomic conditions* in [CG96]. (See Section 7.4.)

Target and *Source* scores of Example 26. Then, the set $\{Q_1\}$ is a cover for Q. In effect, for any $0 \leq g < 1$, we can define $G = 0.5 \cdot (g + 1)$. Thus, if an object t is such that $\mathit{Target}(Q_1, t) \leq g$, then $\mathit{Target}(Q, t) \leq 0.5 \cdot g + 0.5 \cdot p \leq 0.5 \cdot (g + 1) = G$. Similarly, the sets $\{Q_2\}$ and $\{Q_1, Q_2\}$ are also covers for Q. ▮

Example 30: Consider Example 28, using the min and max functions for *Source* and *Target*, respectively. The set $\{Q_1\}$ is not a cover for Q. In effect, an object t with $\mathit{Target}(Q_1, t) = 0$ might still have $\mathit{Target}(Q_2, t) = 1$, making $\mathit{Target}(Q, t) = \max\{0, 1\} = 1$. Therefore, for no $G < 1$ will the definition of cover hold. Similarly, $\{Q_2\}$ is not a cover for Q. However, $\{Q_1, Q_2\}$ is a cover. ▮

The main property of sources that we investigate in the rest of the chapter is defined next. As we will see, if a source satisfies this property for a query, then there are cases where we do not need to extract the entire contents of the source to find the top *Target* objects for the query. Furthermore, we will show that if a source does not satisfy this property, then we always need to extract its entire contents.

Definition 4: *A source S is* tractable *for a query Q if there is a cover C for Q that consists only of queries that are manageable at S (i.e., if there is a manageable cover for Q at S, in short).*

Example 30: (cont.) Although Q is not manageable at source S, as shown above, there is a manageable cover for it, namely $\{Q_1, Q_2\}$. (Q_i is manageable at S because $\mathit{Target}(Q_i, t) = \mathit{Source}(S, Q_i, t)$ $\forall$ object t, $i = 1, 2$.) Therefore, S is tractable for Q. ▮

5.2 Extracting Top Objects from a Tractable Source

In this section we present an algorithm to extract the top *Target* objects for a query from a tractable source (Section 5.2.1), and then we analyze its performance experimentally (Section 5.2.2). Since we will deal with a single source, and to simplify our notation, we sometimes omit mentioning the source explicitly. For example, we use $\mathit{Source}(Q, t)$ as shorthand for $\mathit{Source}(S, Q, t)$.

```
Algorithm 1 Top
Input: A query $Q$ and a source $S$ that is tractable for $Q$.
Method:
(1) Pick a manageable cover $C = \{Q_1, \ldots, Q_m\}$ for $Q$ at $S$.
(2) for $i = 1$ to $m$
(3)     Define $\epsilon_i$ for $Q_i$ as in Definition 1.
(4) Pick $0 \leq g_1, \ldots, g_m, G < 1$ for cover $C$ as in Definition 3.
(5) for $i = 1$ to $m$
(6)     Retrieve all objects $t$ with $Source(Q_i, t) \geq G_i = g_i \Leftrightarrow \epsilon_i$.
(7) Compute $Target(Q, t)$ for all objects $t$ retrieved.
(8) if $\exists i$ such that $G_i \leq 0$ then
        /* We have retrieved all objects in $S$ */
(9)     Go to Step (14).
(10)if $\forall t$ retrieved, $Target(Q, t) \leq G$ then
(11)        Find new $0 \leq g'_1, \ldots, g'_m, G' < 1$ for $C$
            as in Definition 3 such that:
               * $g'_i \leq g_i \; \forall i = 1, \ldots, m$.
               * $\exists j$ such that either $g'_j = 0$ or $g'_j \leq g_j \Leftrightarrow \delta$, for some
                  arbitrary, predefined constant $\delta > 0$.
(12)        Replace $g_i$ by $g'_i$ ($i = 1, \ldots, m$) and $G$ by $G'$.
(13)        Go to Step (5).
(14)Output those objects retrieved that have the highest Target score.
```

Figure 5.1: Algorithm to retrieve the top *Target* objects for a query from a tractable source.

5.2.1 Algorithm *Top*

Consider a query Q and a source S that is tractable for Q. The algorithm in Figure 5.1, which we refer to as *Top*, extracts the top *Target* objects for Q from S [3].

Example 31: Consider the real-estate agent and the scenario of Example 26. Then, Algorithm *Top* can choose $\{Q_1, Q_2\}$ as the cover for query Q (Step (1)). Since *Target* and *Source* agree on single-attribute queries, it follows that $\epsilon_1 = \epsilon_2 = 0$ (Steps (2)

[3]Algorithm *Top* reduces the problem of finding the top *Target* objects for Q in S to the problem of finding all objects t in S with $Target(Q, t) > G$, for some G. [CG96] uses a similar strategy for processing queries over a multimedia repository.

and (3)). We can use any $0 \leq g_1, g_2 < 1$ and $G = 0.5 \cdot (g_1 + g_2)$ in the definition of cover (Definition 3). Suppose that Algorithm *Top* then picks, say, $g_1 = g_2 = 0.8$ with $G = 0.8$ (Step (4)). Then, the algorithm retrieves from S all objects t with $Source(Q_1, t) \geq 0.8$ or $Source(Q_2, t) \geq 0.8$ (Steps (5) and (6)). There is only one such house, the Palo Alto house, that matches the first condition, and no house that matches the second condition.

At this point, the algorithm has extracted all objects t with $Target(Q_1, t) \geq 0.8 + \epsilon_1 = 0.8$ or with $Target(Q_2, t) \geq 0.8 + \epsilon_2$, because Q_1 and Q_2 are manageable for S (see below). If a house t has not been retrieved, then $Target(Q_1, t) < 0.8$ and $Target(Q_2, t) < 0.8$. Because $\{Q_1, Q_2\}$ is a cover, then $Target(Q, t) \leq G = 0.8$. The *Target* score for Q for the Palo Alto house is $0.6 \leq 0.8$ (Step (7)), as discussed above. Consequently, the algorithm goes to Step (11) and lowers g_1 to, say, 0.7, and g_2 to, say, 0.45, assuming $\delta = 0.1$, for example.

No new objects are retrieved in Steps (5) and (6), since all of the Mountain View houses have a *Source* score for Q_1 of 0.6 ($\not\geq g_1 = 0.7$) and a *Source* score for Q_2 of 0.4 ($\not\geq g_2 = 0.45$). The Palo Alto house is retrieved again, of course. Since G for g_1 and g_2 is now 0.575, which is less than 0.6, the *Target* score for the Palo Alto house for Q, then the algorithm stops (Step (14)) and returns the object with the highest score found so far, i.e., the Palo Alto house. ∎

Theorem 1: *Let Q be a query and S a source that is tractable for Q. Then, Algorithm* Top *extracts the top Target objects for Q from S.*

Proof: The algorithm terminates, since the original g_i values are decreased (Step (11)) either to zero, in which case the algorithm stops after Steps (8) and (9), or by at least δ, for a constant $\delta > 0$.

If the algorithm stops because there is some $G_i \leq 0$, then it has extracted all objects t with $Source(Q_i, t) \geq 0$, i.e., all of the objects in S. In particular, it has retrieved the top *Target* objects.

If when the algorithm stops $G_i > 0 \; \forall i = 1, \ldots, m$, then it has extracted all objects t with $Target(Q_i, t) \geq g_i$. Also, it has retrieved an object t' with $Target(Q, t') > G$ (Step (10)). Consequently, from the fact that C is a cover for Q and the choice of G, it follows that any object t that has not been retrieved has $\mathit{Target}(Q,t) \leq G$. Object t' is already better for Q than any unretrieved object. Hence, the top *Target* objects are among the objects already extracted from S. ∎

Consider a source S that is tractable for a query Q. We cannot guarantee that Algorithm *Top* never extracts all the objects in S. As a trivial example, consider the case when there is only one object t in S, and t is such that $\mathit{Target}(Q,t) = 1$. The algorithm then necessarily extracts all the objects in S, namely, object t.

Nevertheless, in many cases Algorithm *Top* is much more efficient than this. In particular, if Q has a manageable cover with high associated g_i values (Definition 3) and low associated ϵ_i values (Definition 1), then the algorithm might stop after examining just a few of the objects in S. (See Section 5.2.2.) Furthermore, as the following theorem shows, we can always define the contents of S in such a way that the algorithm stops without retrieving all of these objects from S.

Theorem 2: *Let Q be a query and S a source that is tractable for Q. Assume that there is a manageable cover $C = \{Q_1, \ldots, Q_m\}$ for Q such that $g_i \Leftrightarrow \epsilon_i > 0$ $\forall i = 1, \ldots, m$ (ϵ_i and g_i are as in Definitions 1 and 3, respectively). Then, there exist instances of S where Algorithm* Top *might find the top Target objects from Q before extracting all of the objects in S.*

Proof: We will "populate" S in such a way that Algorithm *Top* stops (correctly, from Theorem 1) before examining all the objects in S.

Define the contents of S as just two objects, t and t'. Object t is such that $\mathit{Target}(Q,t) > G$, for some G that is suitable for $g_1, \ldots, g_m$. Consequently, from the definition of cover, $\mathit{Target}(Q_j,t) > g_j$ for some j. Therefore, from the choice of ϵ_j, $\mathit{Source}(Q_j,t) > g_j \Leftrightarrow \epsilon_j = G_j$. Now, define object t' in such way that $\mathit{Source}(Q_i,t') < G_i$ $\forall i = 1, \ldots, m$.

Let Algorithm *Top* choose cover C in Step (1), and $g_1, \ldots, g_m, G$ in Step (4). Then, the algorithm would retrieve t in Step (6), since $\mathit{Source}(Q_j,t) \geq G_j$ for some j, and it would not retrieve t', since $\mathit{Source}(Q_j,t) < G_i$ $\forall i$. Furthermore, $\mathit{Target}(Q,t) > G$. Consequently, the algorithm stops after checking that the condition in Step (10) is false and executing Step (14), without ever extracting object t'. ∎

5.2.2 Performance of Algorithm *Top*

For some sources, Algorithm *Top* retrieves most of their objects to find the top *Target* objects for a query. However, for an important class of sources this algorithm manages to extract only a few objects. In this section, we show a preliminary analysis of the efficiency of Algorithm *Top*. (A more exhaustive analysis is part of our future work in this area.) For this analysis, we focus on two important *Target* functions:

- $\mathit{Target}(Q,t) = \min\{\mathit{Target}(Q_1,t),\ldots,\mathit{Target}(Q_n,t)\}$, and

- $\mathit{Target}(Q,t) = \max\{\mathit{Target}(Q_1,t),\ldots,\mathit{Target}(Q_n,t)\}$.

To analyze the behavior of Algorithm *Top* for a source S and a query Q, we define certain options that the algorithm leaves open. In particular, for the $\mathit{Target} = \min$ case we will choose an arbitrary single-attribute query Q_i and take $\{Q_i\}$ as the cover in Step (1). For the $\mathit{Target} = \max$ case we choose all single-attribute queries and take $\{Q_1,\ldots,Q_n\}$ as the cover in Step (1). Furthermore, we let $g_1 = \ldots = g_n = G_0$ for some arbitrary $0 \leq G_0 < 1$ in Step (4). To simplify our discussion, we will also assume that $\epsilon_1 = \ldots = \epsilon_n = \epsilon$, for some arbitrary $0 \leq \epsilon < 1$. In other words, we assume that all single-attribute queries behave equally in terms of the relationship between their associated *Source* and *Target* scores. Finally, Algorithm *Top* decrements the value of G (and hence, the g_i values) by a fixed δ in Step (11).

To estimate the number of objects retrieved by *Top*, we start by studying the number of objects retrieved in one iteration of the algorithm. For this, we assume that source S accepts a single-attribute query Q_i and a score G, and returns all objects with a *Source* score no less than G for Q_i. If we contact S twice with the same query Q_i but with scores G_1 and G_2, $G_1 > G_2$, then the answer for G_2 includes all the objects returned for G_1. Thus, S does not accept requests for the *next* best unretrieved objects with a certain minimum score, for example. Considering such sources is part of our future work.

For a given value of G, and for each single-attribute query Q_i in the cover of Step (1), *Top* retrieves an expected $R(G)$ objects:

$$R(G) = \sum_{t \in S} Pr(Source(Q_i, t) \geq G \Leftrightarrow \epsilon) \cdot 1$$

If we focus on the class of sources S such that the *Source* scores for Q_i in S are uniformly distributed, then $Pr(Source(Q_i, t) \geq g) = 1 \Leftrightarrow g$ for all objects $t \in S$. Consequently, *Top* retrieves an expected $(1 \Leftrightarrow G + \epsilon) \cdot N$ objects for Q_i, where N is the number of objects in S. Then, in one iteration, *Top* retrieves $R(G) = (1 \Leftrightarrow G + \epsilon) \cdot m \cdot N$ objects, where m is the number of single-attribute queries in the cover of Step (1). For the *Target* = min case, $m = 1$. For the *Target* = max case, $m = n$.

Given a value for G, Algorithm *Top* might not find the top *Target* objects for Q in this iteration (i.e., the condition in Step (10) is satisfied). This is the case if and only if there is no object $t \in S$ with $Target(Q, t) > G$. Thus, the algorithm will not stop in this first iteration with probability $F(G)$:

$$\begin{aligned} F(G) &= Pr(\forall t \in S : Target(Q, t) \leq G) \\ &= \Pi_{t \in S} Pr(Target(Q, t) \leq G) \end{aligned}$$

As with the *Source* scores, we will restrict our analysis to sources where the *Target* scores for the Q_i queries are uniformly distributed [4]. Then, for the *Target* = min case we have:

$$\begin{aligned} F(G) &= \Pi_{t \in S} Pr(\min_{i=1}^{n}\{Target(Q_i, t)\} \leq G) \\ &= \Pi_{t \in S} Pr(\exists 1 \leq i \leq n | Target(Q_i, t) \leq G) \\ &= \Pi_{t \in S}(1 \Leftrightarrow Pr(\forall 1 \leq i \leq n : Target(Q_i, t) > G)) \\ &= \Pi_{t \in S}(1 \Leftrightarrow \Pi_{i=1}^{n} Pr(Target(Q_i, t) > G)) \end{aligned}$$

[4] Our analysis in this section focuses on sources for which both the *Target* and *Source* scores for single-attribute queries are uniformly distributed. Hence, although these scores might differ for particular objects, the overall distributions are the same. This is not a fundamental limitation of our analysis, and we will consider other distributions as part of our future work.

$$
\begin{aligned}
&= (1 \Leftrightarrow \Pi_{i=1}^{n}(1 \Leftrightarrow G))^{N} \\
&= (1 \Leftrightarrow (1 \Leftrightarrow G)^{n})^{N}
\end{aligned}
$$

On the other hand, for the *Target* = max case we have:

$$
\begin{aligned}
F(G) &= \Pi_{t \in S} Pr(\max_{i=1}^{n}\{\mathit{Target}(Q_i, t)\} \leq G) \\
&= \Pi_{t \in S} Pr(\forall 1 \leq i \leq n : \mathit{Target}(Q_i, t) \leq G) \\
&= \Pi_{t \in S}(\Pi_{i=1}^{n} Pr(\mathit{Target}(Q_i, t) \leq G)) \\
&= (\Pi_{i=1}^{n} G)^{N} \\
&= G^{n \cdot N}
\end{aligned}
$$

We now estimate the expected number of objects retrieved during all iterations of Algorithm *Top*. In effect, the algorithm starts with some value G_0 for G. If the algorithm successfully finds the top *Target* objects (with probability $1 \Leftrightarrow F(G_0)$) then it stops. Otherwise, the algorithm lets $G_1 = G_0 \Leftrightarrow \delta$, and continues until it reaches $G_k = 0$ or the condition in Step (10) is not satisfied. (k is the smallest value for which $G_{k-1} \Leftrightarrow \delta \leq 0$.) Consequently, the expected total number of objects retrieved by *Top* when it starts with score G_0, $T(G_0)$, is:

$$
\begin{aligned}
T(G_0) &= R(G_0) + F(G_0) \cdot T(G_1|G_0) \\
&= R(G_0) + F(G_0) \cdot (R(G_1) + F(G_1|G_0) \cdot T(G_2|G_1)) \\
&= R(G_0) + F(G_0) \cdot R(G_1) + F(G_0) \cdot F(G_1|G_0) \cdot T(G_2|G_1) \\
&= R(G_0) + F(G_0) \cdot R(G_1) + F(G_0 \wedge G_1) \cdot T(G_2|G_1) \\
&= R(G_0) + F(G_0) \cdot R(G_1) + F(G_1) \cdot T(G_2|G_1) \\
&= \cdots \\
&= R(G_0) + (\sum_{i=1}^{k-1} F(G_{i-1}) \cdot R(G_i)) + F(G_{k-1}) \cdot T(G_k|G_{k-1}) \\
&= R(G_0) + (\sum_{i=1}^{k-1} F(G_{i-1}) \cdot R(G_i)) + F(G_{k-1}) \cdot m \cdot N
\end{aligned}
$$

where:

Parameter	*Description*
G_0	Initial score used by *Top*
δ	Value by which the initial score is decreased in each iteration of *Top*
ϵ	Bound on how much lower *Source* scores might be than *Target* scores
N	Number of objects in source
n	Number of significant attributes in query
m	Number of single-attribute queries in cover

Figure 5.2: The main parameters in our experiments for Algorithm *Top*.

