Predicate Rewriting for Translating Boolean Queries in a Heterogeneous Information System

Chen-Chuan K. Chang and Héctor García-Molina and Andreas Paepcke
Stanford University

Searching over heterogeneous information sources is difficult in part because of the non-uniform query languages. Our approach is to allow users to compose Boolean queries in one rich front-end language. For each user query and target source, we transform the user query into a subsuming query that can be supported by the source but that may return extra documents. The results are then processed by a filter query to yield the correct final results. In this paper we introduce the architecture and associated mechanism for query translation. In particular, we discuss techniques for rewriting predicates in Boolean queries into native subsuming forms, which is a basis of translating complex queries. In addition, we present experimental results for evaluating the cost of post-filtering. We also discuss the drawbacks of this approach and cases when it may not be effective. We have implemented prototype versions of these mechanisms and demonstrated them on heterogeneous Boolean systems.

Categories and Subject Descriptors: D.2.12 [**Software Engineering**]: Interoperability; H.2.3 [**Database Management**]: Languages—*query languages*; H.2.5 [**Database Management**]: Heterogeneous Databases; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation, search process*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*performance evaluation (efficiency and effectiveness)*; H.3.7 [**Information Storage and Retrieval**]: Digital Libraries—*Systems issues*

General Terms: Algorithms, Languages, Experimentation, Measurement

Additional Key Words and Phrases: Boolean queries, content-based retrieval, query translation, predicate rewriting, query subsumption, filtering

This material is based upon work supported by the National Science Foundation under Cooperative Agreement IRI-9411306. Funding for this cooperative agreement is also provided by DARPA, NASA, and the industrial partners of the Stanford Digital Libraries Project.
Name: Chen-Chuan K. Chang
Address: Electrical Engineering Department, Stanford University, Stanford, CA 94305; email: changcc@cs.stanford.edu
Name: Héctor García-Molina and Andreas Paepcke
Address: Computer Science Department, Stanford University, Stanford, CA 94305; email: {hector, paepcke}@cs.stanford.edu

1. INTRODUCTION

Emerging Digital Libraries can provide a wealth of information. However, there are also a wealth of search engines behind these libraries, each with a different document model and query language. Our goal is to provide a front-end to a collection of Digital Libraries that hides, as much as possible, this heterogeneity, and that enables distributed search over them.

As a first step, we have focused on supporting Boolean queries [Salton 1989; Frakes and Baeza-Yates 1992; Faloutsos 1985] at the front-end. The Boolean query model may not be favorable in all situations; in particular, it does not produce ranked output [Harman 1993]. However, most current commercial online services (such as *Dialog*, *BRS*, *Lexis-Nexis*, *Orbit*, *STN*, *etc.*) as well as traditional library systems (such as those at universities) support the Boolean query model to access their text databases offering well-maintained information in fields such as science, business, and law. Furthermore, more and more web search engines are adopting Boolean queries in their "advanced" interfaces (*e.g.*, *AltaVista*, *WebCrawler*, and *Excite*). Therefore, we believe that supporting the Boolean model is critical for providing integrated access to those modern or legacy systems, in order to make available their valuable contents. We are also extending our work to the vector space model – see Section 7.

We adopt the approach of supporting integrated access to heterogeneous systems through an intelligent front-end system responsible for query mapping and post-filtering. The front-end provides a powerful query language that may not be fully supported by the underlying systems. Users do not access the underlying services directly; instead they submit queries to the front-end. The front-end translates the user queries into (native) *subsuming queries* that are supported by the target systems but that may return extra documents. This translation allows the queries to be evaluated by multiple services in parallel. Because the preliminary results may contain extra documents that the users did not ask for, the front-end also generates *filter queries* to process the preliminary results locally and produce the final answers. (Of course, front-end translation and filtering have also been used in related areas. See Section 2.) The following examples illustrate our approach.

Example 1. Suppose that a user is interested in documents discussing multiprocessors and distributed systems. Say the user's query is originally formulated as

User Query: Title Contains multiprocessor AND distributed (W) system

This query selects documents with the three given words in the Title field; furthermore, the W proximity operator specifies that the word "distributed" must immediately precede "system."

Assume the user wishes to query the *Inspec* database managed by the Stanford University *Folio* system. Unfortunately, this source does not understand the W operator. In this case, our approach approximates the predicate "distributed (W) system" by the closest predicate supported by *Folio*, "distributed AND system." This predicate requires that the two words appear in matching documents, but in any position. Thus, the subsuming query (written in *Folio*'s syntax) sent to *Folio-Inspec* is

Subsuming Query: Find Title multiprocessor AND distributed AND system

The subsuming query will return a preliminary result set that is a superset of what the user expects. Therefore, the front-end needs an additional post-filtering step to eliminate (from the preliminary results) those documents that do not have the words "distributed" and "system" occurring next to each other. Therefore, the required filter query is

Filter Query: Title Contains distributed (W) system

Example 2. To illustrate a little more complexity, let's suppose the user is interested in the documents with the exact title "gone with the wind." The query is formulated as follows:

User Query: Title Equals "gone with the wind"

Searching for exact values of a field is quite common in library citation systems such as *Folio*. However, this feature is not available in, say, Dialog Corporation's *Dialog* system, a commercial information provider. Specifically, the Title field in *Dialog* can only be searched using an expression consisting of individual indexed words. Since the complete phrase values are not indexed, one can only test the Title field through the Contains operator instead of Equals. Therefore, our first attempt at translation could be:

Subsuming Query: Title Contains gone (W) with (W) the (W) wind

Unfortunately, this query will surely return zero hits from *Dialog*, because it contains the stopwords "with" and "the." Therefore, a correct translation must remove these stopwords from the expression which then yields:

Subsuming Query: Title Contains gone (2W) wind (or gone (2W) wind / Ti, in *Dialog* syntax)

This means that there are (at most) two words in between "gone" and "wind." Post-filtering is again required as the translation gives a native query broader than the user query. Because the user query consists of only one predicate which may not be satisfied in the native query, the filter query is simply the user query in its entirety:

Filter Query: Title Equals "gone with the wind"

Figure 1 shows some of the main components of the proposed front-end system, specifically illustrating the query translation process. Users interact with the front-end interface and formulate queries in a powerful language that provides the combined functionality of the underlying sources. The figure shows how the query is then processed before being sent to a target source; if the query is intended for multiple sources, the process can be repeated (or done in parallel). First, the query is parsed into a tree of operators. Then the operators are compared against the capabilities and document fields of the target source. The operators are mapped to ones that can be supported and the query tree is transformed into the native query tree (which subsumes the user query) and the filter query tree. Using the syntax of the target, the native query tree is translated into a native query and sent to the source. After the documents are received and parsed according to the syntax for source documents, they are processed against the filter query tree, yielding the final answer.

Even though heterogeneous search engines have existed for over 20 years, the approach we advocate here, full search power at the front-end with appropriate query transformations, has not been studied in detail. The main reason is that our approach has a significant cost, *i.e.*, documents that the end users will not see have

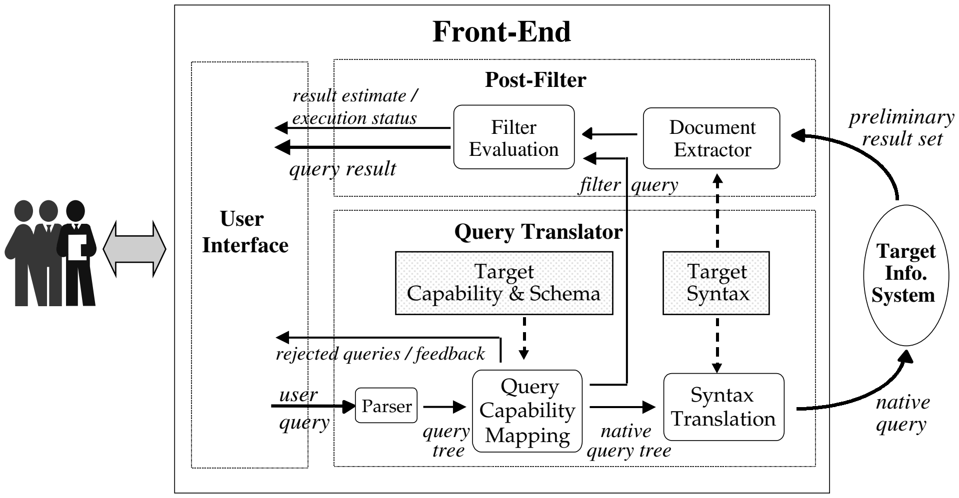


Fig. 1. The (partial) architecture of the front-end system illustrating query translation and post-filtering. The shaded boxes represent metadata defining the target's syntax and capabilities.

to be retrieved from the remote sites. This involves more work for the sources, the network, and the front-end. It may also involve higher dollar costs if the sources charge on a per-document basis. Because of these costs, other alternatives have been advocated in the past for coping with heterogeneity. They generally fall into four categories (see Section 2 for more details):

(1) Eliminate the heterogeneity by standardization of the query languages;
(2) Present inconsistent query capabilities specific to the target systems with no intention to hide the heterogeneity, and have the end users write queries specifically for each;
(3) Provide a "least common denominator" front-end query language that can be supported by all sources;
(4) Copy all the document collections that users may be interested in to a single system that uses one search engine and one language.

While these alternatives may be adequate in some cases, we do not believe they scale well and are adequate for supporting a truly globally distributed Digital Library. End users really require powerful query languages to describe their information needs, and they do require access to information that is stored in different systems. At the same time, increasing computer power and network bandwidths are making the full front-end query power approach more acceptable. Furthermore, many commercial sources are opting for easy-to-manage broad agreements with customers that provide unlimited access. Thus, in many cases it may not be that expensive to retrieve additional documents for front-end post-filtering. Even if there is a higher cost, it may be worth paying the cost to get users the required documents with less effort on their part.

In summary, given the benefits of full query power, we believe that it is at least worth studying this approach carefully. A critical first step is understanding how query translation actually works and when it does not. Furthermore, because the post-filtering cost is of major concern in our approach, we performed experiments to quantify the overhead. In [Chang et al. 1996a; Chang et al. 1996b] we gave an overview of the translation process. We noted that predicates composing user queries must be rewritten to be acceptable to the target sources. We then explained how the resulting pieces are combined, and how a filter query is derived. In this paper we focus on the predicate rewriting process (not covered in [Chang et al. 1996a; Chang et al. 1996b]) which is at the heart of the approach. In addition, this paper also summarizes some of the results of our cost evaluation experiments. As we will see, in many cases the overhead is reasonable (reference [Chang and García-Molina 1997] provides more details). Although this paper does not discuss implementation details for the algorithms, we do note that the algorithms presented here have been implemented and used to transform queries for systems such as *Dialog* (Dialog Corporation), Stanford's *Folio*, DEC's *AltaVista*, *WebCrawler*, and *NCSTRL* (an on-line library of computer science technical reports), each with different Boolean query syntax and functionality.

Note that this paper concentrates specifically on the query translation process, and not on other related important problems. In particular, as a consequence of distributed search (*i.e.*, *meta-search*) over heterogeneous systems, many other challenging front-end issues also arise. For instance, we need a way to find potentially relevant sources for searching [Gravano et al. 1994], a flexible payment mechanism [Ketchpel et al. 1997] to handle on-line shopping (of information), *etc.* We and others have investigated these issues, but they are clearly not in the scope of this paper, and therefore are not shown in Figure 1. In addition, many issues need to be revisited in a distributed environment. In particular, it is critical to support a good user interface that integrates various service components and interacts with users in the process of meta-searching, namely, formulating queries, browsing and clustering results. For instance, *SenseMaker* [Baldonado and Winograd 1997], one of our front-end interface, employs *CSQuest* (accessible at `http://ai.bpa.arizona.edu/html/mcsquest/`, developed in the Illinois Digital Libraries project) for term suggestion. While we are not able to discuss interface issues here, we do note that the query translator can provide feedback information for user interaction, as discussed in Section 7.

In addition, several important issues directly related to query translation are not covered in our work yet. For one, currently we do not consider semantic mapping issues for query terms (*e.g.*, mapping "fault tolerant systems" to "reliable systems"). This can be very important for systems that employ controlled vocabulary. Similarly, the general problem of the semantic mapping of fields (*e.g.*, mapping Author to Creator) is another major barrier to distributed search over very different sources. Such semantic mapping problems are themselves still major open research issues.

Furthermore, our study shows that in some cases the approach can have serious drawbacks. For instance, it may be hard or even impossible to obtain the "metadata" that the algorithms need, such as source vocabularies. In addition, in some rare cases there may not be an appropriate translation, in which case the native queries degenerate to *True*. Another problem is that, to simulate unsupported features such as stemming, the algorithms may create queries containing too many enumerated terms. We discuss these issues and how we may cope with them in Section 7.

We start by briefly reviewing the alternative approaches suggested for accessing heterogeneous search engines. In Section 3 we describe the formal language used at the front-end and its semantics, while in Section 4 we provide a brief overview of the query translation process. Section 5 then describes the rewriting of single predicates in detail. Finally, Section 6 summarizes the experimental results for evaluating the post-filtering overhead.

2. RELATED WORK

The problem of multiple and heterogeneous on-line information retrieval (IR) systems has been observed since the early 1970's. In 1973, T.H. Martin made a thorough comparative feature analysis of on-line systems to encourage the unification of search features [Martin 1974]. Since then, many solutions have been proposed to address the heterogeneity of IR systems. Obviously, one solution is standardization, as suggested by the development of the Common Command Language (CCL) done by Euronet [Negus 1979], Z39.58 [National Information Standards Organization 1993], and ISO 8777 [ISO 1993]. However, none of them has been well accepted as an IR query standard.

Another approach for accessing multiple databases transparently is through the use of *front-ends* or *intermediary systems*, which is also the approach that we advocate. This approach does not enforce any standard that requires the cooperation of the underlying services and thus maintains their autonomy. Reference [Williams 1986] and [Hawkins and Levy 1985] provide overviews of these systems. Like ours, these front-end systems provide automated and integrated access to many underlying sources. However, unlike ours, none of them tried to support a uniform yet comprehensive query language by post-filtering. As we mentioned in the preceding section, their approaches generally fall into the following categories.

