Query Routing in Large-scale Digital Library Systems *

Ling Liu

Oregon Graduate Institute of Science and Technology
Department of Computer Science and Engineering
P.O.Box 91000 Portland Oregon 97291-1000 USA
email: lingliu@cse.ogi.edu

Abstract

Modern digital libraries require user-friendly and yet responsive access to the rapidly growing, heterogeneous, and distributed collection of information sources. However, the increasing volume and diversity of digital information available online have led to a growing problem that conventional data management systems do not have, namely finding which information sources out of many candidate choices are the most relevant to answer a given user query. We refer to this problem as the query routing problem.

We introduce in this paper the notation and issues of query routing, and present a practical solution for designing a scalable query routing system based on multi-level progressive pruning strategies. A key idea is to create and maintain user query profiles and source capability profiles independently, and to provide algorithms that can dynamically discover relevant information sources for a given query through the smart use of user query profiles and source capability profiles, including the mechanisms for interleaving query routing with query parallelization and query execution process to continue the pruning at run-time. Comparing with the keyword-based indexing techniques adopted in most of the search engines and software, our approach offers fine-granularity of interest matching, thus it is more powerful and effective for handling queries with complex conditions.

Keywords: *Digital libraries, Distributed and interoperable information servers, Query routing, Query refinement, Scalability, Relevance Reasoning.*

*This research is partially supported by ARPA grant MDA972-97-1-0016.

1 Introduction

The Internet today contains millions of hosts that provide file service, document sources, free and commercial information servers specializing in topics such as news, books, technical reports, biology, geography, jobs, real-estate, airline schedules, stock market listings, politics, and entertainment. Locating and accessing information in the rapidly growing, heterogeneous, distributed collection of data sources available in the Internet is a difficult problem of growing importance [2, 5, 6, 12, 17, 21], especially for large-scale digital libraries where a single point of contact is desired. However, neither the organization of the data nor the available Internet tools for associative access to the data is adequate for quickly discovering relevant information.

The difficulty of providing associative access to a large number of distributed information servers lies primarily in problems of scale. *Query routing* is a process of directing user queries to appropriate servers by constraining the search space through query refinement and source selection. Borrowing from the information retrieval terminology, query routing reduces both false positives (useless delivered answers that fail to fulfill user's needs) and false negatives (useful answers that the system fails to deliver to the user). Concretely, effective query routing not only reduces the query response time and the overall processing cost, but also eliminates a lot of unnecessary communication overhead over the global networks and over the individual information sources.

We divide the query routing into two cooperating processes. First, since a broadly defined query inevitably produces many false positives, the main mechanism to reduce them is by *query refinement*, which helps a user narrow the query definition to focus on the useful answers. Second, to reduce false negatives we use *source selection* to help users identify and locate the current set of relevant information providers in a fast changing environment. At the same time, source selection also prunes irrelevant information sources for a user query, thus reducing the overhead of contacting the information servers that do not contribute to the answer of the query.

In this paper we first outline the main research issues in query routing for large-scale distributed information systems such as digital libraries. Then we present a query routing system architecture that combines query refinement with source selection. The paper has two main contributions. The first is a simple and effective query refinement mechanism. We create and maintain a meta-description of query refinement information, called *user query profile* that accumulates the information collected about the semantic context (and scope) of what the user wants in a specific query to narrow the focus of the query. The second contribution is a source selection algorithm based on multi-level progressive pruning strategies. Using *source capability profile*, which describes an information server's content and capabilities, source selection matches the user query profile and source capability profiles to prune efficiently the set of irrelevant information sources for a given query. We discuss the query routing techniques in the context of the DIOM, a prototype system [15, 10, 19] that provides uniform access to a distributed collection of information sources on the World Wide Web (WWW).

In the reminder of this paper, we review the related work in Section 2, and discuss the query routing architecture and the mechanisms for query refinement and source selection in Section 3. Then we present the design of our query routing system in Section 4, outline the multi-level progressive pruning algorithms for query routing and for generating executable query plan in Section 5, and offer our conclusions and directions for future work in Section 6.

2 Related Work

Significant systems and theoretical results have been developed in the past decade towards uniform access of the rapidly growing, heterogeneous, and distributed collection of information sources. We review the related work in the following four categories: network-based resource discovery systems, network-based information retrieval systems, subject directories, and mediator-based information integration systems.

Network-based resource discovery systems like Harvest, GLOSS, Discover, gather information about other servers and provide some form of query-based mechanism for users to find out about servers relevant to a query.

- Harvest [2] provides a modular system for gathering information from servers and providing query-based browsing of descriptions of those servers. A broker is a kind of simple query router, and a SOIF (Structured Object Interchange Format) object is a kind of simple content summary. However, Harvest has no clear architecture for composing brokers. The Harvest Server Registry, like the WAIS directory of servers, does not support interaction with multiple brokers.

- The GLOSS system [6] also provides a mechanism for finding servers relevant to a query, but it uses a probabilistic scheme. GLOSS characterizes the contents of an information server by extracting a histogram of its word occurrences. The histograms are used for estimating the result size of each query and are used to choose appropriate information servers for searching query answers. However, GLOSS's estimates of query result set sizes are not at all accurate. Furthermore, GLOSS does not search remote data sources, it only suggests them to the user who must search them and retrieve documents manually.

- The Discover system [20] provides query refinement and content routing to over 500 WAIS servers. It is based on content labels which are constructed from WAIS source and catalog files. Unlike GLOSS, Discover does not use document frequency as a measure of server relevance. Instead, Discover first returns a collection of information servers for a user query based on the content-labels maintained locally at the Discover server, each information server is described by a set of keyword/frequency pairs. Then Discover suggests to the user a list of terms that are related to the query, by click query refinement the user may choose a term from the suggested list and run the query again. A reduced list of information servers is returned as the refined query result. The user may repeat this content routing process until he/she is satisfied with the list of information servers. However, the content labels in Discover are restricted to content summaries, and the content routing process assumes all information servers have similar capabilities, such as those described in WAIS catalog files and source summary files.

Query routing can be seen as a particular case of generic resource discovery in the Internet since the goal of query routing is to match a user query with the most relevant information sources that can provide the best answer. One could imagine using network-based resource discovery tools, such as Harvest [2], GLOSS [6, 7], and Discover [21, 20], to achieve query routing. However, typical network-based resource discovery tools so far are syntactically limited (to web pages, for example) and semantically shallow (keyword match, for example). We believe that specific information describing both the queries and information sources are needed for a good match required in effective query routing.

Network-based information retrieval systems, e.g., popular search engines like Altavista, Excite, Infoseek, and search software like Inktomi [8], WAIS [9], provide associative access to information on remote

servers. These systems combines full text indexing of documents with network-based access. A query in such systems yields a set of content-summary/URL pairs that are then used to retrieve the relevant documents manually. Both search engines and resource discovery tools can be seen as serving the same purpose of query routing, i.e., bringing the best answers to user queries. While they have been quite effective for keyword-based queries over web pages collected over the Internet, their strength is in the caching of content summaries and indexing information at the search engine server to answer simple queries quickly. In contrast, the strength of query routing discussed in this paper is the ability to redirect queries with sophisticated conditions to relevant distributed information sources, while minimizing or reducing false positives and false negatives of the query results.

Another way to digest and filter Internet information is to rely on human experts to classify and update information manually, referred to as *Subject-based directories* of information, e.g., Yahoo. While such a directory helps human users browsing the web to reach their destination faster, the classification is usually fixed and designed for naive users looking for the most common information. In comparison, query routing has the flexibility of dynamically matching individual user query profile to source capability profiles.

The recent mediator-based systems like DISCO [4], TSIMMIS [23, 13], Garlic [3], and Information Manifold [12] have used source capability descriptions to address the problem of capability-sensitive query processing. Information Manifold is among the very few systems [1, 12, 17] that provide an algorithm for source selection through relevance reasoning. While the solution from these mediator-based systems are useful in the selection of appropriate information sources, the absence of using user query profile for query refinement limited their effectiveness in the selection process.

Our proposal presented in this paper was initially inspired by the Information Manifold for capability-sensitive mediation and by Information Retrieval research on term revision. The main thrust is its smart coordination of the query refinement powered by user query profiles and the source selection powered by multi-level progressive pruning and source capability profiles.

3 Query Routing: Notation and Issues

The goal of a query routing system is to provide efficient associative access to the contents of a very large, heterogeneous, distributed collection of information providers. Such a system must be designed to scale as the network of information providers grows and must provide to key services: query refinement and source selection.

In this section we review the architectural structure of a query routing system, and characterize the query routing systems by the techniques used for implementing query refinement and query routing services.

3.1 System Architecture

A query routing system combines a large, distributed set of information providers into a single coherent framework. A query routing system consists of a hierarchical network (a directed acyclic graph) with external information providers at the leaves and query routers as mediating nodes (see Figure 1). The end-point information providers support query-based access to their documents. At a query router node, a user may browse and query the meta information about information providers registered at that query router as well as make use of the router's facilities for query refinement and source selection. The meta

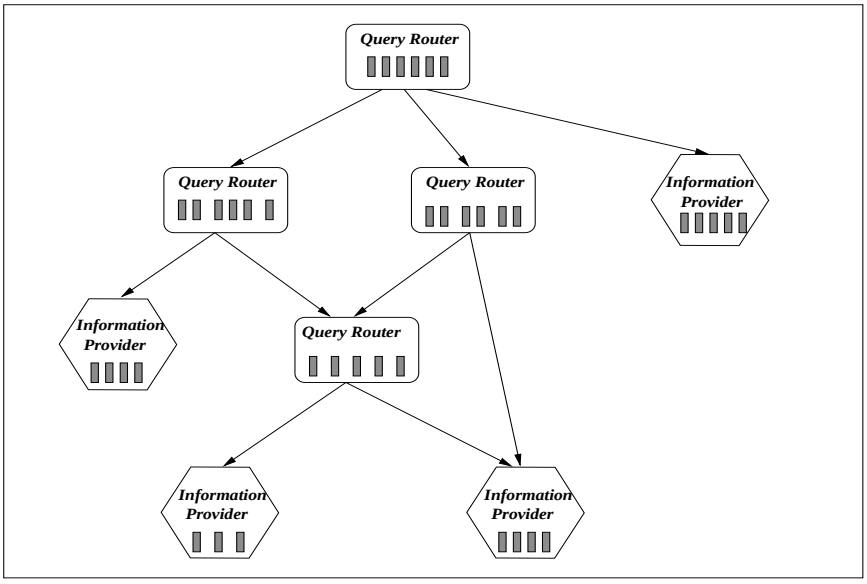

Figure 1: The query routing system architecture

information about an information provider contains content summary and query capability description of that provider, and is called the *source capability profile*. A source capability profile contains a query that is true of all documents available from the information provider. It also contains information that can be used for query refinement and source selection. For example, the following is a simple source capability profile, saying that every document at its server has a collection type of `techreports`, is from an institution named `OGI`, and is from the department of `computer science and engineering`:

```
(institution: ogi) and (collection-type: techreports) and
(department: computer science and engineering)
```

The precise nature of source capability profiles is not specified by the query routing architecture, but is left to the information providers and query routers themselves. A source capability profile may therefore be a simple predicate consisting only of the host name of the computer acting as the information server. It may also be a disjunction of all the terms in all the documents at the information server. We expect typical source capability profiles to consist of a subset of the attributes derived from the indexed documents as well as *value-added* attributes that describe the collection of documents but may not occur in the documents themselves. A query router may restrict the form and size of the source capability profiles it accepts from registering information providers. Constraining the size of source capability profiles allows the system to scale, and may require an information provider to eliminate some terms from its source capability profile.