- $T(G_{i+1}|G_i)$ is the expected number of objects retrieved when *Top* starts with score G_{i+1}, given that $\forall t \in S : \mathit{Target}(Q,t) \leq G_i$.
- $F(G_{i+1}|G_i)$ is $Pr(\forall t \in S : \mathit{Target}(Q,t) \leq G_{i+1} | \forall t \in S : \mathit{Target}(Q,t) \leq G_i)$.
- $F(G_i \wedge G_{i+1})$ is $Pr((\forall t \in S : \mathit{Target}(Q,t) \leq G_{i+1}) \wedge (\forall t \in S : \mathit{Target}(Q,t) \leq G_i))$.
- $T(G_k|G_{k-1}) = T(0|G_{k-1}) = m \cdot N$, because $G_k = 0$.

For our experiments, we use the expressions above to numerically compute the total number of objects that Algorithm *Top* is expected to retrieve. We assume that $\delta = 0.01$ (G decreased in steps of 0.01), $n = 4$ (four significant attributes in query Q), and $N = 10,000$ (10,000 objects in source S). Figure 5.2 summarizes the main parameters in our experiments.

To see the impact of the initial score G_0 used by *Top*, we set ϵ to 0 (i.e., the *Source* scores for single-attribute queries are never lower than the corresponding *Target* scores), and vary G_0. Figure 5.3 shows the percentage of the source objects that *Top* is expected to retrieve (i.e., the expected value of $\frac{T(G_0)\cdot 100}{N}$). For *Target* = max, the best values for G_0 are quite high: 1 and 0.99. The reason is that the probability that *Top* does not stop after one iteration, $F(G_0)$, is quite low even for $G_0 = 0.99$. Therefore, the best strategy for *Target* = max is to start with a high value for G_0, and stop after one iteration, with high probability. In this case, only around 4% of the objects in the source are expected to be retrieved. For *Target* = min, the best

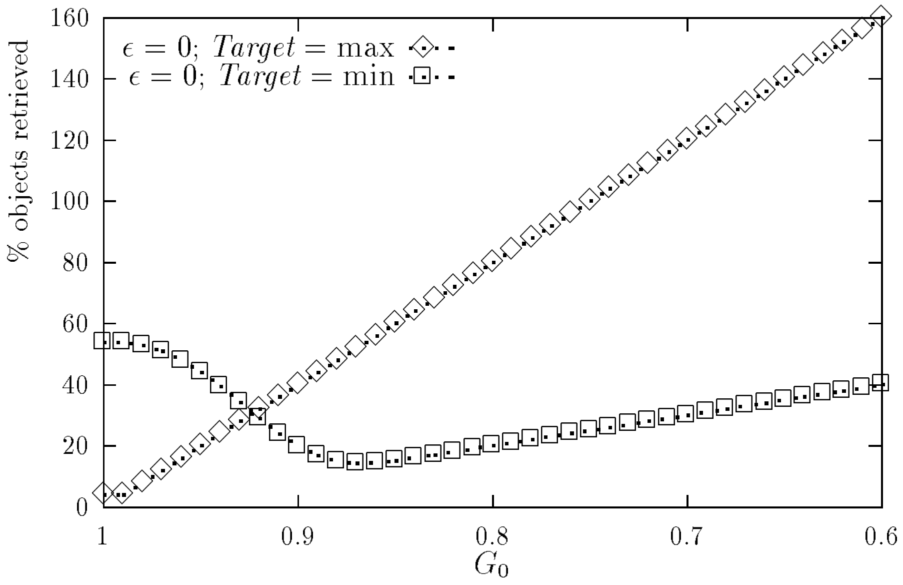

Figure 5.3: The percentage of objects retrieved by Algorithm *Top* as a function of the initial score G_0 used ($\epsilon = 0$).

value for G_0, $G_0 = 0.87$, is lower than that for *Target* = max. The reason is that the probability that there is an object t with *Target*$(Q, t) > G_0$ is lower: such an object needs high *Target* scores for all attributes in Q, not just for one attribute as is the case for *Target* = max. Consequently, by starting with a lower value of G_0, *Top* does not retrieve several times those objects with high *Source* scores for the single-attribute query being used. Then, only around 14% of the objects in the source are expected to be retrieved. Note that to take advantage of these "good" values for G_0, we need to know that *Target* and *Source* scores follow reasonably uniform distributions.

To check the effect of higher values of ϵ on the above figures, Figure 5.4 also shows results for $\epsilon = 0.10$. The curves for $\epsilon = 0$ are the same as in Figure 5.3. As we see from this figure, the shape of the curves for both *Target* = max and *Target* = min is quite similar to that of the corresponding curves for $\epsilon = 0$. However, more objects are retrieved in both cases. More specifically, exactly $\epsilon \cdot m \cdot N$ more objects are retrieved in each iteration of the algorithm, where $m = n$ for *Target* = max, and $m = 1$ for *Target* = min.

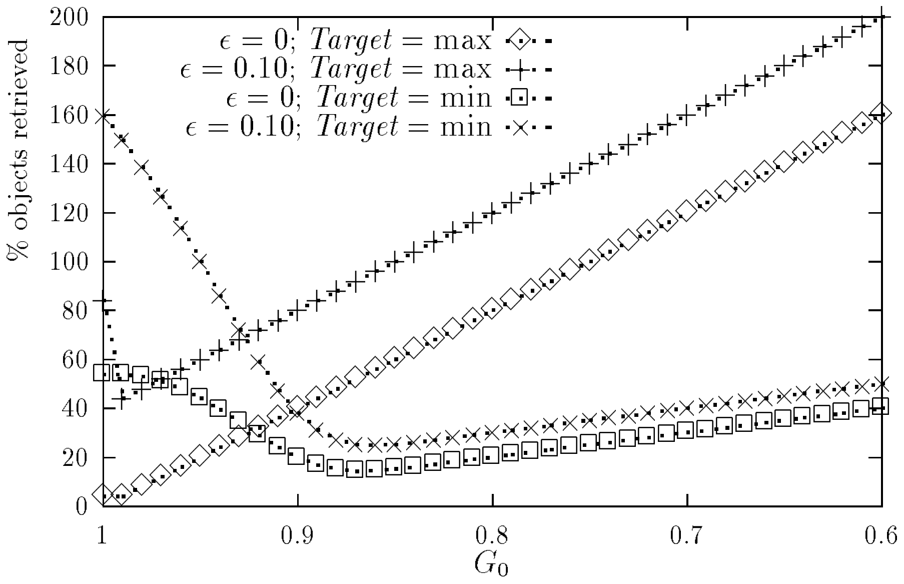

Figure 5.4: The percentage of objects retrieved by Algorithm *Top* as a function of the initial score G_0 used ($\epsilon = 0$ and $\epsilon = 0.10$).

Finally, to study the impact of higher values of ϵ, we fix G_0 to the best values (0.99 and 0.87 for *Target* = max and *Target* = min, respectively), and vary ϵ. As expected, Figure 5.5 shows that the number of objects retrieved increases steadily as ϵ increases. For high values of ϵ, Algorithm *Top* retrieves more than N objects. For such values, a better strategy than to use Algorithm *Top* is to just retrieve all of the N objects in S directly. Interestingly, the case *Target* = max is affected more strongly, because the expected number of iterations of *Top* does not vary with ϵ, and in each iteration, the algorithm retrieves n times more objects for *Target* = max than it does for *Target* = min, as discussed above.

The experiments above are preliminary. However, they give evidence that for an important family of sources Algorithm *Top* manages to find the top *Target* objects by inspecting only a small fraction of the objects in the sources. However, in some cases (e.g., when ϵ is high), our algorithm does worse than just retrieving all of the sources' contents. In such cases, there is probably no good way to answer the queries. These results complement Theorem 2 (Section 5.2.1), which shows that source tractability, together with the assumption in the theorem that $\forall i$, $g_i \Leftrightarrow \epsilon_i > 0$, forms a *sufficient*

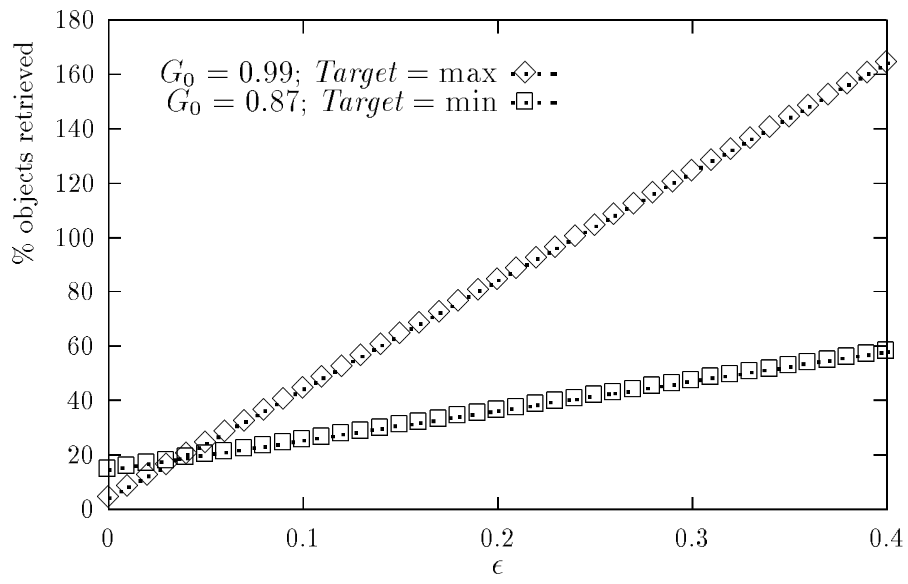

Figure 5.5: The percentage of objects retrieved by Algorithm *Top* as a function of ϵ ($G_0 = 0.87$ for min and $G_0 = 0.99$ for max).

condition for being able to sometimes extract a top *Target* object from a source without accessing all of its objects. As we will see in Section 5.4, source tractability is also a necessary condition: if a source is not tractable for a query, we must *always* access all of its contents to extract the top *Target* objects for the query.

5.3 Varying Source Types

Section 5.2 presented an algorithm to extract top objects from sources that satisfied a number of properties. However, the sources that a metasearcher has to deal with are intrinsically autonomous and heterogeneous. Some sources reveal how they process queries, while others conceal this. Some sources return quite complete information together with their query results, while others just provide quite basic data. In this section we revisit the properties of Section 5.2 and see in what cases we can adapt Algorithm *Top* for sources where these properties do not hold.

Property 1: Information in Query Results

Algorithm *Top* requires that sources return the values of the objects for those attributes with significant values in a query. In effect, Step (7) of the algorithm computes the *Target* scores for the objects retrieved using these values. However, some sources might return just object ids, or just a few of these attribute values in the query results. In such a case, a possibility for Algorithm *Top* is to access each object retrieved in its entirety to obtain all the information needed for the *Target* scores, which could be quite time consuming.

Alternatively, if the metasearcher knows how to map *Source* scores into *Target* scores for single-attribute queries (like in the real-estate agent scenario of Example 26), then it might compute the *Target* scores for the original query without accessing the actual attribute values for each object. This requires, of course, that the sources report their *Source* scores. If these scores are not available, then the metasearcher needs the attribute values.

Example 32: Consider the real-estate agent of Example 26. In this case, the *Target* function for our metasearcher coincides with the agent's *Source* function for single-attribute queries. Consider a query Q with significant values for both the *Location* and *Price* attributes. Then, if an object t is retrieved by both the single-attribute queries for *Location* and *Price* with *Source* scores s_1 and s_2, respectively, then the metasearcher can compute $Target(Q, t)$ as $0.5 \cdot s_1 + 0.5 \cdot s_2$. However, if t is retrieved by only one of these queries, then the metasearcher cannot compute the *Target* score this way, and it has to obtain the missing attribute value for t. ▮

Property 2: Information about Source Manageability

Algorithm *Top* requires that a metasearcher know what single-attribute queries are manageable at a source. Furthermore, a metasearcher needs to know the ϵ values (Definition 1) that bound how much lower than the *Target* scores the *Source* scores might be (Steps (2) and (3)). All this information can be derived from the *Source* scoring function of a source. Unfortunately, this function might not be publicly known, as the sources view it as their competitive advantage.

If the *Source* function for a source is not known, and Property 2 does not hold either (i.e., the metasearcher does not know whether an attribute is manageable or not, or the ϵ values), then a metasearcher can only try to guess all this information by issuing sample queries to the sources. However, whatever conclusion the metasearcher draws about a *Source* function would only be a statistical guess, since there is no way to guarantee (unless more information is available) that the corresponding source would not behave differently in the future, for example. Thus, users would still get ranked query results from the metasearcher, but they should be warned that high ranking objects might be missing from these results.

Example 33: Consider the real-estate agent of Example 26. Suppose that a metasearcher does not know whether a single-attribute query on *Location* is manageable at the source. Suppose that the metasearcher, off-line, issued a series of single-attribute queries on *Location* to the source and computed, for each such query L_i, $e_i = \max_{t \text{ retrieved}}\{Target(L_i, t) \Leftrightarrow Source(L_i, t)\}$. Based on the e_i values retrieved, the metasearcher might then decide that indeed such single-attribute queries are always manageable at the source, with associated $\epsilon = \max\{0, \max_i\{e_i\}\}$. In particular, in our real-estate scenario, ϵ would be determined to be zero, which is the right decision. ∎

To proceed as in the example above, a metasearcher needs the *Source* scores for each object retrieved. If a source does not even report these scores, then a metasearcher would have to resort to other forms of "guessing" for the ϵ values.

Other Implicit Properties of the Source Behavior

Algorithm *Top* asks sources for all objects with *Source* score G_i or higher for a single-attribute query and for arbitrary values of G_i (Steps (5) and (6)). However, a source interface might fail to allow this in several ways.

First, a source might not accept a single-attribute query for a particular attribute. For example, the real-estate agent of Example 26 might not accept queries that specify a target *Price* but not a target *Location*. In this case, we can redefine cover (Definition 3) to allow for multiple-attribute queries.

Example 34: Consider a source S and a query Q over attributes A_1, A_2, and A_3. Suppose that S does not accept single-attribute queries on A_1. However, S accepts multi-attribute query $Q_{1,2}$, which is the restriction of Q to A_1 and A_2, and S also accepts single-attribute query Q_3. Assume that $\exists 0 \leq g_{1,2}, g_3, G < 1$ such that $\forall$ object t, if $\mathit{Target}(Q_{1,2}, t) \leq g_{1,2}$ and $\mathit{Target}(Q_3, t) \leq g_3$ then $\mathit{Target}(Q, t) \leq G$. Then, $C = \{Q_{1,2}, Q_3\}$ is a cover for Q if we now allow multi-attribute queries like $Q_{1,2}$ in a cover. ▮

Thus, if we can find a manageable cover using multiple-attribute queries, then Algorithm *Top* might proceed as before. Otherwise, the metasearcher will not be able to extract the top *Target* objects from the source (Section 5.4).

As a second problem that a metasearcher might have with a source, the source might only return the top objects for a query, without including the *Source* scores for the objects returned. In such a case, a metasearcher does not know if it has retrieved all the objects with a *Source* score of at least G_i or not, and Step (6) needs this information. Unfortunately, the definition of manageability does not allow us to infer much about the *Source* score of an object given its *Target* score. For example, consider a source that assigns most objects a *Source* score of 1 for a given query. Then, the top k *Source* objects for that query might not include any of the top *Target* objects. Therefore, to work with such a source a metasearcher would need to know some bound on how different the *Source* and *Target* scores might be.

Finally, a source might always return a fixed maximum of, say, 200 objects per query, for efficiency reasons or to prevent users from downloading all the source's valuable contents, for example. In such a case, a metasearcher that wants all objects t with $\mathit{Source}(Q_i, t) \geq G_i$ might retrieve only those objects with $\mathit{Source}(Q_i, t) \geq G'_i$, for some higher G'_i. If these higher values (and their associated G', as in Definition 3) are not low enough to make the condition in Step (10) false, then the metasearcher cannot guarantee that it has obtained the top *Target* objects from the source, and will have to return only approximate results.

In summary, ranking objects from autonomous sources is a difficult problem. For Algorithm *Top* to work, the sources need to provide a query interface that permits "powerful enough" searches based on scores, and the sources must return "sufficient" information on the matching objects so that the metasearcher can compute its *Target* scores. Finally, the metasearcher needs to know some "fundamental properties" of the source scoring functions.

Given all that is needed by our algorithm, one may wonder if there could be some *other* algorithms that require less source functionality or less knowledge of the sources. In the next section, we show how under some very broad assumptions, essentially there is no algorithm that can rank results in a meaningful way for a source that is not tractable for a given query.

5.4 Source Tractability as a Necessary Condition

In this section, we will see that if our source is not tractable, then any strategy to extract the top *Target* objects from the source using single-attribute queries must *always* retrieve all the objects. To prove this, we need to make some assumptions about the *Source* and *Target* scoring functions. We believe that these assumptions are not restrictive, and all reasonable scoring functions that we can think of meet these criteria. These assumptions are in addition to the properties in Section 5.2.

Our first assumption about the *Source* scores for a query is that these scores can take values ranging all the way from 0 to 1. Using this assumption we rule out "constant" *Source* score functions.

Assumption 4: Variability of *Source*: *Let Q be a query. Then, $\exists t_1$, t_2 objects such that $Source(Q, t_1) = 0$ and $Source(Q, t_2) = 1$.*

Our second assumption affects both the *Target* and *Source* scores for a query Q. In essence, these scores must only depend on the attributes corresponding to the significant values in Q. Thus, the attribute values for "don't care" attributes are irrelevant for *Target* and *Source*.