The first approach is to present non-uniform query capabilities specific to the target services. As the user moves from one service to another, the capabilities of the system are modified automatically to reflect specific limitations. Examples of such systems are *TSW* [Preece and Williams 1980], OCLC's *Intelligent Gateway Service* [Zinn et al. 1986], and the more recent internet search services such as the *All-in-One Search* (`http://www.albany.net/allinone/`). This kind of system actually does not provide transparent access to multiple sources. The user must be aware of the capability limitation of the target systems and formulate queries for each. It is therefore impossible to search multiple sources in parallel with a single query, since it may not be interpretable by all of them.

The second approach is to provide a simple query language, the least common denominator, that can be supported by all sources. Most front-end systems adopt this approach. Examples include *CONIT* [Marcus 1982], *OL'SAM* [Toliver 1982], and *FRED* [Crystal and Jakobson 1982]. These systems unify query functionality at the expense of masking some powerful features available in specific sources. To use particular features not supported in the front-ends, the user must issue a query in the "pass-through" mode, in which the query is sent untranslated. This again compromises transparency.

Along this line, another popular technique for dealing with language heterogeneity is for the front-end to use some form of natural language (*e.g.*, vector-space) queries. As discussed in Section 1, we decided to support Boolean queries because they are used by most of the systems we are interested in. Consequently, to access those Boolean systems, if the front-end supports natural-language queries, it must first convert a natural language query to some Boolean format (*e.g.*, as the conjunction or proximity of keywords found in the query), and then translate it into the target (Boolean) query languages. In other words, we still need a technique for translating Boolean queries.

Finally, there are systems that manage and resell multiple collections and do the search by themselves. For example, Dialog Corporation's *Dialog* system manages over 450 databases from a broad scope of disciplines. Clearly, this centralized approach does not scale well as the amount of information keeps increasing.

Closest to our work, in terms of the shared goal, are the evolving meta-searchers on the internet, such as *MetaCrawler* (`http://metacrawler.cs.washington.edu/`) [Selberg and Etzioni 1995] and *SavvySearch* (`http://guaraldi.cs.colostate.edu:2000/`). These services provide a single, central interface for Web document searching. They represent the meta-searchers which use no internal databases of their own and instead rely on external search services (*e.g.*, *WebCrawler*, *Lycos*) to answer user queries. Like ours, they also do query mapping and (optional) post-filtering. However, they provide relatively simple front-end query languages that are only slightly more powerful than the least common denominator supported by the external sources. For example, they support a subset of Boolean queries instead of arbitrary ones.

Furthermore, information integration has long been recognized as a central problem of modern database systems, with the goals to query legacy systems, to cope with semantic or schematic inconsistency, and to handle unstructured data [Ullman 1997]. Our front-end architecture is consistent with the notion of *mediators* [Wiederhold 1992], which has been widely adopted in information integration efforts such as TSIMMIS [García-Molina et al. 1995] and Information Manifold [Kirk et al. 1995]. In addition, the approach of (query) subsumption and post-filtering have been generally applied in complementing the lack of full capabilities in query processing (*e.g.*, [Papakonstantinou et al. 1995]); similar idea can also be found in the work on signature files [Salton 1989; Faloutsos 1985], which may generate "false drops" at initial processing.

However, compared to the work on information integration, our contribution is unique in the following aspects.

—We study selection queries (in terms of relational algebra [Ullman 1988]), *i.e.*, arbitrary Boolean combination of predicates to be evaluated over individual sources, which are exactly the type of queries used in Boolean IR systems. We have developed the algorithms that generate minimal subsuming and filter queries for this type of queries [Chang et al. 1996a; Chang et al. 1996b]. In contrast, the above-mentioned work focuses on selection-join queries that consist of only conjunctive predicates (*i.e.*, without disjunction and negation).

—To the best of our knowledge, the related efforts in information integration assume a fixed set of "uninterpreted" predicates, in the sense that a predicate is either supported by a target system, or not at all. In contrast, in this paper, we discuss techniques for semantically rewriting predicates when they are not fully supported, rather than dropping them blindly. This certainly gives a better

Syntactic Construct				Associated Semantics
1. Query Tree				
(1.1)	*Query*	:=	$Query_1$ OR $Query_2$	$\langle Query_1\rangle \cup \langle Query_2\rangle$
(1.2)		|	$Query_1$ AND $Query_2$	$\langle Query_1\rangle \cap \langle Query_2\rangle$
(1.3)		|	$Query_1$ NOT $Query_2$	$\langle Query_1\rangle - \langle Query_2\rangle$
(1.4)		|	*Pred*	$\langle Pred\rangle$
2. Predicate Subtree				
(2.1)	*Pred*	:=	Contains(Field, WPat)	$\{D \mid \exists(D; \{s_{F1}, s_{F2}, \cdots, s_{Fk}\}) \in \langle \text{Field}\rangle,$ $(D; \{s_{E1}, s_{E2}, \cdots, s_{Em}\}) \in \langle \text{WPat}\rangle,$ *such that every* s_{Ei} *is contained in some* $s_{Fj}\}$
(2.2)		|	Equals(Field, PPat)	$\{D \mid \exists(D; \{s_{F1}, s_{F2}, \cdots, s_{Fk}\}) \in \langle \text{Field}\rangle,$ $(D; \{s_{E1}, s_{E2}, \cdots, s_{Em}\}) \in \langle \text{PPat}\rangle,$ *such that every* s_{Ei} *is equal to some* $s_{Fj}\}$
3. Word Pattern Subtree				
(3.1)	WPat	:=	WPat_1 OR WPat_2	$\langle \text{WPat}_1\rangle \cup \langle \text{WPat}_2\rangle$
(3.2)		|	WPat_1 AND WPat_2	$\{(D; S) \mid (D; S_1) \in \langle \text{WPat}_1\rangle,\ (D; S_2) \in \langle \text{WPat}_2\rangle;\ S = S_1 \cup S_2\}$
(3.3)		|	WPat_1 (nW) WPat_2	$\{(D; S) \mid (D; S_1 = \{s_{11}, s_{12}, \cdots, s_{1k}\}) \in \langle \text{WPat}_1\rangle,$ $(D; S_2 = \{s_{21}, s_{22}, \cdots, s_{2m}\}) \in \langle \text{WPat}_2\rangle,$ $s_{11} \bullet s_{12} \bullet \cdots \bullet s_{1k}$ *precedes* $s_{21} \bullet s_{22} \bullet \cdots \bullet s_{2m},$ $Dist(s_{11} \bullet s_{12} \bullet \cdots \bullet s_{1k}, s_{21} \bullet s_{22} \bullet \cdots \bullet s_{2m}) \leq n;\ S = S_1 \cup S_2\}$
(3.4)		|	WPat_1 (nN) WPat_2	$\{(D; S) \mid (D; S_1 = \{s_{11}, s_{12}, \cdots, s_{1k}\}) \in \langle \text{WPat}_1\rangle,$ $(D; S_2 = \{s_{21}, s_{22}, \cdots, s_{2m}\}) \in \langle \text{WPat}_2\rangle,$ $Dist(s_{11} \bullet s_{12} \bullet \cdots \bullet s_{1k}, s_{21} \bullet s_{22} \bullet \cdots \bullet s_{2m}) \leq n;\ S = S_1 \cup S_2\}$
(3.5)		|	Word	$\{(D; \{m{:}m\}) \mid D[m{:}m]$ *is a (single-word) segment in* $\mathcal{I}(\text{Word})\}$
4. Phrase Pattern Subtree				
(4.1)	PPat	:=	PPat_1 OR PPat_2	$\langle \text{PPat}_1\rangle \cup \langle \text{PPat}_2\rangle$
(4.2)		|	PPat_1 AND PPat_2	$\{(D; S) \mid (D; S_1) \in \langle \text{PPat}_1\rangle,\ (D; S_2) \in \langle \text{PPat}_2\rangle;\ S = S_1 \cup S_2\}$
(4.3)		|	Phrase	$\{(D; \{k{:}m\}) \mid D[k{:}m]$ *is a segment in* $\mathcal{I}(\text{Phrase})\}$

Fig. 2. The abstract syntax of the front-end Boolean queries and the associated semantics.

translation.

3. BOOLEAN QUERY LANGUAGES

To discuss query translation we need a formal query language for the front-end and a document model to define the semantics. Although other models exist (see [Loeffen 1994; Navarro and Baeza-Yates 1995] for a brief survey), we believe that the one we present here is especially well suited as a query translation framework because of its compactness and ability to model most functionalities of commercial Boolean search engines. We assume that readers are familiar with Boolean systems, so we present the formalism in an abbreviated fashion.

3.1 Syntactic Structure of Boolean Queries

We start with the syntactic structure of Boolean queries. To avoid defining a complete syntax, we represent queries as "trees", where the nodes represent operators. The left column of Figure 2 describes the components of query trees. (The right column shows the associated semantics – these will be discussed later.) At the top level, queries consist of predicates connected by the operators AND, OR, and NOT (Figure 2, Construct 1). As required by most Boolean systems, the NOT operator is always implicitly used as the binary operator AND-NOT. A document D is in the result set of a query if and only if the query evaluates to *True* for D.

In Boolean systems, a document consists of a set of *fields*, each representing a particular kind of information such as Title, Author, and Abstract. Usually referred to as *fielded search*, a predicate specifies a pattern to be matched against the content of a field (Figure 2 , Construct 2). Typically, for each searchable field, IR systems build indexes [Salton 1989; Frakes and Baeza-Yates 1992; Faloutsos 1985] to direct the search engine to find documents with some given term, such as the word cat or phrase "Joe Doe". The indexing schemes of a field restrict how it can be queried. Generally, there are two ways of indexing[1].

First, the system may index every single word appearing in a field (except some common words), with additional information such as the positions where the word appears in the field. This *word-index* scheme allows efficient evaluation to find the set of documents containing some keywords. For example, the query Contains(Title, cat) searches for documents with the word cat in the Title field, and can only be used if Title is word-indexed. Furthermore, the positional information facilitates the evaluation of proximity queries, such as Contains(Title, cat (5W) dog) (cat appears within 5-word distance of dog). However, word-indexed fields generally do not allow the Equals operator, *e.g.*, Equals(Title, "Database Systems") or Equals(Title, Database (W) Systems), assuming Title is only word-indexed. The evaluation would be expensive simply because the complete values are not indexed, and therefore the field of each document must be accessed to evaluate the equality.

Second, the index may be built on the complete contents of the field (which is called *phrase-index*), typically for short fields such as Author. The system can thus efficiently support the Equals operator to search with the complete value of the field, *e.g.*, Equals(Author, "Joe Doe"). However, this type of index does not support Contains, *e.g.*, Contains(Author, "Joe Doe"). It also does not allow queries with word expressions, *e.g.*, Equals(Author, Joe (1W) Doe). Because the index entries are the complete values of the field, such queries can only be evaluated by scanning all the entries, which is expensive.

Therefore, for a word-indexed field, the Contains predicate can be used to test if the field contains a *word pattern* (Figure 2 , Construct 3), which is an expression consisting of words (the terminal Word) connected by AND, OR, or the proximity operators. The nW proximity operator specifies that its first operand must precede the second by no more than n words. The W operator is used when the distance is implicitly zero. For instance, we can use color (W) printer to search for the phrase "color printer" (and thus it is not necessary to support phrases in Contains predicates). If the order does not matter, operators nN and N may be used instead. The terminal Word can be either an *exact* word like cat, or an *expanded* word like cat* (which matches any words starting with cat if truncation is supported) or stem(cat) (which matches any words with the same stem as cat under some stemming algorithm [Lovins 1968; Porter 1980]).

On the other hand, for a phrase-indexed field, the Equals predicate can be used to test the equality of the field to some *phrase pattern*. Phrase patterns (Figure 2, Construct 4) are expressions consisting of phrases (the terminal Phrase) connected by AND or OR operators. A phrase is a quoted string, in our notation, which is

[1]The discussion on indexing schemes is to (informally) give the intuition of why some fields can be queried only with either Contains or Equals, but not both. It is not meant to represent all the access methods (see [Faloutsos 1985] for a general survey).

supposed to be the complete content of a field. Like words, an exact phrase is completely specified (*e.g.*, "Joe Doe"); otherwise, it may be truncated (*e.g.*, "Joe *") to form an expanded phrase.

Figure 2 shows all the syntactic constructs of the front-end query language. The underlying systems may not support all these constructs and interpret them uniformly. Boolean systems mainly differ in how they process predicates. First, they may have different fields in their documents, disallow searches over some fields (*e.g.*, because they are not indexed), or support only Contains or Equals for certain fields (as a consequence of particular indexing schemes used). Second, they may not support certain operators (*e.g.*, proximity operators). Third, they may not support features, such as stemming or truncation, for query expansion, or they may define stopwords that cannot be used in queries. Finally, there are some other minor details that are different across Boolean systems, for instance, the tokenization rules (*e.g.*, OS/2 may be considered as two words). Moreover, a supported feature may be interpreted non-uniformly across sources. For example, different systems may have different algorithms for stemming, or they may interpret, say, transitive proximity expression differently.

Note that in this paper we only discuss the *query capability mapping* process (Figure 1), which is the major challenge of query translation. The process generates syntax-neutral native query trees expressible in the target's syntax. In particular, the syntax differences (*e.g.*, operator precedence) of the target query languages are handled in the ensuing *syntax translation* step, which actually produces the native query strings.