3.2 Query Refinement

In a large, rapidly evolving network of information servers, there are no expert users because any user's knowledge is quickly out of date. Users will inevitably submit broadly defined queries that produce enormous result sets with many false positives (useless answers that fail to fulfill the user's needs). Such

enormous result sets are likely to adversely impact system performance and overwhelm the user with unwanted material. Any system that provides global information access must therefore help the user formulate queries that will return more useful results and that can be processed efficiently. A mechanism that recommends such query modifications to reduce false positives is called *query refinement*.

Query refinement, initially introduced in the Discover system [21], intends to assist a user in formulating well focused queries by suggesting terms related to a query. These terms can be used for helping the user narrow the query definition to focus it on documents of interest by either formulating new queries or refining a query with more focus. As shown in Figure 2, the Discover query refinement paradigm assumes that the user starts with a simple general query that would match many relevant documents in a broader context. Query refinement incrementally drives the process of focusing the general query by iteratively offering the user an automatically generated set of terms that can be chosen to make the query more specific.

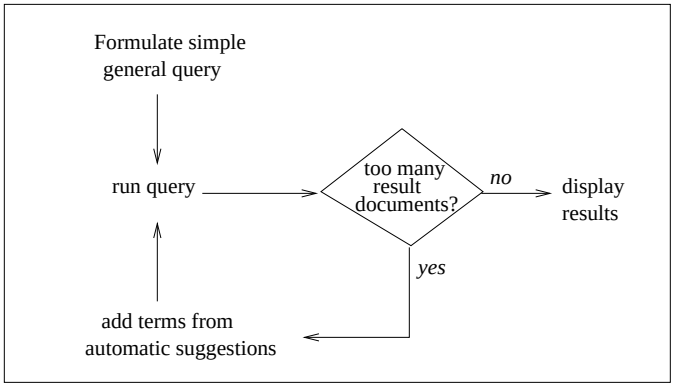

Figure 2: The Discover query refinement paradigm

There are a number of methods for deriving recommended terms to add to a user's query. Typical query refinement algorithms rely on collocation of terms. A commonly used approach consists of two basic steps: it first computes the set of documents that contain one or more terms from the user's query and then suggests to the user the terms with the highest cumulative frequency over the above set of documents. For example, in response to the query "scheme", the HyPersuit [25] prototype that indexes documents from several MIT web sites suggests terms including: "applications", "compiler", "interpreter", "abelson" and "closures". In Discover prototype [20], the conditional probability of term collocation[1] is used to compute and recommend terms that are related to a given query. Each refinement iteration offers the user the top 40 terms with the highest conditional probabilities. Any concrete query refinement algorithm may incrementally drive the refinement process by iterating these two steps.

In this paper we propose a new approach to query refinement, which uses the user query profiles as a means to assist a user in formulating well focused queries (see Figure 3). The main idea is to derive recommended terms based on the semantic context and scope of what the user wants in a particular query, and replace the terms that are too broad in the original query definition with the recommended terms. For instance, in response to a query "book suppliers", the query routing system will derive the

[1]The conditional probability p_i that term t_i occurs in a document that satisfies query Q is the size of the document space for Q conjoined with t_i divided by the size of the document space for Q: p_i = size(D($Q \wedge t_i$)) / size(D(Q)). The document space of Q is the set of all documents that match Q, i.e., D(Q) = $\{d|Q(d)\}$.

following recommended terms: "book store", "book club", "publisher". These recommended terms are either obtained directly from the user's feedback on the query context or derived from domain specific knowledge (e.g., book related application domain in this case).

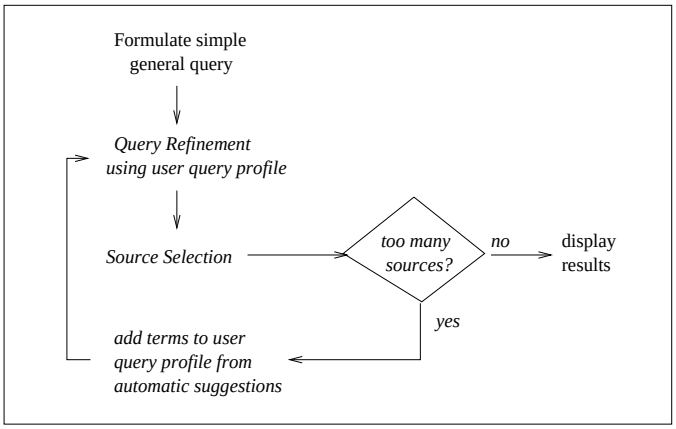

Figure 3: The user-query-profile based query refinement paradigm

A significant difference between the user-query-profile approach to query refinement and the Discover approach, which relies on collocation of terms in the source documents, lies in the ways by which additional terms are derived. The query refinement approach, driven by user-query-profiles, computes and recommends terms to focus a query primarily based on the domain knowledge of the terms used in the original user query, and thus is independent of the collection of source documents over which the query is posed. An obvious advantage of such an approach is its ability to reduce false positives before the actual run of the query, thus enhancing the efficiency and accuracy of the query refinement algorithms. For further detail see Section 4.

3.3 Source Selection

Internet-scale distributed information systems such as digital libraries consists of a large, continuously changing set of information sources. Quite often, of the large collection of available information sources, only a small subset may actually contribute to the answer of a user query. For instance, objects or documents being queried upon sometimes may not be physically stored in the given source (e.g., they may be only conceptual entities or links to other external sources), or be very costly to access, or be impossible to consult (e.g., they are stored in a remote server that is temporarily unavailable to a mobile computing device). Thus, any system that offers global information access must not only provide a single point of contact but also help the user to identify and locate the current set of relevant information providers. *Source selection* is a mechanism that helps the user locate relevant information by identifying and pruning irrelevant information sources for a user query, thus reducing the overhead of contacting the information servers that do not contribute to the answer of the query. Source selection is a process in query routing, aiming at reducing or minimizing false negatives in the result set of a query.

Given a query and a collection of information sources, the first decision one needs to make is which of the following four different search semantics [7] is required for this specific query:

- *Exhaustive Search.* The user is interested in the set of all information sources that contain any matching documents.
- *All-Best Search.* The user is interested in all the best data sources for the query. The best data sources are the ones that contain more maching documents than any other data sources and have the highest payoff. In this case, the user is willing to miss some data sources, as long as they are not the best ones. Usually, a threshold-based approach is used for qualification of the best sources.
- *Sample-Best Search.* The user is interested in only to search some of the best data sources that yield the highest payoff, because of limited resources (e.g., time, budget).
- *Sample Search.* The user is in "browsing" mode and simply want to obtain some matching documents (normally with the shortest responsive time).

Without loss of generality we assume the first search scenario in this paper, unless otherwise specified.

Given a query and its search semantics, there are three alternative approaches to compute the best data sources for a particular query. The first one is to use the size of the source documents as a measure of server relevance. It is widely used in most of the Internet search software. For instance, the GLOSS system [6] uses this approach by providing a mechanism based on a probabilistic scheme. Concretely, GLOSS characterizes the contents of each information server by extracting a histogram of its word occurrences. The histograms are then used for estimating the result size of each query. Those data sources having the highest payoff (i.e., the largest number of documents) are chosen as the best information servers for searching in the context of All-Best and Only-Best semantics. The main problem with this approach is two folds: On the one hand, the estimate of word occurrences may not at all accurate. On the other hand, without explicit use of query refinement, a broader definition of query terms in the original query may inevitably produce many false positives, and thus invalidates some of the source selection results.

Another approach to compute the best data sources for a query is to use the query capabilities of data sources as a measure of server relevance. It is mostly used in mediator-based systems. For example, In TSIMISS [23, 13] and Information Manifold [12], data sources are characterized by their content and capability descriptions. Such source capability descriptions are then used to evaluate the relevance of a particular query. Those data sources having the capability to handle the query and returning non-empty results are then chosen as the best information servers for executing the query. Although the capability-sensitive approach is more adequate for handling queries with complex conditions, it also suffers the problem of missing explicit incorporation of query refinement into the query routing process.

We present our proposal as the third approach to query routing, which uses both the user query profile and the source capability profile as a measure of server relevance. By using user query profiles for query refinement, a well focused query is produced to ensure the matching data sources selected have zero or minimized false positives. By using the query capability profiles and a multi-level progressive pruning algorithm, the result of source selection will have zero or minimized false negatives (see Section 4 for further discussion.

4 Query Routing: Our Approach

In this section we describe the design of a query routing system that combines the user-query-profile based query refinement with the stepwise source selection. The distinct features of our approach include

a simple and effective query refinement mechanism based on user query profiles and the smart use of user query profile and source capability profiles to coordinate and enhance the query routing result through multi-level progressive irrelevance pruning.

We begin by introducing the motivating examples and a predicate metadata model in which user queries, user query profiles, and source capability profiles are captured. Then we present the design of our query routing algorithms.

4.1 The Motivating Example

Data sources over the Internet have a wide range of query processing capabilities. Many information servers where the document sources reside provide a single-table view, and allow only limited types of selection queries. This introduces some interesting query processing challenges as illustrated by the following examples.

Example 1 Consider the Internet bookstore Amazon.com. Suppose one wants to search for books written by Robert Orfali or Dan Harkey on the topic of CORBA or Java. We can express the query condition as ((*author = "Robert Orfali"* $\vee$ *author = "Dan Harkey"*) $\wedge$ (*title contains "CORBA"* $\vee$ *title contains "Java"*). Let us assume for the time being that we have captured the source content and capability of Amazon.com site in its source capability profile.

From the source capability profile, we know that the Amazon.com's query interface does not support the search for two or more authors or titles at once. Thus to process this query, our system needs to break down the query into queries that can be executable at the amazon.com site. One possible plan is to break the query into the following four queries: (1) (*author = "Robert Orfali"* $\wedge$ *title contains "CORBA"*); (2) (*author = "Dan Harkey"* $\wedge$ *title contains "CORBA"*); (3) (*author = "Robert Orfali"* $\wedge$ *title contains "Java"*); (42) (*author = "Dan Harkey"* $\wedge$ *title contains "Java"*). The results of these four queries are then unionized to obtain the answer to the original query.

Through this example, we observe two interesting facts. First, the extraction and use of source capability profile about Amazon.com site plays a critical role in the query parallelization and execution process of the query. Second, even simple selection queries against a single data source across the Internet may have more complications due to the source-specific content and its limited query capability. The situation will become more sophisticated when we have queries over multiple distributed data sources that are heterogeneous in both their information content and their query capabilities.