Assumption 5: Locality of *Source* **and** *Target*: *Let Q be a query and $A_1, \ldots, A_m$ the attributes with significant values in Q. Let t and t' be two objects such that*

$t[A_i] = t'[A_i]$ for $i = 1, \ldots, m$ (i.e., t and t' agree on all the significant attributes in Q). Then, Target(Q, t) = Target(Q, t') and Source(Q, t) = Source(Q, t').

Our final assumption affects the *Target* scores for a query Q, and is related to Assumption 5. If we "improve" an object t for Q by changing its value for A_j so that it is better for Q_j, for some j, then *Target*(Q, t) should not decrease. Also, this assumption bounds the effect of a change in *Target*(Q_j, t) over *Target*(Q, t).

Assumption 6: Monotonicity of *Target*: *Let Q be a query and $A_1, \ldots, A_m$ the attributes with significant values in Q. Let t and t' be two objects such that $t[A_i] = t'[A_i]$ for $i = 1, \ldots, m$, $i \neq j$ for some j. Also, Target(Q_j, t) $\geq$ Target(Q_j, t') $\Leftrightarrow \delta$, for some $\delta \geq 0$. Then, Target(Q, t) $\geq$ Target(Q, t') $\Leftrightarrow \delta$.*

Next, we define the class of executions for a query Q that we analyze in this section. In short, these executions follow the methodology of Algorithm *Top* in that they query the source using single-attribute queries for Q, until they have obtained "enough" objects and, hopefully, the top *Target* objects for Q. These executions decide when they have retrieved enough objects based only on the objects that they retrieve. They do not, for example, have any "magic" information about the unseen contents of the source.

Definition 5: *Let S be a source, Q a query, and $C = \{Q_1, \ldots, Q_m\}$ a set of single-attribute queries for Q. Then, a* partial retrieval *for Q and S using C is a set of objects $\{t \in S | Source(Q_i, t) > g_i$, for some $i = 1, \ldots, m\}$, with $0 < g_i < 1$, $i = 1, \ldots, m$[5]. The g_i values are determined based on the objects retrieved, and not on the rest of the source contents.*

To prove the main result of this section, we first need the following lemma, which identifies a condition that implies manageability.

[5]This definition excludes executions that request all objects with a non-zero *Source* score for Q_i, since g_i has to be greater than zero. However, this is not a limitation for most sources, where *Source* scores have finite precision.

Lemma 1: *Let Q be a query and S a source for which $\exists 0 < x \leq y < 1$ such that $\forall$ object t, either Source$(Q,t) > x$ or Target$(Q,t) < y$. Then, Q is manageable at source S.*

Proof: We need to find $0 \leq \epsilon < 1$ such that $\forall$ object t, *Source*$(Q,t) \geq$ *Target*$(Q,t) \Leftrightarrow \epsilon$. Let $\epsilon = \max\{1 \Leftrightarrow x, y\}$. ($\epsilon > 0$, since $\epsilon \geq y > 0$, and $\epsilon < 1$, since $1 \Leftrightarrow x < 1$ and $y < 1$.) Consider an object t. From the assumptions, it follows that either *Source*$(Q,t) > x$ or *Target*$(Q,t) < y$:

1. *Source*$(Q,t) > x$:

$$\textit{Source}(Q,t) > x \geq \textit{Target}(Q,t) \Leftrightarrow 1 + x = \textit{Target}(Q,t) \Leftrightarrow (1 \Leftrightarrow x)$$

   because *Target*$(Q,t) \leq 1$.

   Furthermore, $1 \Leftrightarrow x \leq \epsilon$. Then, *Source*$(Q,t) \geq$ *Target*$(Q,t) \Leftrightarrow \epsilon$.

2. *Target*$(Q,t) < y$:

$$\textit{Target}(Q,t) < y \leq \epsilon$$

   Then, *Target*$(Q,t) \Leftrightarrow \epsilon < 0$. Consequently, *Source*$(Q,t) \geq$ *Target*$(Q,t) \Leftrightarrow \epsilon$, because *Source*$(Q,t) \geq 0$.

$\blacksquare$

We are now ready for our main result. Consider a partial retrieval for a query Q and a source S that is not tractable for Q and that has no objects with a *Target* score of 1. The following theorem shows that such a partial retrieval might miss objects that are better than any object retrieved. In fact, we can *always* build better objects and "include" them in the source. These objects would not be retrieved, because the execution that built the partial retrieval at hand would see exactly the same top *Source* objects for each single-attribute query. Thus, this execution would stop at exactly the same point as before for each of the single-attribute queries (Definition 5), hence missing the (new) top *Target* objects. Consequently, such a partial retrieval might always be incorrect, leaving no alternative but to extract the entire source contents to obtain the top *Target* objects for Q.

Theorem 3: *Consider a query Q and a* minimal *cover $C = \{Q_1, \ldots, Q_m\}$ for Q. Assume that $\exists j$ such that Q_j is not manageable at source S, and Q_i is manageable at source S, $\forall i \neq j$. Consider a partial retrieval for Q and S using C, and let $G = \max_{t \text{ retrieved}}\{\mathit{Target}(Q, t)\}$. Assume that $G < 1$. Then, we can build an object l not in the partial retrieval such that $\mathit{Target}(Q, l) > G$.*

Proof: Let $0 < g_i < 1$, $i = 1, \ldots, m$, be the values used by the partial retrieval for Q and S using C (Definition 5). For every $i \neq j$, pick an object t_i such that $\mathit{Source}(Q_i, t_i) \leq g_i$. (Such objects exist from Assumption 4.) From the choice of t_i and the definition of partial retrieval, it follows that t_i is not retrieved by query Q_i. Let $a_i = \mathit{Target}(Q_i, t_i)$ $(0 \leq a_i \leq 1)$.

From the minimality of C it follows that $C \Leftrightarrow \{Q_j\}$ is not a cover for Q. Then, there is an object l_0 such that $\mathit{Target}(Q_i, l_0) \leq a_i$ $\forall i \neq j$ and $\mathit{Target}(Q, l_0) > G$. Otherwise, $C \Leftrightarrow \{Q_j\}$ would be a cover for Q. (If $m = 1$, just pick any object l_0 with $\mathit{Target}(Q, l_0) > G$.) Furthermore, $\mathit{Target}(Q_i, l_0) \leq a_i = \mathit{Target}(Q_i, t_i)$ $\forall i \neq j$.

We now build an object l_1 using the t_is and l_0:

$$l_1[i] \quad = \quad \begin{cases} t_i[i] & \text{if } i = 1, \ldots, m,\ i \neq j \\ l_0[i] & \text{otherwise} \end{cases}$$

From the choice of l_1 it follows that:

- $i = 1, \ldots, m$, $i \neq j$: $\mathit{Target}(Q_i, l_1) = \mathit{Target}(Q_i, t_i)$, because $l_1[i] = t_i[i]$ and using Assumption 5. Furthermore, $\mathit{Target}(Q_i, t_i) = a_i \geq \mathit{Target}(Q_i, l_0)$.
- Otherwise: $\mathit{Target}(Q_i, l_1) = \mathit{Target}(Q_i, l_0)$, because $l_1[i] = l_0[i]$ and using Assumption 5.

Then, $\mathit{Target}(Q_i, l_1) \geq \mathit{Target}(Q_i, l_0)$, $\forall i$. Hence, from Assumption 6, it follows that $\mathit{Target}(Q, l_1) \geq \mathit{Target}(Q, l_0) > G$. Also, for $i = 1, \ldots, m$, $i \neq j$, $\mathit{Source}(Q_i, l_1) = \mathit{Source}(Q_i, t_i) \leq g_i$. Hence l_1 is not retrieved by any of the Q_i queries, $i \neq j$.

Next, we build another object l_2. We will use l_1 and l_2 to construct the final object l that we need for our proof. Let $0 < \delta < \mathit{Target}(Q, l_1) \Leftrightarrow G$. Now, let $x = g_j$ and $y = \max\{x, \mathit{Target}(Q_j, l_1) \Leftrightarrow \delta\}$. (Then, $0 < x \leq y < 1$.) Since Q_j is not manageable at

S, from Lemma 1 it follows that there is an object l_2 such that $\mathit{Source}(Q_j, l_2) \leq x$ and $\mathit{Target}(Q_j, l_2) \geq y$. Then, $\mathit{Source}(Q_j, l_2) \leq g_j$ and $\mathit{Target}(Q_j, l_2) \geq \mathit{Target}(Q_j, l_1) \Leftrightarrow \delta$.

Finally, let us define object l by letting $l[i] = l_1[i]\ \forall i \neq j$ and $l[j] = l_2[j]$. Then,

- $i \neq j$: $\mathit{Target}(Q_i, l) = \mathit{Target}(Q_i, l_1)$.
- Otherwise: $\mathit{Target}(Q_j, l) = \mathit{Target}(Q_j, l_2) \geq \mathit{Target}(Q_j, l_1) \Leftrightarrow \delta$.

Then, from Assumption 6 it follows that $\mathit{Target}(Q, l) \geq \mathit{Target}(Q, l_1) \Leftrightarrow \delta > \mathit{Target}(Q, l_1) \Leftrightarrow \mathit{Target}(Q, l_1) + G = G$. Also,

- $i = 1, \ldots, m$, $i \neq j$: $\mathit{Source}(Q_i, l) = \mathit{Source}(Q_i, l_1) \leq g_i$.
- Otherwise: $\mathit{Source}(Q_j, l) = \mathit{Source}(Q_j, l_2) \leq g_j$.

Thus, we have constructed an object l that satisfies the conditions in the theorem. ▮

Corollary 1: *Let $C = \{Q_1, \ldots, Q_m\}$ be a (not necessarily minimal) cover for the query Q of Theorem 3 such that it does not contain any manageable cover for Q. Then, we can still build an object l as in Theorem 3 for any partial retrieval for Q and S using C.*

Proof: Let $Q_{i_1}, \ldots, Q_{i_r}$ be all the manageable queries in C. Since they do not constitute a cover for Q, we can still build object l_1 as in Theorem 3. Then, we "fill" each of the values for each Q_j that is not manageable in exactly the same way as we did for l_1, using the fact that Q_j is not manageable and Lemma 1. ▮

Note that the main results of this section only cover algorithms that work via multiple single-attribute queries. We believe that this is not a restriction for most sources, since we expect the *Source* scores to match the *Target* scores for single-attribute queries more often than for multi-attribute queries. Consequently, our result has broad applicability, and points out the fundamental properties that are required for extracting the top objects for a query across multiple autonomous sources.

5.5 Conclusion

Many sources rank the objects in query results according to how well these objects match the original query. In this environment, metasearchers usually query multiple autonomous, heterogeneous sources that might use varying result-ranking strategies. In this chapter we have studied two crucial problems that a metasearcher faces: guaranteeing that it has extracted all the top objects for a user query from the underlying sources, and re-ranking these objects according to its own criterion. These are difficult problems, and our goal is to characterize the sources where we have some hope of dealing with these problems efficiently. We have presented necessary properties that any source should satisfy, under broad assumptions. If a source does not verify these properties, then a metasearcher might miss top objects from the source, unless all of the source's contents are retrieved. We have also described a simple algorithm to extract the top objects from a source where our properties hold.

The results in this chapter, and Algorithm *Top* in particular, do not guarantee efficient executions. If not implemented carefully, Algorithm *Top* might retrieve large portions of a source when searching for top *Target* objects. We touched on the efficiency of *Top* in Section 5.2.2. However, the experiments in that section are still preliminary. We will conduct a more exhaustive experimental analysis of the algorithm in the near future. Another interesting open issue is the optimization of queries over multiple sources, perhaps using statistics on the sources' contents. A promising direction is to adapt the work in [CG96] and [Fag96] to our distributed, heterogeneous scenario. Another interesting issue is how to deal with sources that do not satisfy the properties and assumptions that our results need. We discussed this issue in Section 5.3, but we need to explore further, for example, how to deal with sources that return no more than, say, 200 objects per query. These characteristics also impact the optimization of queries over these sources.

Chapter 6

dSCAM: A Non-Traditional Metasearcher

In a renowned 1995 case [Den95], an author, who we will refer to as Mr. X for legal reasons, plagiarized several technical reports and conference papers, and resubmitted them under his own name to other conferences and journals. Unfortunately, most of these papers (nearly 18) passed undetected through the paper review process and were accepted to these conferences and journals. The topics of these papers ranged from Steiner routing in VLSI CAD, to massively parallel genetic algorithms, complexity theory, and network protocols. Mr. X also plagiarized papers from the database field, notably a paper in DAPD by Tal and Alonso [TA94] on three-phase locking, a paper in VLDB '92 by Ioannidis et al. [INSS92] on parametric query optimization, and a paper in ICDE '90 by Leung and Muntz [LM90] on temporal query processing. The Stanford Copy Analysis Mechanism (SCAM) [SGM95, SGM96] played an important role in identifying the papers that Mr. X had plagiarized. (See [Den95] for further details.)

SCAM is a registration server mechanism that helps flag document-copyright violations in Digital Libraries. The target is not simply academic plagiarism, but any type of copying that can financially hurt authors and commercial publishers. SCAM is also useful for removing duplicates and near-duplicates in information retrieval systems [YGM95a]. Essentially, SCAM keeps a large database of documents along with indices to support efficient retrieval of stored documents that are "potential copies." SCAM attempts to find not just identical copies, but also cases of "substantial" overlap. For example, if a document contains several paragraphs or sections that were copied from a registered document, it should be flagged as a potential copy even if there are also significant portions where the documents differ. Documents flagged by SCAM have to be checked manually for actual violations since the copying may have been legal and since SCAM may produce some false positives.

The basic SCAM system requires a database of registered documents. In the future, publishers may indeed establish such "copyright registration servers" [Kah92], and these servers can then automatically check public sources such as netnews articles and WWW/FTP sites for copies of the registered documents. However, if there are multiple registration servers, and one has a suspicious document to check, then the distributed copy detection problem is essentially a metasearching problem, as described in Chapter 1. In effect, we first have to decide what servers to check, since it may be impractical to go to all of them. Furthermore, we may also want to include in our search databases that may not be running a SCAM system. In this case, not only do we have to identify these databases, but we also need to pull out candidate documents so that SCAM can analyze them.

This is precisely what we had to do in Mr. X's case. Initially, we only had the abstracts of the papers that Mr. X had "written," i.e., we had the suspicious documents. Then we proceeded as follows:

1. First we selected existing databases that we thought were likely to contain the matching registered documents. Based on the contents of the suspicious documents we decided that the INSPEC and NCSTRL databases were the most appropriate. (As we mentioned before, INSPEC is a commercial database of electrical engineering and computer science abstracts; NCSTRL is an emerging digital library of computer science technical reports, see `http://www.ncstrl.-org`.)

2. We manually chose some keywords from Mr. X's abstracts, and issued queries such as $\textit{VLSI} \vee \textit{Steiner} \vee \textit{Routing}$ against the above databases.

3. We retrieved the abstracts (about 35,000 overall) that matched the above queries, registered them in SCAM, and then tested the suspicious documents against them. In this way we found a total of 14 cases of plagiarism, most of them previously unknown.

In this chapter we develop *dSCAM*, a metasearcher that automates the entire copy search process. Automating this process is crucial as the number of document databases grows and as publishers rely more on digital publishing. As a matter of fact, appropriate safeguards for intellectual property rights are essential in a large scale public digital library, and an automated *dSCAM* can be one of the tools.

- *Text Source Discovery:* We present the *dSCAM* mechanism that, given many databases and a suspicious document, efficiently identifies the databases that may contain documents that SCAM would consider copies. This problem is fundamentally different from a conventional text source discovery problem (e.g., from that of Chapters 3 and 4): *dSCAM* has to flag a database even if it contains a single document that overlaps significantly with the suspicious document.

- *Query Translation:* We present the *dSCAM* strategies for automatically generating queries from the underlying databases that retrieve potential copies for subsequent analysis by SCAM.

For the source discovery phase, we build on the *GlOSS* approach of Chapters 3 and 4. The idea is to collect in advance "metainformation" about the candidate databases. This can include, for example, information on how frequently terms appear in documents at a particular database. This metainformation is much smaller than a full index of the database or than what SCAM would actually need to detect copies. Then, based on this information, *dSCAM* can rule out databases that do not contain documents that SCAM would consider copies.

Notice that while *dSCAM* is structurally similar to *GlOSS*, there is a fundamental difference. *GlOSS* attempts to discover databases that satisfy a given query (Boolean or vector space). The *GlOSS* problem is a simpler one since all we need to know is that the candidate database contains the necessary terms. However, for *dSCAM*, finding documents that have similar terms to those of the suspicious document is not enough. For example, if the suspicious document only contains a subset that is a copy, then there are terms in the non-copy portion that are not relevant. Thus, simply treating the suspicious document as a *GlOSS* query will not lead us to the right databases.

Instead, *dSCAM* will need to keep more sophisticated statistics than *GlOSS* does, enough to let it identify sites that may have even a single copy, not just documents that are "similar" to the suspicious one in an information retrieval sense. The key challenge is to collect as little information as possible in *dSCAM* to be able to perform this difficult discovery task. Section 6.6 reports experiments that indeed show that our techniques are successful at isolating the databases with potential copies, with relatively few false positives.

In this chapter we consider two types of discovery techniques: *conservative* and *liberal* ones. Conservative techniques only rule out a database if it is certain that SCAM would not consider *any* document there a potential copy. The clear advantage of conservative techniques is that they do not miss potential copies. In contrast, liberal techniques might in principle miss databases with potential copies. However, this is rarely the case, as we will see, and the liberal techniques search fewer unnecessary databases. In practice, the choice between conservative and liberal depends on the application: how exhaustive the search must be and what resources are available.