3.2 Boolean Retrieval Model

This section formally defines the semantics (Figure 2, right column) of Boolean queries in terms of the retrieval model for query evaluation. We first define the data model that underlies query evaluation. A *document* D is a finite length string logically structured as a number-indexed sequence of words. A *segment* $m{:}n$ of document D, denoted by $D[m{:}n]$, where m and n are a pair of integers, is a contiguous subsequence from the m-th to the n-th word of D. During query evaluation, usually only some particular segments of a document, which we call a subdocument, are of importance. A *subdocument* of D, denoted by $(D; \{s_1, s_2, ..., s_n\})$, is a set of segments from D, $\{D[s_1], D[s_2], ..., D[s_n]\}$, that collectively satisfy some property, *e.g.*, match a pattern. A *collection* is in general a set of subdocuments. Notice that a full document D is just a special instance of a subdocument.

A text retrieval system manages a collection of documents which we call the *source collection*. Query evaluation is modeled as an algebra on collections starting with the source collection and yielding a subset as the collection of answer documents. All the nodes in a query tree are operators which take one or two collections as operands and return collections. Notationally, we use $\langle X \rangle$ to denote the collection returned by a subtree X.

Because most operations are defined around the notion of segments, we first describe the positional relationships and operations of segments. For any two segments (of the same document), one can contain, equal, overlap, or precede the other. Segment $s_1 = m_1{:}n_1$ is said to *contain* segment $s_2 = m_2{:}n_2$, if $m_1 \leq m_2 \leq n_2 \leq n_1$. We say that s_1 *equals* s_2 if they contain each other. Segment s_1 *precedes* s_2 if

$n_1 < m_2$. Otherwise, s_1 *overlaps* s_2 if neither of them precedes or contains the other. The *distance* between s_1 and s_2, $Dist(s_1, s_2)$, is the number of words between the nearest endpoints of s_1 and s_2 if one precedes the other, or -1 if one contains or overlaps the other. The *concatenation* of s_1 and s_2, denoted by $s_1 \bullet s_2$, is $min(m_1, m_2){:}max(n_1, n_2)$, *i.e.*, the smallest segment containing both s_1 and s_2.

Referring to the right column of Figure 2, the evaluation is a bottom-up, post-order process. Not defined in the figure are the field operator and the interpretation function $\mathcal{I}(\cdot)$. A field operator F returns a collection $\langle \mathsf{F} \rangle$ in which each subdocument is the field F of a document in the source collection. Notice that each subdocument may contain more than one segment because a field, say Author, can be multi-valued (*i.e.*, having more than one author), in which case each individual value will be represented by a segment in the subdocument of the field.

The evaluation of a terminal pattern (Figure 2 – 3.5, 4.3) is based on the notion of interpretation. The *interpretation* of a terminal pattern t, denoted by $\mathcal{I}(t)$, is the set $\{\, x \mid x \text{ is a string matching } t \}$, *e.g.*, $\mathcal{I}$(cat?) = { cat, cats}, and $\mathcal{I}$("text retrieval") = { "text retrieval"}. Remember that various features can be used to expand the interpretation, and that the target systems might not understand such expansion, in which case the interpretation is simply $\emptyset$. In addition, the target systems may not agree on the interpretation. In particular, many systems define a set of non-searchable words called stopwords (*e.g.*, the, an). A *stopword* w is a word that, by definition, does not match anything, *i.e.*, $\mathcal{I}(w) = \emptyset$ (and thus $\langle w \rangle = \emptyset$), although it may actually appear frequently. We use S_T to represent the stopwords defined by a system T (which is called the stopword list[2] of T). In contrast, the set of all the words appearing in T other than stopwords is called the *vocabulary* of T and denoted V_T. Besides, a system may apply implicit expansion such that an exact word is expanded automatically to match a set of words.

3.3 Atomic Predicates

Predicates are the basic constructs of queries and hence the basis of query mapping. Sometimes a predicate contains logical conjunctions or disjunctions, and it is more effective to break the complex predicate into simpler atomic predicates. For example, consider the predicate Contains(Title, multiprocessor AND distributed (W) system). It is equivalent to the conjunction of the simpler predicates: Contains(Title, multiprocessor) AND Contains(Title, distributed (W) system). This *atomization* separates predicates that may not be supported at a target from those that are, and hence simplifies translation. The predicate rewriting process (Section 5) assumes atomic predicates as inputs.

Atomic predicates do not contain the OR operator – all the other operators can distribute over OR [Mitchell 1973] (see Figure 2), so it can be "pulled out" from predicates. For example, Contains(F, (A OR B) (nW) C) = Contains(F, (A (nW) C) OR (B (nW) C)) = Contains(F, A (nW) C) OR Contains(F, B (nW) C).

However, atomic Contains-predicates may contain the AND operator (in addition

[2] Some systems use context-sensitive stopwords, *e.g.*, "in" is ordinarily a stopword, but it is searchable in certain contexts such as mother (W) in (W) law. To take advantage of those special cases when a stopword is actually searchable, the front-end can record the source's stopword list along with the "exceptional" contexts.

to the proximity operators) since the proximity operators do not distribute over AND [Mitchell 1973], *e.g.*, (A AND B) (nW) C $\neq$ (A (nW) C) AND (B (nW) C). In contrast, notice that in an atomic Equals-predicate the phrase pattern is simply a single phrase, either exact or expanded.

4. QUERY CAPABILITY MAPPING

As discussed in Section 1, our goal is to transform a user query into a native query that can be supported by the target source. Furthermore, we would like the native query to return as few "extra" documents as possible. In this case, we say that the native query *minimally subsumes* the user query with respect to the target system. Note that the notion of query subsumption is directly related to *query containment* (for conjunctive queries) in deductive databases [Ullman 1988] and has been applied extensively in information integration [Ullman 1997]. The following definitions formalize these concepts. The notation $\langle Q \rangle$ represents the result set of a query Q.

Definition 1. (Query Subsumption) A query Q' *subsumes* query Q ($Q' \supseteq Q$) if $\langle Q' \rangle \supseteq \langle Q \rangle$ regardless of the contents of the collection. If $\langle Q' \rangle$ is a proper superset of $\langle Q \rangle$ for some collection, then Q' *properly subsumes* Q ($Q' \supset Q$), *i.e.*, Q' subsumes but is not equivalent to Q.

Definition 2. (Minimal Subsuming Query) A query Q^S is the *minimal subsuming query* of query Q, or Q^S *minimally subsumes* Q, with respect to a target system T, if

(1) Q^S is supported by T,
(2) Q^S subsumes Q, and
(3) there is no query Q' that also satisfies 1 and 2, and is properly subsumed by Q^S.

Our goal thus is to transform the input query tree Q into its minimal subsuming query Q^S. We do this in three steps. The first step is to convert Q into a disjunctive normal form (DNF) query Q^d where the predicates are atomic. Having Q in this form simplifies the following two steps. The DNF query will be of the form $Q^d = C_1 \vee C_2 \vee \cdots \vee C_m$, where each conjunction term C_i is of the form $\tilde{P}_1 \wedge \tilde{P}_2 \wedge \cdots \wedge \tilde{P}_n$, *i.e.*, conjunction of predicates. Each predicate $\tilde{P}_j$ is either an atomic predicate P_j or a negated atomic predicate $\neg P_j$. Converting queries (which are Boolean expressions) into DNF is a well know process [McCluskey 1986], so we will not discuss it further here.

The second step is to rewrite each atomic predicate in Q^d into one that can be supported by the target. The proper substitutes are those supported constructs with weaker (or stronger for negated predicates) selectivity that are as close as possible to the original predicates. We illustrate this through an example.

Example 3. (Predicate Rewriting) Consider the predicate P = Contains(Title, color (5W) printer), which means Title must contain the two words appearing no more than 5 words apart and in that order. Assume the target system only supports the immediate adjacency operator W, of which the distance is always implicitly zero. In this case, we replace 5W with AND, because it is the closest *weaker* substitute.

The substitution results in P^S = Contains(Title, color AND printer). Notice that $P \subset P^S$.

Next, consider what happens if P is negated in the query. In this case, it is not correct to replace $\neg P$ with $\neg P^S$, since $\neg P \not\subset \neg P^S$. Indeed, the subsumption relationship is reversed by the negation, *i.e.*, $\neg P \supset \neg P^S$. It is thus possible that some answers of $\neg P$ may be lost in $\neg P^S$. This fact suggests that unsupported operators in a negated predicate should be replaced with the closest *stronger* substitute; in other words, 5W should be replaced with W in this case. Therefore, we obtain the *negative* form of the predicate, P^- = Contains(Title, color (W) printer). We see that $\neg P \subset \neg P^-$, and hence we can replace $\neg P$ in our query with $\neg P^-$ and get a broader result set.

As Example 3 suggests, we need different subsuming forms for positive and negative predicates. We formally define these subsuming forms in the following.

Definition 3. (Predicate Subsuming Forms)

(1) A query P^S is the *positive subsuming form* of a predicate P with respect to a target system T, if P^S minimally subsumes P with respect to T.

(2) A query P^- is the *negative subsuming form* of a predicate P with respect to a target system T, if $\neg P^-$ minimally subsumes $\neg P$ with respect to T.

Notice that the subsumption relationship is $P^S \supseteq P \supseteq P^-$ (and also $\neg P^- \supseteq \neg P \supseteq \neg P^S$). In some extreme cases, there may not exist *non-trivial* rewritings for either positive or negative subsuming forms. That is, P can only be rewritten *trivially* as $P^S = True$ (or $P^- = False$). This can happen, for instance, when P specifies a natively non-searchable field (see Section 5.1). In effect, this trivial rewriting of P will remove it from the native query (and thus P will be processed in post-filtering). Section 7 discusses more on the implication of trivial rewriting.

Furthermore, if a predicate P is logically equivalent to P' expressible in T, then P' is both the positive and negative subsuming form of P, which we call the *equivalent subsuming form* of P, *i.e.*, $P \equiv P'$. Note that P and P' are not necessarily identical. For example, Contains(Title, text*) is logically equivalent but different from Contains(Title, text OR textual OR ...). Of course, when a predicate is directly supported by the target system, the predicate itself is its equivalent subsuming form.

Once we have rewritings for all the predicates in Q^d, the third step generates the final minimal subsuming query Q^S to be sent to the target. It turns out that in the vast majority of cases, Q^S is simply obtained by replacing the predicates in Q^d with their (positive or negative) rewritings. If Q^d were not in DNF, this would not be true. (This is why we converted the original query to DNF.) Even if Q^d is in DNF, there are certain rare cases where predicates are not "independent" and we do not get a minimal query by simply replacing the predicates with their rewritings. The precise condition when this occurs is given in [Chang et al. 1996a; Chang et al. 1996b], together with proofs that in the remaining cases the resulting query is the desired minimal subsuming query.

Note that in [Chang et al. 1996a; Chang et al. 1996b] we assume that all target systems support the Boolean operators AND, OR, and NOT. That is, if the source supports predicates P_1 and P_2, then it supports P_1 AND P_2, P_1 OR P_2, and so on.

We surveyed many commercial Boolean search engines and found this to be true. For systems that impose syntactic restrictions (*e.g.*, AND must appear at the top level of query expressions), the restrictions can be handled in the syntax translation step (Figure 1) by conversion of the query expressions. Furthermore, in the rare cases when Boolean systems do not support arbitrary combination of predicates, one additional step (between capability mapping and syntax translation) must be taken to formulate query plans consisting of supported subqueries [García-Molina et al. 1998].

Thus, the critical difficult step in the whole process is predicate rewriting, which is the main contribution of this paper. In Section 5 we discuss the predicate rewriting rules. Keep in mind that after Q^S is submitted to the target, we may still need to filter the results, as Example 1 and 2 suggest. The filter query for post-processing must include the conditions that were not "pushed down" to the target. Reference [Chang et al. 1996b] provides details for constructing good filters.

5. PREDICATE REWRITING

In this section we present systematic procedures for predicate rewriting, in which each step rewrites a particular syntactic construct of a predicate subtree (Figure 2). To obtain minimal translations, we first consider equivalent transformation whenever there are native constructs equivalent to unsupported features. Otherwise, we consider transformations into either positive or negative subsuming forms, *i.e.*, the *positive* or *negative transformations*. Notationally, we use $U \Longrightarrow V$, $U \stackrel{+}{\Longrightarrow} V$, and $U \stackrel{-}{\Longrightarrow} V$ to denote equivalent, positive, and negative transformations respectively, where U is an unsupported construct, and V the rewritten native construct.

The procedure for rewriting a predicate starts at the schema level transformation (Section 5.1), in which we transform between Contains and Equals, if either is not supported. We then process the patterns. The rewriting of word patterns (for Contains predicates) is quite different from that of phrase patterns (for Equals predicates); they are discussed in Sections 5.2 and 5.3 respectively. Figure 3 summarizes the rewriting rules that we will explain in this section.

5.1 Schema Level Transformation

Each information source defines its specific view of the documents it manages. From the users' perspective, the *schema* of a collection defines the set of *searchable* fields (that can be constrained in queries) and *retrievable* fields (that can be returned in query results). The schema also specifies how each searchable field can be queried. In practice, users would consult the source's documentation to formulate valid queries. In a front-end system, users do not query the heterogeneous systems directly; instead they formulate queries on a *common schema* defined at the front-end.

There are generally two approaches for *schema unification*, *i.e.*, the specification of a common schema that represents a set of interested sources. In the first approach, the front-end supports a single universal schema fixed for all target services, *e.g.*, GAIA [Rao et al. 1994; Rao et al. 1993]. This may be too restrictive if the front-end supports a wide range of targets and the set of targets actually involved in queries can change dynamically depending on user interest.