Example 2 Consider the online book supplier sites (sources) for book purchase such as Amazon.com, Barnes and Nobel, Morgan Kaufmann, Addison Wesley, and so on. Suppose we want to search for title, price, The parameters of interest to this query are the price, year, title, the supplier name, and the book reviews. We may ask query Q: *get the title, price, the supplier, and the reviews of all books published in 1998 about* `cancer`. We may enter the query using a SQL-like on-line form (e.g., The DIOM Interface Query Language: IQL [15]) as follows:

```
SELECT  B.title, B.price, R.review, B.supplier
FROM    Book B, Review R
WHERE   B.description CONTAINS 'cancer' .AND. B.year = 1998
        .AND. R.booktitle = B.title;
```

Suppose the query Q is annotated in the user query profile (see Section 4.2.2) that the suppliers we are interested in include only *book stores* and *publishers*, and the description attribute of books corresponds

to *title*, *abstract*, or *subject* of books. We can refine this user query by replacing the term `supplier` with *book stores* and *publishers*, and the term `description` with *title, abstract*, or *subject* of the book. Also suppose we have extracted and collected the source capability profiles of the information servers as shown in Figure 4, among many others.

Source 1: Reviews for books published during 1970 - 1980 **Category**: Book Review; **Content**: Reviews, $1970 \leq b\_publish\_year \leq 1980$; **Query Capabilities**: Accept as input b_title or $b\_authors$, and optionally $b\_publish\_year$, $reviewer\_name$. Output is the book reviews for that b_title or $b\_authors$, and optionally $b\_publish\_year$.
Source 2: Barnes & Nobel **Category**: Retail Book Store Company; **Content**: Books; **Query Capabilities**: Accept as input title or authors or category, and optionally year, price, publisher. Output is the qualifying books, with title, authors, year, price, or contact information.
Source 3: BookClubs **Category**: Book Club Database; **Content**: Book, BookClub; **Query Capabilities**: Accept as input title or authors, and optionally year or price or category or clubNo. Output is the qualifying books or bookclubs, with title, authors, year, price, clubName, or contact information.
Source 4: Morgan Kaufmann Publishers Inc. **Category**: Publisher; **Content**: Books, Journals; **Query Capabilities**: Accept as input title or authors, and optionally year, price for books, and title or editor-in-chief, and optionally year for journals. Output may include title, authors, price, year, ISBN, purchase contact information for books; or title, editor-in-chief, subjectDesc, or contact information for journals.
Source 5: Books For Sale Databases (books published after 1970) **Category**: Book Store Database; **Content**: Book, $year > 1970$; **Query Capabilities**: Accept as input title, authors, and optionally price, category. Output is the qualifying books, with title, authors, price, publisher, or contact information.
Source 6: Book Reviews Database **Category**: Book Review Database; **Content**: Reviews; **Query Capabilities**: Accept as input title, authors, and optionally year. Output is the book reviews for that title, authors and/or year(s).
Source 7: TechReport Database **Category**: Technical Report Database; **Content**: TechReport; **Query Capabilities**: Accept as input title or authors or organization, and optionally year. Output is the technical reports for that title or authors or organization or year.
Source 8: Amazon.com Book Store database **Category**: Online Book Store; **Content**: Books; **Query Capabilities**: Accept as input title or authors, and optionally publisher. Output may include title, authors, abstract, year, price, publisher for qualifying books,

Figure 4: Example information sources

By taking a close look at the source capability profiles, it is interesting to observe that some of the sources are obviously not contributing to the answer of Q. For example, we can immediately determine that Source 7 is not relevant to this query, because it has no information about books. We can also conclude that Source 1, 3, 5 are not able to contribute to the answer of Q. However, the reasoning here is more subtle: we are interested only in books published in 1998 and supplied by either book stores or publishers, whereas Source 1 has review information only on books published in between 1970 and 1980, Source 3 has only books supplied by book clubs, and Source 5 neither takes a particular year as its input argument nor provides year information on its books as output. We are left with sources 2, 4, 6, and 8. Thus only three independent subquery execution plans are generated by the query routing system powered by query refinement and source selection:

1. Ask Source 2 for the title, authors, and price of books about `cancer`, published in 1998. Assign the source name `Barnes&Nobel` to the supplier field of qualifying books. For each book, obtain a review from the Source 6.

2. Ask Source 4 for title, authors, and price of books about `cancer` and published in 1998. Assign the source name `Morgan Kaufmann Publishers, Inc.` to the supplier field of qualifying books. For each book, obtain a review from the Source 6.

3. Ask Source 8 for the title, authors, price, year of books about `cancer`. Assign the source name `Amazon.com` to the supplier field of qualifying books, and select only those books published in 1998. For each selected book, obtain a review from the Source 6.

The query result is the union of the results returned by executing these three subqueries. Note that

- In all three subqueries, we use title to select books about "`cancer`" because according to the user query profile of Q, the term `description` in the original query Q may be replaced by title or abstract or subject of books.
- We have to add authors into the list of output arguments for all three subqueries when querying the sources 2, 4, and 8, even though the user query Q does not ask for the authors in its output argument list. This is because Source 6 requires both title and authors as the mandatory input parameters in its query capability description.
- In Subquery 1 and Subquery 2, the query capabilities of Source 2 and Source 4 allow external queries to select a specific year, whereas in Subquery 3 we have to do the selection on `year` ourselves because Source 8 provides the argument `year` in its output parameter list but not in its input parameter list.

From this example, we observe that by utilizing the user query profile in query refinement of Q, the original query is refined to focus more on the documents of interest, thus reducing the amount of potential false positives in the query results. By further incorporating source capability profiles (e.g., shown informally in Figure 4) into the source selection process of Q, we can prune the information sources that are incapable of contributing to the query answer, either due to the restriction on the scope of query interest (e.g., Source 7 and Source 3 in our example), or due to the constraints on the list of mandatory input or output arguments of the sources (e.g., Source 5 does not have `year` as input or output argument), or due to the conflict of query interest with the access constraints associated with the sources (e.g., Source 1 only provides reviews for books published in between 1970 and 1980, whereas the query is interested in only books published in 1998).

This example can also be used as a concrete elaboration of our discussion in Section 3. Obviously, without query routing capability, a distributed query system has to perform exhaustive search over all available information sources one by one. In addition to the long delay for query responsiveness and high cost of query processing and communication overhead, the amount of false positives and false negatives in the query result will be huge. With a query routing system powered by source selection only (no explicit query refinement), such as some of the early source-capability sensitive mediator systems [23, 13, 3, 12] or Internet search engines, the query responsiveness is improved, the query processing cost and the amount of false negatives are also reduced with respect to the terms given in the original query. However, the inaccuracy and broadness of query terms may introduce many false positives in the result set, thus reducing the quality of the final result of source selection. Based on our experience, a query routing system powered with both query refinement and capability-sensitive source selection presents a step forward to the ultimate solution for Internet-scale distributed information systems such as the digital libraries.

4.2 The Metadata Description Model

4.2.1 The Preliminaries

The metadata description model is designed to be an object relational model. It uses the relational model as the basis and is augmented with essential object-oriented features that are useful for describing and reasoning about the scopes of user queries and the contents and query capabilities of information sources. Typical components of the metadata model are classes, a set of attributes associated with each class, a class hierarchy described by a subclass-superclass partial order. Instances of a class are called objects. Each object has a unique identifier. A class also inherits attributes from its superclasses. Attributes may be single valued or multi-valued, atomic or object identifier.

In order to model relations and classes uniformly, we use a unary relation to describe each class and a binary relation to describe each attribute. The metadata description model also includes certain integrity constraints such as disjoin dependencies, inclusion dependencies, functional dependencies, and existence dependencies. In this paper we only consider the role of the first two types of dependencies in query routing.

We consider queries that involve select, project, join, and union and that use the built-in comparison predicates such as $\leq, <, =$ and $\neq$. We assume set semantics for queries. For the convenience of our analysis, in this paper we use the notation of *conjunctive queries* [22] to express queries. A conjunctive query Q consists of a head predicate with arguments, denoting the result template, and a body, representing a binding pattern [18] of Q. The arguments of the predicate that are provided as input parameters of the query are expected to be bound. The arguments of the predicate that are produced as outputs of the query are free variables. We use lower case letters for variable names and uppercase letters with bars to denote tuples of variables and constants. Formally, a conjunctive query is of the form:

$$Q(\vec{X}) :- C_1(\vec{Y_1}), ..., C_m(\vec{Y_m}), F_Q$$

where $Q(\vec{X})$ is the head, $\vec{X} \subseteq \bigcup_{1 \leq i \leq m} \vec{Y_i}$, and the rest is the body. $C_1(\vec{Y_1}), ..., C_m(\vec{Y_m})$ are called non built-in atoms in the body, and $C_1, ..., C_m$ are names of classes or attributes used in Q; m is the number of non built-in atoms in the body and is called the *length* of the conjunctive query Q. F_Q is a conjunction of built-in comparison atoms of the form $a\theta v$, $a \in \bigcup_{1 \leq i \leq m} \vec{Y_i}$, v is a constant value. If a is numeric attribute, $\theta \in \{<, \leq, >, \geq\}$; otherwise θ denotes a string matching predicate, i.e., $\theta \in \{=, \text{CONTAINS}\}$ (precise or partial matching). We describe a conjunctive query Q by a quadruple $(Q_{from}, Q_{in}, Q_{out}, Q_{cond})$ where Q_{from} is the set of virtual interface classes used in Q, Q_{in} is the set of input arguments, Q_{out} is the set of output arguments, and Q_{cond} is the conjunction of comparison atoms.

Example 3 Recall the query Q in Example 2. It can be described as a conjunctive query of the following form:

$$\begin{aligned} query(t, p, v, s) :- \; & Book(b), Review(r), title(b, t), description(b, d), price(b, p), \\ & year(b, y), supplier(b, s), booktitle(r, t), review(r, v), \\ & y = 1998 \wedge \text{CONTAINS}(d, \text{`}cancer\text{'}). \end{aligned}$$

$query(t, p, v, n)$ is the *head* of the query, and its arguments title t, price p, review v, supplier name n are its *distinguished variables*. The distinguished variables of the query correspond to attributes appearing

in the `SELECT` clause. The rest are *atoms* of the *body* of the query, and are the bounding pattern of the query. Note that the equality predicates in the `WHERE` clause are represented by equating variables in different atoms of a conjunctive query. The atom "$y = 1998$" is called an atom of a built-in numeric comparison predicate (= in this case). The atom CONTAINS(t, '*cancer*') is called an atom of a built-in string comparison predicate (CONTAINS in this case).

$$
\begin{aligned}
Q_{from} &= \{Book(b), Review(r)\},\\
Q_{in} &= \{description(b, \text{`\%cancer'}), year(b, 1998)\}, \text{ where \% denotes CONTAINS}\\
Q_{out} &= \{title(b,t), price(b,p), review(r,v), supplier(b,s)\},\\
Q_{cond} &= \{title(b,t), booktitle(r,t), description(b,\text{`\%cancer'}), year(b,1998)\}.
\end{aligned}
$$

For presentation convenience, we sometimes refer to the unary relation atoms (e.g.,$Book(b)$, $Review(r)$) as **unary subgoals** of Q and the binary relation atoms (e.g., $title(b,t)$, $review(r,v)$) **binary subgoals**.