For the query translation problem (i.e., the problem of generating queries to retrieve the potential copies from a database), a naive solution is to simply query each database (identified in the discovery phase) for all documents containing any of the terms in the suspicious document. That is, if the suspicious document contains words $w_1, \ldots w_N$, we could submit the query $w_1 \vee \ldots \vee w_N$ (or alternatively, request all the documents with w_1, then all the ones with w_2, and so on). Clearly, any potential copy would be extracted in this way. However, our goal here is to extract all the potential copies without having to perform such a massive query. Thus, to solve this problem we show how to find the minimal query, under two different cost metrics, that can extract all the desired documents. For example, one of our cost measures is the number of words in the submitted query. We will see that we can reduce such a number drastically by bounding the maximum "contribution" of every word to a potential copy.

We start in Section 6.1 by giving an overview of SCAM. In Section 6.2 we describe the data that *dSCAM* keeps about the databases. We use this data in Section 6.3 to define the conservative copy discovery schemes for *dSCAM*, while in Section 6.4 we relax these schemes to make them liberal. In Section 6.5 we present the extraction (query translation) mechanisms. Finally, in Section 6.6 we discuss an experimental evaluation of our techniques, using a collection of 50 databases and two sets of suspicious documents [GMGS96].

6.1 Using SCAM for Copy Detection

Given a suspicious document s and a registered document d, SCAM detects whether d is a potential copy of s by deciding whether they overlap significantly. We have explored [SGM95, SGM96] a variety of overlap measures. For example, we can say that s and d overlap if they contain at least some fraction of common sentences. A problem with this scheme is that it is often hard to detect sentence boundaries (e.g., periods in abbreviations get confused with the end of sentences). Also, it cannot detect partial-sentence overlaps.

A different measure we have studied uses *similarity*, in the information retrieval (IR) sense, as a starting point. Traditionally, two documents are said to be similar if the frequency with which words occur is correlated. If the distribution of word frequencies between s and d is identical, we say that the similarity is maximal at 1. As the distributions differ, the similarity decreases. This measure does not work for copy detection because the matching documents can have portions that are very different causing the word frequency distributions to differ significantly. However, this measure can be modified for copy detection as we will explain in this section. In this chapter we will use this modified IR measure as the basis for *dSCAM* because our experimental results show it works best, at least for the relatively small documents found on the Internet.

To evaluate s and d using this modified IR measure, SCAM first focuses on the words that appear a similar number of times in s and d, and ignores the rest of the words. More precisely, given a fixed $\epsilon > 2$, the *closeness set* for s and d, $c(s,d)$, contains the words w_i with a similar number of occurrences in the two documents [SGM95]:

$$w_i \in c(s,d) \Leftrightarrow \frac{F_i(s)}{F_i(d)} + \frac{F_i(d)}{F_i(s)} < \epsilon$$

where $F_i(d)$ is the frequency of word w_i in document d. If either $F_i(s)$ or $F_i(d)$ are zero, then w_i is not in the closeness set. Given ϵ, s determines a range of frequencies $Accept(w_i, F_i(s))$ such that w_i is in the closeness set for s and d if and only if $F_i(d) \in Accept(w_i, F_i(s))$.

The intuition behind this is as follows. If s and d share a substantial portion of identical text, then there ought to be a set of words unique to that text that will occur with similar frequencies. Focusing on words in the closeness set diminishes the effects of unrelated portions of text.[1]

Example 35: Consider a suspicious document s and a database db with two documents, d_1 and d_2. There are four words in these documents, w_1, w_2, w_3, and w_4. The following table shows the frequency of the words in the documents.

Document	F_1	F_2	F_3	F_4
s	1	3	3	9
d_1	1	3	0	0
d_2	0	8	5	0

For example, w_3 appears three times in s ($F_3(s) = 3$), five times in d_2 ($F_3(d_2) = 5$), and it does not appear in d_1 ($F_3(d_1) = 0$). Assuming $\epsilon = 2.5$ (a value that worked well in the experiments in [SGM95]), $Accept(w_3, F_3(s)) = Accept(w_3, 3) = [2,5]$. Thus, w_3 is in $c(s, d_2)$, the closeness set for s and d_2, because $F_3(d_2) = 5$ is in $Accept(w_3, F_3(s))$. Although $F_3(d_2)$ is higher than $F_3(s)$, these two values are sufficiently close for $\epsilon = 2.5$. In effect, $\frac{F_3(s)}{F_3(d_2)} + \frac{F_3(d_2)}{F_3(s)} = \frac{3}{5} + \frac{5}{3} = 2.27 < \epsilon = 2.5$. For the remaining cases,

[1]It also helps to ignore altogether words that occur frequently across documents [SGM96]. Our experiments of Section 6.6 use the stop words in [SGM96].

$Accept(w_1, F_1(s)) = [1, 1]$, $Accept(w_2, F_2(s)) = [2, 5]$, and $Accept(w_4, F_4(s)) = [5, 17]$. Then, $c(s, d_1) = \{w_1, w_2\}$, and $c(s, d_2) = \{w_3\}$. ▮

After finding the closeness set for s and d, SCAM computes the similarity $sim(s, d)$ between the two documents. We would like to use traditional IR similarity measures (using only words in the closeness set), but this does not work because those measures give low values when s is a subset of d or vice versa. Instead we compute two measures, one for the case where s might be a subset of d and one for the reverse case, and take the maximum. In the former we ignore the norm (see below) of d since it could be a much larger document; in the latter we ignore the norm of s. That is, $sim(s, d) = \max\{subset(s, d), subset(d, s)\}$ where:

$$subset(d_1, d_2) = \sum_{w_i \in c(d_1, d_2)} \frac{F_i(d_1)}{|d_1|} \cdot F_i(d_2)$$

($|d| = \sum_{i=1}^{N} F_i^2(d)$ is the norm of document d and N is the number of terms.) If $sim(s, d) > T$, for some user-specified threshold T, then SCAM flags document d as a potential copy of the suspicious document s.

Example 35: (cont.) Continuing with our example above, $|s| = F_1^2(s) + F_2^2(s) + F_3^2(s) + F_4^2(s) = 1^2 + 3^2 + 3^2 + 9^2 = 100$. Similarly, $|d_1| = 10$ and $|d_2| = 89$. To compute the similarity $sim(s, d_2)$ we just consider w_3, the only word in the closeness set for s and d_2. Then,

$$\begin{aligned} sim(s, d_2) &= \max\{F_3(d_2) \cdot \frac{F_3(s)}{|s|}, \frac{F_3(d_2)}{|d_2|} \cdot F_3(s)\} \\ &= \max\{5 \cdot \frac{3}{100}, \frac{5}{89} \cdot 3\} = 0.17 \end{aligned}$$

Similarly, $sim(s, d_1) = 1$, because SCAM regards d_1 as a strict "subdocument" of s. So, for $T = 0.80$, SCAM would not consider d_2 to be a potential copy of s. However, SCAM would find d_1 suspiciously close to s. ▮

Even though the SCAM similarity does not take into account word sequencing, the experiments in [SGM95, SGM96] show that it detects potential copies relatively well.

In these experiments, conducted with 50,000 netnews articles, false positives were very rare: the similarity measure flagged unrelated documents as copies (because they shared common vocabulary) in only 0.01% of the cases. False negatives were more common but still only 5% of the cases tested: in these cases, documents with relatively small overlap were not detected. Overall, the similarity measure performed better than the sentence overlap measure described earlier.

6.2 The *dSCAM* Information about the Databases

dSCAM needs information to decide whether a database db has potential copies of a suspicious document s. This information should be concise, but also sufficient to identify any such database. *dSCAM* keeps the following statistics (or a subset of them) for each database db and word w_i, where db_i is the set of documents in db that contain w_i:

- $f_i(db) = \min_{d \in db_i} F_i(d)$: $f_i(db)$ is the minimum frequency of word w_i in any document in db that contains w_i
- $F_i(db) = \max_{d \in db_i} F_i(d)$: $F_i(db)$ is the maximum frequency of word w_i in any document in db that contains w_i
- $n_i(db) = \min_{d \in db_i} |d|$: $n_i(db)$ is the minimum norm of any document in db that contains w_i
- $R_i(db) = \max_{d \in db_i} \frac{F_i(d)}{|d|}$: $R_i(db)$ is the maximum value of the ratio $\frac{F_i(d)}{|d|}$ for any document $d \in db$ that contains w_i
- $d_i(db)$ is the number of documents in db that contain word w_i

Example 35: (cont.) The following table shows the *dSCAM* metadata for our sample database db. Note that there are no entries for w_4, since it does not appear in any document in db.

Statistics	w_1	w_2	w_3
f_i	1	3	5
F_i	1	8	5
n_i	10	10	89
R_i	$\frac{1}{10}$	$\frac{3}{10}$	$\frac{5}{89}$
d_i	1	2	1

As mentioned earlier, db has two documents, d_1 and d_2. Document d_1 contains w_2 three times, and document d_2, eight times. Therefore, $f_2(db) = \min\{3, 8\} = 3$ and $F_2(db) = \max\{3, 8\} = 8$. Also, $|d_1| = 10$ and $|d_2| = 89$, so $n_2(db) = \min\{10, 89\} = 10$. Finally, $R_2(db) = \max\{\frac{3}{10}, \frac{8}{89}\} = \frac{3}{10}$, and $d_2(db) = 2$, since w_2 appears in both d_1 and d_2. ∎

Notice that the table above is actually larger than our earlier table that gave the complete word frequencies. This is just because our sample database contains only two documents. In general, the information kept by *dSCAM* is proportional to the number of words or terms appearing in the database, while the information needed by SCAM is proportional to the number of words times the number of times the words appear in different documents. In a real database, many words appear in hundreds or thousands of documents, and hence the SCAM information can be much larger than the *dSCAM* information. We will return to this issue in Section 6.6. To obtain the necessary statistics, *dSCAM* periodically polls each potential source database, which then extracts the data from its index structures.

6.3 The Conservative Approach

Given a set of databases, a suspicious document s, and a threshold T, *dSCAM* selects all databases with potential copies of s, i.e., all the databases with at least one document d with $sim(s, d) > T$. To identify these databases, *dSCAM* uses the metadata of Section 6.2. In this section we focus on conservative techniques that never miss any database with potential copies. In other words, *dSCAM* cannot produce any *false negatives* with the techniques of this section. However, *dSCAM* might produce *false* *positives*, and consider that a database has potential copies when it actually does not. In Section 6.6 we report experimental results that study how often the latter takes place.

The information described in Section 6.2 can be used by *dSCAM* in a variety of ways. We present two alternatives, starting with the simplest. The more sophisticated technique will be less conservative: it will always identify the databases with potential copies of a document, but it will have fewer false positives than the simpler technique.

Given a database db, a suspicious document s, and a technique A, *dSCAM* computes an upper bound $Upper_A(db, s)$ on the similarity of any document in db and s. In other words, $Upper_A(db, s) \geq sim(s, d)$ for every document $d \in db$. Thus, if $Upper_A(db, s) \leq T$, then there are no documents in db close enough to s as determined by the threshold T, and we can safely conclude that database db has no potential copies of s. The two strategies below differ in how they compute this upper bound.

The *Range* Strategy

Consider a word w_i in s. Suppose that w_i appears in some document d in db. We know that d contains w_i between $f_i(db)$ and $F_i(db)$ times. Also, w_i is in the closeness set for s and d if and only if $F_i(d) \in Accept(w_i, F_i(s))$. So, w_i is in the closeness set for s and d if and only if $F_i(d) \in [m_i, M_i] = [f_i(db), F_i(db)] \cap Accept(w_i, F_i(s))$. If this range is empty, then w_i is not in the closeness set for s and d, for any document $d \in db$, and therefore w_i does not contribute to $sim(s, d)$ for any d. If the range $[m_i, M_i]$ is not empty, then w_i can be in the closeness set for s and d, for some document d. For any such document d, $F_i(d) \leq M_i$. We then define the *maximum frequency* of word $w_i \in s$ in any document of db, $M_i(db, s)$, as:

$$M_i(db, s) = \begin{cases} M_i & \text{if } [m_i, M_i] \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Putting everything together, we define the upper bound on the similarity of any document d in db and s for technique *Range* as:

$$Upper_{Range}(db, s) = \max\{Upper1_{Range}(db, s), Upper2_{Range}(db, s)\}$$

where:

$$Upper1_{Range}(db, s) = \sum_{i=1}^{N} M_i(db, s) \cdot \frac{F_i(s)}{|s|} \tag{6.1}$$

$$Upper2_{Range}(db, s) = \sum_{i=1}^{N} \frac{M_i(db, s)}{n_i(db)} \cdot F_i(s) \tag{6.2}$$

Note that since $n_i(db) \leq |d|$ for every $d \in db$ that contains w_i, then:

$$Upper_{Range}(db, s) \geq sim(s, d)$$

for every $d \in db$. Also note that the *Range* technique does not use the R_i statistics.

Example 35: (cont.) Consider the db statistics and the suspicious document s. We have already computed $Accept(w_1, F_1(s)) = [1, 1]$, $Accept(w_2, F_2(s)) = [2, 5]$, $Accept(w_3, F_3(s)) = [2, 5]$, and $Accept(w_4, F_4(s)) = [5, 17]$. Also, *dSCAM* knows, for example, that word w_2 appears in db with in-document frequencies between $[f_2(db), F_2(db)] = [3, 8]$. Then, the interesting range of frequencies of w_2 in db is $[m_2, M_2] = [3, 8] \cap [2, 5] = [3, 5]$. The maximum such frequency is $M_2(db, s) = 5$. (Notice that there is no document d in db with $F_2(d) = 5$. $M_2(db, s)$ is in this case a strict upper bound for the frequencies of w_2 in db that are in $Accept(w_2, F_2(s))$.) Similarly, $M_1(db, s) = 1$, $M_3(db, s) = 5$, and $M_4(db, s) = 0$. Therefore,

$$\begin{aligned} Upper1_{Range}(db, s) &= 1 \cdot \frac{1}{100} + 5 \cdot \frac{3}{100} + 5 \cdot \frac{3}{100} \\ &= 0.31 \\ Upper2_{Range}(db, s) &= \frac{1}{10} \cdot 1 + \frac{5}{10} \cdot 3 + \frac{5}{89} \cdot 3 \\ &= 1.77 \\ Upper_{Range}(db, s) &= 1.77 \end{aligned}$$

Therefore, if our threshold T is, say, 0.80, we would search db. This is of course the right decision since d_1 in db is indeed a potential copy. ∎

The *Ratio* Strategy

This technique is similar to the previous one, but uses the R_i statistics. Thus,

$$Upper_{Ratio}(db, s) \quad = \quad \max\{Upper1_{Range}(db, s), Upper2_{Ratio}(db, s)\}$$

where:

$$Upper2_{Ratio}(db, s) \quad = \quad \sum_{i|M_i(db,s)\neq 0} \min\{\frac{M_i(db, s)}{n_i(db)}, R_i(db)\} \cdot F_i(s) \qquad (6.3)$$

It is immediate from the definition above that $Upper_{Ratio}(db, s) \leq Upper_{Range}(db, s)$ for every database db and query document s. Therefore, *Ratio* is a less conservative technique than *Range*, and will tend to have fewer false positives than *Range*. Nevertheless, *Ratio* will always detect databases with potential copies of s, because $sim(s, d) \leq Upper_{Ratio}(db, s)$ for every $d \in db$.

Example 35: (cont.) We have already computed $Upper1_{Range}(db, s) = 0.31$. Now,

$$\begin{aligned} Upper2_{Ratio}(db, s) &= \frac{1}{10} \cdot 1 + \frac{3}{10} \cdot 3 + \frac{5}{89} \cdot 3 \\ &= 1.17 \end{aligned}$$

which is lower than $Upper2_{Range}(db, s)$. ∎

6.4 The Liberal Approach

The techniques of Section 6.3 are conservative: they never fail to identify a database with potential copies of a suspicious document (i.e., these techniques have no false negatives). A problem with these techniques is that they usually produce too many false positives. (See Section 6.6.) Consequently, we now introduce *liberal* versions of the *Range* and *Ratio* techniques. In principle, the new techniques might have false negatives. As we will see, false negatives occur rarely, while the number of false positives is much lower than that for the conservative techniques.

We modify the techniques of Section 6.3 in two different ways. First, we allow these techniques to focus only on the "rarest" words that occur in a suspicious document, instead of on all its words (or on all the words that SCAM uses). (See Section 6.4.1.) This way *dSCAM* can prune away databases where these rare words do not appear, thus reducing the search space. Second, we allow these techniques to use probabilities to estimate (under some assumptions) how many potential copies of a suspicious document each database is expected to have. (See Section 6.4.2.) Thus, the probabilistic techniques no longer compute upper bounds, again reducing the search space.

6.4.1 Counting Only Rare Words

The techniques of Section 6.3 considered every word in a suspicious document s (i.e., every word that SCAM uses) to decide which databases to search for potential copies of s. Alternatively, *dSCAM* can just focus on the *rarest* words in s, i.e., on the words in s that appear in the fewest number of databases. *dSCAM* then decides to search a database only if at least a few of these rare words appear in it. If *dSCAM* uses enough of the rare words in s, any potential copy of s will tend to contain a few of these words. Furthermore, since these words appear in only a few databases, they will help *dSCAM* dismiss a significant fraction of the databases, thus reducing the number of false positives.

One specific way to implement these ideas is as follows. Given a suspicious document s, *dSCAM* just considers k percent of its words. These are the $k\%$ words in s that appear in the fewest available databases. *dSCAM* can tell which words these are from the metadata about the databases (Section 6.2). The remaining words in s are simply ignored.