Syntactic Construct	Rewriting Rules
Predicate Subtree	
Predicate with non-searchable field	$\stackrel{+}{\Longrightarrow}$ *True*
	$\stackrel{-}{\Longrightarrow}$ *False*
(*2.1*) *Pred* := Contains(Field, WPat)	$\Longrightarrow$ Equals(Field, OR(*ToPhrase*(Field, WPat)))
(*2.2*) | Equals(Field, PPat)	$\stackrel{+}{\Longrightarrow}$ Contains(Field, *ToWord*(PPat))
	$\stackrel{-}{\Longrightarrow}$ *False*
Word Pattern Subtree	
(*3.1*) WPat := WPat_1 OR WPat_2	(always supported; but will not appear in an atomic predicate.)
(*3.2*) | WPat_1 AND WPat_2	(always supported)
(*3.3*) | WPat_1 (nW) WPat_2	$\stackrel{+}{\Longrightarrow}$ [WPat_1 (m_1W) WPat_2] AND [WPat_1 (m_2N) WPat_2] AND [WPat_1 AND WPat_2]
	$\stackrel{-}{\Longrightarrow}$ WPat_1 (m_1W) WPat_2
(*3.4*) | WPat_1 (nN) WPat_2	$\stackrel{+}{\Longrightarrow}$ [(WPat_1 (m_1W) WPat_2) OR (WPat_2 (m_1W) WPat_1)] AND [WPat_1 (m_2N) WPat_2] AND [WPat_1 AND WPat_2]
	$\stackrel{-}{\Longrightarrow}$ [(WPat_1 (m_1W) WPat_2) OR (WPat_2 (m_1W) WPat_1)] OR [WPat_1 (m_2N) WPat_2]
	(See Fig. 6 for the detailed rules for the proximity operators.)
(*3.5*) | Word	
• exact word w :	(ideal case) ; *if* $\mathcal{I}_T(w) = \{w\}$,
	$\stackrel{+}{\Longrightarrow}$ w, $\stackrel{-}{\Longrightarrow}$ NoWord ; *if* $\mathcal{I}_T(w) = \{w, w_1, w_2, \ldots, w_n\}$,
	$\stackrel{+}{\Longrightarrow}$ AnyWord, $\stackrel{-}{\Longrightarrow}$ NoWord ; *if* $\mathcal{I}_T(w) = \emptyset$, *i.e.*, *w is a stopword.*
	(See Sec. 5.2.1 for the processing of AnyWord and NoWord.)
• expanded word x :	(ideal case) ; *if* $\mathcal{I}_T(x) = \mathcal{I}_F(x)$.
	$\Longrightarrow$ OR($\mathcal{I}_F(x)$) ; *if* $\mathcal{I}_T(x) \neq \mathcal{I}_F(x)$
	(or by approximation: $\approx x$, $\approx$ x OR($\mathcal{I}_F(x) - \mathcal{I}_T(x)$).)

Fig. 3. Summary of the predicate rewriting rules corresponding to the syntactic constructs. Rules for phrase patterns are not shown.

Consequently, the second approach is to determine the common schemas based on the set of involved targets. Along this line, various techniques can be used to determine the common schema. The simplest approach is to compute the common schema as the "intersection" of those searchable fields supported at all targets, assuming they share a degree of semantic consistency. For instance, if the user is interested in querying bibliographic citations, the schema intersection of the interested bibliographic sources may support Title, Author, Publisher, and so on. Another interesting approach, as described in [Paepcke 1993], is to organize the target services in a "type" hierarchy according to their subject areas, from which a reasonable common schema can be computed.

Note that, in the determination of common schemas, we assume that there is a way to decide the semantic mapping of equivalent (or similar) fields across sources. This assumption may not be true when the interested sources represent a wide range of materials, *e.g.*, news articles, patent records, bibliographic citations, *etc.* Indeed, the general semantic mapping problem of fields is itself a significant barrier to automatic distributed search that warrant serious investigation. In [Baldonado et al. 1997a; Baldonado et al. 1997b], we present our initial study to represent source schema and the mapping of fields as part of "source metadata." Furthermore, in most cases a user will be interested in a small subset of sources that share a common subject area. In such cases, as we discussed in the second approach, we believe a common schema and the mapping of fields can be reasonably specified.

We therefore consider the specification of common schema an orthogonal issue to be addressed separately from the query translation problem. We assume that in the front-end there is an independent component, the *common schema service*, that constructs a common schema for the intended targets specified by users. Furthermore, we assume that any field supported in the common schema is at least retrievable for any intended target so that post-filtering is possible. However, it is not required that the targets support the same set of predicates for a particular field; in fact, the field may not even be searchable at some targets. If the field is not searchable at a target, it can only be processed in post-filtering. That is, if a predicate P is for a natively non-searchable field, P will be rewritten trivially as $P \overset{+}{\Longrightarrow} True$ and $P \overset{-}{\Longrightarrow} False$.

Otherwise, the predicate, either $P = \mathsf{Equals(F, PPat)}$ or $P = \mathsf{Contains(F, WPat)}$, refers to a field F that can be mapped to a searchable field at the target. (The mapping can be simply given in a table, or provided by other front-end components responsible for maintaining the mapping of fields. For instance, references [Baldonado et al. 1997a; Baldonado et al. 1997b] describe such service.) The first problem we may face is that the target does not support the Contains or Equals operators we need.

First we consider the rewriting of predicate $P = \mathsf{Equals(F, PPat)}$ when the target only supports Contains for field F. Clearly, there is no non-trivial negative rewriting, *i.e.*, $P \overset{-}{\Longrightarrow} False$, because Equals is more selective than Contains. On the other hand, positive rewriting is possible because the phrase PPat does tell what must be contained in the field. Assume $\mathit{ToWord}(\mathsf{PPat})$ is a procedure that converts PPat into its word pattern counterpart by tokenizing PPat into a list of words connected with appropriate operators, either W or AND. In particular, internal truncation is replaced with AND; otherwise, W is used. Therefore, the positive rewriting is $\mathsf{Equals(F,\ PPat)} \overset{+}{\Longrightarrow} \mathsf{Contains(F,}\ \mathit{ToWord}(\mathsf{PPat}))$.

Note that different systems may apply different tokenization rules. For instance, some systems tokenize the term OS/2 as two words OS and 2, while others recognize it as a single word. Therefore, the procedure $\mathit{ToWord}(\mathsf{PPat})$ is target specific that tokenizes PPat according to the target system. In most cases we can simply encode the native token definitions in regular expressions (*e.g.*, "`[A-Za-z]+`" for tokens consisting of only the alphabets). Otherwise, when the tokenization rules are extremely complex, they can be directly coded in the procedure $\mathit{ToWord}(\cdot)$.

Example 4. (Equals-Predicates) Let $P_1 = \mathsf{Equals(Title,\ \text{"gone with the wind"})}$. Suppose the target (*e.g.*, *Dialog*) does not support Equals for the Title field. In this case, the rewriting is:

$$\begin{aligned} P_1 &\overset{+}{\Longrightarrow} \mathsf{Contains(Title,}\ \mathit{ToWord}(\text{"gone with the wind"})) \\ &= \mathsf{Contains(Title, gone\ (W)\ with\ (W)\ the\ (W)\ wind)}; \\ P_1 &\overset{-}{\Longrightarrow} False. \end{aligned}$$

As another example, if $P_2 = \mathsf{Equals(Title,\ \text{"introduction to database * principles *"})}$, in which case the phrase pattern is truncated, then the positive rewriting is as follows. Note that the operator connecting the last two words is AND (instead of W), because the truncation symbol indicates that there may be other words in between.

$$P_2 \stackrel{+}{\Longrightarrow} \text{Contains(Title, } \textit{ToWord}(\text{"introduction to database * principles *"}))$$
$$= \text{Contains(Title, introduction (W) to (W) database AND principles).}$$

The filter query for either P_1 or P_2 is simply the predicate itself. In general, if an atomic predicate is supported at the target, then no filtering is needed; otherwise, the predicate itself (in its entirety) must be the filter. The filter construction algorithm for complex queries consisting of more than one predicate is discussed in [Chang et al. 1996a; Chang et al. 1996b].

Next we show the rewriting of P = Contains(F, WPat). If the field F can only be queried with Equals, we must "promote" the word pattern WPat (representing a partial value) to the corresponding phrase pattern (matching the complete values of the field). This rewriting requires reference to the *phrase vocabulary* (*i.e.*, the set of complete values) of the field F. Without the vocabulary, the predicate can only be rewritten trivially, *i.e.*, $P \stackrel{+}{\Longrightarrow} \textit{True}$ and $P \stackrel{-}{\Longrightarrow} \textit{False}$. Otherwise, if the vocabulary is accessible, we can enumerate from it all the phrases containing WPat. The disjunction (OR) of these exhaustively enumerated phrases gives an equivalent rewriting. That is, Contains(F, WPat) $\Longrightarrow$ Equals(F, OR(*ToPhrase*(F, WPat))), where *ToPhrase*(F, WPat) = { p | p *appears in the phrase vocabulary of* F, p *contains* WPat}. In some cases when *ToPhrase*(F, WPat) returns so many phrases that the rewriting is unwieldy, users may be asked to choose those that best match their intentions.

Example 5. (Contains-Predicates) Suppose the predicate P = Contains(Author, garcia-molina), and the target does not support Contains for the Author field. If *ToPhrase*(Author, garcia-molina) = { "garcia-molina, h.", "garcia-molina, r."}, the equivalent rewriting is: Contains(Author, garcia-molina) $\Longrightarrow$Equals(Author, "garcia-molina, h." OR "garcia-molina, r.").

5.2 Word Patterns for Contains-Predicates

We are now ready to transform the third predicate component, *i.e.*, the pattern. This section studies the word patterns WPat in predicates Contains(F, WPat). (We discuss phrase patterns for Equals-predicates in Section 5.3.) The rewriting process starts with the removal of stopwords (Section 5.2.1), then the replacement of unsupported proximity operators (Section 5.2.2), and finally the rewriting of expanded words (Section 5.2.3). While the order of processing is not critical, we do assume this order to simplify the presentation.

5.2.1 *Exact Words.* We first process the exact words, which are at the leaves of the WPat subtree (Figure 2). We assume that the words are valid tokens as defined by the target system. Otherwise, we can apply the procedure *ToWord*(w) just discussed to tokenize and convert a word w into a proximity expression. For instance, *ToWord*("OS/2") = OS (W) 2, if the target system tokenizes the term as two words. (The proximity operator W, if not supported, will be processed in the ensuing step.)

An exact word may not be interpreted consistently across systems. We denote the interpretation functions (Section 3.2) of the target T and the front-end by $\mathcal{I}_T(\cdot)$ and $\mathcal{I}_F(\cdot)$ respectively. We assume that the front-end does not apply implicit expansion, *i.e.*, $\mathcal{I}_F(w) = \{w\}$, for an exact word w. However, the target may not interpret exact words this way. There are three cases:

(1) $\mathcal{I}_T(w) = \{w\}$.
(2) $\mathcal{I}_T(w) = \{w, w_1, w_2, \ldots, w_n\}$, *i.e.*, the target implicitly expands to words other than w.
(3) $\mathcal{I}_T(w) = \emptyset$, *i.e.*, w is a stopword of T ($w \in S_T$).

Case 1 is the most common case and exactly what we want. For case 2 and 3, as $\mathcal{I}_T(w) \neq \mathcal{I}_F(w)$, a transformation is required. For our exposition, we define two "imaginary" word patterns: AnyWord, which matches any words in a document (*i.e.*, $\mathcal{I}_T(\mathsf{AnyWord}) = V_T \cup S_T$), and NoWord, which matches nothing (*i.e.*, $\mathcal{I}_T(\mathsf{NoWord}) = \emptyset$). Note that the target system does not understand these imaginary patterns. The transformation process can therefore be divided into two phases. In Phase 1 we replace w with the imaginary patterns when appropriate, then in Phase 2 we actually rewrite these imaginary patterns.

Phase 1. For case 2, there is no way to suppress implicit expansion and restrict the matchings to only w. Consequently, we have $w \stackrel{+}{\Longrightarrow} w$ (for any word w); note that the positive rewritings are always broader than the user queries because of expansion. Similarly, $w \stackrel{-}{\Longrightarrow} \mathsf{NoWord}$.

For case 3, because $\mathcal{I}_T(w) = \emptyset$ (w is a stopword at T), $\langle w \rangle = \emptyset$ (Figure 2). That is, the target system cannot evaluate if stopwords appear in a document (because they are not indexed); this can only be done in post-filtering. Note that we assume the front-end does not specify any stopwords, *i.e.*, $\mathcal{I}_F(w) = \{w\}$. Thus, in rewriting, we must allow w to match any words so that no possible matches can be excluded, *i.e.*, $w \stackrel{+}{\Longrightarrow} \mathsf{AnyWord}$, and similarly $w \stackrel{-}{\Longrightarrow} \mathsf{NoWord}$.

Phase 2. This phase processes the imaginary patterns introduced in Phase 1, because they are not natively interpretable. Note that in Phase 1 an exact word w may be negatively replaced with NoWord ($w \stackrel{-}{\Longrightarrow} \mathsf{NoWord}$) or positively with AnyWord ($w \stackrel{+}{\Longrightarrow} \mathsf{AnyWord}$). In the following we discuss both in turn.

First, the negative rewriting $w \stackrel{-}{\Longrightarrow} \mathsf{NoWord}$ effectively makes the containing predicate P become *False* ($P \stackrel{-}{\Longrightarrow} \mathit{False}$). To see this, note that only the proximity (nW and nN) and AND operators can appear in an atomic predicate (Section 3.3). As Figure 2 shows, for any such operator op and arbitrary pattern E, clearly $\emptyset$ op $\langle E \rangle = \emptyset$. Because $\mathcal{I}_T(\mathsf{NoWord}) = \emptyset$, $\langle \mathsf{NoWord} \rangle = \emptyset$. Therefore, by construction, the rewritten predicate also evaluates to $\emptyset$.