A user may pose queries on the fly (without using any pre-defined views or classes). For each user query, we create a *virtual interface schema* that describes all the arguments used in the query, including the classes or relations, the data types, the domain constraints, and the usage (i.e., as input or output parameter) of the arguments. Note that the notion of virtual interface schema is not the same as the view concept in relational or object-oriented database systems although it may appear similar to views at the first glance. There are two *subtle differences*: First, a view in conventional databases is defined as an installed query over a pre-existing database schema, whereas a virtual interface schema describes a semantic and structured context of a query posed by the user without using any pre-existing database schema or views. Second, views in conventional databases are defined against pre-existing database schemas. The users who define views often know what the corresponding database schema is about. While a virtual interface schema is created by the system in terms of a user query posed on the fly. The user who pose the query does not need to be aware of what information sources are currently available and which data schemas or pre-defined views should be used to access them.

For instance, once our example query Q is issued, a virtual interface schema will be created for this query, which consists of the following two virtual interface classes, each is described by a *head* predicate with arguments, denoting the interface name and list of attributes of the interface class, and a *body* containing a set of atoms.

$$
\begin{aligned}
Book(b,t,p,y,s) &:- Book(b), title(b,t), price(b,p), year(b,y), supplier(b,s)\\
Review(r,b,w) &:- Review(r), review(r,w), booktitle(r,tl)
\end{aligned}
$$

The interface class `Book` receives objects describing the title, price, year, and supplier name of books selected by Q. The interface class `Review` receives objects of reviews for those selected books. The interface class `Supplier` receives supplier information for the qualifying books.

Atoms used in a virtual interface schema are not restricted to unary or binary relation atoms. The built-in comparison atoms are often used to model constraints associated with a virtual interface schema.

A *union* of conjunctive queries is a set of conjunctive queries that have the same arity in their heads. Unless otherwise specified, we assume that a user query is a union of conjunctive queries and an information source is a database instance. Given an information source S and a query Q, we denote by $answer(Q,S)$ the result of evaluating Q over S, and $answer(Q)$ the *final* (assembled) result of Q. A query Q is said to be *satisfiable* if there exists an information source S such that $answer(Q,S) \neq \emptyset$. Put differently, a query is satisfiable if there is at least one information source where it can be evaluated

and it returns a non empty set of objects or tuples. We say that two queries Q_1 and Q_2 are *equivalent*, if Q_1 and Q_2 are satisfiable, and for any information source S, $answer(Q_1, S) = answer(Q_2, S)$. A conjunctive query Q is said to be *minimal* if we cannot remove any of the non-comparison atoms from its body and still obtain a query equivalent to Q. Unless otherwise specified, in the rest of the paper we assume that queries passed to the query routing process are minimal conjunctive queries.

4.2.2 The User Query Profile

One of the objectives of creating a user query profile for each query posed on the fly is to capture the semantic scope and context of the query. For each consumer query Q, we create a user query profile which consists of two key components: the query scope description and the query capacity description.

The *query scope description* describes the synonyms (alternative descriptions) for each atom name used in the `FROM` clause of Q and how these synonyms participate in the alternation relationship with the term, namely the *alternation constraints*. Four different types of alternations are considered here: *total* & *disjoint*, *total* & *overlap*, *partial*& *disjoint*, *partial* & *overlap*. We model the query scope, denoted as Q_{scope}, using *scope records* of the form (VI, SY, AL), where VI is the name of a virtual Interface, SY denotes a set of synonyms of VI, and AL denotes the alternation constraint over SY.

The *query capacity description* describes the qualification requirements for the arguments of the query. We model the query capacity, denoted as Q_{capa}, using *capacity records* of the form $(Att, SY, AL, Dtype, Dunit, IOtype, Bopt)$, where Att denote an attribute argument, SY denotes a set of synonyms of Att, AL is the alternation constraint over the set of synonyms SY, $Dtype$ and $Dunit$ denote the data type and the data unit of the attribute Att respectively, $IOtype$ indicates if the attribute is used as an input argument or output argument or both, and $Bopt$ denotes the binding option (*mandatory* or *optional*) for the argument Att. Note that all the input arguments in a query need to be satisfied and thus their binding options are mandatory, but sometimes we may allow certain outputs to be absent [15] in the answer by indicating their $Bopt$ to be optional.

There are a number of ways to construct user query profiles. For example, one may use domain-specific ontology, such as thesauri or concept hierarchies, to sharpen a user query, or use an interactive dialog interface program that pops up a dialog screen for users to enter meanings or synonyms whenever the user poses a query on the fly.

In our current implementation, we use simple domain-specific thesauri as the default value-added means to create a user profile. When an item in the user query is not annotated explicitly, this thesauri will be used. For instance, when the query Q in Example 2 is posed, the user may annotate each virtual interface class in the `FROM` clause and save the annotation in the query scope description record (see Figure 5(a)), and annotate each input or output attribute of the query Q in the user query capacity record (Figure 5(b)). For example, one may restrict the term *supplier* to the semantic context that only two types of suppliers are of interest: *bookstore*s and *publisher*s. We describe such context by indicating *bookstore* and *publisher* as user-specific "synonyms" of *supplier* with *total&disjoint* alternation constraint (see Figure 5(a)).

Note that the item `price` is not explicitly annotated, the system-supplied thesauri is then used and two synonyms "cost" and "value" are added. In Figure 5, all the items in `Courier` fonts are user-annoted semantics and all the items in Times fonts are generated using the system default thesauri.

The *total&disjoint* alternation constraint means that the given set of synonyms of supplier (*bookstore* and *publisher* in this case) presents a total and disjoint categorization of all supplier objects. The

Virtual Interface	Synonyms	Alternation
$Book(b)$	`novel, textbook`	`partial` & `disjoint`
$Review(r)$	`book review`	partial & overlap

(a) The query scope description of the example query Q

Attribute	Synonyms	Alternation	DType	Dunit	IOtype	Bopt
$title(b, t)$			String	VarChar	out	`mandatory`
$description(b, d)$	`title, abstract, subject`	`partial` & `disjoint`	String	VarChar	in	mandatory
$year(b, y)$			String	VarChar	in	mandatory
$price(b, p)$	cost, value	partial & overlap	float	`US$`	out	`mandatory`
$review(r, v)$			String	VarChar	out	optional
$booktitle(r, bt)$	`title`		String	VarChar	in	mandatory
$supplier(b, s)$	`book store, publisher`	`total` & `disjoint`	String	VarChar	out	`mandatory`

(b) The query capacity description of the example query Q

Figure 5: Fragements of the user query profile of the example query Q

default alternation constraint is set to *partial* & *overlap*. Similarly, the default for all the non-numeric attributes is String type with VarChar as the unit scale. The default for all the numeric attributes is the corresponding data type of their numeric values such as integer, float, double. The default for binding option of an output argument is optional. Figure 5 shows an example user query profile for the query Q in Example 2. Data are not in `this` font are entered by the system using default settings derived from the query statement of Q or inferred using available on-line ontology.

Important to note is that, for each user query posed to the global information system, we create a virtual interface schema as its result place holder and a user query profile *independently* from the number of information sources available in the system. Since the collection of information sources available is large and frequently changing, the logical independence as such allows us to add new information sources seamlessly into the system at any time without affecting the way how queries are posed and how answers are delivered, thus higher scalability is obtained.

4.2.3 The Source Capability Profile

A source capability profile tells what is in an information source and what types of services are provided about its content. It contains not only the content and query capability description but also statistics on the local data (e.g., size of relations), availability of the source with respect to the access cost and access authorization, as well as update frequency and capabilities of the source. In this section we will focus only on the source category, content, and query capability descriptions, since they are the essential components of the source capability profile and are used extensively in each step of the query routing process. In addition, each source may export information about itself by giving values to a list of meta attributes [5] such as `FieldSupported` (list of optional fields), `Linkage` (the URL where the source should be required), `ContentSummaryLinkage` (the URL of the content summary of the source).

The category and content description of an information source

The category and content description of an information source describe what is in the information

source. The content description of an information source tells us what types of objects (or tuples) are in the source. The category description tells us what type of domain the data in the source are used for and is used for the *IsA* categorization of the source. The source category description often contains information that can be used to verify an input (selection) condition or fill in an output parameter of a query. Consider the query in Example 2. Source 2 does not have supplier information in its content and capability description, but using the source category description, we can easily infer that the source name is the supplier for books from Source 2. Similar to Source 4. We model the contents of an information source in terms of the object types and the object access constraints that the source objects must satisfy. Each source object type is described by a unary relation. Each access constraint is described using a conjunction of built-in comparison atoms of the form $a\theta v$ where a is an attribute of a source type and v is a constant drawn from a domain that is compatible to the domain of a. Formally, given a source S of k relations, the content description of S is described by a set of *content records*: $\{(R_1(\vec{Y_1}), IC_1), (R_2(\vec{Y_2}), IC_2) \ldots, (R_t(\vec{Y_t}), IC_t)\}$, where R_i $(i = 1, ..., t)$ are relations and IC_i $(i = 1, ..., k)$ are access constraints over relations R_i. We may view a source content description as a collection of views defined over the source. Each R_i describes one type of source objects. Every object in R_i satisfies the insertion constraint IC_i.

Source Query Capability Description

The query capability description of an information source tells which types of queries the source can answer about its content. We model the query capabilities of an information source S using *capability records*, each is denoted by $(S_{in}, S_{out}, S_{cond})$. S_{in} denotes the set of permissible input arguments. S_{out} denotes the set of permissible output arguments. S_{cond} denotes the logical constraint ($\wedge$ or $\vee$) on the mandatory input arguments. Figure 6 shows the set of content records and the set of capability records of the information sources listed in Figure 4. From now on, we denote each information source by a triplet $(S_{cat}, S_{cnt}, S_{qpd})$ where S_{cat} denotes the text description of the category of the source, S_{cnt} is a set of source relations, each may be associated with some access constraints, i.e., $(R_i(\vec{Y_i}), IC_i)$, S_{qpd} denotes a set of query capability descriptions, each of the form $(S_{in}, S_{out}, S_{cond})$.

Consider Source 5 in Figure 6. S_{cat} = `Book Store Database`, $S_{cnt} = \{(Book(b), year(b, y) \wedge y > 1970)\}$, $S_{qpd} = (S_{in}, S_{out}, S_{cond})$ where
$S_{in} = \{title(b,t), authors(b,a), price(b,p), category(b,c)\}$,
$S_{out} = \{title(b,t), authors(b,a), price(b,p), publisher(b,s), contact(b,i)\}$,
$S_{cond} = title(b,t) \wedge authors(b,a)$.