Example 35: (cont.) Consider suspicious document s, with words w_1, w_2, w_3, and w_4. Suppose that w_1 appears in 1 database, w_2 in 2, w_3 in 70, and w_4 in 20 databases. If *dSCAM* uses only 50% of the words in s ($k = 50$), it chooses w_1 and w_2, and ignores

w_3 and w_4. ▮

As we mentioned before, *dSCAM* now ignores words in s that SCAM uses for copy detection. Therefore, *dSCAM* might in principle miss a database with potential copies of s. However, as we will see in Section 6.6, we can find values for k for which *dSCAM* has very few false negatives, while producing much fewer false positives than with the conservative techniques of Section 6.3.

Given k, we adapt the $Upper_{Range}$ and $Upper_{Ratio}$ bounds of Section 6.3 (Equations 6.1, 6.2, and 6.3) to sum only over the k% rarest words in s. We refer to the new values as Sum_{Range} and Sum_{Ratio}, because they are no longer upper bounds on the similarities of the documents in the databases and s.

As we use fewer words in s (i.e., only k% of them), we need to adjust the threshold T (Section 6.1) for *dSCAM* accordingly. We refer to the adjusted threshold as T^k. For example, if we are just considering 10% of the words in s, we could compensate by reducing the threshold $T^{10} = 0.10 \cdot T$. We explore different values for T^k in Section 6.6. If $Sum_{Range}(db, s)$ (respectively, $Sum_{Ratio}(db, s)$) is higher than T^k, *dSCAM* will search db for potential copies of s.

Example 35: (cont.) In Section 6.3 we computed $Upper_{Range}(db, s) = 1.77$. Now, if *dSCAM* only considers the 50% rarest words in s (i.e., w_1 and w_2), only those words are counted, and we have:

$$\begin{aligned} Sum1_{Range}(db, s) &= 1 \cdot \frac{1}{100} + 5 \cdot \frac{3}{100} = 0.16 \\ Sum2_{Range}(db, s) &= \frac{1}{10} \cdot 1 + \frac{5}{10} \cdot 3 = 1.6 \\ Sum_{Range}(db, s) &= 1.6 \end{aligned}$$

The original SCAM threshold was $T = 0.80$. Since we are now considering only half of the words, we could scale down T to, say, $T^{50} = 0.5 \cdot T = 0.40$. At any rate, we would still search db, because $1.6 > 0.4$. This is the right decision, since d_1 in db is indeed a potential copy. ▮

6.4.2 Using Probabilities

So far, the techniques for *dSCAM* compute the maximum possible contribution of each word considered, and add these contributions. However, it is unlikely that any document in a database will contain all of these words with this maximum contribution. In this section, we depart from this "deterministic" model, and, given a database db, try to bound the probability that db has potential copies of a suspicious document. If this probability is high enough, *dSCAM* will search db.

Our goal is to bound the probability that a document in db has a similarity with s that exceeds the adjusted threshold T^k. For this, we define two random variables $XRange1$ and $XRange2$ (corresponding to $Sum1_{Range}$ and $Sum2_{Range}$, respectively). These variables model the similarity of the documents in db and s. Then,

$$Prob_{Range} = \max\{P(XRange1 > T^k), P(XRange2 > T^k)\}$$

If $Prob_{Range} \geq \frac{1}{|db|}$, *dSCAM* will search db for potential copies of s, since there is at least one expected document that exceeds the adjusted threshold T^k.

Actually, instead of computing $P(XRange1 > T^k)$ and $P(XRange2 > T^k)$, we use an upper bound for these values as given by Chebyshev's inequality. This bound is based on the expected value and the variance of $XRange1$ and $XRange2$.

We now define random variable $XRange1$, following the definition of $Sum1_{Range}$. (Random variable $XRange2$ is analogous, using the definition of $Sum2_{Range}$.) The $XRange1$ is actually a sum of random variables: $XRange1 = XRange1_{i_1} + \ldots + XRange1_{i_s}$ where $w_{i_1}, \ldots, w_{i_s}$ are the $k\%$ rarest words in s. Random variable $XRange1_i$ corresponds to word w_i:

$$XRange1_i = \begin{cases} M_i(db, s) \cdot \frac{F_i(s)}{|s|} & \text{with probability } \frac{d_i(db)}{|db|} \\ 0 & \text{with probability } 1 \Leftrightarrow \frac{d_i(db)}{|db|} \end{cases}$$

This variable models the occurrence of word w_i in the documents of database db. Word w_i occurs in $d_i(db)$ documents in db, so the probability that it appears in a randomly chosen document from db is $\frac{d_i(db)}{|db|}$. To use Chebyshev's inequality and compute the variance of $XRange1$ and $XRange2$, we assume that words appear in documents following independent probability distributions. We define $Prob_{Ratio}$ in a completely analogous way.

6.5 Searching the Databases with Potential Copies

Once *dSCAM* has decided that a database db might have potential copies of a suspicious document s, it has to extract these potential copies from db. If database db happens to run a local SCAM server, *dSCAM* can simply submit s to this server and get back exactly those documents that SCAM considers potential copies. However, if db does not run a SCAM server, we need an alternative mechanism to extract the potential copies automatically. For this, we will assume that db can answer Boolean "or" queries, which most commercial search engines support. For example, we can retrieve from db all documents containing the words "copyright" or the word "SCAM" by issuing the query $copyright \vee SCAM$. (Alternatively, if some search engine does not support "or" queries, we could issue a sequence of queries, and then merge the sequence of results.)

Let $w_1, \ldots, w_N$ be the words in s. In principle, we could issue the query $w_1 \vee \ldots \vee w_N$ to db and obtain all documents that contain at least one of these words. However, such a query is bound to return too many documents that are not potential copies of s. In this section, we study how to choose a smaller set of words $\{w_{i_1}, \ldots, w_{i_n}\}$ that will not miss any potential copy from db. Furthermore, the resulting queries will tend not to extract documents that are not potential copies of s.

To choose a set of words to query, we define the *maximum contribution* $C_i(db, s)$ of word w_i in db as an upper bound on the amount that w_i can add to $sim(s, d)$, for any $d \in db$. We give two definitions of this maximum contribution, each corresponding to a technique of Section 6.3. The first of these is more conservative but uses less information. The other is less conservative but uses more information.

$$C_i(db,s) \;=\; \begin{cases} \max\{M_i(db,s)\cdot \frac{F_i(s)}{|s|},\;\; \frac{M_i(db,s)}{n_i(db)}\cdot F_i(s)\} \text{ for } \textit{Range} \\ \max\{M_i(db,s)\cdot \frac{F_i(s)}{|s|},\;\; \min\{\frac{M_i(db,s)}{n_i(db)}, R_i(db)\}\cdot F_i(s)\} \text{ for } \textit{Ratio} \end{cases}$$

Now, let $C(db,s) = \sum_{i=1}^{N} C_i(db,s)$, and let T be the SCAM similarity threshold that the users specified. Then, any set of words $\{w_{i_1},\ldots,w_{i_n}\}$ with the following property is sufficient to extract all the potential copies of s from db:

$$\sum_{j=1}^{n} C_{i_j}(db,s) \geq C(db,s) \Leftrightarrow T \tag{6.4}$$

To see why it is enough to use the query $w_{i_1} \vee \ldots \vee w_{i_n}$, consider a document $d \in db$ that does not contain any of these n words. Then, $sim(s,d) \leq C(db,s) \Leftrightarrow \sum_{j=1}^{n} C_{i_j}(db,s) \leq T$. Therefore, the similarity of d and s can never exceed the required threshold T. This approach is conservative: we cannot miss any potential copy of a document by choosing the query words as above. Alternatively, we explored a liberal approach that would retrieve all potential copies most of the time, and has much fewer "false positives." We do not describe this liberal technique further, but we report some experimental results in Section 6.6.

To choose among all sets of words that satisfy Condition 6.4, we associate a cost p_i with each word w_i. We then choose a set of words $\{w_{i_1},\ldots,w_{i_n}\}$ that satisfies Condition 6.4 and minimizes $\sum_{j=1}^{n} p_{i_j}$. We consider two different cost models for a query:

The *WordMin* Cost Model

In this case we minimize the number of words that will appear in the query. Thus, $p_i = 1$ for all i. Then, our problem reduces to finding the smallest set of words that satisfies Condition 6.4, which we can do optimally with a simple greedy algorithm.

The *SelMin* Cost Model

In this case we consider the selectivity of each word w_i that will appear in the query, i.e., the fraction of the documents in the database that contain word w_i. Thus, $p_i = Sel(w_i, db)$. By minimizing the added selectivity we will tend to minimize the number of documents that we retrieve from db.

We will find an optimal solution for this problem by reducing it to the *0-1 knapsack problem* [CLR91]. The new formulation of the problem is as follows. A thief robbing a store finds N items (the words). The ith item is worth p_i dollars (the selectivity of word w_i) and weighs $C_i(db, s)$ pounds (the maximum contribution of w_i). The thief wants to maximize the value of the load, but can only carry up to T pounds. The problem is to find the right items (words) to steal. This formulation of the problem actually finds the words that will not appear in the final query, and maximizes the added selectivity of these words. The weight of the words is at most T. Therefore, the words that are not chosen weigh at least $C(db, s) \Leftrightarrow T$, satisfy Condition 6.4, and have the lowest added selectivity among the sets satisfying Condition 6.4. Assuming that T, the C_i's, and the p_i's have a fixed number of significant decimals, we can use dynamic programming to solve the problem in $O(T \cdot N)$ time, where N is the number of words in the suspicious document [CLR91].

6.6 Experiments

This section presents experimental results for *dSCAM*. We focus on three sets of issues: How many false positives do the *dSCAM* techniques report, how many false negatives do the liberal *dSCAM* techniques produce, and how effective is the document extraction step?

For the registered-document databases, our experiments used a total of 63,350 ClariNet news articles. We split these articles evenly in 50 databases so that each database consists of 1,267 documents.

For the suspicious documents, our experiments used two different document sets. The first set, which we refer to as *Registered*, contains 100 documents from the 50

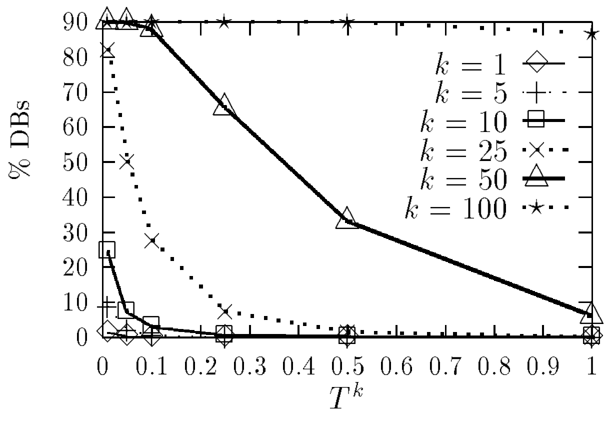

Figure 6.1: The percentage of the 50 databases that are searched as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

databases. Therefore, each suspicious document has at least one perfect copy in some database. (There could be more copies due to crosspostings of articles.) The second set, which we refer to as *Disjoint*, contains 100 later articles that do not appear in any of the 50 databases. This set models the common case when the suspicious documents are actually new documents that do not appear anywhere else.

Our first experiments are for the Sum_{Ratio} technique, which proved to work the best among the *dSCAM* techniques, as we will see later. Figures 6.1 through 6.4 show different interesting metrics as a function of the adjusted threshold T^k, and for different values of k. In all of these plots, the SCAM threshold T is set to 1. For example, the curves for $k = 10$ correspond to considering only 10% of the words (the rarest ones) in the suspicious documents. Note that for $k = 100$ all of the words in the suspicious documents are used. In this case, Sum_{Ratio} coincides with the conservative technique $Upper_{Ratio}$.

One way to evaluate the *dSCAM* strategies is to look at $d(s)$, the percentage of databases returned by *dSCAM* for a suspicious document s. Figure 6.1 shows the average $d(s)$ (averaged over all s in the *Registered* set), as a function of T^k. The more words *dSCAM* considers from the suspicious documents (i.e., the higher k), the more databases are searched: *dSCAM* considers the words as ordered by how rare they are. Therefore, when *dSCAM* starts considering "popular" words, more databases

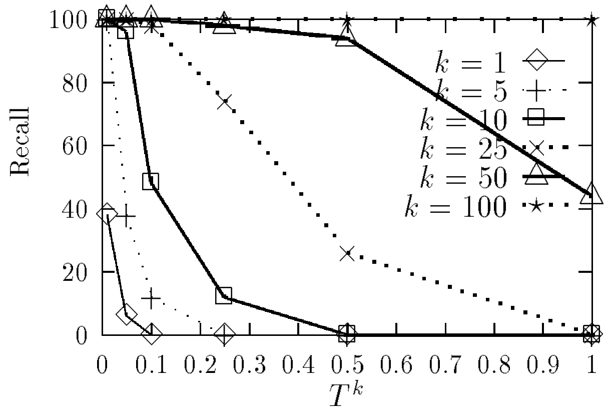

Figure 6.2: The average recall as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

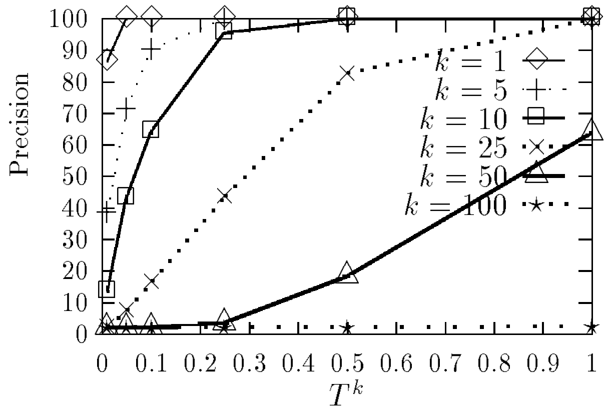

Figure 6.3: The average precision as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

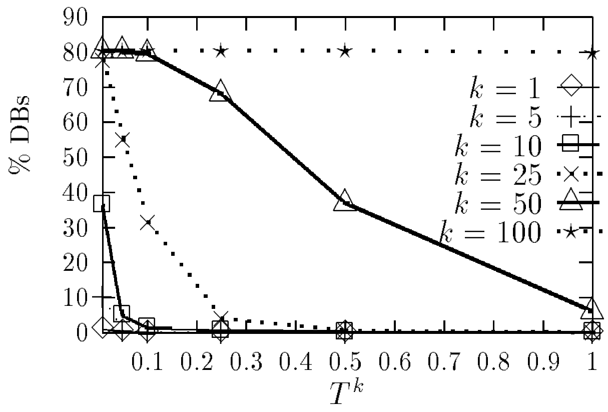

Figure 6.4: The percentage of the 50 databases that are searched as a function of the adjusted similarity threshold T^k (*Disjoint* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

will tend to exceed the similarity threshold T^k. Also, for a fixed k, the higher T^k, the fewer databases that *dSCAM* searches, since only databases that exceed T^k are searched. For low values of k, *dSCAM* searches very few databases. For example, for $k = 10$ and $T^k = 0.05$, less than 10% of the databases are searched.

As we know, Sum_{Ratio} may produce false negatives for $k < 100$, i.e., it may tell us not to search databases where SCAM would find potential copies. It is interesting to study what percentage of the databases with potential copies *dSCAM* actually searches (or equivalently, what percentage of these databases are not false negatives). Let $Right(s, DB)$ be the set of databases in DB with potential copies of s according to SCAM, and let $Chosen(s, DB)$ be the set of databases that *dSCAM* searches. Then, the *recall* of the technique used by *dSCAM* is the average value of:

$$\frac{100 \cdot |Chosen(s, DB) \cap Right(s, DB)|}{|Right(s, DB)|}$$

over our suspicious documents s, as in Section 3.3.

Figure 6.2 shows the recall values for Sum_{Ratio} as a function of the adjusted threshold T^k. This figure is very similar to Figure 6.1: the more databases a technique searches, the higher its recall tends to be. Note, however, that some techniques have very few false negatives, while they search a low percentage of the databases. For example, for $k = 10$ and $T^k = 0.05$, recall is above 90%, meaning that for the average suspicious document, 90% of the databases with potential copies are chosen by *dSCAM*. As we have seen, just under 10% of the databases are searched for this value of k and T^k.

As we mentioned above, *dSCAM* produces false positives. We want to measure what percentage of the databases selected by *dSCAM* actually contains potential copies. The *precision* of the technique used by *dSCAM* is the average value of:

$$\frac{100 \cdot |Chosen(s, DB) \cap Right(s, DB)|}{|Chosen(s, DB)|}$$

over our suspicious documents s. Figure 6.3 shows the precision values for Sum_{Ratio} as a function of the adjusted threshold T^k. As expected, the more databases a technique searches, the lower its precision tends to be. For $k = 10$ and $T^k = 0.05$, precision is over 40%, meaning that for the average suspicious document, over 40% of the databases that *dSCAM* searches have potential copies of the document. Actually, this choice of values for k and T^k is a good one: *dSCAM* searches very few databases while achieving high precision and recall values.

We are evaluating *dSCAM* in terms of how well it predicts the behavior of SCAM at each database. However, SCAM can sometimes be wrong. For example, SCAM can wrongly flag a document d in db as a copy of a suspicious document s. *dSCAM* might then also flag db as having potential copies of s. However, we do not "penalize" *dSCAM* for this "wrong" choice: the best *dSCAM* can do is to predict the behavior of SCAM, and that is why we define precision and recall as above. It would be unreasonable to ask a system like *dSCAM*, with very limited information about the databases, to detect copies more accurately than a system like SCAM, which has complete information about the database contents. (See Section 3.4.3.)