Second, for positive rewritings we need to process AnyWord. To illustrate, consider the following rewriting (*e.g.*, continued form Phase 1 for the pattern gone (W) with (W) the (W) wind, where "with" and "the" are stopwords):

$$\mathsf{gone\ (W)\ AnyWord\ (W)\ AnyWord\ (W)\ wind} \stackrel{+}{\Longrightarrow} \mathsf{gone\ (2W)\ wind}.$$

While the two AnyWords representing stopwords are removed, their occupancy is properly reflected by modifying the proximity operators. (Note that the rewriting

```
procedure StopWordRemoval(thisNode)
/* Remove AnyWord (representing stopwords) from the subtree rooted at thisNode. */
/* Return a 4-tuple [WPat', g_l, g_r, width]. */
/* The symbol "~" represents don't-care (i.e., unimportant) return values. */
begin
    /* thisNode is a terminal Word (see Fig. 2), possibly rewritten to AnyWord. */
    if (thisNode is a leaf node):
        if (thisNode.op = AnyWord): /* thisNode.op is the operator represented by thisNode. */
            /* an AnyWord is nullified, and its occupancy (1 word) is returned in width. */
            return [Null, ~, ~, 1]
        else: /* thisNode is a normal (non-stop) word. */
            return [thisNode.op, 0, 0, ~] /* simply return the operator as is. */
    else:/* thisNode is an interior node in {nW, nN, AND}. */
        /* recursively traverse the subtree in post-order. */
        [WPat_1 , g_l1, g_r1, width_1] := StopWordRemoval(thisNode.leftChild)
        [WPat_2 , g_l2, g_r2, width_2] := StopWordRemoval(thisNode.rightChild)
        if (thisNode.op = nW):
            case (WPat_1 ≠ Null and WPat_2 ≠ Null):
                return [WPat_1 ((g_r1 + n + g_l2) W) WPat_2 , g_l1, g_r2, ~]
            case (WPat_1 ≠ Null and WPat_2 = Null):
                return [WPat_1 , g_l1, g_r1 + n + width_2, ~]
            case (WPat_1 = Null and WPat_2 ≠ Null):
                return [WPat_2 , width_1 + n + g_l2, g_r2, ~]
            case (WPat_1 = Null and WPat_2 = Null):
                return [Null, ~, ~, width_1 + n + width_2]
        else: /* thisNode.op = nN, including AND (n = ∞). */
            case (WPat_1 ≠ Null and WPat_2 ≠ Null):
                return [WPat_1 (max(g_r1 + n + g_l2, g_r2 + n + g_l1) N) WPat_2,
                        max(g_l1, g_l2), max(g_r1, g_r2), ~]
            case (WPat_1 ≠ Null and WPat_2 = Null):
                return [WPat_1 , width_2 + n + g_l1, g_r1 + n + width_2, ~]
            case (WPat_1 = Null and WPat_2 ≠ Null):
                return [WPat_2 , width_1 + n + g_l2, g_r2 + n + width_1, ~]
            case (WPat_1 = Null and WPat_2 = Null):
                return [Null, ~, ~, width_1 + n + width_2]
end
```

Fig. 4. A recursive procedure, written in pseudo code, for removing stopwords.

is not equivalent because the left hand side requires exactly, rather than at most, two words in between "gone" and "wind".)

Figure 4 presents a recursive procedure for *stopword removal* from a pattern subtree. Essentially, the procedure is a post-order traversal of the subtree, during which stopwords are removed, and their occupancy is propagated upward. It takes as input the pattern subtree represented by its root *thisNode*, and returns a 4-tuple [WPat$'$, $g_l, g_r, width$]. The first component WPat$'$ is the rewritten pattern (of the input subtree) with all stopwords removed. Note that WPat$'$ may be *Null* if the subtree contains only stopwords. Depending on if WPat$'$ is *Null*, different occupancy information of AnyWord will be returned. First, when WPat$'$ is *Null*, its width (the maximal number of words supposed to be occupied by the nullified pattern) is returned in *width*. Otherwise, the left and right gaps (the maximal number of AnyWords adjacent to the pattern at either side) are returned in g_l and

g_r respectively. Note that, the AND operator can be modeled as and processed just like the ∞N operator, where ∞ is an arbitrarily large number.

In summary, the positive rewriting of P = Contains(F, WPat) is $P \stackrel{+}{\Longrightarrow}$Contains(F, WPat′), in which WPat is rewritten to WPat′ with the procedure just discussed. If WPat consists of only stopwords, then WPat′ becomes *Null*, which means $P \stackrel{+}{\Longrightarrow}$ *True*.

5.2.2 *Proximity Operators.* The front-end supports the proximity operators nW and nN, which must be translated because not all systems support them. They are standard features available in common command languages such as Z39.58 [National Information Standards Organization 1993] and ISO-8777 [ISO 1993], and some commercial systems such as *Dialog*. Other systems either do not support the proximity operators (*e.g.*, Stanford's *Folio*), or support them only partially[3], *e.g.*, the distance and/or order parameters may not be specified freely. For instance, *WebCrawler* does not support nW for arbitrary n, although it does support 0W (written as ADJ) and the unordered proximity operator nN (written as NEAR/n). As another example, *AltaVista* has only the operator NEAR, which means 10N.

In general, translation for unsupported operators is possible only if the target supports some semantically related operators. Therefore, we must first identify the subsumption relationships of those semantically related operators. Given arbitrary patterns A and B as operands, the compound patterns constructed using the proximity-related operators hold the subsumption relationships illustrated in Figure 5. In the figure, an arc $U \longrightarrow V$ indicates that expression U properly subsumes V. The subsumption relationships can be summarized as follows:

—A (m'N) B properly subsumes A (mN) B if $m' > m$. Similarly, A (n'W) B properly subsumes A (nW) B if $n' > n$. This represents the relaxation of the distance constraint.

—A (mN) B properly subsumes A (nW) B (and also B (nW) A) if $m \geq n$. This represents the relaxation of the order constraint. Furthermore, A (mN) B can be approximated by A (mW) B OR B (mW) A. In fact, in most cases they are equivalent, except when the segments matching A do not precede those matching B or vice versa (see Section 3.2). For instance, if A = (database (2N) principles) and B = (distributed (2N) systems), the text "... principles of distributed database systems ..." matches A (3N) B but not A (3W) B OR B (3W) A, because the segments matching A and B overlap.

—A AND B properly subsumes A (mN) B, because AND is equivalent to ∞N, where ∞ is an arbitrarily large number.

Based on the subsumption relationships, we can construct the rewriting rules for unsupported operators. Figure 6 shows the rules for rewriting nW and nN. In general, for an unsupported pattern U, the positive rewriting is the conjunction of

[3]We are also told that some systems interpret transitive proximity expressions (*e.g.*, A (W) B (W) C) as conjunctions of binary proximity expressions (*e.g.*, A (W) B AND B (W) C), which is inconsistent with the usual interpretation. For strict consistency, this can be handled by treating transitive proximity expressions as subsuming (rather than equivalent) queries of themselves, and processing them in the filter queries.

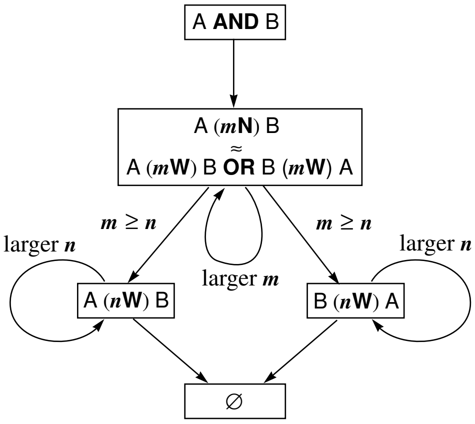


Fig. 5. Subsumption relationships of the proximity-related operators.

all the supported patterns U_i that subsume U, *i.e.*, $U \overset{+}{\Longrightarrow} \wedge(U_i)$. For example, Rule (1.1a) (Figure 6) states that A (nW) B (for a particular n) can be rewritten as the conjunction of all its subsuming patterns: A (m_1W) B, A (m_2N) B, and A AND B, for the smallest m_1 ($m_1 > n$) and m_2 ($m_2 \geq n$). Rule (1.1b) then shows the more specific rewritings according to what the target actually supports. Note that it is not necessary to include those terms that are "broader" than (*i.e.*, subsume) some other term already in the conjunction. For instance, in case (4) of Rule (1.1b), the term A AND B was removed, because it is broader than the remaining conjuncts.

On the other hand, the negative rewriting of an unsupported pattern U is the disjunction of all the supported patterns U_i that U subsumes, *i.e.*, $U \overset{-}{\Longrightarrow} \vee(U_i)$, as Figure 6 also illustrates. Note that, when the target does not support any such U_i, $U \overset{-}{\Longrightarrow} \emptyset$, in which case the enclosing predicate P becomes *False*, *i.e.*, $P \overset{-}{\Longrightarrow}$ *False* (*e.g.*, Rule (1.2b), case (1)). In contrast, for positive rewritings, the worst-case substitute is AND (*i.e.*, $U \overset{+}{\Longrightarrow}$A AND B), because the AND operator is by definition always supported by a Boolean system.

Example 6. (Proximity Operators) Suppose the predicate P = Contains(Title, distributed (2W) system). For the target *AltaVista* (which supports only 10N), referring to Figure 6, we rewrite

$$P \overset{+}{\Longrightarrow} \text{Contains(Title, distributed (10N) system)} \quad \text{(Rule(1.1b), case(2))}$$
$$P \overset{-}{\Longrightarrow} \textit{False}. \quad \text{(Rule(1.2b), case(1))}$$

As another example, assume the target is *WebCrawler*. Because it supports 0W and nN (for arbitrary n), P can be rewritten as:

$$P \overset{+}{\Longrightarrow} \text{Contains(Title, distributed (2N) system)} \quad \text{(Rule(1.1b), case(2))}$$
$$P \overset{-}{\Longrightarrow} \text{Contains(Title, distributed (0W) system)} \quad \text{(Rule(1.2b), case(2))}$$

5.2.3 *Expanded Words.* This section discusses the rewriting for expanded word patterns in general. For instance, stemming, synonym expansion, truncation, *etc.* are all expansion features to broaden the interpretation of words. Given an ex-

1. A (nW) B

- **Equivalent rewriting:** none.
- **Positive rewriting:**
  – Find the smallest m_1 and m_2 such that $m_1 > n$, $m_2 \geq n$, and m_1W and m_2N are supported.

$$(1.1a) \quad \text{A } (n\text{W}) \text{ B} \stackrel{+}{\Longrightarrow} [\text{A } (m_1\text{W}) \text{ B}] \text{ AND } [\text{A } (m_2\text{N}) \text{ B}] \text{ AND } [\text{A AND B}]$$

$$(1.1b) \quad = \begin{cases} (1) \ \text{A AND B} & \text{if neither } m_1 \text{ nor } m_2 \text{ exists;} \\ (2) \ \text{A } (m_2\text{N}) \text{ B} & \text{if } m_1 \text{ does not exist;} \\ (3) \ \text{A } (m_1\text{W}) \text{ B} & \text{if } m_1 \leq m_2 \text{ or } m_2 \text{ does not exist;} \\ (4) \ [\text{A } (m_1\text{W}) \text{ B}] \text{ AND } [\text{A } (m_2\text{N}) \text{ B}] & \text{if } m_2 < m_1. \end{cases}$$

- **Negative rewriting:**
  – Find the largest m_1 such that $m_1 < n$, and m_1W is supported.

$$(1.2a) \quad \text{A } (n\text{W}) \text{ B} \stackrel{-}{\Longrightarrow} \text{A } (m_1\text{W}) \text{ B}$$

$$(1.2b) \quad = \begin{cases} (1) \ \emptyset & \text{if } m_1 \text{ does not exist;} \\ (2) \ \text{A } (m_1\text{W}) \text{ B} & \text{otherwise.} \end{cases}$$

---

2. A (nN) B

- **Equivalent rewriting:**
  – If nW is supported (otherwise, no equivalent rewriting):

$$(2.0) \quad \text{A } (n\text{N}) \text{ B} \Longrightarrow (\text{A } (n\text{W}) \text{ B}) \text{ OR } (\text{B } (n\text{W}) \text{ A}) \qquad \text{(approximation)}$$

- **Positive rewriting:**
  – Find the smallest m_1 and m_2 such that $m_1 > n$, $m_2 > n$, and m_1W and m_2N are supported.

$$(2.1a) \quad \text{A } (n\text{N}) \text{ B} \stackrel{+}{\Longrightarrow} [(\text{A } (m_1\text{W}) \text{ B}) \text{ OR } (\text{B } (m_1\text{W}) \text{ A})] \text{ AND } [\text{A } (m_2\text{N}) \text{ B}] \text{ AND } [\text{A AND B}]$$

$$(2.1b) \quad = \begin{cases} (1) \ \text{A AND B} & \text{if neither } m_1 \text{ nor } m_2 \text{ exists;} \\ (2) \ \text{A } (m_2\text{N}) \text{ B} & \text{if } m_2 \leq m_1 \text{ or } m_1 \text{ does not exist;} \\ (3) \ (\text{A } (m_1\text{W}) \text{ B}) \text{ OR } (\text{B } (m_1\text{W}) \text{ A}) & \text{if } m_1 < m_2 \text{ or } m_2 \text{ does not exist.} \end{cases}$$

- **Negative rewriting:**
  – Find the largest m_1 and m_2 such that $m_1 < n$, $m_2 < n$, and m_1W and m_2N are supported.

$$(2.2a) \quad \text{A } (n\text{N}) \text{ B} \stackrel{-}{\Longrightarrow} [(\text{A } (m_1\text{W}) \text{ B}) \text{ OR } (\text{B } (m_1\text{W}) \text{ A})] \text{ OR } [\text{A } (m_2\text{N}) \text{ B}]$$

$$(2.2b) \quad = \begin{cases} (1) \ \emptyset & \text{if neither } m_1 \text{ nor } m_2 \text{ exists;} \\ (2) \ \text{A } (m_2\text{N}) \text{ B} & \text{if } m_2 \geq m_1 \text{ or } m_1 \text{ does not exist;} \\ (3) \ (\text{A } (m_1\text{W}) \text{ B}) \text{ OR } (\text{B } (m_1\text{W}) \text{ A}) & \text{if } m_1 > m_2 \text{ or } m_2 \text{ does not exist.} \end{cases}$$

Fig. 6. Rewriting rules for the proximity operators.

panded word x (*e.g.*, stem(running), cat$*$), we assume that the front-end interpretation is $\mathcal{I}_F(x) = \{w_1, w_2, ..., w_n\}$, where w_i's are the exact words supposed to match x.