How to obtain a source capability profile?
There are several ways that a source may export its content and capability description. One possible way is to publish the contents and capabilities of an information source on the World Wide Web. This is what current "search engines" like Alta Vista (`http://www.altavista.com`) or "metasearch crawlers" like AskJeeves do. Another way is to write a robot that can extract the list of source content and capability summaries from the resources periodically. An example of such robots is the Harvest robots (`http://www.harvest.transac.com`). It is also possible to have the sources export their content and capability profiles to a specific system through a registration process. If a source does not export any kind of content summary, it becomes hard for a metasearch system to assess what kind of information the source covers.

Furthermore, there is no restricted format in which a source exports or publishes its contents and capability. It can be simple text files or structured files or directories. A typical example is the *ZDSR* for Z39.50 profile (for simple Distributed Search and Ranked Retrieval) [5].

Source 1: Reviews for books (published during 1970 - 1980) **Category**: *Book Review;* **Content**: $Reviews(r)$; $b\_publish\_year(r, y) \wedge 1970 \leq y \leq 1980$; **Query Capabilities**: $\{b\_title(r, bt), b\_authors(r, ba), b\_publish\_year(r, by), reviewer\_name(r, n)\}$, $\{review(r, v), (b\_title(r, bt), b\_authors(r, ba), b\_publish\_year(r, by), b\_title(r, bt) \vee b\_authors(r, ba)$.
Source 2: Barnes & Nobel **Category**: *Retail Book Store Company;* **Content**: $Books(b)$; **Query Capabilities**: $\{title(b, t), authors(b, a), category(b, ct), year(b, y), price(b, p), publisher(b, s)\}$. $\{title(b, t), authors(b, a), year(b, y), price(b, p)\}$, $title(b, t) \vee authors(b, a) \vee category(b, ct)$.
Source 3: BookClubs **Category**: *Book Club Database;* **Content**: $Book(b)$, $BookClub(cl)$; **Query Capabilities**: $\{title(b, t), authors(b, a), clubName(cl, cln)\}$, $\{title(b, t), authors(b, a), price(b, p), year(b, y), clubName(cl, cln), contact(cl, info)\}$, $(title(b, t) \vee authors(b, a)) \wedge clubName(cl, cln)$.
Source 4: Morgan Kaufmann Publishers Inc. **Category**: *Publisher;* **Content**: $Books(b)$, $Journals(j)$; **Query Capabilities**: $\{title(b, t), authors(b, a), year(b, y), price(b, p), j\_title(j, tl)\}$, $\{title(b, t), authors(b, a), price(b, p), year(b, y), ISBN(b, isbn), j\_title(j, tl), j\_desc(j, d)\}$, $title(b, t) \vee authors(b, a)$.
Source 5: Books For Sale Databases (books published after 1970) **Category**: *Book Store Database;* **Content**: $Book(b)$, $year(b, y) \wedge y > 1970$; **Query Capabilities**: $\{title(b, t), authors(b, a), price(b, p), category(b, c)\}$, $\{title(b, t), authors(b, a), price(b, p), publisher(b, s), contact(b, i)\}$, $title(b, t) \wedge authors(b, a)$.
Source 6: Book Reviews Database **Category**: *Book Review Database;* **Content**: $Reviews(w)$; **Query Capabilities**: $\{book\_title(w, bt), book\_authors(w, ba), book\_year(w, by)\}$, $\{review(w, r), book\_title(w, bt), book\_authors(w, ba), book\_year(w, by)\}$, $book\_title(w, bt) \wedge book\_authors(w, ba)$.
Source 7: TechReport Database **Category**: *Technical Report Database;* **Content**: $TechReport(tr)$; **Query Capabilities**: $\{title(tr, t), authors(tr, a), organization(tr, og), year(tr, y)\}$, $\{title(tr, t), authors(tr, a), organization(tr, og), year(tr, y)\}$, $title(tr, t) \vee authors(tr, a)$.
Source 8: Amazon.com Book Stores Database (in Portland) **Category**: *Book Stores;* **Content**: $Book(b)$; **Query Capabilities**: $\{title(b, t), authors(b, a)\}$, $\{title(b, t), authors(b, a), abstract(b, abs), year(b, y), price(b, p)\}$, $title(b, t) \vee authors(b, a) \vee abstract(b, abs)$.

Figure 6: The metadata description of example information sources

4.3 The Main Steps of Query Routing

As stated earlier, the ultimate goal of query routing is to constrain the search space for a query over a large collection of available information sources by reducing the overhead of contacting the information sources that do not contribute to the query answer. Given a user query Q, a user query profile of Q, and a set of source content and capability descriptions, we design the query routing service as a two-phase process. At the query refinement phase, mechanisms are applied to refine the original query into a well focused query, aiming at reducing the false positives in the query result set and enhancing the quality and the degree of accuracy of the results produced from source selection. We have discussed a number of query refinement mechanisms in Section 3.2 and illustrated the user-query-profile based query refinement in Section 4.2.2. In what follows, we concentrate on the second-phase of the query routing task – source selection and its three-step pruning process. We first outline the three steps below and then discuss the detailed strategies and algorithms used in each of the three steps in Section 5.

Step 1: Level-one relevance pruning This step discovers the candidate information sources whose content descriptions are related to the scope of a query Q (e.g., in terms of substring matching used in the DIOM implementation). For level-one relevance pruning we use the user query scope description of Q and the content and category description of the sources. The redundancy or replication of the sources will be detected and removed. Other factors such as unavailability of the sources or affordability of the sources should be considered at this step too. We call the set

of sources selected by this step as *target* information sources of *level-one relevance*.

Step 2: Level-two relevance pruning This step prunes the information sources that have level-one relevance but do not offer enough query capability to contribute to the answer of Q, either due to the restriction on the scope of query parameters of Q, or the restriction on the list of input or output arguments of the sources, or due to the conflict of query interest with the access constraints associated with the sources. The user query capacity description of Q and the source capability profiles are used in the level-two relevance pruning. We call the set of sources selected by Step 2 as *target* information sources of *level two relevance*.

Step 3: Level-three relevance pruning This step explores run-time information to further prune irrelevant information sources to answer Q. For the atoms in Q that are of form $a\theta v$ where a is an attribute and v is a constant, if there are sources which take a or its alternative as one of the input arguments, then it is in general worthwhile to ask some subqueries to these information sources and use the run-time information returned (intermediate results) to continue to prune the number of information sources selected. We call the set of sources selected by this step as *target* information sources of *level three relevance*. The level-three relevance pruning is built as a plug-and-play component and can be turned on or tuned to play any time when the need arises. We provide an illustrative example in Section 5.3.

5 The Query Routing Algorithms

In this section we discuss algorithms for query routing and for generating an executable query plan for a user query Q. It is well known that the main source of complexity in distributed query processing is the fact that there are exponential number of candidate rewritings that need to be considered. This is especially significant in the context of open environments such as Internet, because algorithms would be exponential in the number of information sources. Our query routing algorithms can drastically reduce the number of unnecessary candidate rewritings considered by performing three-step progressive pruning, such that each step considerably reduces the number of possibilities considered in the next step.

5.1 Algorithm for Level-one Relevance Pruning

Level-one relevance pruning algorithm looks at the large collection of available information sources and selects the candidate information sources which are relevant to the user query Q in terms of only the user query scope description and the source category and content description. This step serves as a first-round selection which rules out all the information sources that are irrelevant in terms of only the query scope description and the source content descriptions. The redundancy or replication of the sources will be detected and removed. Other factors such as unavailability of the sources or affordability of the sources should be considered at this step too. For level-one relevance pruning we use the user query scope description of Q and the content and category description of the sources.

Three general strategies are used in the level-one relevance pruning process: (1) Select only the information sources that have a source relation in the source content or source category descriptions, which in some way matches one of the virtual interface classes used in the query or one of its alternations specified in the user query scope description of Q. (2) From the sources selected by using the first strategy, detect those that are replica with one another or which are in some ways duplicated in their

content. Make a decision to reduce the redundant and remove the replicated information sources. For example, if there is an information source S that is answer-complete [11] with respect to a subset of atoms (subgoals) of Q, say $G \subseteq Atom(Q)$, then we can remove other information sources that are used only for evaluating the subgoals in G (see Section 5.3 for an example). (3) For each source selected from the first two strategies, check if the cost to access it is affordable We call the set of sources selected by Step 1 as *target* information sources of *level-one relevance*. The details of the first strategy are given in Figure 7. Due to the space restriction we omit the algorithms for implementing the second and third strategies in this paper.

Algorithm Level-one Relevance Pruning$(Q, U, \mathcal{S}, \varphi)$
Inputs: Q is a conjunctive query of the form $Q(\vec{X}) : -R_1(\vec{Y_1}), ..., R_m(\vec{Y_m}), F_Q$,
U is the user query profile of Q, of the form (Q_{scope}, Q_{capa}),
$\mathcal{S}$ is a set of information source descriptions, each of the form $(S_{cat}, S_{cnt}, S_{qcp})$.
$\varphi(R)$ is a mapping function, $\varphi(R) = T$ only if $R = T$ or R is a substring of T.
Outputs: $Bucket[q]$, $q = |Q_{scope}|$ is the number of unary subgoals in Q.

For each unary subgoal $R_i \in Q_{scope}$ $(i = 1, ..., q)$ do:
Set $Bucket[R_i] \leftarrow \emptyset$;
For each information source $S_j \in \mathcal{S}$ $(j = 1, ..., |\mathcal{S}|)$ do:
Let S_j be of the form: $T_1(\vec{Z_1}), ..., T_n(\vec{Z_n}), \bigwedge_{k=1}^{n} IC_k$,
where $(T_k(\vec{Z_k}), IC_k) \in S_{cnt}$, $k = 1, ..., n$;
For $k = 1, ..., n$ do:
If $\varphi(R_i) = T_k \vee \varphi(R_i) = S_{cat}$ then
add S_j to $Bucket[R_i]$, break;
else For each synonym M of R_i, $M \in SY(R_i)$, do:
If $\varphi(M) = T_k \vee \varphi(M) = S_{cat}$ then
add S_j to $Bucket[R_i]$, break;

End.

Figure 7: *Algorithm to find the information sources of level-one relevance to a query Q. It creates a bucket for each virtual interface class, i.e., each unary subgoal of $Q(\vec{X})$, which holds names of the information sources that are considered relevant in terms of the mapping function φ.*

The algorithm **Level-one relevance pruning** scan through the large collection of information sources, and, for each unary subgoal, say R_i, (i.e., virtual interface class) in the query Q, it finds the information sources that are relevant to R_i and create a bucket, denoted as $bucket[R_i]$ to record the selected candidate information sources. The number of buckets created and computed is the same as the number of unary subgoals in Q. The level-one relevance pruning step considerably reduces the number of information sources considered in the level-two relevance pruning step, and is particularly useful for the systems that serve for customers of diverse business objectives and business processes.