To illustrate the storage space differences between *dSCAM* and SCAM, let us consider the data that we used in our experiments. In this case, there are around 4 million word-document pairs, which is the level of information that a SCAM server needs, whereas there are only around 791,000 word-database pairs, which is the level of information that a *dSCAM* server needs. As the databases grow in size, we expect this difference to widen too, since the *dSCAM* savings in storage come from words appearing in multiple documents. For example, if we consider our 50 databases as a single, big database, *dSCAM* needs only 138,086 word-database pairs, whereas the SCAM data remains the same. Therefore, *dSCAM* has just around 3.36% as many entries as SCAM. We are considering alternatives to reduce the size of the *dSCAM* data even further. As an interesting direction for future work, *dSCAM* can store information on, say, only the 10% rarest words. Most of the time, the 10% rarest words that appear in a suspicious document will be among these 10% overall rarest words, so *dSCAM* can proceed as usual. With this scheme, the *dSCAM* space requirements would be cut further by an order of magnitude.

Figure 6.4 shows results for the *Disjoint* set of suspicious documents, again for the Sum_{Ratio} technique and $T = 1$. There are no potential copies of these documents in any of the 50 databases. Therefore, recall is always 100%, and precision is 0% if some database is selected. It then suffices to report the percentage of databases chosen for these documents (Figure 6.4). These values tend to be lower in general than those for the *Registered* suspicious documents of Figure 6.1, which is the right trend, since no database contains potential copies of the suspicious documents. For example, for $k = 10$ and $T^k = 0.05$, less than 5% of the databases are searched.

So far we have presented results just for the Sum_{Ratio} technique. Figures 6.5 through 6.7 show results also for Sum_{Range}, $Prob_{Range}$, and $Prob_{Ratio}$, as a function of the SCAM threshold T. In all of these plots, we have fixed $k = 10$ and $T^k = 0.05 \cdot T$, which worked well for both the *Registered* and the *Disjoint* suspicious documents when $T = 1$. In Figure 6.5, $Prob_{Range}$ and $Prob_{Ratio}$ search fewer databases than Sum_{Range} and Sum_{Ratio}, at the expense of significantly lower recall values (Figure 6.6). Sum_{Range} and Sum_{Ratio} have very high recall values (above 95% for all values of T). Precision is also relatively high, especially for the Sum_{Ratio} strategy (Figure 6.7). From all these plots, Sum_{Ratio} appears as the best choice for *dSCAM*, because of its high recall and precision, and low percentage of databases that it searches. Also, note that Sum_{Ratio} does not need the d_i statistics, resulting in lower storage requirements than those of $Prob_{Ratio}$, for example. However, if we want to be conservative, and be sure that we do not miss any potential copy of a document, then the best choice is also Sum_{Ratio}, but

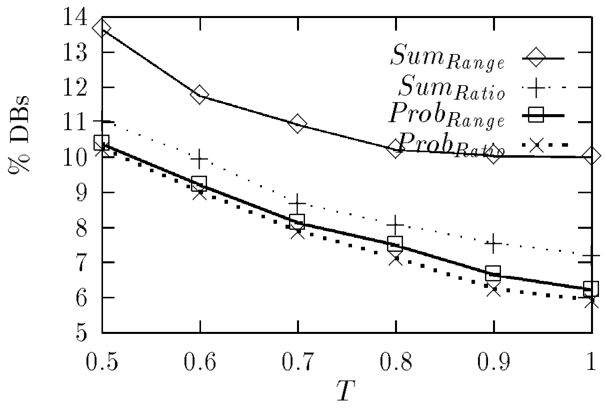

Figure 6.5: The percentage of the 50 databases that are searched as a function of the SCAM threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

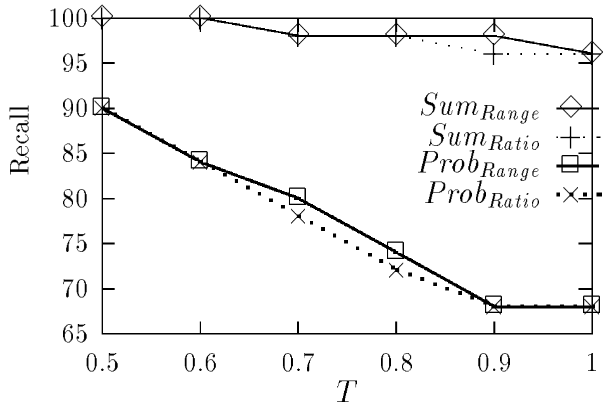

Figure 6.6: The average recall as a function of the SCAM threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

with $k = 100$ and $T^k = T$. (This technique coincides with the conservative $Upper_{Ratio}$ technique of Section 6.3.)

To determine whether the results above will still hold for larger databases, we performed the following experiment. Initially we have a single database with 1,267 documents (one of the databases that we used in this section). *dSCAM* decides whether this database should be searched or not for each of the *Disjoint* suspicious documents, with $T = 1$, the Sum_{Ratio} strategy, $k = 10$, and $T^k = 0.05$. The answer should be "no" for each of these documents, of course. Figure 6.8 shows that *dSCAM* decides to search this database for less than 10% of the tested documents. This

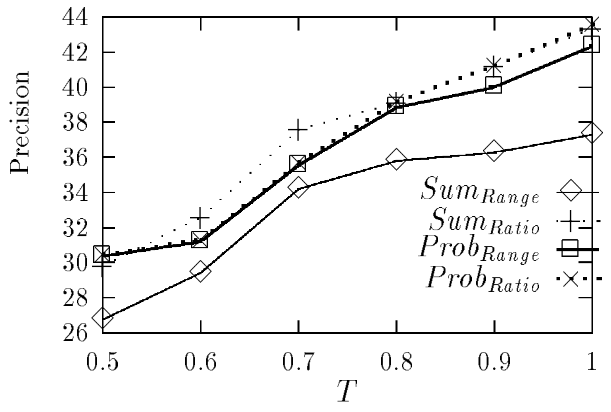

Figure 6.7: The average precision as a function of the SCAM threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

corresponds to a 0.10 probability of false positives. Then, we keep enlarging our only database by progressively adding the documents from our original databases, until the database consists of all 63,350 documents. As we see from Figure 6.8, after an initial deterioration, *dSCAM* stabilizes and chooses to search the database around 25% of the time. These important results show that *dSCAM* scales relatively well to larger databases. That is, the probability of false positives is relatively insensitive (after an initial rise) to database size. Notice, incidentally, that the 25% false-positive probability can be made smaller by changing the T^k and k values (at a cost in false negatives). So the key observation from this figure is simply that the value is flat as the database size grows.

Our final set of experiments is for the results of Section 6.5. In that section we studied how to choose the query for each database that *dSCAM* selects. These queries retrieve all potential copies of the suspicious documents. There are many such queries, though. We presented two cost models, and showed algorithms to pick the cheapest query for each model.

Under our first cost model, *WordMin*, we minimize the number of words in the queries that we construct. Thus, we choose a minimum set of words for our query from the given suspicious document. Figure 6.9 shows the percentage of words in the suspicious document that are chosen to query the databases, for the *Registered* documents and for different values of T. The number of words in the queries decreases

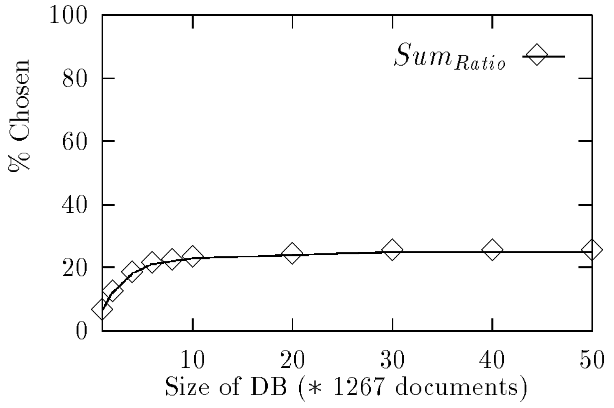

Figure 6.8: The average number of times that *dSCAM* (incorrectly) chooses to search the (growing) database, as a function of the size of the database (*Disjoint* suspicious documents; Sum_{Ratio} strategy).

as T increases. In effect, Condition 6.4 in Section 6.5 becomes easier to satisfy for larger values of T. For example, for $T = 0.80$ and *Ratio*, we need on average 9.99% of the suspicious-document words for our queries. If a particular database cannot handle so many words in a query, we should partition the query into smaller subqueries, and take the union of its results. As expected, the number of words chosen using the *SelMin* cost model is higher, because this cost model focuses on the selectivity of the words, and not on the number of words chosen.

While our second cost model, *SelMin*, uses the word selectivities, the *WordMin* cost model ignores these selectivities. Therefore, we analyze the selectivity for the queries to know what fraction of each database we will retrieve with such queries. Figure 6.10 shows the average value of this selectivity for the *Registered* suspicious documents.

The number of query words and the added selectivity of the query words are relatively high. However, if all a database has is a Boolean-query interface, we have no choice but to ask the right queries to the database to extract all the potential copies of a suspicious document. The results above show that we can do substantially better than the brute-force approach (i.e., when we use all the words in the suspicious document to build a big "or" query) by writing the queries as in Section 6.5.

We have also explored liberal techniques to extract the potential copies from a

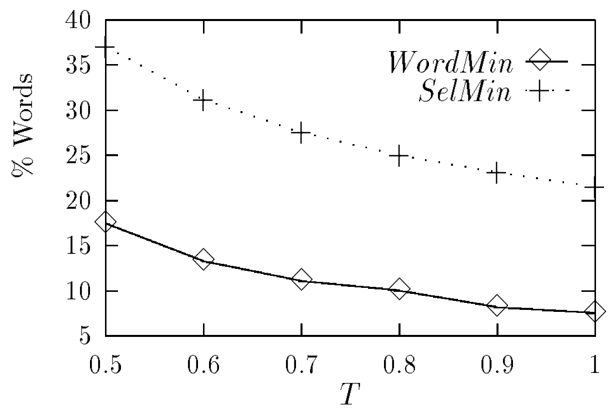

Figure 6.9: The percentage of words of the suspicious documents that are included in the query to extract the potential copies from the databases (*Registered* suspicious documents; *Ratio* strategy).

database. These liberal techniques might have false negatives (i.e., they might miss some potential copies) and they have much fewer false positives (i.e., they retrieve fewer documents that are not potential copies). Although we do not describe these techniques here, we report some numbers for $T = 1$ to give an idea of the promising results that we obtained. For example, we queried the databases using only the 10% rarest words in the suspicious documents. These queries had an average selectivity of 0.49% (i.e., these "or" queries retrieved on average less than 1% of the database documents), and an average recall of 94% (i.e., these queries retrieved on average 94% of the potential copies). In contrast, the *WordMin* queries for $T = 1$ have fewer words on average (around 8% of the words), but their selectivity is much higher (around 16%). The *SelMin* queries for $T = 1$ have over 20% of the document words in them, and their average selectivity is still higher than that of the liberal technique (over 5%). Of course, recall is perfect for *WordMin* and *SelMin*, while this is not the case for the liberal techniques.

6.7 Conclusion

Discovering a potential copy that might exist in one of many databases is a fundamentally difficult problem. One might say that it is harder than finding a "needle in

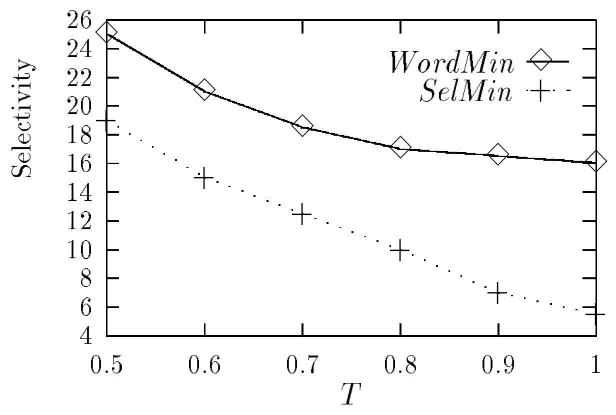

Figure 6.10: Average selectivity of the queries used to extract the potential copies from the databases, as a function of the SCAM threshold T (*Registered* suspicious documents; *Ratio* strategy).

a haystack:" the haystack is distributed across the Internet, we do not want *similar* items (e.g., a nail), and we also want to find any *piece* of the needle if it exists. It is a harder problem than simply finding similar items, as in traditional information retrieval. Given this difficulty it is somewhat surprising that *dSCAM* performs as well as we have found, especially when one considers the relatively small amount of index information it maintains. It is true that *dSCAM* can miss some potential copies or can lead us to sites without copies, but with the right algorithm and parameter settings, these errors can be made tolerable. For example, we found that *dSCAM* can miss fewer than 5% of the sites with potential copies, and for the sites it does lead us to, they actually have a potential copy roughly half the time.

dSCAM performs best when it only considers about 10% of the words in the suspicious document, those that are the "rarest." Intuitively, these rare words act as a "telltale signature" that makes it easier to pick out the target databases. We believe that this is the main reason that *dSCAM* performs better than one would expect, given the difficulty of the problem at hand. Some pirates may make it harder for *dSCAM* to detect these signatures by changing these rare words, but this is not a significant problem since our goal is to prevent widespread and direct copying of documents.

We believe that copy discovery will be an important service in distributed information systems. It will not prevent people from making illegal copies, but having effective discovery mechanisms (together with copy tracing schemes) may dissuade people from large scale duplication.

Chapter 7

Related Work

This chapter reviews the literature that is relevant to this thesis. Section 7.1 starts by describing how existing metasearchers address the key metasearching issues that we identified in Chapter 1. Then, Section 7.2 reviews protocols relevant to our *STARTS* work of Chapter 2. Section 7.3 discusses different approaches to solving the text-source discovery problem of Chapters 3 and 4. Section 7.4 focuses on work on the result merging problem of Chapter 5, most notably on relevant work from the information retrieval field. Finally, Section 7.5 gives an overview of work related to the distributed copy detection problem of Chapter 6.

7.1 Metasearchers

Several metasearchers already exist on the Internet for querying multiple World-Wide Web indexes. However, not all of them support the three major metasearch tasks described in Chapter 1 (i.e., selecting the best sources for a query, querying these sources, and merging the query results from the sources). Examples of metasearchers include MetaCrawler [SE95] (`http://www.metacrawler.com`), SavvySearch (`http://guaraldi.cs.colostate.edu:2000/`), and ProFusion [GW96]. Also, the Stanford InfoBus, designed within the Digital Library project [PCGM$^+$96, RBC$^+$97], hosts a variety of metasearchers. In [BCGP97a, BCGP97b], we discuss a metadata architecture for the InfoBus. This architecture is based on the requirements of the

InfoBus services, and uses the *STARTS* information that sources should export.

MetaCrawler, SavvySearch, and Profusion support the three metasearch tasks above to some degree. First, they provide some sort of source selection. For example, SavvySearch ranks its accessible sources for a given query based on information from past searches and estimated network traffic. Second, they support a unified query interface for accessing the underlying sources, although this interface tends to be the least common denominator of that of the underlying sources. For query features that are not supported uniformly by the underlying sources, a post-filtering step is required for the metasearcher to locally implement the missing functionality. For instance, MetaCrawler processes phrase searches in the "verification mode." Third, these meta-searchers re-rank the documents in the query results. Specifically, MetaCrawler re-ranks the documents by actually retrieving and analyzing them. SavvySearch simply reports the documents according to the originating sources, and using the source rank mentioned above. Profusion re-ranks each documents by scaling its score with the *confidence factor* for the source where the document originates. These confidence factors measure how useful each source was for a set of 25 sample queries.

7.2 Protocols

The most relevant standards effort in terms of shared goals is the Z39.50-1995 standard [Org95], which provides most of the functionality we have described for *STARTS*. For instance, its Explain facility requires the Z39.50 servers to export their "source metadata" so that the clients can dynamically configure themselves to match individual servers, thus providing the option to support more than the least common denominator of the servers. The standard also specifies query languages such as the type-101 query language, which we used in Section 2.3.1. In addition, the Scan service allows the clients to access the sources' contents incrementally.

While similar in functionality, our *STARTS* proposal is much simpler than Z39.50, and keeping it simple was one of our main concerns. Moreover, as our proposal is specifically tailored to facilitate metasearching, we require some information not exported in Z39.50. For example, we need the term and document statistics as part of the query results to help in merging multiple document ranks. However, we do see our proposal as a step toward bridging the gap between the library community where Z39.50 has been widely used and the Internet search community. As we mentioned in Chapter 2, there are currently efforts under way to define a simple profile of the Z39.50 standard based on *STARTS* [Z3997].

In addition to the Z39.50 standard effort, other projects focus on providing a framework for indexing and querying multiple document sources. One such project, Harvest [BDH$^+$94], includes a set of tools for gathering and accessing information on the Internet. The Harvest *gatherers* collect and extract indexing information from one or more sources. Then, the *brokers* retrieve this information from one or more gatherers, or from other brokers. The brokers provide a querying interface to the gathered information. A project related to Harvest is Netscape's RDM (Resource Description Messages) [Har96], which focuses on indexing and accessing structured metadata descriptions of information objects. Our work complements Harvest and RDM in that we define the information and functionality that sources should export to help in metasearching. Thus, the Harvest brokers (or RDM clients) could act as metasearchers, and benefit from the *STARTS* information that sources export.