If the target T also supports the pattern with a consistent interpretation, *i.e.*, $\mathcal{I}_T(x) = \mathcal{I}_F(x)$, then no rewriting is necessary for this ideal case. For example, there are usually standard interpretations across systems for the truncation features.

Otherwise, if $\mathcal{I}_T(x) \neq \mathcal{I}_F(x)$ (either x is not supported or its interpretations are inconsistent), a rewriting may be necessary. Our primary goal is to rewrite x into a pattern x' such that $\mathcal{I}_T(x') = \mathcal{I}_F(x)$, resulting in an equivalent rewriting. If this is not possible, the alternative goals are $\mathcal{I}_T(x') \supset \mathcal{I}_F(x)$ for positive rewriting, and $\mathcal{I}_T(x') \subset \mathcal{I}_F(x)$ for negative rewriting. In the following we study different cases for the expansion features.

(1) Pattern x is unsupported at T, *i.e.*, $\mathcal{I}_T(x) = \emptyset$ (*e.g.*, the target does not support stemming). Our strategy is to enumerate all the words in $\mathcal{I}_F(x)$, *i.e.*, $x \Longrightarrow \text{OR}(\mathcal{I}_F(x)) = w_1 \text{ OR } w_2 \text{ OR } \cdots \text{OR } w_n$. An exhaustive enumeration will result in an equivalent rewriting, if each exact word w_i is interpreted as is (*i.e.*,

no implicit expansion). Otherwise, with implicit expansion, the results can be broader than expected, as discussed in Section 5.2.1.

(2) Pattern x is supported but with an inconsistent interpretation at T. For example, the target may support a different stemming algorithm [Lovins 1968; Porter 1980]. In this case we have the following strategies:
   - *Tolerate the interpretation inconsistency.* We may regard the target interpretation as an acceptable approximation of the desired expansion, because users may not insist on (and usually are not aware of) the actual interpretation algorithms. Therefore, as long as the interpretation is expanded, it should be acceptable for the minor details to be determined by the target systems. In fact, insisting on a particular interpretation (such as that of the front-end) may not be necessary as there is no single algorithm proven to be the best in terms of retrieval effectiveness.
     Another approximation is to enumerate the words from $\mathcal{I}_F(x) - \mathcal{I}_T(x)$ in disjunction with x, *i.e.*, $x \Longrightarrow x' \approx x\ \mathsf{OR}(\mathcal{I}_F(x) - \mathcal{I}_T(x))$. Note that the extra expansions in $\mathcal{I}_T(x) - \mathcal{I}_F(x)$ are still ignored.
   - *Make the interpretation consistent.* As long as $\mathcal{I}_T(x) - \mathcal{I}_F(x) \neq \emptyset$ (*i.e.*, there are extra expansions), the only way to obtain a consistent interpretation is to directly enumerate the desired words, *i.e.*, $x \Longrightarrow \mathsf{OR}(\mathcal{I}_F(x))$, as case (1) discusses.

For the approaches suggested above we need an *enumerator* that, when given an expanded word x, will return all the words in $\mathcal{I}_F(x)$. For the enumerator not to miss any words that the target might have, it is required that the native vocabulary V_T be accessible. Otherwise, we can instead use a *common vocabulary* that is appropriate for the subject domains of a set of targets, but that is not specific to any of them. Although this may give us translations that are approximations at best, it is a reasonable alternative if native vocabularies are hard to obtain or costly to maintain. In addition, some systems have an upper bound on input query length, thus instead of enumerating all the words, we may approximate by enumerating some of them.

Example 7. (Stemming) Suppose the target system does not support stemming, then

$$\mathsf{Contains(Title, gone\ AND\ stem(wind))} \Longrightarrow \mathsf{Contains(Title, gone\ AND\ (wind\ OR\ winds))},$$

if $\mathcal{I}_F(\mathsf{stem(wind)}) = \{\mathsf{wind}, \mathsf{winds}\}$.

Finally, we discuss more specifically on truncation, an expansion feature that almost every system supports but to various extents. The truncation support differs in the sophistication of the allowed patterns. For example, Stanford-*Folio* does not allow open truncation to be used more than once in an expanded word (*e.g.*, com*ta*). As just discussed, we can use enumeration for unsupported expansion patterns. However, note that there are great numbers of possible truncation patterns, say comp*, compute?, comput??, *etc.* Consequently, it is not possible to pre-compute a database of expansions (with the patterns as keys) for fast look-up. In other words, the enumeration requires a sequential scan over the full vocabulary to match the pattern and thus can be costly.

Therefore, an alternative (for positive transformation only) is to rewrite with supported, simpler patterns, *e.g.*, com∗ta∗ $\stackrel{+}{\Longrightarrow}$com∗. Such rewriting of unsupported patterns can be directed by a set of rules for pattern translation. Of course, the target must at least support some sort of truncation; otherwise, the truncation can only be emulated by enumeration.

5.3 Phrase Patterns for Equals-Predicates

This section discusses the rewriting for phrase patterns, which are part of Equals-predicates. Because a phrase pattern (in an atomic Equals-predicate) is a single phrase, either exact or truncated (Section 3.3), we only need to deal with truncation, if not supported.

The truncation of phrases is similar to that of words, so our discussion for word truncation also applies. That is, for an unsupported phrase pattern, one way of (equivalent) rewriting is to enumerate possible matches from the phrase vocabulary (if available), as Section 5.2.3 describes. However, phrase vocabularies can be much larger than word vocabularies, which implies more storage cost and larger search space for enumeration. As also discussed in Section 5.2.3, another way for (positive) rewriting is to translate unsupported patterns to their positive subsuming forms directed by some pattern translation rules, *e.g.*, "Introduction to database ∗ principles ∗" $\stackrel{+}{\Longrightarrow}$"Introduction to database ∗".

Another alternative (for positive transformation) is to rewrite the Equals-predicate using the Contains operator, as shown in the rewriting of P_2 in Example 4. Of course, this is possible only if the search field also supports Contains-predicates. Note that the resulting Contains-predicate must then be processed as described in Section 5.2. For instance, referring to the rewriting of P_2 in Example 4, the word "to" might be a stopword that must be removed, or the target may not support the W operator.

6. COST EVALUATION

Because rewritten queries (minimally) subsume original queries, the front-end needs to post-filter preliminary results that may contain extra documents with respect to the original queries. Post-filtering incurs more work at the font-end, the networks, and the underlying services, because extra documents that users will not see have to be retrieved and processed. Because the post-filtering is of major concern in our approach, we performed experiments to study the overhead. This section presents some of the experimental results. A more detailed report is available in [Chang and García-Molina 1997].

Our experiments evaluated both *batch* and *incremental* processing, which are generally the two ways to implement post-filtering. Given a query Q and its translation Q^S, with batch processing the front-end retrieves and filters all the documents in $\langle Q^S \rangle$ (the preliminary results) and produces the final answers $\langle Q \rangle$ all at once. The cost is therefore proportional to the size of the preliminary result set, *i.e.*, $Size(\langle Q^S \rangle)$. In contrast, with the second approach, the front-end processes the documents incrementally when a matching document is requested. In other words, if a user wishes to see, say a screenful of documents, only some of the source's documents must be retrieved and filtered. Therefore, with incremental processing, the interesting cost metric is the per-document cost, *i.e.*. the batch cost amortized by the number of documents matching Q. We call this metric the *selectivity ratio*, denoted by $SR(Q^S, Q)$, because it indicates the reduction of selectivity from Q to Q^S; *i.e.*,

$$SR(Q^S, Q) = \frac{Size(\langle Q^S \rangle)/\text{size of the source collection}}{Size(\langle Q \rangle)/\text{size of the source collection}} = \frac{Size(\langle Q^S \rangle)}{Size(\langle Q \rangle)}.$$

Note that in the above cost metric we ignore the sizes (numbers of words) of documents in $\langle Q^S \rangle$ and $\langle Q \rangle$. We believe that in most cases $\langle Q^S \rangle$ and $\langle Q \rangle$ statistically share the same average document size, and thus it is not an interesting cost factor to focus on. (In other words, we believe that whether a document satisfies Q or Q^S is independent of its document size.) However, for certain kind of queries (see Section 6.1) this claim may be challenged. For such queries we also measured the document sizes to verify their significance as a cost factor (Section 6.1).

We set up the experiments to measure the cost metrics, $Size(\langle Q^S \rangle)$ and $SR(Q^S, Q)$, for some of the translation rules over sets of sample queries. Each set of the experiments focused on a specific translation rule using sample queries automatically generated and appropriate for the rule. We focused on evaluating positive rewriting, because post-filtering is not necessary for equivalent rewriting and negation (the NOT operator) is rarely used in practice[4]. Specifically, we evaluated the translation rules for the proximity operators, stopwords, and the Equals operator. Furthermore, we did not evaluate the cost of translation that degenerates the query predicates trivially to *True*, namely, when the search fields are not supported by the target. A predicate rewritten to *True* will be effectively removed from the query, and the cost will depend on the remaining predicates in the query, thus making it impossible to isolate this kind of translation. We discuss this in Section 7.

6.1 Proximity Experiments

We first studied the cost of rewriting proximity queries by operator substitution, as suggested by the rules in Figure 6. In other words, the experiments compared the selectivity of queries with different proximity operators. The automatically generated sample queries are of the parametric form Q_{W} = Contains(F, w_1 (W) w_2 (W) $\cdots$ (W) w_n), where F is a field designation like Title, and w_i's are the words in a phrase. First, for the field parameter, we selected three common fields representing different typical lengths: Title, Abstract, and Text (the body of text of documents). We expect that the typical lengths of the fields may impact the selectivity of proximity queries. Second, to generate the proximity expression "w_1 (W) w_2 (W) $\cdots$ (W) w_n", we first selected some phrase vocabularies, then randomly picked a phrase from the vocabularies, and finally extracted words $w_1, w_2, \cdots, w_n$ from the phrase. In particular, we chose as vocabularies The Free On-line Dictionary of Computing (VOC_{Foldoc}) [Howe 1997] and the *Inspec* Thesaurus (VOC_{Inspec}) [IEE 1991]. VOC_{Foldoc} is an evolving dictionary of computing-related terms, and VOC_{Inspec} is a set of controlled subject terms that IEE compiles to categorize the documents in the *Inspec* collection. Third, the sample queries, and their subsuming queries

[4] For instance, we analyzed a two-week user trace collected in our university library system, and found that negation was used in only 22 out of the total 15595 queries, *i.e.*, 0.14%.

(where W is replaced with less selective operators) were evaluated in the *Dialog* service, because of its sophisticated search engine. In particular, among the many collections *Dialog* provides, we chose to query the *Dialog-275* (*Computer Database*) and *Dialog-2* (*Inspec*) because they support the desired search fields, and their subject domains are appropriate for the vocabularies generating the queries.

Table 1 shows the four different configurations used for our experiments, where each configuration defines a search field, a phrase vocabulary, and a source collection. We performed two sets of experiments, each sharing these configurations, but with different subsuming queries. The first set of experiments evaluated the costs when the W operator is replaced with AND, which represents the worst-case substitution (Figure 6, case (1) of Rule (1.1b)). In the second set of experiments, we investigated at a finer granularity how progressively weaker operators impact query selectivities.

Table 1. Configurations of the proximity experiments.

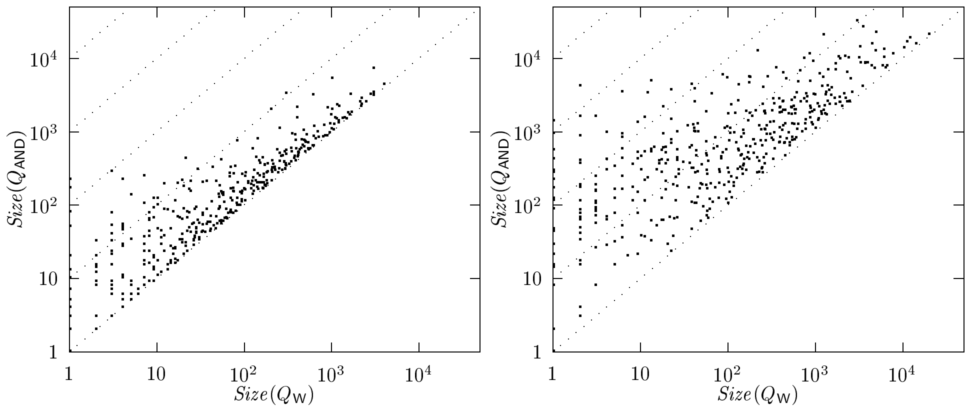
configuration parameter	Inspec-Ti	Inspec-Ab	Comput-Ti	Comput-Tx
Search field	Title	Abstract	Title	Text
Phrase vocabulary	VOC_{Inspec}	VOC_{Inspec}	VOC_{Foldoc}	VOC_{Foldoc}
Source collection	*Dialog-2*	*Dialog-2*	*Dialog-275*	*Dialog-275*

Figure 7 sketches the results of the first set of experiments, where a user query Q_{W} with the W operator is compared with a native query Q_{AND} that uses the AND operator instead. We only report on two of the configurations; the results for the remaining two configurations are similar. Specifically, the figures plot the pairs [$Size(\langle Q_{\mathsf{W}}\rangle)$, $Size(\langle Q_{\mathsf{AND}}\rangle)$] for the queries of the corresponding configurations. In other words, they illustrate the distribution of $Size(\langle Q_{\mathsf{AND}}\rangle)$ with respect to $Size(\langle Q_{\mathsf{W}}\rangle)$.

To illustrate the ranges of $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ values in the figures, the diagonal dotted lines (representing the $y/x = m$ axes, where m is labeled on the y–axis) partition the space into different bands, each representing a range of $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ (*i.e.*, y/x). For instance, for all the data points falling in the lowest band (y/x between 1 and 10), the subsuming query Q_{AND} fetches between 1 and 10 times as many documents as the original query Q_{W} would have fetched, were it supported by the source. Note that all the points fall on the upper-left side of the $y/x = 1$ axis because Q_{AND}'s subsume Q_{W}'s. In other words, the closer the points accumulate to the $y/x = 1$ axis, the better the Q_{AND}'s approximate the Q_{W}'s.

