Boolean Query Mapping Across Heterogeneous Information Sources

Kevin Chen-Chuan Chang, Hector Garcia-Molina, Andreas Paepcke[*]

***Abstract*---Searching over heterogeneous information sources is difficult because of the non-uniform query languages. Our approach is to allow a user 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 result. In this paper we introduce the architecture and associated algorithms for generating the supported subsuming queries and filters. We show that generated subsuming queries return a minimal number of documents; we also discuss how minimal cost filters can be obtained. We have implemented prototype versions of these algorithms and demonstrated them on heterogeneous Boolean systems.**

***Index Terms*---Boolean queries, query translation, information retrieval, heterogeneity, digital libraries, query subsumption, filtering.**

I. 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. As a first step, in this paper we focus on translating Boolean queries [18][6], from a generalized form, into queries that only use the functionality and syntax provided by a particular target search engine. We initially look at Boolean queries because they are used by most current commercial systems; eventually we will incorporate other types of queries such as vector space and probabilistic-model ones [18][6]. The following example illustrates our approach.

Example 1.1 Suppose that a user is interested in documents discussing multiprocessors and distributed systems. Say the user's query is originally formulated as follows:

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 word "distributed" must immediately precede "system."

Now assume the user wishes to query the INSPEC database managed by the Stanford University Folio system.

*. K.C.-C. Chang is with the Dept. of Electrical Engineering, Stanford University, Stanford, CA 94305; e-mail: changcc@cs.stanford.edu.
H. Garcia-Molina and A. Paepcke are with the Dept. of Computer Science, Stanford University, Stanford, CA 94305; e-mail: {hector, paepcke}@cs.stanford.edu.

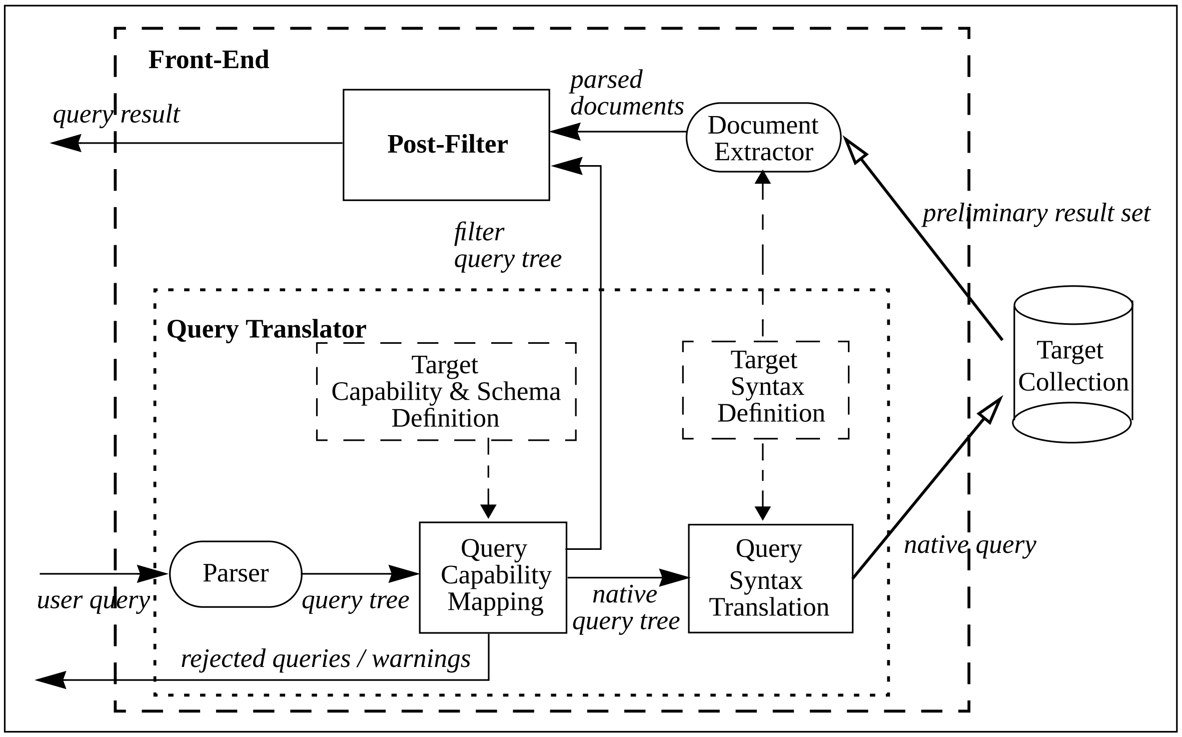

Fig. 1. The architecture of the front-end system illustrating query translation and post-filtering. The dashed boxes are target-specific metadata defining the target's syntax and capabilities.