Other related efforts focus on defining attribute sets for documents and sources. As discussed in Chapter 2, we have built on some of these efforts in defining our protocol. Relevant attribute sets for documents include the Z39.50 Bib-1 attribute set [Age95], the Dublin Core [WGMJ95], and the Warwick Framework [LLJ96]. The Bib-1 attribute set registers a large set of bibliographic attributes that have been widely adopted in library cataloging. On the other hand, the focus of the Dublin Core is primarily on developing a simple yet usable set of attributes to describe the essential features of networked documents (e.g., World-Wide Web documents), which is also the intention of our "Basic-1" set. The Warwick Framework proposes a container architecture as a mechanism for incorporating attribute values from different sets in a single information object. In contrast, we chose to support only a simple, "flat" document model, albeit with the ability to mix different attribute models [GCGMP96]. Regarding source-metadata attribute sets, the most notable efforts include the Z39.50 Exp-1 attribute set [Org95] and the GILS profile [Chr97], upon which we based our

"MBasic-1" attribute set (Section 2.3.3).

7.3 Text-Source Discovery

Many solutions have been presented recently for the text-source discovery problem, or, more generally, for the resource-discovery problem: the text-source discovery problem is a subcase of the resource-discovery problem, since the latter generally deals with a larger variety of types of information [ODL93, SEKN92].

One solution to the text-source discovery problem is to let the database selection be driven by the user. Thus, the user will be aware of and an active participant in this selection process. Different systems follow different approaches to this: one such approach is to let users "browse" through information about the different resources. A typical example of this paradigm is Yahoo! (`http://www.yahoo.com`). As another example, the Prospero File System [Neu92] lets users organize information available in the Internet through the definition (and sharing) of customized views of the different objects and services available to them.

A different approach is to keep a database of "meta-information" about the available databases and have users query this database to obtain the set of databases to search. For example, WAIS [KM91] provides a "directory of servers." This "master" database contains a set of documents, each describing (in English) the contents of a database on the network. The users first query the master database, and once they have identified potential databases, direct their query to these databases. One disadvantage is that the master-database documents have to be written by hand to cover the relevant topics, and have to be manually kept up to date as the underlying database changes. However, freeWAIS [FW$^+$93] automatically adds the most frequently occurring words in an information server to the associated description in the directory of servers. Another drawback is that in general, databases containing relevant documents might be missed if they are not chosen during the database-selection phase. [DS94] shows sample queries for which very few of the existing relevant servers are found by querying the WAIS directory of servers (e.g., only 6 out of 223 relevant WAIS servers).

Reference [Sch90] follows a probabilistic approach to the resource-discovery problem, and presents a resource-discovery protocol that consists of two phases: a dissemination phase, during which information about the contents of the databases is replicated at randomly chosen sites, and a search phase, where several randomly chosen sites are searched in parallel. Also, sites are organized into "specialization subgraphs." If one node of such a graph is reached during the search process, the search proceeds "non-randomly" in this subgraph, if it corresponds to a specialization relevant to the query being executed. See also [Sch93].

In Indie (shorthand for "Distributed Indexing") [DLO92], information is indexed by "Indie brokers," each of which has associated, among other administrative data, a Boolean query (called a "generator rule"). Each broker indexes (not necessarily local) documents that satisfy its generator rule. Whenever a document is added to an information source, the brokers whose generator rules match the new document are sent a descriptor of the new document. The generator objects associated with the brokers are gathered by a "directory of servers," which is queried initially by the users to obtain a list of the brokers whose generator rules match the given query. See also [DANO91]. [BC92], [OM92], and [SA89] are other examples of this type of approach in which users query "meta-information" databases.

A "content-based routing" system is used in [SDW$^+$94] to address the resource-discovery problem. The "content routing system" keeps a "content label" for each information server (or collection of objects, more generally), with attributes describing the contents of the collection. Users assign values to the content-label attributes in their queries until a sufficiently small set of information servers is selected. Also, users can browse the possible values of each content-label attribute.

The WHOIS++ directory service (`http://www.ucdavis.edu/whoisplus`) organizes the WHOIS++ servers into a distributed "directory mesh" that can be searched: each server automatically generates a "centroid" listing the words it contains (for the different attributes). Centroids are gathered by index servers, that in turn must generate a centroid describing their contents. The index server centroids may be passed to other index servers, and so on. A query that is presented to an index server is forwarded to the (index) servers whose centroids match the query.

In [FY93], every site keeps statistics about the type of information it receives along each link connecting to other sites. When a query arrives in a site, it is forwarded through the most promising link according to these statistics. References [MDT93], [ZC92], and [MTD92] follow an expert-systems approach to solving the related problem of selecting online business databases.

A complementary approach to *GlOSS* is taken by Chamis [Cha88]. Briefly, the approach this paper takes is to expand a user query with thesaurus terms. The expanded query is compared with a set of databases, and the query terms with exact matches, thesauri matches, and "associative" matches are counted for each database. Each database is then ranked as a function of these counts. We believe that this approach is complementary in its emphasis on thesauri to expand the meaning of a user query.

Reference [CLC95] has applied inference networks (from information retrieval) to the text-source discovery problem. Their approach summarizes databases using document-frequency information for each term (the same type of information that *GlOSS* keeps about the databases), together with the "inverse collection frequency" of the different terms. An inference network then uses this information to rank the databases for a given query.

Two interesting alternative approaches are Pharos and the Information Manifold. The Pharos system [DADA96] combines browsing and searching for resource discovery. This system keeps information on the number of objects that each source has for each category of a subject hierarchy like the Library of Congress's LC Classification System. Alternatively, the Information Manifold system [KLSS95, LRO96] uses declarative, hand-written descriptions of the sources' contents and capabilities. These descriptions are useful to prune the search space for evaluating user queries efficiently.

7.4 Result Merging

The problem of merging document ranks from multiple sources has been studied in the information retrieval field, where it is often referred to as the *collection fusion* problem. Given a query, the goal is to extract as many of the *relevant* documents as possible from the underlying document collections. As with our problem of Chapter 5, key decisions include how far "down" each document rank to explore, and how to translate *Source* scores (*local* similarity measures) into *Target* scores (usually *global* similarity measures). An approach to address these problems is to learn from the results of training queries. Given a new query, the closest training queries are used to determine how many documents to extract from each available collection, and how to interleave them into a single document rank [VGJL95, VT97]. Another approach is to calibrate the document scores from each collection using statistics about the word distribution in the collections [CLC95]. One important difference between this line of work and ours is that we want to *guarantee* that metasearchers extract the top *Target* objects from the sources and return these objects ordered according to their *Target* scores. In contrast, the work on the collection fusion problem develops *heuristics* or techniques for placing relevant documents (a subjective notion) as high as possible in the combined document ranks for a query, sometimes using the *Source* scores as indicators of relevance.

For document collections, it is particularly hard to compute the *Target* score for a document from the query results that are typically returned by text search engines. In effect, these results do not include entire documents, and have very little information other than the *Source* scores. To address this problem, the *STARTS* protocol proposal that we described in Chapter 2 specifies what information should accompany the query results that a text search engine returns so that document rank merging is facilitated. A metasearcher can then use this information to merge multiple document ranks by computing *Target* scores without accessing the documents themselves.

A closely related problem is how to query a repository of complex, multimedia objects. These objects might have attributes like images and text. Thus, the matches between query values and such multimedia attributes are inherently fuzzy, and the objects are ranked according to how well they match the query values. The work in [CG96] and [Fag96] studies how to query such repositories efficiently. In particular, [Fag96] studies upper and lower bounds on the number of objects that we need to extract from a repository so that the overall top objects are retrieved and returned to the user that issued a query. [CG96] addresses the cost-based optimization of queries over such repositories. This work assumes that a single repository handles all attributes of an object. Therefore, there is no need to "calibrate" the scores that an object gets for a particular attribute, for example. Using our terminology, all single-attribute queries are manageable with $\epsilon = 0$. (See Section 5.5 for further discussion.)

Finally, there has been a significant amount of work on querying multiple heterogeneous sources. In Chapter 5 we assume that all sources export a uniform interface so they can all answer queries over the same set of attributes. We can use the techniques in [FK93, PGMGU95], for example, to build *wrappers* around the sources and provide the illusion of such a uniform interface.

7.5 Distributed Copy Detection

Protecting digital documents from illegal copying has received a lot of attention recently. Some systems favor the copy *prevention* approach, for example, by physically isolating information (e.g., by placing information on stand-alone CD-ROM systems), by using special-purpose hardware for authorization [PK79], or by using *active* documents (e.g., documents encapsulated by programs [Gri93]). We believe such prevention schemes are cumbersome, and may make it difficult for honest users to share information. Furthermore, such prevention schemes can be broken by using software emulators [BDGM95] and recording documents. Instead of placing restrictions on the distribution of documents, another approach to protecting digital documents (one we subscribe to) is to *detect* illegal copies using registration server mechanisms such as SCAM [SGM95, SGM96] or COPS [BDGM95]. Once we know a document to be an illegal copy, it is sometimes useful to know the originator of the illegal copy. There have been several proposals [BLMO94, CMPS94] to add unique "watermarks" to documents (encoded in word spacing or in images) so that one can trace back to the original buyer of that illegal document.

A variety of mechanisms have been suggested for registration servers. In [MW94], a few words in a document are chosen as *anchors* and checksums of a following window of characters are computed. "Similar" files can then be found by comparing these checksums that are registered into a database. This tool is mainly intended for file management applications, and detection of files that are very similar, but not for detecting small text overlaps. The COPS and SCAM registration servers however were developed to detect even small overlaps in text.

dSCAM builds on work in the resource-discovery area. (See Section 7.3.) This work usually focuses on finding the "best" sources for a query, where the best sources are usually those with the largest number of "relevant" documents for the query. These schemes are not tuned to choose databases with a potential copy of a suspicious document, in the sense of Section 6.1. The distributed copy detection problem requires that we identify databases even if they contain a single document that overlaps a suspicious document significantly.

Chapter 8

Future Work

Users should be able to express their information needs and receive the relevant data even when finding this data requires accessing multiple, heterogeneous sources, or sources that do not cooperate by exporting content summaries. Furthermore, users should receive this data ordered starting from those objects that are potentially most useful, because the number of objects that match a query might be very large. This thesis has addressed some of the issues involved in building sophisticated metasearchers. In particular, we specified a protocol, *STARTS*, that sources should support to make all metasearching tasks easier (Chapter 2). Then, we developed a system, *GlOSS*, that relies on the *STARTS* content summaries provided by cooperative sources for text-source discovery, an important task that metasearchers perform (Chapters 3 and 4). We also studied the result merging problem, and characterized what sources are "good" with respect to result merging (Chapter 5). Finally, we designed *dSCAM*, a metasearcher for a novel application: distributed copy detection, with challenging specific requirements for text-source discovery and query translation (Chapter 6). However, many problems still need to be solved before we can provide users with sophisticated, seamless, and transparent access to the large number and variety of Internet sources. Below is a description of some of these problems, which range from improving systems that already exist (e.g., WWW search engines for HTML documents), to dealing with sources that are currently largely ignored by WWW search engines (e.g., "uncooperative," non-HTML text sources, relational databases, image repositories).

Smart Query Processing over World-Wide Web Documents

Current WWW search engines generally do a poor job at ranking pages for a given user query. Typically, these engines rank the available WWW pages for the query based on the pages' contents. These page ranks are computed by following variants of the vector-space and probabilistic retrieval models developed over the years by the information retrieval community. The number of WWW pages and the wide difference in their quality and scope make this approach inappropriate in many cases: users are overwhelmed with large numbers of highly ranked, low quality pages that happen to include the query words many times.

Departing from more traditional approaches, the BackRub search engine developed by Larry Page and others at Stanford (`http://backrub.stanford.edu`) exploits the HTML link information to rank pages for queries. The more times a page is cited, the more important this page is considered by this system. Furthermore, if an "important" page points to another page, the latter inherits part of the former's importance. Citation information has also been used, together with other factors, in [YL96].

A fascinating research issue is how to use all the information available on the WWW to do a better job at ranking pages for queries. The page citations, coupled with additional knowledge available on the WWW, contain valuable nuggets of information to be mined. For example, we can map every WWW page to a location based on where its hosting site resides. Then, we can consider the location of all the pages that point to, say, the Palo Alto Weekly home page [1]. By examining the distribution of these pointers we can conclude that the Palo Alto Weekly home page is of interest mainly to residents of the San Francisco Bay Area. This information can then be used to answer queries. For example, if a midwest user requests home pages of periodicals, then the Palo Alto Weekly should be ranked low for that user. However, the New

[1]Citations from pages hosted on national access providers like America On Line would be ignored in this process, unless we can map these citations to the physical location of their creator.

York Times home page might be ranked high for such a user: although this page does not reside in the midwest, it is cited all across the USA, and will therefore be judged relevant for our user.

The example above illustrates the kind of more sophisticated query processing strategies that are possible if we start exploiting all the information available on the WWW. A key challenge in *mining* all this information for query processing is efficiency, since the volume of the information at hand is extremely large, and growing fast. Other sources of information to employ include available query logs, response times, user feedback, and quality reports. For example, initial work on mining query logs tries to predict what pages are likely to be useful to users based on their browsing behavior [Lie95], and that of previous users [YJGMD96].

Source Discovery over Uncooperative Text Sources

Metasearchers choose the best sources to evaluate queries by using summaries of each source's contents, assuming that they can extract these summaries with the sources' cooperation by using the *STARTS* protocol of Chapter 2, for example. Alternatively, a metasearcher can follow the WWW crawlers' model, and extract the entire full text contents of the sources by following HTML links, if possible. Once the metasearcher has the full text contents of a source, it might choose to index or summarize it in any way. Unfortunately, sometimes uncooperative text sources are hidden behind search interfaces, and offer little more to users than a sophisticated query interface (e.g., the Internet Movie Database, at `http://www.imdb.com`). A WWW crawler will not index the text contents of such a source, because the crawler cannot download the source's contents by following links, and neither will any metasearcher, because the source does not export content summaries.

To summarize the contents of such a source with no further help, a metasearcher might resort to periodically querying the source using a reasonably small set of carefully chosen queries. By interpreting the answers to these queries, the metasearcher might decide if the source is likely to be useful when it receives a user query. An interesting direction to design this small set of queries is to use the Latent Semantic

Indexing technique (LSI) [FDD+88, BDO94, Dum94]. LSI is used in the information retrieval community for document retrieval. LSI constructs compact representations of the sources' contents. These representations could in turn be approximated by carefully choosing a limited set of queries to issue to the sources. This way, we could approximate a sensible representation of the contents of a source by just querying the source a small number of times, instead of relying on the source to directly export content summaries.

Source Discovery over Overlapping Text Sources

Knowing how sources' contents overlap is especially important on the Internet, where mirroring of sources is commonplace. The Information Manifold system developed at AT&T Bell Laboratories [KLSS95] uses description logic to summarize the source contents. Such descriptions can express when a source has *complete* information for a query, thus making other sources redundant for the query at hand. As an alternative to human-generated descriptions, *GlOSS* uses automatically generated summaries to rank the sources for the given queries. Unfortunately, this ranking of the sources does not take into account how sources might overlap. In fact, the source ranks for a query could be radically different in the presence of overlap information.

Example 36: Consider a query q that asks for documents with the word "mining" in their title. Suppose that *GlOSS* has three sources available, S_1, S_2, and S_3. To rank these sources for q, *GlOSS* knows how many documents match q at each collection. For example, *GlOSS* knows that S_1 has a total of 100 documents that contain the word "mining" in their title, S_2 has 60 such documents, and S_3 also has 60 such documents. *GlOSS* will suggest S_1 as the best source for the query (100 matching documents), and S_2 and S_3 as the next best sources (60 matching documents). Assume that sources S_2 and S_3 are disjoint, that 50 documents in S_2 matching q are also in S_1, and that 50 documents in S_3 matching q are also in S_1. If *GlOSS* does not have overlap information, then it would still rank source S_1 first, followed by S_2 and S_3. However, if *GlOSS* knows how sources overlap, the rank it produces for q will have to depend on the expected user's interests. Thus, if the user that issued q will be satisfied with the contents of only one source, then source S_1 should be at the top of the source rank. Otherwise, *GlOSS* should rank S_2 and S_3 at the top of the list. In effect, *GlOSS* would obtain 120 documents by accessing S_2 and S_3, whereas it would obtain only 110 documents by accessing S_1 and S_2, for example. ▮

The example above illustrates the need to take into account the users' interests when designing incremental query plans using source overlap information. Different users have different needs, and these needs should be accessible to the metasearchers so that users find the information they are seeking in the most effective way. A challenging problem to study is modeling different users and designing query execution plans accordingly. A possible approach is to design a probabilistic model for the users' behavior. For example, we can define the probability that users will ask for the contents of the $i + 1^{st}$ source in an incremental query plan given that they have accessed up to the i^{th} source in the rank. Based on these probabilities, we can then design optimal, or close to optimal, plans, for different definitions of optimality. One possibility is to minimize the number of sources accessed before users are satisfied with the answers received.

Another challenging problem is defining and extracting source overlap information. We could start by modeling pairwise source overlap. Given two sources S_1 and S_2, a possibility is to resort to document sampling to determine how they overlap. Thus, we can conclude that, say, 20% of all S_1 documents are also in S_2. To gather finer information at the query level, we will use more sophisticated schemes. A possibility is to obtain a document sample from S_1 and S_2, and cluster these documents using some predefined clustering scheme [Sal89]. For each cluster, we can analyze how the two sources overlap. When a query arrives, the metasearcher classifies it into the most relevant clusters, and uses the overlap information for these clusters. Thus, we can conclude that, say, 20% of all S_1 documents *that match the query* are also in S_2.