The cost of batch post-filtering indicated by $Size(\langle Q_{\mathsf{AND}}\rangle)$ varies greatly, from as little as 1 to on the order of 10^3. In principle, the only upper bound is the size of the queried collection. Given this significant variation, batch post-filtering may not always be feasible; the front-end can choose to do batch post-filtering when the preliminary result sizes are manageable.

In contrast, the results in terms of selectivity ratios indicate that incremental post-filtering is feasible in most cases. As Table 2 (middle column) shows, the overall average $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$'s range from 2.12 for Comput-Ti to 139.38 for Comput-Tx. Interestingly, the ratios decrease greatly if we exclude those samples of which Q_{W}'s get very few hits (*e.g.*, less than 5) (Table 2, right column). These "odd" samples



(a) Configuration Inspec-Ti (b) Configuration Inspec-Ab

Fig. 7. Results of AND-queries versus W-queries for Inspec-Ti and Inspec-Ab.

may simply suggest that the queried collection is not an appropriate source for the underlying phrases to begin with. Note that, in both cases, Q_{AND}'s tend to approximate Q_{W}'s better for shorter fields (*e.g.*, Title).

Interestingly, in all the four configurations, $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ values tend to decrease as $Size(\langle Q_{\mathsf{W}}\rangle)$ increases. This means that incremental processing complements batch processing well. That is, if batch processing does not work well because of large result sizes, then it is likely that incremental processing will be effective.

Table 2. Average selectivity ratios of AND-queries versus W-queries.

configuration	Avg. $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ (*all samples*)	Avg. $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ (*samples with* $Size(\langle Q_{\mathsf{W}}\rangle) \geq 5$)
Inspec-Ti	5.25	2.69
Inspec-Ab	32.17	13.74
Comput-Ti	2.12	1.62
Comput-Tx	139.38	38.48

As stated earlier, our cost metric, $SR(Q^S, Q)$, does not consider the sizes of documents in $\langle Q^S\rangle$ and $\langle Q\rangle$, which might be questionable in some cases. Specifically, $\langle Q^S\rangle$ and $\langle Q\rangle$ may not share the same average document size when the length of the search field can affect the query results, and in addition the search field length also dominates the document size. Table 2 shows that the results for the proximity translation depend greatly on the field lengths. Therefore, we also evaluated the average sizes for documents in the query answers. As expected, for short fields such as Title (Comput-Ti), the average document sizes are almost identical, with the ratio of $\langle Q_{\mathsf{AND}}\rangle$ documents to $\langle Q_{\mathsf{W}}\rangle$ documents being 1.05. In contrast, for longer fields (that determine the document sizes) such as Text, we do see a difference: in Comput-Tx, the average (document) size ratio is 2.03. Notice that this ratio is still insignificant in determining the total cost, as compared to the $SR(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ of 139.38. In other words, the document size is not a dominating factor of the total cost, although it does have slight implication.

We next report on the second set of the proximity experiments, in which we compared series of weaker operators to the W operator. Because a proximity operator specifies both the order and distance constraints, we investigated how the query selectivities degenerate as we relax either of the constraints.

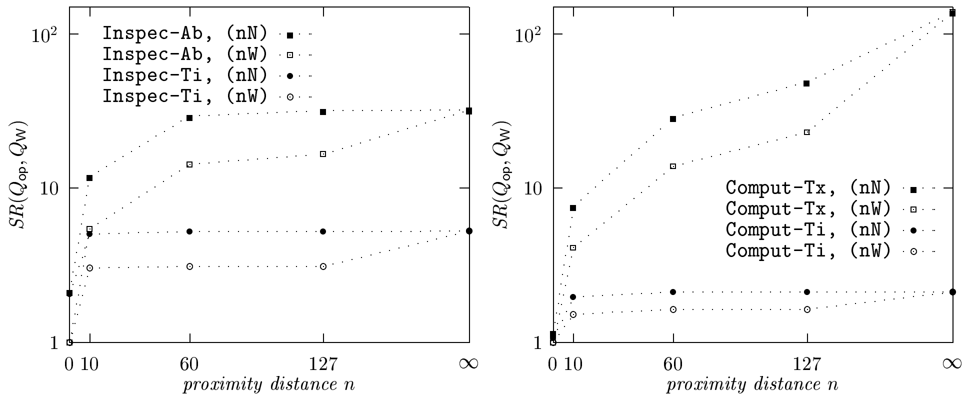
For each configuration, Figure 8 gives the average $SR(Q_{\mathsf{op}}, Q_{\mathsf{W}})$ values, when a query Q_{op} (with operator op) is compared to a query Q_{W} (with the most selective operator W). For each configuration we plot two curves: the first curve consists of a series of pairs [n, $SR(Q_{n\mathsf{W}}, Q_{\mathsf{W}})$], where $n \in \{0, 10, 60, 127\}$. That is, this curve represents the selectivity of the *ordered* proximity operators as the distance constraints are progressively relaxed. Similarly, the second curve is for the *unordered* proximity operators, *i.e.*, it plots the pairs [n, $SR(Q_{n\mathsf{N}}, Q_{\mathsf{W}})$], where $n \in \{0, 10, 60, 127, \infty\}$. (Note that ∞N represents the AND operator.) For example, looking at the configuration Inspec-Ti, we see in Figure 8(a) that the 60W operator has an overhead of $SR(Q_{\mathsf{60W}}, Q_{\mathsf{W}}) = 3.1$, while the 60N operator has an overhead of $SR(Q_{\mathsf{60N}}, Q_{\mathsf{W}}) = 5.2$.

Several remarks are noteworthy. First, the results for the nN-queries are relatively close to those of the nW-queries, with the former being no more than two times greater than the latter, which indicates the range of overhead for systems that do not provide the order constraint (*e.g.*, Figure 6, case (2) of Rule (1.1b)). Second, for systems with even only partial support of the proximity operators, the incremental post-filtering cost decreases significantly compared to those with no support. In other words, in a system that supports some operators stronger than AND (*e.g.*, *AltaVista*'s NEAR operator meaning 10N), the incremental cost may be greatly reduced with these operators. Third, the (unordered) proximity operators nN approximate the AND operator for n greater than some threshold value depending on the typical lengths of the search fields. For instance, for short fields as Inspec-Ti and Comput-Ti show, the nN operators start to approximate AND for $n \geq 10$. For longer fields such as Abstract, this threshold is about 60. Therefore, for queries on short fields (*e.g.*, bibliographic fields), lack of full support of the proximity operators may not be a crucial restriction, because the AND operator can approximate them well.

6.2 Summary of Other Experiments

The stopword experiments evaluated the rewriting rule for stopword removal (Section 5.2.1). We compared the result size of a sample query Q (*e.g.*, Contains(Text, video (W) on (W) demand)) containing stopwords (*e.g.*, on) to that of its subsuming query Q^S (*e.g.*, Contains(Text, video (1W) demand)), with stopwords removed (by the procedure in Figure 4). The sample queries are of the parametric form $Q =$ Contains(Text, w_1 op w_2 op $\cdots$ op w_n). We set up two configurations, which differ only in the connecting operator op: configuration Config-Prox uses W, while Config-Conj uses AND. To generate the queries, we selected from VOC_{Foldoc} all the phrases containing at least one stopword specified by the information service *Britannica Online* (`http://www.eb.com/`).

Figure 9 illustrates the distribution of the pairs [$Size(\langle Q \rangle), Size(\langle Q^S \rangle)$]. The



(a) Configurations Inspec-Ti and Inspec-Ab

(b) Configurations Comput-Ti and Comput-Tx

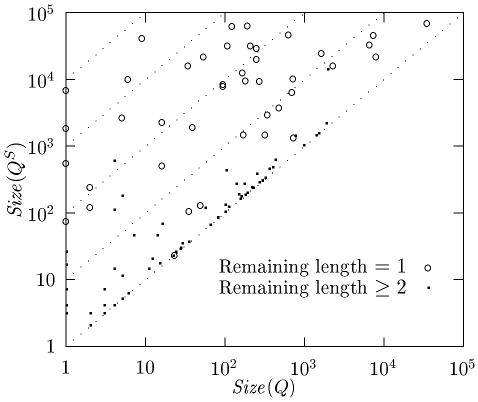
Fig. 8. Average $SR(Q_{\mathsf{op}}, Q_{\mathsf{W}})$'s for different operator op's.

results depend not only on the connecting operators (*i.e.*, AND or W), but also on the *remaining lengths* of the subsuming queries, *i.e.*, the numbers of search words remaining in the subsuming queries after the removal of stopwords.

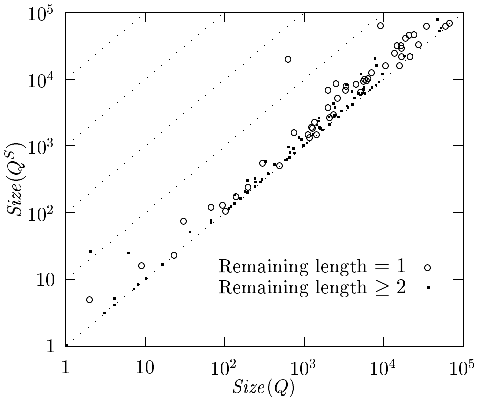
In all the cases except when the remaining length is 1 in configuration Config-Prox, the subsuming queries closely approximate the sample queries. In summary, first, unless a query contains mostly stopwords, the subsuming query with stopwords removed closely approximates the original query. Second, notice that stopword removal from conjunctive expressions does not reduce selectivity as much as was the case with proximity expressions, because AND only tests the occurrence of terms, which is almost guaranteed for stopwords.

In the last set of experiments, we studied the effects of rewriting predicates with the Equals operator to those with the Contains operator, *i.e.*, the schema level transformation (Section 5.1). The sample queries are of the form $Q =$ Equals(Title, "$w_1\ w_2\ \cdots\ w_n$"). Because we used *Inspec* (with a subset since year 1988) as the source collection, the experiments generated the sample queries using the complete Title values of *Inspec* citations (provided by the Stanford library). Because, in practice, users usually rely on truncation instead of giving the full phrase, we configured two sets of experiments: configuration Config-Full queries with full phrases, and Config-Trun assumes truncation to the first 5 words. For both configurations, the experiments generated subsuming queries of the form Contains(Title, w_1 op w_2 op $\cdots$ op w_n), where n is no greater than 5 for Config-Trun. Specifically, the first subsuming query Q_{W} uses the operator W to connect the words, which represents the best-case translation. However, if the target system does not support W, the query must be further transformed. Therefore, the second subsuming query Q_{AND} assumes that the underlying service does not support the proximity operators, and therefore uses AND as the connecting operator.

Surprisingly, the results (not shown fully here) demonstrate high consistency for



(a) Configuration Config-Prox



(b) Configuration Config-Conj

Fig. 9. Results of the stopword experiments.

all the sample queries. Because both the Equals–queries and the Contains-queries are extremely selective, almost all the Equals-queries return only one hit, and the Contains-queries closely approximate the Equals-queries with the hits ranging from 1 to 10. Because of the small result sizes, batch post-filtering is always feasible, while in the previous two sets of experiments there are cases when it is not. For the incremental cost, Table 3 shows the corresponding metric, *i.e.*, the average $SR(Q^S, Q)$'s. Overall, the selectivity reduction is about 2, which indicates that incremental costs may also be acceptable. Also, the costs increase slightly for truncated phrases, and for systems that do not support the proximity operators.

Table 3. Average selectivity ratios of the equality experiments.

configuration	Average $SR(Q_{\mathsf{W}}, Q)$	Average $SR(Q_{\mathsf{AND}}, Q)$
Config-Full	1.28	1.69
Config-Trun	1.29	2.14

7. CONCLUDING DISCUSSION

Search services support different query languages and varying access capabilities. To address this heterogeneity, we proposed a unifying front-end that provides the illusion of uniform capabilities across underlying services. A front-end does not internally manage data of its own; instead it relies on external services to provide necessary information for answering queries. To unify search, the front-end must translate user queries in a unified language into those natively supported by the underlying services. This translation also makes possible *multi-search* (over multiple sources) with a single user query. In this paper we gave an overview of the query translation process, and focused on predicate rewriting in particular. The front-end must also perform local post-filtering, in order to implement missing functionalities. This paper also summarized our experimental results illustrating the batch and incremental post-filtering costs.

We have implemented the algorithms presented in this paper in Digital Library testbed system. Currently, we translate queries for heterogeneous search services including *Dialog*, (Stanford's) *Folio*, *AltaVista*, *WebCrawler*, and *NCSTRL*, each with different Boolean query syntax and functionality. The results are quite encouraging: in most cases users get their results quickly, without having to know the different query languages. However, as pointed out earlier, there are situations in which our approach has drawbacks. We discuss these in turn, suggesting ways to cope with them.

The first problem is that a source may not provide the necessary information for translation. In summary, our rewriting algorithms require the following *metadata* defining the target's capability and schema (Figure 1): (1) the schema definition, (2) the supported operators, (3) the stopword list, (4) the vocabulary, and (5) the details of expansion features (*e.g.*, the supported truncation patterns). While most of them (*e.g.*, items 1, 2, 3) are usually documented, others (*e.g.*, the vocabulary) are currently harder to obtain. Note that the availability of service metadata is essential for interoperability in general, not just for query translation. Consequently, various standards or agreements have been developed for metadata acquisition. For example, the Z39.50 [National Information Standards Organization 1995] Explain Facility and the *STARTS* [Gravano et al. 1997] protocols require services to export their metadata. This metadata includes useful information for query translation, *e.g.*, searchable fields and the operators for searching the fields. Along these lines, we have also developed a metadata architecture [Baldonado et al. 1997a; Baldonado et al. 1997b] to facilitate metadata management.