5.2 Algorithms for Level-two Relevance Pruning

Level-two relevance pruning algorithm takes the set of candidate information sources of level-one relevance as input, and prunes the information sources that do not offer enough query capability to contribute to the answer of Q. The decision is made based on the input and output arguments of Q, the user query profile of Q, and the query capability descriptions of the sources.

The process for level-two relevance pruning has two phases. In the first phase we prune the information sources (1) that have no input or output arguments, which are relevant to the arguments used in the user query, or (2) that have conflict with the interest of the user query (such as the query selection conditions do not match the access constraints of the sources), or (3) that have arguments corresponding to the mandatory input parameters of the user query but these arguments can only be used as input and are not included in the list of output arguments of the sources. The information sources selected in the first phase will be passed to the second phase where more sophisticated pruning is conducted in the process of generating an executable plan of the query. For example, the following two additional cases are handled: (4) We prune the information sources (say S_i) whose output capability are not enough to satisfy the input requirement of the other sources (say S_j) when an inter-site join from S_i to S_j is required. (5) We also prune those information sources whose mandatory input requirement is higher than the input arguments that the user query provides, and there are no other information sources executed earlier which would have enough output capability to complement such requirement.

The main difference between the two phases is the following: In the first phase we only check the obvious capability mismatches between the user query capacity description and each individual information source that cannot be complemented through using inter-site joins, such as those in cases (1), (2) and (3), whereas in the second phase, we further prune the information sources which cannot match the input requirements of a user query even through utilization of inter-site join with other possible sources. The first phase of the level-two relevance pruning considerably reduces the number of possibilities considered in generating executable query plans.

5.2.1 Phase 1: Level-two relevance pruning Algorithm

Figure 8 presents the details of the first phase of the level-two relevance pruning process, we refer to the algorithm for Phase 1 simply by **Level-two relevance pruning** to distinguish from the algorithm for Phase 2 which performs pruning during the process of generating executable plans. In the **Level-two relevance pruning** algorithm, for each candidate information source S, we check (1) if each input and output arguments of the given query Q has a correspondence in the source capability description of S. (2) if there is any inconsistency between the given query selection conditions and the access constraints of the source relations. (3) if each source selected has enough output capability that can satisfy the mandatory output requirements of Q.

Example 4 We illustrate these two algorithms on our example. Consider our query asking for books about "`cancer`" topic and published in 1998:

$$query(t,p,v,s) :- Book(b), Review(r), title(b,t), description(b,d), price(b,p), year(b,y), supplier(b,s), booktitle(r,t), review(r,v), y = 1998 \wedge \textsc{Contains}(d, `cancer').$$

When algorithm **Level-one Relevance Pruning** looks at the two unary subgoals of Q: $Book(b)$, $Review(r)$, and the set of source descriptions in Figure 6, Source 7 is pruned because its content is about technical reports and it has no correspondence with either $Book(b)$ or $Review(r)$. The following buckets are created: $Bucket[Book(b)]$ includes Sources 2, 3, 4, 5, 8 and $Bucket[Review(v)]$ includes Sources 1, 6.

Now let the algorithm **Level-two Relevance Pruning** look at the first binary subgoal $title(b,t)$ and one of the sources in $Bucket[Book(b)]$, say Source 2. Since $title(b,t)$ has a matching argument in Source 2

Algorithm Level-two Relevance Pruning$(Q, U, \mathcal{B}, \phi)$
Inputs: Q is a conjunctive query of the form $Q(\vec{X}) : -R_1(\vec{Y_1}), ..., R_m(\vec{Y_m}), F_Q$,
U is the user query profile of Q, of the form (Q_{scope}, Q_{capa}),
$\mathcal{B}$is the set of information sources of level-one relevance in $Bucket[q]$, $q = |Q_{scope}|$,
each information source is of the form $(S_{cat}, S_{cnt}, S_{qcp})$.
$\phi(R, S)$ is a mapping function that maps a binary subgoal name in Q
to an input or output argument name (say A) in the source S.
$\phi(R, S) = A$ only if $R = A$ or R is a substring of A.
Outputs: $Buckets[q]$, each holds a set of information sources of level-two relevance.

For each unary subgoal $R_k \in Q_{scope}$ $(k = 1, ..., q$ and $q < m)$ do:
For each binary subgoal $R_i \in Q_{capa}$ $(i \in \{q+1, ..., m\})$ do:
For each information source $S_j \in Bucket[R_k]$ do:
Let S_j be of the form: $T_1(\vec{Z_1}), ..., T_n(\vec{Z_n}), \bigwedge_{i=1}^{n} IC_i$, $(T_i(\vec{Z_i}), IC_i) \in S_{cnt}$;
If $\forall A_l \in S^j_{in} \cup S^j_{out}$, $\phi(R_i, S_j) \neq A_l$, and $\nexists B \in SY(R_i)$ such that $\phi(B, S_j) = A_l$
If $\phi(R_i, S_j) \neq S^j_{cat}$ and $\nexists B \in SY(R_i)$ such that $\phi(B, S_j) = S^j_{cat}$
If R_k is the virtual interface class of R_i
remove S_j from $Bucket[R_k]$, i.e., $Bucket[R_k] \leftarrow (Bucket[R_k] - \{S_j\})$, break;
else
If $Q_{cond}(R_i) \neq \emptyset \wedge \exists (T_h(\vec{Z_h}), IC_h) \in S^j_{cnt}$, $h \in \{1, ..., n\}$
such that $IC_h(A_l)$ is in conflict with $Q_{cond}(R_i)$ for $\phi(R_i, S_j) = A_l$
$Bucket[R_k] \leftarrow (Bucket[R_k] - \{S_j\})$, break;
else
If $R_i \in Q_{out} \wedge Bopt(R_i) = mandatory \wedge \phi(R_i, S_j) = A_l \notin S^j_{out}$ then
$Bucket[R_k] \leftarrow (Bucket[R_k] - \{S_j\})$, break;
End.

Figure 8: *Algorithm to prune the candidate information sources selected by level-one relevance pruning. It updates the set of buckets created by the level-one relevance pruning algorithm through removing the information sources whose query capabilities are not enough for them to participate in answering the query Q.*

and there is no conflict between them, thus Source 2 stays in the bucket. The algorithm continues on the next source in $Bucket[Book(b)]$. Once all the sources in $Bucket[Book(b)]$ are examined, the algorithm moves on to the next binary subgoal and proceeds the same examination.

Consider what happens when the algorithm looks at $year(b, y)$ and Source 1 in $Bucket[Review(v)]$, a conflict between the query selection condition "$year = 1998$" and the access constraint of Source 1 "$1970 \leq b\_publish\_year \leq 1980$" is found. Thus Source 1 is removed from $Bucket[Review(v)]$. Let the algorithm look at Source 5 and $year(b, y)$. Source 5 is removed from $Bucket[Book(b)]$ because there is no argument in $S^5_{in} \cup S^5_{out}$ which "matches" `year`, the input argument of the query. Similarly, when the algorithm looks at $supplier(b, s)$ and Source 3, it will remove Source 3 from $Bucket[Book(b)]$ because there is no argument in either $S^3_{in} \cup S^3_{out}$ or the source category description S^3_{cat}, which "matches" the query output argument `supplier` or one of its synonyms (i.e., `book store` or `publisher`).

Algorithm **Level-two Relevance Pruning** results in the following updated buckets: $Bucket[Book(b)]$ contains Sources 2, 4, 8 and $Bucket[Review(v)]$ contains Source 6.

5.2.2 Phase 2: Algorithm for Generating An Executable Query Plan

After the algorithm **Level-two Relevance Pruning**, we can start the process of generating executable query plan for a user query Q and try to perform further pruning by checking the cases (4) and (5) mentioned at the beginning of Section 5.2.

The first step is to try to find an ordering on the subgoals of Q that is executable. The main idea of finding such an ordering is to start from the built-in atoms of Q to find all the variables that are bound by values in Q, and then follow such variable(s) to find the first unary subgoal. This conforms well with the common tactics in conventional database query processing, such as performing selection before join. If more than one unary subgoals are found, the order should start with the unary subgoal that has the lowest selectivity factor. To deal with the situation that most of the information sources may not have their local statistics accessible to external systems, the selectivity factors used in this stage are a rough estimation by means of the following heuristics: the selectivity factor of a built-in comparison atom with one of the following operators "<=, <, >, >=" or "CONTAINS" is higher than the selectivity factor of the atom with the operator "=", but lower than the selectivity factor of the atom with "$\neq$" operator. This follows the well recognized fact that point query atoms (e.g., $a =$`v`) are generally more restrictive than range query atoms (e.g., $a \geq$`v`). If there is only one unary subgoal in the user query Q, then finding an ordering on the subgoals of Q is trivial.

Once we find the first unary subgoal, we can find the next unary subgoal by following the equating variable(s) in the subgoals. If there are more than one equating variables and they lead to more than one unary subgoals, then the most restrictive subgoal will be chosen first (i.e., the subgoal that have lowest selectivity factor in the Q).

Once found the partial order on the subgoals of Q, we can start to generate the executable query plan. During this process, we will further explore the pruning possibility. For instance, for each information source (say S_h) in the bucket of the first subgoal, if there is an information source (say S_l) in the bucket of the second subgoal, then we check (1) if the output of S_h matches the mandatory input requirement of S_l, and (2) if the mandatory input requirements of S_h and S_l are enough for the input requirements of the query Q. Figure 9 presents the details on generating an executable query plan and the pruning possibility during the query execution plan generation.

Example 5 We illustrate the algorithm for generating an executable query plan on our example query. First, by looking at the built-in atoms $year(b, y) \wedge y = 1998$ and $description(b, d) \wedge$CONTAINS$(d,$ `cancer`$)$", we will find the corresponding unary subgoal (e.g., $Books(b)$ in this case). Second, by putting $Books(b)$ as the first unary subgoal, we are able to find two binary subgoals that have the equating variable: $title(b, t)$ and $booktitle(r, t)$. Following $booktitle(r, t)$, we know that $Review(r)$ will be the next unary subgoal. We generate the following partial order for the unary subgoals of our query Q: $\langle Books(b), Review(r)\rangle$.

Now, for each information source in $Bucket[Book(b)]$, say Source 2, consider an information source in $Bucket[Review(b)]$ (Source 6 in this case). Before generating an executable plan for Q, we first verify the relevance of the two sources to the answer of Q: By looking at the mandatory input requirement of Source 2, i.e., S^2_{cond}: $title(b, t) \vee authors(b, a) \vee category(b, ct)$, the mandatory input requirement of Source 6, i.e., S^2_{cond}: $book\_title(w, bt) \wedge book\_authors(w, ba)$, and the input requirement of Q, i.e., Q_{cond}: $year(b,$`1998`$) \wedge description(b,$`'%cancer'`$)$, we can conclude that both Source 2 and Source 6 have enough capability to answer Q. Thus, the order of Source 2 followed by Source 6 is executable. We generate the first subquery, denoted by $\langle$[Source 2, Q(Source 2)], [Source 6, Q(Source 6)]$\rangle$, where Q(Source 2) $= (V^2_{in}, V^2_{out}, V^2_{cond})$, Q(Source 6) $= (V^6_{in}, V^6_{out}, V^6_{cond})$, and

- $V^2_{in} = \{title(b,t), year(b,y)\}$,
  $V^2_{out} = \{title(b,t), authors(b,a), price(b,p), supplier(b,s)\}$, $supplier(b,s) =$ 'Barnes&Nobel',
  $V^2_{cond} : y = 1998 \wedge$ CONTAINS$(t,$ 'cancer').