Source Discovery over Non-Textual Sources

So far we have discussed the source discovery problem over sources of text documents. However, many sources on the Internet host other kinds of information, like "relational-like" data, images, etc. A particularly challenging open issue is how to summarize the contents of such sources in an automatic and scalable way so that metasearchers can reason about the sources when processing user queries.

Image features like color histograms are commonly used to search over image repositories [FBF$^+$94, NBE$^+$93, OS95, CSM$^+$97]. These features are typically represented as weight vectors. Similarly, the vector-space retrieval model also models text documents as weight vectors. An interesting direction to investigate is whether we can use our techniques for text sources in order to compactly summarize image repositories, given this similarity in representation. However, there are significant semantic differences between the vectors for text and for image features. For example, it is unlikely that we could successfully summarize the color histograms of an image repository by assuming that colors appear independently in the images. Alternatively, we could cluster similar image feature vectors at a source, and export one *centroid* per cluster as its representative. We will then use these centroids as the content summary for the image source.

An approach for summarizing relational-like sources is to use human-generated descriptions of the sources (e.g., the Information Manifold). An interesting direction for generating source summaries automatically is to adapt results from the database community on result size and selectivity estimation of queries over relational data (e.g., [PIHS96]). These results have been used extensively for query optimization. Thus, we will consider using frequency histograms for succinctly describing relational tables. A challenging problem is answering *point* queries (as opposed to *range* queries), while keeping the size of the histograms orders of magnitude smaller than that of the original data.

Putting All the Pieces Together

Ultimately, our goal is to allow transparent query processing over sources with varying data types. For example, users should be able to issue queries whose processing involves accessing text, relational, and image sources. The discussion above focused on dealing with, say, text sources and image sources *separately*. However, before we can process queries that span several source and data types, we need to address the following issues:

- **Defining the meaningful combinations of data types and operations.** To extract the information that users need, a metasearcher might perform join-like operations involving, say, two repositories of text documents. Reference [CDY95] is an interesting step towards integrating relational-like and text sources for querying. We will explore the meaningful combinations of data types and operations, and define their semantics precisely so that metasearchers can translate user requests into potentially complex queries spanning multiple sources.

- **Defining expressive query languages.** Users should express their requests using simple interfaces. Metasearchers should translate user requests into queries written in a query language that models the wide variety of sources and data types available on the Internet. We will start with recent work on query languages for the WWW like WebSQL [MMM96], and incorporate the notion of sources as first class objects, so that we can express source discovery in our queries, and optimize the queries using source properties. We will also include the notion of fuzzy matches of conditions and objects, to model that users typically want the *best* objects for their queries, not all possible matching objects.

- **Defining efficient execution plans for queries spanning several source and data types.** Finally, once a metasearcher produces a complex query expressed in the query language discussed above, it has to design efficient, incremental query plans to execute it. Producing these plans involves putting together all the pieces that we have discussed in this thesis: deciding what sources are relevant for evaluating the different query pieces (*source discovery*), evaluating these pieces at the sources using the available interfaces and query models (*query translation*), and finally combining the answers produced by the sources into a coherent query result for the user that issued the query (*result merging*).

Bibliography

[Age95] Z39.50 Maintenance Agency. Attribute set Bib-1 (Z39.50-1995): Semantics, September 1995. Accessible at `ftp://ftp.loc.gov/pub/z3950/-defs/bib1.txt`.

[BC92] Daniel Barbará and Chris Clifton. Information Brokers: Sharing knowledge in a heterogeneous distributed system. Technical Report MITL-TR-31-92, Matsushita Information Technology Laboratory, October 1992.

[BCGP97a] Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke. Metadata for digital libraries: Architecture and design rationale. In *Proceedings of the Second ACM International Conference on Digital Libraries (DL'97)*, July 1997.

[BCGP97b] Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke. The Stanford Digital Library Metadata Architecture. *International Journal of Digital Libraries*, 1(2), 1997.

[BDGM95] Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD'95)*, May 1995.

[BDH+94] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado-Boulder, August 1994.

[BDO94] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. Technical Report CS-94-270, Computer Science Department, University of Tennessee, December 1994.

[BLMO94] J. Brassil, S. Low, N. Maxemchuk, and L. O'Gorman. Document marking and identification using both line and word shifting. Technical report, AT&T Bell Laboratories, 1994.

[CDY95] Surajit Chaudhuri, Umeshwar Dayal, and Tak W. Yan. Join queries with external text sources: execution and optimization techniques. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD'95)*, May 1995.

[CG96] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, June 1996.

[CGMP96a] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, August 1996.

[CGMP96b] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Predicate rewriting for translating Boolean queries in a heterogeneous information system. Technical Report SIDL-WP-1996-0028, Stanford University, 1996. Accessible at `http://www-diglib.stanford.edu/-cgi-bin/WP/get/SIDL-WP-1996-0028`.

[Cha88] Alice Y. Chamis. Selection of online databases using switching vocabularies. *Journal of the American Society for Information Science*, 39(3), 1988.

[Chr97] Eliot Christian. Application profile for the government information locator service GILS, Version 2, August 1997. Accessible at `http://-www.usgs.gov/gils/prof_v2.html`.

[CLC95] James P. Callan, Zhihong Lu, and W. Bruce Croft. Searching distributed collections with inference networks. In *Proceedings of the Eighteenth ACM International Conference on Research and Development in Information Retrieval (SIGIR'95)*, July 1995.

[CLR91] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms.* The MIT Press, 1991.

[CMPS94] A. Choudhury, N. Maxemchuk, S. Paul, and H. Schulzrinne. Copyright protection for electronic publishing over computer networks. Technical report, AT&T Bell Laboratories, 1994.

[CSM+97] Shih-Fu Chang, John R. Smith, Horace J. Meng, Hualu Wang, and Di Zhong. Finding images/video in large archives. *D-Lib Magazine*, February 1997.

[DADA96] Ron Dolin, Divyakant Agrawal, Laura Dillon, and Amr El Abbadi. Pharos: A scalable distributed architecture for locating heterogeneous information sources. Technical Report TRCS96-05, Computer Science Department, University of California at Santa Barbara, July 1996.

[DANO91] Peter B. Danzig, Jongsuk Ahn, John Noll, and Katia Obraczka. Distributed indexing: a scalable mechanism for distributed information retrieval. In *Proceedings of the Fourteenth ACM International Conference on Research and Development in Information Retrieval (SIGIR'91)*, October 1991.

[Den95] Peter J. Denning. Editorial: Plagiarism in the web. *Communications of the ACM*, 38(12), December 1995.

[DLO92] Peter B. Danzig, Shih-Hao Li, and Katia Obraczka. Distributed indexing of autonomous Internet services. *Computer Systems*, 5(4), 1992.

[DS94] Andrzej Duda and Mark A. Sheldon. Content routing in a network of WAIS servers. In *Proceedings of the Fourteenth IEEE International Conference on Distributed Computing Systems*, June 1994.

[Dum94] Susan T. Dumais. Latent semantic indexing (LSI) and TREC-2. In *Proceedings of the Second Text Retrieval Conference (TREC-2)*, March 1994.

[Fag96] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Fifteenth ACM Symposium on Principles of Database Systems (PODS'96)*, June 1996.

[FBF$^+$94] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[FDD$^+$88] George W. Furnas, Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, Richard A. Harshman, Lynn A. Streeter, and Karen E. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure. In *Proceedings of the Eleventh ACM International Conference on Research and Development in Information Retrieval (SIGIR'88)*, June 1988.

[FK93] Jean-Claude Franchitti and Roger King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. In *Advanced Database Systems*, pages 313–36. Springer-Verlag, 1993.

[FW$^+$93] Jim Fullton, Archie Warnock, et al. Release notes for freeWAIS 0.2, October 1993.

[FY93] David W. Flater and Yelena Yesha. An information retrieval system for network resources. In *Proceedings of the International Workshop on Next Generation Information Technologies and Systems*, June 1993.

[GCGMP96] Luis Gravano, Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. STARTS: Stanford protocol proposal for Internet retrieval and search. Technical Report SIDL-WP-1996-0043, Stanford University, August 1996. Accessible at `http://-www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0043`.

[GCGMP97] Luis Gravano, Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. STARTS: Stanford proposal for Internet meta-searching. In *Proceedings of the 1997 ACM International Conference on Management of Data (SIGMOD'97)*, May 1997.

[GGM95a] Luis Gravano and Héctor García-Molina. Generalizing *GlOSS* for vector-space databases and broker hierarchies. In *Proceedings of the Twenty-first International Conference on Very Large Databases (VLDB'95)*, pages 78–89, September 1995.

[GGM95b] Luis Gravano and Héctor García-Molina. Generalizing *GlOSS* to vector-space databases and broker hierarchies. Technical Report STAN-CS-TN-95-21, Computer Science Department, Stanford University, May 1995.

[GGM97] Luis Gravano and Héctor García-Molina. Merging ranks from heterogeneous Internet sources. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, August 1997.

[GGMT93] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The efficacy of *GlOSS* for the text-database discovery problem. Technical Report STAN-CS-TN-93-002, Computer Science Department, Stanford University, November 1993.

[GGMT94a] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The effectiveness of *GlOSS* for the text-database discovery problem. In *Proceedings of the 1994 ACM International Conference on Management of Data (SIGMOD'94)*, May 1994.

[GGMT94b] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. Precision and recall of *GlOSS* estimators for database discovery. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS'94)*, September 1994.

[GMGS96] Héctor García-Molina, Luis Gravano, and Narayanan Shivakumar. *dSCAM*: Finding document copies across multiple databases. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS'96)*, December 1996.

[Gri93] Gary N. Griswold. A method for protecting copyright on networks. In *Joint Harvard MIT Workshop on Technology Strategies for Protecting Intellectual Property in the Networked Multimedia Environment*, April 1993.

[GW96] Susan Gauch and Guijun Wang. Information fusion with ProFusion. In *Proceedings of the World Conference of the Web Society (WebNet'96)*, October 1996.

[Har96] Darren Hardy. Resource description messages (RDM), July 1996. Accessible at `http://www.w3.org/pub/WWW/TR/NOTE-rdm.html`.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB'92)*, pages 103–14, August 1992.

[Kah92] Robert E. Kahn. Deposit, registration and recordation in an electronic copyright management system. Technical report, Corporation for National Research Initiatives, Reston, Virginia, August 1992.

[KLSS95] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The Information Manifold. In *Proceedings of the AAAI Spring Symposium Series*, March 1995.

[KM91] Brewster Kahle and Art Medlar. An information system for corporate users: Wide Area Information Servers. Technical Report TMC199, Thinking Machines Corporation, April 1991.

[Lie95] Henry Lieberman. Letizia: An agent that assists web browsing. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, August 1995.

[LLJ96] Carl Lagoze, Clifford A. Lynch, and Ron Daniel Jr. The Warwick Framework: A container architecture for aggregating sets of metadata. Technical Report TR 96-1593, Computer Science Department, Cornell University, June 1996.

[LM90] T. Y. Cliff Leung and Richard R. Muntz. Query processing for temporal databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 200–8, February 1990.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases (VLDB'96)*, September 1996.

[MDT93] Anne Morris, Hilary Drenth, and Gwyneth Tseng. The development of an expert system for online company database selection. *Expert Systems*, 10(2):47–60, May 1993.

[MMM96] Alberto O. Mendelzon, George H. Mihaila, and Tova Milo. Querying the World Wide Web. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS'96)*, December 1996.

[MTD92] Anne Morris, Gwyneth Tseng, and Hilary Drenth. Expert systems for online business database selection. *Library Hi Tech*, 10(1-2):65–68, 1992.

[MW94] Udi Manber and Sun Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.

[NBE$^+$93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Querying images by content using color, texture, and shape. In *Storage and retrieval for image and video databases (SPIE)*, pages 173–187, February 1993.

[Neu92] B. Clifford Neuman. The Prospero File System: A global file system based on the Virtual System model. *Computer Systems*, 5(4), 1992.

[ODL93] Katia Obraczka, Peter B. Danzig, and Shih-Hao Li. Internet resource discovery services. *IEEE Computer*, September 1993.

[OM92] Joann J. Ordille and Barton P. Miller. Distributed active catalogs and meta-data caching in descriptive name services. Technical Report #1118, University of Wisconsin-Madison, November 1992.

[Org95] National Information Standards Organization. Information retrieval (Z39.50): Application service definition and protocol specification (ANSI/NISO Z39.50-1995), 1995. Accessible at `http://-lcweb.loc.gov/z3950/agency/`.

[OS95] Virginia E. Ogle and Michael Stonebraker. Chabot: retrieval from a relational database of images. *Computer*, 28(9), September 1995.

[PCGM$^+$96] Andreas Paepcke, Steve B. Cousins, Héctor García-Molina, Scott W. Hassan, Steven K. Ketchpel, Martin Roscheisen, and Terry Winograd. Towards interoperability in digital libraries: Overview and selected highlights of the Stanford Digital Library Project. *IEEE Computer Magazine*, May 1996.

[PGMGU95] Yannis Papakonstantinou, Hector Garcia-Molina, Ashish Gupta, and Jeffrey Ullman. A query translation scheme for rapid implementation of wrappers. In *Fourth International Conference on Deductive and Object-Oriented Databases*, pages 161–186, 1995.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, June 1996.

[PK79] Gerald J. Popek and Charles S. Kline. Encryption and secure computer networks. *ACM Computing Surveys*, 11(4):331–356, December 1979.

[RBC$^+$97] Martin Roscheisen, Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, Steven Ketchpel, and Andreas Paepcke. The Stanford InfoBus and its service layers: Augmenting the Internet with higher-level information management protocols. In *MeDoc Dagstuhl Workshop: Electronic Publishing and Digital Libraries in Computer Science*, July 1997.

[SA89] Patricia Simpson and Rafael Alonso. Querying a network of autonomous databases. Technical Report CS-TR-202-89, Department of Computer Science, Princeton University, January 1989.

[Sal89] Gerard Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer.* Addison-Wesley, 1989.

[Sch90] Michael F. Schwartz. A scalable, non-hierarchical resource discovery mechanism based on probabilistic protocols. Technical Report CU-CS-474-90, Department of Computer Science, University of Colorado at Boulder, June 1990.

[Sch93] Michael F. Schwartz. Internet resource discovery at the University of Colorado. *IEEE Computer*, September 1993.

[SDW$^+$94] Mark A. Sheldon, Andrzej Duda, Ron Weiss, James W. O'Toole, and David K. Gifford. A content routing system for distributed information servers. In *Proceedings of the Fourth International Conference on Extending Database Technology*, 1994.

[SE95] Erik Selberg and Oren Etzioni. Multi-service search and comparison using the MetaCrawler. In *Proceedings of the Fourth International WWW Conference*, December 1995.

[SEKN92] Michael F. Schwartz, Alan Emtage, Brewster Kahle, and B. Clifford Neuman. A comparison of Internet resource discovery approaches. *Computer Systems*, 5(4), 1992.

[SFV83] Gerard Salton, Edward A. Fox, and Ellen M. Voorhees. A comparison of two methods for Boolean query relevance feedback. Technical Report TR 83-564, Computer Science Department, Cornell University, July 1983.

[SGM95] Narayanan Shivakumar and Héctor García-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second International Conference in Theory and Practice of Digital Libraries*, June 1995.

[SGM96] Narayanan Shivakumar and Héctor García-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of the First ACM International Conference on Digital Libraries (DL'96)*, March 1996.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[TA94] A. Tal and Rafael Alonso. Commit protocols for externalized-commit heterogeneous database systems. *Distributed and Parallel Databases*, 2(2):209–34, April 1994.

[TGL+97] Anthony Tomasic, Luis Gravano, Calvin Lue, Peter Schwarz, and Laura Haas. Data structures for efficient broker implementation. *ACM Transactions on Information Systems*, 1997.

[VGJL95] Ellen M. Voorhees, Narendra K. Gupta, and Ben Johnson-Laird. The collection fusion problem. In *Proceedings of the Third Text Retrieval Conference (TREC-3)*, March 1995.

[VT97] Ellen M. Voorhees and Richard M. Tong. Multiple search engines in database merging. In *Proceedings of the Second ACM International Conference on Digital Libraries (DL'97)*, July 1997.

[WGMJ95] Stuart Weibel, Jean Godby, Eric Miller, and Ron Daniel Jr. OCLC/NCSA metadata workshop report. Accessible at `http://www.oclc.org:5047/oclc/research/publications/-weibel/metadata/dublin_core_report.html`, March 1995.

[YGM95a] Tak W. Yan and Héctor García-Molina. Duplicate detection in information dissemination. In *Proceedings of the Twenty-first International Conference on Very Large Databases (VLDB'95)*, September 1995.

[YGM95b] Tak W. Yan and Héctor García-Molina. SIFT–a tool for wide-area information dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, pages 177–86, January 1995.

[YJGMD96] Tak W. Yan, Matthew Jacobsen, Héctor García-Molina, and Umeshwar Dayal. From user access patterns to dynamic hypertext linking. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[YL96] Budi Yuwono and Dik L. Lee. Search and ranking algorithms for locating resources on the World Wide Web. In *Proceedings of the Twelfth International Conference on Data Engineering*, February 1996.

[Z3997] ZDSR profile: Z39.50 profile for simple distributed search and ranked retrieval, Draft 5, March 1997. Accessible at `http://lcweb.loc.gov/-z3950/agency/profiles/zdsr.html`.

[ZC92] Sajjad Zahir and Chew Lik Chang. Online-Expert: An expert system for online database selection. *Journal of the American Society for Information Science*, 43(5):340–357, June 1992.