A second potential translation problem is that a query may translate to *True*, in which case a source with a large corpus is generally unable to return all of its contents for filtering. However, our study shows that in most interesting and practical cases a better translation is possible. First, predicate rewrites to *True* are actually unlikely in practice. Referring to Figure 3, for positive rewriting, a predicate translates to *True* only when it involves a non-searchable field, or when its pattern contains only stopwords. Both cases are unusual, or can at least be easily avoided. Although negative rewrites to *False* are more likely, it may not be a serious problem because negation is rarely used (see footnote 4). Second, if the predicates translated to *True* appear in a conjunction, then the remaining conjuncts may provide reasonable selectivity.

A third drawback is the potential high cost of post-filtering. Our experiments show that the costs of batch post-filtering can vary greatly; the front-end can advise users of its feasibility based on the numbers of hits for the native queries. (As we discuss later, with user interaction, post-filtering may only have to take place when the results are of manageable sizes.) With incremental post-filtering, costs are incurred only as the user requests matching documents, so the user has control over the costs. With fast search engines and networks, combined with changing information access economics, the additional processing time and cost may be acceptable for users, given that they access information with less effort on their part.

There are variations to the basic translation scheme we have discussed in this paper that may also mitigate the drawbacks. For example, we can apply *approximate translation* that yields "slightly" different answers. To illustrate, consider a query with a truncation term. With exact translation, the term is replaced with all possible expansions, and this can yield an excessively large native query. Instead, we can decide not to enumerate all possible expansions. Similarly, suppose that the source vocabulary is not available at the front-end. We can use instead some common vocabulary, *e.g.*, from a dictionary suitable for the domain of interest. Another useful approximation is to map an unsupported field to "anywhere" (or the default fields supported by a service). These approximations do not guarantee a precise translation, but it may be acceptable, given the inherent uncertainty in information retrieval. We are currently extending our framework to incorporate approximate translation.

Another variation is to skip post-filtering. That is, the user query is still translated according to our algorithms, but all of the results of the subsuming query are given to the user. (Notice that our translation algorithms guarantee that no other native query could return fewer documents [Chang et al. 1996a; Chang et al. 1996b].) We still have to pay the overhead of fetching additional documents, but there is no filtering work at the front-end. The user may get documents that do not match the query, but the costs at the front-end are reduced. This strategy is used by *MetaCrawler* [Selberg and Etzioni 1995], which gives users the option to eliminate post-filtering that verifies phrase queries.

Query translation can also support an interactive environment where a user gets help in constructing queries, estimating their execution costs, and interpreting the results. For example, as a user develops a query, the translation system can indicate what components will be hard to translate, and suggest operators or terms that may be easier to use with the intended sources. The front-end can also estimate the expected post-filtering cost based on some standard cost functions (*e.g.*, our experimental results), identifying the expensive predicates, and advising the user accordingly. After the native query is evaluated, the front-end can report the preliminary result size. If the size is small, the user may decide to do batch post-filtering directly, which yields all the final results at once. Otherwise, the front-end may instead post-filter incrementally and estimate the final result size dynamically by extrapolation of the accumulated results. At any point, based on the execution status, the user can continue to refine the query until the results are manageable.

In this paper we have focused on the Boolean query model, because it is used by most commercial search services and library systems. However, there are also other kinds of popular query models, the most prominent being the vector-space model, and we have started to develop extensions for it. In the vector space model, documents are retrieved and ranked based on their "similarity" with queries. If a front-end decides to support the vector-space model, translation to an underlying service that also supports vector-space queries is relatively straightforward, because there is no strict syntax, there are typically no sophisticated features, and there are not many dialects as with Boolean queries. However, the major challenge is to collate results returned from different services, *i.e.*, to merge the different rankings [Fagin 1996; Voorhees and Tong 1997]. The problem is hard because all the search engines use proprietary ranking algorithms, and the details are not publicly available. To help meta-searchers perform the rank-merging, it is desirable that the services return some ranking information along with the results, as *STARTS* [Gravano et al. 1997] suggests. We have also studied how much data must be retrieved from a ranking source in order to do meaningful merging [Gravano and García-Molina 1997].

It is even harder to translate queries between different retrieval models, *e.g.*, from the Boolean model to the vector-space model, and vice versa. The retrieval semantics are fundamentally different; Boolean queries specify exact matches while vector-space queries are based upon statistical similarities. One way to integrate services that may use either model is to support a combined query model in the front-end, as *STARTS* suggests. That is, users specify both a selection criterion (Boolean) and a ranking criterion (vector-space). The front-end can then translate the appropriate part for each underlying service with the corresponding model, and post-process the other unexecuted part locally. However, there are still many open issues that need to be resolved with such inter-model execution.

REFERENCES

Baldonado, M., Chang, C.-C. K., Gravano, L., and Paepcke, A. 1997b. Metadata for digital libraries: Architecture and design rationale. In *Proceedings of the Second ACM International Conference on Digital Libraries* (Philadelphia, Pa., July 1997), pp. 47–56. ACM Press, New York.

Baldonado, M., Chang, C.-C. K., Gravano, L., and Paepcke, A. 1997a. The Stanford Digital Library metadata architecture. *International Journal on Digital Libraries 1*, 2 (Sept.), 108–121.

Baldonado, M. Q. W. and Winograd, T. 1997. SenseMaker: An information-exploration interface supporting the contextual evolution of a user's interests. In *Proceedings of the Conference on Human Factors in Computing Systems, CHI'97* (Atlanta, Ga., March 1997), pp. 11–18. ACM Press, New York.

Chang, C.-C. K. and García-Molina, H. 1997. Evaluating the cost of boolean query mapping. In *Proceedings of the Second ACM International Conference on Digital Libraries* (Philadelphia, Pa., July 1997), pp. 103–112. ACM Press, New York.

Chang, C.-C. K., García-Molina, H., and Paepcke, A. 1996a. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering 8*, 4 (Aug.), 515–521.

Chang, C.-C. K., García-Molina, H., and Paepcke, A. 1996b. Boolean query mapping across heterogeneous information sources (extended version). Technical Report SIDL-WP-1996-0044 (Sept.), Stanford Univ. Accessible at `http://www-diglib.stanford.edu`.

Crystal, M. I. and Jakobson, G. E. 1982. FRED, a front end for databases. *Online 6*, 5 (Sept.), 27–30.

Fagin, R. 1996. Combining fuzzy information from multiple systems. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Montreal, Canada, June 1996), pp. 216–226. ACM Press, New York.

Faloutsos, C. 1985. Access methods for text. *Computing Surveys 17*, 1 (March), 49–74.

Frakes, W. B. and Baeza-Yates, R. 1992. *Information Retrieval Data Structures & Algorithms.* Prentice Hall, Englewood Cliffs, N.J.

García-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. 1995. Integrating and accessing heterogeneous information sources in TSIMMIS. In *Proceedings of the AAAI Spring Symposium on Information Gathering* (Stanford, Calif., March 1995), pp. 61–64. AAAI Press, Menlo Park, Calif.

García-Molina, H., Labio, W., and Yerneni, R. 1998. Capability sensitive query processing on internet sources. Technical report, Stanford Univ. Accessible at `http://www-db.stanford.edu/`.

GRAVANO, L., CHANG, C.-C. K., GARCÍA-MOLINA, H., AND PAEPCKE, A. 1997. STARTS: Stanford proposal for Internet meta-searching. In *Proceedings of the 1997 ACM SIGMOD Conference* (Tucson, Ariz., May 1997), pp. 207–218. ACM Press, New York.

GRAVANO, L. AND GARCÍA-MOLINA, H. 1997. Merging ranks from heterogeneous internet sources. In *Proceedings of the 23rd VLDB Conference* (Athens, Greece, Aug. 1997), pp. 196–205. VLDB Endowment, Saratoga, Calif.

GRAVANO, L., GARCÍA-MOLINA, H., AND TOMASIC, A. 1994. The effectiveness of *GlOSS* for the text-database discovery problem. In *Proceedings of the 1994 ACM SIGMOD Conference* (Minneapolis, Minn., May 1994), pp. 126–137. ACM Press, New York.

HARMAN, D. 1993. Document detection overview. In *Proceedings TIPSTER Text Program (Phase I)* (Fredricksburg, Va., Sept. 1993). Morgan Kaufmann, San Francisco, Calif.

HAWKINS, D. T. AND LEVY, L. R. 1985. Front end software for online database searching Part 1: Definitions, system features, and evaluation. *Online 9*, 6 (Nov.), 30–37.

HOWE, D. 1997. The free on-line dictionary of computing. Accessible at `http://-wombat.doc.ic.ac.uk/`.

IEE. 1991. *INSPEC Thesaurus*. Institution of Electrical Engineers, London.

ISO. 1993. *ISO 8777:1993 Information and Documentation – Commands for Interactive Text Searching* (First ed.). Int'l Organization for Standardization, Geneva, Switzerland.

KETCHPEL, S. P., GARCIA-MOLINA, H., AND PAEPCKE, A. 1997. Shopping models: A flexible architecture for information commerce. In *Proceedings of the Second ACM International Conference on Digital Libraries* (Philadelphia, Pa., July 1997), pp. 65–74. ACM Press, New York.

KIRK, T., LEVY, A. Y., SAGIV, Y., AND SRIVASTAVA, D. 1995. The Information Manifold. In *Proceedings of the AAAI Spring Symposium on Information Gathering* (Stanford, Calif., March 1995), pp. 85–91. AAAI Press, Menlo Park, Calif.

LOEFFEN, A. 1994. Text databases: A survey of text models and systems. *SIGMOD Record 23*, 1 (March), 97–106.

LOVINS, J. B. 1968. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics 11*, 1-2, 22–31.

MARCUS, R. S. 1982. User assistance in bibliographic retrieval networks through a computer intermediary. *IEEE Trans. on Systems, Man, and Cybernetics smc-12*, 2, 116–133.

MARTIN, T. H. 1974. A feature analysis of interactive retrieval systems. Report SU-COMM-ICR-74-1 (Sept.), Institute of Communication Research, Stanford Univ., Stanford, Calif.

MCCLUSKEY, E. J. 1986. *Logic Design Principles*. Prentice Hall, Englewood Cliffs, N.J.

MITCHELL, P. C. 1973. A note about the proximity operators in information retrieval. In *Proceedings of ACM SIGPLAN-SIGIR Interface Meeting* (Gaithersburg, Md., Nov. 1973), pp. 177–180. ACM Press, New York.

NATIONAL INFORMATION STANDARDS ORGANIZATION. 1993. *Z39.58-1992 Common Command Language for Online Interactive Information Retrieval*. NISO Press, Bethesda, Md.

NATIONAL INFORMATION STANDARDS ORGANIZATION. 1995. *Information Retrieval (Z39.50): Application Service Definition and Protocol Specification (ANSI/NISO Z39.50-1995)*. NISO Press, Bethesda, Md. Accessible at `http://lcweb.loc.gov/z3950/agency/`.

NAVARRO, G. AND BAEZA-YATES, R. 1995. A language for queries on structure and contents of textual databases. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Seattle, Wash., July 1995), pp. 93–101. ACM Press, New York.

NEGUS, A. E. 1979. Development of the Euronet-Diane Common Command Language. In *Proceedings 3rd Int'l Online Information Meeting* (1979), pp. 95–98. Learned Information, Oxford, U.K.

PAEPCKE, A. 1993. An object-oriented view onto public, heterogeneous text databases. In *Proceedings of the 9th International Conference on Data Engineering* (Vienna, Austria, April 1993), pp. 484–493. IEEE Computer Society, Washington, D.C.

PAPAKONSTANTINOU, Y., GARCÍA-MOLINA, H., GUPTA, A., AND ULLMAN, J. 1995. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the 4th Inter-*

national Conference on Deductive and Object-Oriented Databases (Singapore, Dec. 1995), pp. 161–186. Springer, Berlin.

PORTER, M. F. 1980. An algorithm for suffix stripping. *Program 14*, 3 (July), 130–137.

PREECE, S. AND WILLIAMS, M. 1980. Software for the searcher's workbench. In *Proceedings of the 43rd American Society for Information Science Annual Meeting*, Volume 17 (Anaheim, Calif., Oct. 1980), pp. 403–405. Knowledge Industry Publications, White Plains, N.Y.

RAO, R., JANSSEN, B., AND RAJARAMAN, A. 1994. GAIA technical overview. Technical report, Xerox PARC.

RAO, R., RUSSEL, D., AND MACKINLAY, J. 1993. System components for embedded information retrieval from multiple disparate information sources. In *Proceedings of the ACM UIST '93* (Atlanta, Ga., Nov. 1993), pp. 23–33. ACM Press, New York.

SALTON, G. 1989. *Automatic Text Processing.* Addison-Wesley, Reading, Mass.

SELBERG, E. AND ETZIONI, O. 1995. Multi-service search and comparison using the MetaCrawler. In *Proceedings of the 4th International WWW Conference* (Boston, Mass., Dec. 1995).

TOLIVER, D. E. 1982. OL'SAM: An intelligent front-end for bibliographic information retrieval. *Information, Technology and Libraries 1*, 4, 317–326.

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, Rockville, Md.

ULLMAN, J. D. 1997. Information integration using logical views. In *Proceedings of the 6th International Conference on Database Theory* (Delphi, Greece, Jan. 1997). Springer, Berlin.

VOORHEES, E. M. AND TONG, R. M. 1997. Multiple search engines in database merging. In *Proceedings of the Second ACM International Conference on Digital Libraries* (Philadelphia, Pa., July 1997), pp. 93–102. ACM Press, New York.

WIEDERHOLD, G. 1992. Mediators in the architecture of future information systems. *IEEE Computer 25*, 3 (March), 51–60.

WILLIAMS, M. E. 1986. Transparent information systems through gateways, front ends, intermediaries, and interfaces. *Journal of the American Society for Information Science 37*, 4 (July), 204–214.

ZINN, S., SELLERS, M., AND BOHLI, D. 1986. OCLC's intelligent gateway service: Online information access for libraries. *Library Hi Tech 4*, 3, 25–29.