- $V^6_{in} = \{book\_title(r,t), book\_authors(r,a)\}$,
  $V^6_{out} = \{book\_title(r,t), book\_authors(r,a), review(r,v)\}$,
  $V^6_{cond} : (book\_title(r,t) =$ t1 $\wedge\ book\_authors(r,a) =$, a1), where t1, a1 denote values drawn from title and authors of the books selected from Source 2.

Note that by $\psi(S^6_{cond}, S_6) = \{book\_title(w,bt), book\_authors(w,ba)\}$, and $\psi(Q_{in}, S_6) = \{book\_title(w,bt), book\_year(w,by)\}$, we have $\psi(S^6_{cond}, S_6) - \psi(Q_{in}, S_6) = \{book\_authors(w,ba)\}$, which is not empty. It indicates that Source 6 requires both title and authors as the mandatory input arguments, even though query Q does not have authors in its input argument list. After checking out that the authors information that corresponds to $book\_authors(w,ba)$ is available in S^2_{out}, and it is not in $\psi(Q_{out}, S_2) = \{title(b,t), price(b,p), supplier(b,s)\}$, to enable an intersite-join between Source 2 and Source 6, the authors argument $authors(b,a)$ is added to V^2_{out}, the output list of Q(Source 2).

Similarly, a subquery is generated for Source 4 and Source 6, as well as for Source 8 and Source 6. Q(Source 4) $= (V^4_{in}, V^4_{out}, V^4_{cond})$ and Q(Source 8) $= (V^8_{in}, V^8_{out}, V^8_{cond})$ are described below:

- $V^4_{in} = \{title(b,t), year(b,y)\}$,
  $V^4_{out} = \{title(b,t), authors(b,a), price(b,p), supplier(b,s)\}$,
  where $supplier(b,s) =$ 'Morgan Kaufmann Publishers, Inc.',
  $V^4_{cond} : y = 1998 \wedge$ CONTAINS$(t,$ 'cancer') $\wedge$ CONTAINS$(abs,$ 'cancer').

- $V^8_{in} = \{title(b,t)\}$,
  $V^8_{out} = \{title(b,t), authors(b,a), price(b,p), year(b,y), store\_id(b,i)\}$,
  V^8_{cond} : CONTAINS$(t,$ 'cancer'). For books returned from Source 8, we select only those books that satisfy the condition "$year = 1998$".

Note that by $\psi(Q_{in}, S_8) = \{title(b,t), year(b,y)\}$, and and $S^8_{in} = \{title(b,t), authors(b,a), store\_name(s,n)\}$, $\psi(Q_{in}, S_2) - S^2_{in} = \{year(b,y)\}$ which is not empty. It indicates that Source 8 does not allow external queries on a specific year. Since Source 8 is survived from the level-two relevance pruning algorithm, therefore the year information must be available as an output argument of Source 8. By algorithm **Generating Executable Query Plan**, $year(b,y)$ is added into V^8_{out}, the output list of Q(Source 8). Of course, we have do the selection on year ourselves for those books returned from Source 8, before asking for reviews from Source 6.

Note that, although from $Bucket[Book(b)]$ and $Bucket[Review(r)]$, we have total number of 4 information sources that are relevant (i.e., Sources 2, 4, 6, 8), not all the possible enumerations are semantically correct query plans, because only those plans that follow the subgoal ordering: $\langle Books(b), Review(r)\rangle$, computed by algorithm **Generating Executable Query Plan**, are executable. The following three independent subqueries are the executable query plans generated by the algorithm **Generating Executable Query Plan**:

1. $\langle$[Source 2, Q(Source 2)], [Source 6, Q(Source 6)]$\rangle$,
2. $\langle$[Source 4, Q(Source 4)], [Source 6, Q(Source 6)]$\rangle$,
3. $\langle$[Source 8, Q(Source 8)], [Source 6, Q(Source 6)]$\rangle$.

The answer to our example query is the union of these three subqueries, to be performed through the query result packaging and assembly process. A collection of semantic attachment mechanisms are used

in the result assembly phase to handle the representational and semantic heterogeneity in the subquery results returned from different information sources. This subject is beyond the topic of this paper. Readers who are interested in the detail may refer to [14].

This example shows that our algorithms for query routing effectively prune the information sources that do not contribute to a user query Q. As a result, not only the number of candidate rewritings considered for query planning is considerably reduced, but we also avoid unnecessary overhead of contacting those information sources that cannot contribute to the answer of Q.

5.3 Algorithm for Level-three Relevance Pruning

Although the level-one and level-two relevance pruning algorithms considerably reduce the number of information sources to be accessed, there are still cases where using run-time information may drastically reduce the number of sources to be contacted in producing a complete answer to a query.

Example 6 Suppose we are interested in finding the telephone number of bookstores in "Portland, OR", we may ask query Q: "*get the name and the phone-number of all book stores in* `Portland, OR`". The interface classes of interest to us are local yellow book directories, area code and phone directories that can be searched by location. The parameters of interest include store type, store name, phone No, area code and area name. This query may be expressed as follows:

$$\begin{aligned} query(sn, ph) :- \; & ACDirectory(ad), LocalDirectories(ld), phoneNo(ld, ph), store\_name(ld, sn), \\ & store\_type(ld, st), areaCode(ld, ac), area(ad, a), code(ad, ac), \\ & a = \text{`Portland, OR'} \wedge \textsc{Contains}(st, \text{`book store'}). \end{aligned}$$

Suppose that we currently have n (say $n \geq 1000$) directory information sources accessible. Also suppose that the user query profile has got the information that only directories that are related with phone and book stores are interested. By using the level-one and level-two relevance pruning algorithms, we can reduce the number of sources to m (say $m \geq n$). It is clear that without having the area code of "`Portland, OR`" available at the query compile time, the query plan would have to treat all other directory information sources, which are outside the `503` area code, as relevant to the query, even though they do not at all contribute to the answer of this query. However, once we have the subquery evaluated,

$$query(ac) :- ACDirectory(ad), area(ad, a), code(ad, ac), a = \text{`Portland, OR'}.$$

the binding for area code ac (in this case just 503) can be used to regulate the set of relevant directory information sources to those that have telephone listings only in the `503` area code.

Obviously, when the number m of candidate sources is large with respect to the actual number of information, it becomes critical for the query processing system to take advantage of run-time information, such as the intermediate result of evaluating the subquery for area code based on the location information in this example. The key issue here is how to find the right subqueries that have great reduction power and can be used to effectively prune many of the irrelevant information sources.

Consider our query in Example 6. The first two steps of query routing – through the compile-time analysis, will prune those information sources that do not contain telephone directory information and

those directories that do not allow to use location as input argument. However, without information on area code for 'Portland, OR', we will not be able to prune the rest of m Directories information sources, even though we know the majority of the m information sources are not contributing to the answer of Q. If we use the executable query plan generated based on the m information sources, the answer to Q would still be quite ineffective since we would need to contact each of the m directory information sources, and in the worst case where there is one information source that contains the complete answer to Q (i.e., the complete information about book stores in "Portland, OR"), the work for contacting the other $m-1$ sources will be a total waste.

One obvious solution is to examine the selection constraints (the input parameters) of Q and select the one(s) that have lowest selectivity and that need a relatively small number of sources to produce the complete answer. If such selection constraints exist, then we can find an equivalent query Q' (i.e., $answer(Q) = answer(Q')$) by means of query rewriting, such that $Q' = \bowtie_F (Q'_1, Q'_2)$ and Q'_1 is satisfiable and can be evaluated before Q'_2 to allow the use of output of Q'_1 to reduce the search space accessed to answering Q'_2, and thus the overall search space for answering Q can be reduced substantially. This is the primary focus of the algorithm for level-three relevance pruning. Due to the space limitation, we omit the algorithm in this paper and present the flavor of the level-three relevance pruning algorithm through an example. Readers may refer to our technical report [14] for detail.

Recall our query in Example 6, of the two input arguments of Q, the most restrictive one is the atom that requires area to be equal to "Portland, OR". If we (the user) also know that only one source (e.g., the yellow book directory in this case) may be needed to produce the complete answer once the area code is known, then after level-two relevance pruning process, we can simply plug in the level-three relevance pruning algorithm and proceed the following steps:

- (1) Apply the algorithm to decompose the query into the following two dependent subqueries:

  $Subquery_1(ac)$:– ACDirectory(ad), $area(ad, a)$, $code(ad, ac)$, $a =$ '$Portland, OR$'.

  $Subquery_2(sn, ph)$:– $LocalDirectories(ld)$, $phone(ld, ph)$, $store\_name(ld, sn)$, $store\_type(ld, st)$, $areaCode(ld, ac)$, $ac = Subqquery_1(ac) \wedge st =$ '$bookstore$'.

- (2) Use the algorithm **Generating Executable Query Plan** to create an executable query plan for $Subquery_1$, obtain the result of $Subquery_1$, and add the result of $Subquery_1$ as the additional selection atom in $Subquery_2$;

- (3) Apply the level-one and level-two relevance pruning algorithms on $Subquery_2$ and the m information sources selected earlier to further prune away the irrelevant ones. Then run the algorithm **Generating Executable Query Plan** on the information sources that are survived from the step (2) pruning, and find an executable query plan for $Subquery_2$.

6 Conclusion

We have described the notation and issues of query routing, and present a practical solution for designing a scalable query routing system based on multi-level progressive pruning strategies. One of the key ideas is to create user-specific ontology to capture what users want (namely user query profiles), and source-specific ontology to capture what the sources can offer (namely source capability profile), and then to provide mechanisms that can dynamically discover relevant information sources for a given query

through the smart use of user query profiles and source capability profiles, including the mechanisms for interleaving query routing with query parallelization and query execution process to continue the pruning at run-time.

This paper has three main contributions. First, our approach, to the best of our knowledge, is the first one that identifies and implements the coordination of query refinement powered by user query profile with capability-sensitive source selection powered by the multi-level progressive irrelevance pruning algorithms. Second, our approach emphasizes the importance of creating and maintaining user query profiles and source capability profiles independently to enhance the flexibility and scalability of the query routing process with respect to the dynamic change of available information sources. The third and interesting contribution of the paper is the multi-level progressive pruning strategies and the set of query routing algorithms to discover and prune effectively the information sources that do not contribute to a query. A novel aspect of our algorithms is the step-wise utilization of the user query profile and the source capability descriptions to implement the multi-level relevance pruning and to ensure that each step considerably reduces the number of candidate rewritings considered in the next step. Comparing with the keyword-based indexing techniques adopted in many of the search engines and search software outlined in Section 3, our query routing approach offers fine-granularity of interest matching and reduces many false positives and false negatives in the early stage of the distributed query processing, thus it is more powerful and effective for handling queries with complex conditions.

Our research on query routing continues alone several directions. On the practical side, a prototype of the query routing algorithms is being implemented on top of the DIOM (www.cse.ogi.edu/DISC/DIOM) system with Oracle 8.0 as the back-end service repository support. The first demonstration package is implemented as a WWW application. All the components and its interface programs are created using Perl CGI-scripts [16, 24] or Java programs. We are actively working towards experimentally evaluating our query routing algorithms with more than 100 information sources on the WWW. Second, we are interested in exploring tools for automating the process of extracting source capability profiles that are currently created and maintained manually. On the theoretical side, we are working on extending our query routing framework to allow a seamless incorporation of other richer domain-specific ontology, thesaurus, and document-frequency based query refinement mechanisms available on-line into our multi-level pruning process, and to add ability of inferring that a source is relevant to a query with some degree of likelihood. For instance, if we could use some sort of domain-specific thesaurus or ontology to infer that the constant `'cancer'` is a term in medical and health science, then we can further prune away the information sources that have only books on, for example, computer science and engineering subject.

Acknowledgement

This work was carried out under the DIOM project. I thank the DIOM implementation team, especially David Buttlers, Yooshin Lee, and Kirill Richine, for their discussions and implementation endeavor. My thanks are also due to Calton Pu and the Continual Queries project team at OGI for many interesting discussions on technical issues related to DIOM.

References

[1] Y. Arens and et al. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[2] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, M. F. Schwartz, , and D. P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, March 1995. http://newbruno.cs.colorado.edu/harvest/papers.html.

[3] M. Carey, L. Haas, P. Schwarz, and et al. Towards heterogeneous multimedia information systems: the garlic approach. In *IEEE Int. Workshop on Research Issues in Data Engineering*, 1995.

[4] D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 1995.

[5] L. Gravano, C. Chang, and H. Garcia-Molina. Starts: Stanford proposal for internet meta-searching. In *SIGMOD'97*, 1997.

[6] L. Gravano and H. Garcia-Molina. Generalizing gloss to vector-space databases and broker hierarchies. In *VLDB'95*, 1995.

[7] L. Gravano, H. Garcia-Molina, and A. Tomasic. The effectiveness of gloss for text database discovery problem. In *Proceedings of ACM SIGMOD*, 1994.

[8] Inktomi Corp. The technology behind hotbot. White Paper, http://www.inktomi.com/, 1996.

[9] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers. In *Technical Report TMC-199, Thinking Machines, Inc., Version 3*, 1994.

[10] Y. Lee. Rainbow: A prototype of the diom interoperable system. MSc. Thesis, Department of Computer Science, University of Alberta, July, 1996.

[11] A. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, 1996.

[12] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, 1996.

[13] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability based mediation in tsimiss. In *Proceedings of ACM SIGMOD Conference*, 1997.

[14] L. Liu. Query routing: An intelligent query service for accessing heterogeneous information sources. Technical report, TR97-10, Department of Computing Science, University of Alberta, Edmonton, Alberta, Feb. 1997.

[15] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous databases. In *Proceedings of the International Conference on Coopertive Information Systems*, Brussels, June 19-21 1996.

[16] NCSA HTTPd Development Team. The common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi.

[17] J. Ordille and B. Miller. Distributed active catalogs and meta-data caching in descriptive name services. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 120–129, 1993.

[18] A. Rajarman, Y. Sagiv, and H. Ullman. Answering queries using templates with binding patterns. In *Proceedings of PODS*, 1995.

[19] K. Richine. Distributed query scheduling in the context of diom: An experiment. MSc. Thesis, Department of Computer Science, University of Alberta, April, 1997.

[20] M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proceedings of the 3rd International Conference on World Wide Web*, 1996.

[21] M. A. Sheldon, A. Duda, R. Weiss, J. W. O'Toole, and D. K. Gifford. Content routing for distributed information servers. In *Proceedings of the 4th Int. Conf. on Extending Database Technology (EDBT)*, 1994.

[22] J. D. Ullman. *Principles of Database and Knowledge-based Systems.* Computer Science Press, 1989.

[23] V. Vassalos and Y. Papakonstantinou. Describing and using the query capabilities of heterogeneous sources. In *Proceedings of VLDB Conference*, 1997.

[24] L. Wall and R. L. Schwartz. *Programming Perl.* O'Reilly and Associates, Jan. 1991.

[25] R. Weiss, B. Velez, M. A. Sheldon, C. Namprempre, P. Szilagyi, and D. K. Gifford. Hypursuit: A hierarchical network search engine that exploits content-link hypertext clustering. In *Proceedings of the 7th ACM Conf. on Hypertext*, 1996.

Algorithm Generating Exectuable Query Plan$(Q, U, \mathcal{B})$
Inputs: Q is a conjunctive query of the form $Q(\vec{X}) :- R_1(\vec{Y_1}), ..., R_m(\vec{Y_m}), F_Q$, and
is described by a triplet $(Q_{in}, Q_{out}, Q_{cond})$,
U is the user query profile of Q, of the form (Q_{scope}, Q_{capa}),
$\mathcal{B}$ is the set of information sources in $Bucket[q]$, $q = |Q_{scope}|$, each described by a triplet $(S_{cat}, S_{cnt}, S_{qcp})$.
Outputs: an executable plan of Q, denoted by $ExePlan(Q)$.
$ExePlan(Q) = \{\langle [S_i, Q(S_i)], [S_j, Q(S_j)]\rangle \mid S_i, S_j \in \bigcup_{i=1}^{q} Bucket[R_i]$, $Q(S_i) = (V^i_{in}, V^i_{out}, V^i_{cond})$, $i \in \{1, ..., t\}, t = |\mathcal{B}|\}$

If $q = 1$, i.e., there is only one unary atom in Q,
For each source S_j in $Bucket[R_q]$,
Let ψ be defined as a transformation function that translates query arguments in Q into the corresponding source input or output arguments in S_j;
Create a subquery $Q(S_j)$ by computing $(V^j_{in}, V^j_{out}, V^j_{cond})$, i.e.,
$V^j_{in} = \psi(Q_{in}, S_j)$, $V^j_{out} = \psi(Q_{out}, S_j)$, $V^j_{cond} = \psi(Q_{cond}, S_j)$;
else For each atom in Q_{cond} that is bounded by values, do:
Set $SF(\alpha) = 1/10$ if the atom is of the form "$\alpha = a$",
Set $SF(\alpha) = 1$ if the atom is of the form "$\alpha \neq a$", Set $SF(\alpha) = 1/3$ otherwise;
For each unary atom $R_i \in VI$ of Q_{scope}, $(i = 1, ..., q)$, do:
compute $SF(R_i)$ by multipling all the $SF(\alpha)$ for $\alpha \in Att(R_i)$;
Sort all the $R_i \in VI$, $i = 1, ..., q$, by SF in an ascending order, i.e., $ASort(VI, SF)$;
Let the sorting result be the partial order of $\{\langle R_{i_l}, R_{i_k}\rangle \mid SF(R_{i_l}) \leq SF(R_{i_k}), i_l, i_k \in \{1, ..., q\}\}$;
Set $E^i_{out} = \emptyset$, $E^i_{cond} = \emptyset$, $G^i_{in} = \emptyset$ $(i = 1, ..., t = |\mathcal{B}|)$;
For each R_{i_j} in the above partial order $\{\langle R_{i_l}, R_{i_k}\rangle\}$, $i_j \in \{1, ..., q-1\}$, do:
For each $R_{i_k}, k \in \{1, ..., j-1, j+1, ..., q\}$ such that
there is one or more equating variable(s) between R_{i_j} and R_{i_k}
If $Bucket[R_{i_j}] \cap Bucket[R_{i_k}] = \emptyset$, do:
For each information source $S_h \in Bucket[R_{i_j}]$ do:
For each information source $S_l \in Bucket[R_{i_k}]$ do:
If **PruningBeforeGenPlan**$(S_h, S_l) = -1$, break;
else i.e., when $Bucket[R_{i_j}] \cap Bucket[R_{i_k}] \neq \emptyset$, do:
For each information source $S_h \in Bucket[R_{i_j}]$ do:
For each information source $S_l \in Bucket[R_{i_k}]$ do:
If $S_h = S_l$
$V^h_{in} = \psi(G^h_{in}, S_h)$, $V^h_{out} = \psi(Q_{out}, S_h)$, $V^h_{cond} = \psi(Q_{cond}, S_h)$;
Add $\langle [S_h, Q(S_h)], nil\rangle$ to the executable query plan $ExePlan(Q)$;
else If **PruningBeforeGenPlan**$(S_h, S_l) = -1$, break;
End.

Int **PruningBeforeGenPlan**(S_h, S_l)
BeginProcedure
If $\psi(S^h_{cond}, S_h) - \psi(Q_{in}, S_h) \neq \emptyset$
remove S_h from $Bucket[R_{i_j}]$, i.e., $Bucket[R_{i_j}] \leftarrow (Bucket[R_{i_j}] - \{S_h\})$, return -1;
For each input argument α involved in $\psi(S^l_{cond}, S_l) - \psi(Q_{in}, S_l)$,
If there is a corresponding argument α' in S^h_{out}, then add α' in E^h_{out},
else (i.e., S^h_{out} is not enough to cover the mandatory arguments in S^l_{cond})
remove S_h from $Bucket[R_{i_j}]$, i.e., $Bucket[R_{i_j}] \leftarrow (Bucket[R_{i_j}] - \{S_h\})$, return -1;
Set $E^l_{cond} = E^l_{cond} \cup \{\alpha = a | \alpha \in V^l_{in}, a \in answer(Q, S_h)\}$;
For $i = h, l$ do:
If $\phi(Q_{in}, S_i) - S^i_{in} = \emptyset$ then set $G^i_{in} = G^i_{in} \cup \phi(Q_{in}, S_i)$
else set $G^i_{in} = G^i_{in} \cup (\phi(Q_{in}, S_i) - (\phi(Q_{in}, S_i) - S^i_{in}))$, $E^i_{out} = E^i_{out} \cup (\phi(Q_{in}, S_i) - S^i_{in})$,
Create a subquery $Q(S_i)$ by computing $(V^i_{in}, V^i_{out}, V^i_{cond})$, i.e.,
¿$V^i_{in} = G^i_{in}$, $V^i_{cond} = \psi(Q_{cond}, S_i) \cup E^i_{cond}$, $V^i_{cond} = \psi(Q_{cond}, S_i) \cup E^i_{cond}$;
Add $\langle [S_h, Q(S_h)], [S_l, Q(S_l)]\rangle$ to the executable query plan $ExePlan(Q)$; return 0;
End Procedure

Figure 9: *Algorithm for generating an executable plan for answering the query Q.*