Unfortunately, this source does not understand the (W) operator. In this case, our approach will be to approximate 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 native query that is sent to Folio-INSPEC is

Native Query: Find Title multiprocessor AND distributed AND system

Notice that now this query is expressed in the syntax understood by Folio. The native query will return a preliminary result set that is a super-set of what the user expects. Therefore, an additional post-filtering step is required at the front-end to eliminate from the preliminary result documents that do not have words "distributed" and "system" occurring next to each other. In particular, the filter query that is required is:

Filter Query: Title Contains distributed (W) system ❑

Fig. 1 shows the main components of the proposed front-end system. The user submits (lower left) a query in a powerful language that provides the combined functionality of the underlying sources. The figure shows how the query is then processed before sending to a target source; if the query is intended for multiple sources, the process can be repeated. First, the incoming 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 (by a process we will describe here) into the native query tree 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 user will not see have 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 three categories:

(1) Present inconsistent query capabilities specific to the target systems with no intention to hide the heterogeneity and have the end user write queries specifically for each;

(2) Provide a “least common denominator” front-end query language that can be supported by all sources;

(3) Copy all the document collections that a user may be interested 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. And even if there is a higher cost, it may be worth paying it to get the user the required documents with less effort on his 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, since there are many different operators provided by Boolean systems, and it is challenging to determine what other weaker operators can provide a super-set of results. Furthermore, as we will see, the transformation process also needs to consider the structure of the query tree, not just the individual operators.

Due to space limitations, in this short paper we only study the central query transformation algorithms (Query Capability Mapping box in Figure 1), and furthermore, leave some of the details for an extended technical report [1]. Also, there are several important issues that are not covered here. First, we only focus on the *feasibility* of the translations, not their cost. Some feasible translations may be too expensive to execute, so a system component (not discussed here) must inform the user that their query cannot be translated with a reasonable cost. (In such cases, the user will have to reformulate the query.) Second, we do not consider semantic mapping issues (e.g., how to know whether “author” on one system is really the same as “author” on another.) Here we simply assume we are given tables (and possible transformation functions) that specify how fields or attributes map to each other. Third, we do not discuss the implementation of the algorithms. However, we do note that the algorithms presented here have been implemented and used to transform queries for three systems, Knight-Ridder’s DIALOG, Stanford’s Folio, and Alta Vista (Digital Equipment Corporation), each with different Boolean query syntax and functionality. We are in the process of extending our query transformation system to other Boolean sources.

We start by briefly reviewing the alternative approaches suggested for access to heterogeneous search engines. In Section III we provide a brief overview of the Boolean query languages, while in Section IV we discuss the preliminary steps that are required for query transformation. Section V then describes the central algorithms that yield the query for the target source and the filter query.

II. 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 [11]. 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 [15], Z39.58 [14], and ISO 8777 [8]. 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 what we advocate. Reference [22] and [7] 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 previous section, their approaches generally fall into three 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 [17], OCLC’s Intelligent Gateway Service [23], and the more recent internet search services such as the All-in-One Search [3]. 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 [10], OL'SAM [20], and FRED [4]. 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 the query in the "pass-through" mode, in which the query is sent untranslated. This again compromises transparency.

Finally, there are systems that actually manage numbers of collections and do the search by themselves. For example, Knight-Ridder'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.

The closest works to ours are the recent development of *meta-searcher* on the internet such as MetaCrawler [19] and SavvySearch [5]. 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 other existing search services (e.g., WebCrawler, Lycos) to provide information necessary to fulfill 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.

III. Boolean Query Languages

In Boolean retrieval systems, queries are Boolean expressions consisting of *predicates* connected by the Boolean operators OR, AND, and NOT. A document is in the *result set* of a query if and only if the query evaluates to *True* for the document.

In Boolean systems, a document consists of a set of fields, each representing a particular kind of information such as Title, Author, and Abstract. In general a predicate consists of three components: a *predicate operator,* a *field designation*, and a *value expression*. For example, the predicate Contains(Title, cat$*$) evaluates to *True* for a document if it contains a word starting with the letters "cat" in its Title field. The predicate Equals(Author, "Joe Doe") is satisfied if the Author field is exactly equal to the string "Joe Doe." As seen in Example 1.1, value expressions can be compound, formed by connecting expressions by AND, OR, and proximity operators. For processing, we represent a predicate as a syntax tree, where the root is the predicate operator, the left child is the field designation, and the right child is a subtree representing the value expression. The predicates of a query are then combined into a query tree with the appropriate AND, OR, NOT operators; see Figure 2(a).

Boolean systems mainly differ in how they process predicates. First, they may have different fields in their documents, and may disallow searches over some fields (e.g., because they have not built an index). Second, they may support different types of operators and value expressions. For example, systems may support various kinds of proximity

Features	Z39.58 CCL	Stanfod Folio	U.C. Melvyl	Knight-Ridder's DIALOG	DEC's Alta Vista	CNIDR's freeWAIS
Boolean operators	√	√	√	√	√	√
fielded search	√	√	√	√	√	√
proximity operators:						
-(nN)	√	×	×	√	only (10N)	×
-(nW)	√	×	×	√	×	×
-same field	×	×	×	√	×	×
-sentence/paragraph	×	×	×	×	×	×
truncation						
-open right (wom∗)	√	√	√	√	√	√
-controlled right (wom??)	×	×	×	√	×	×
-internal (wom?n)	×	×	×	√	×	×
stemming	×	×	×	×	×	√
synonym expansion	×	×	×	×	×	√
Soundex/Phonix	×	×	×	×	×	√
stopwords	overrid-able	ignore	reject	no hits	no stop-words	no hits

Table 1. Feature comparison of query languages supported at various information sources.

expressions and operators for them. In the DIALOG language, the "(nW)" proximity operator specifies that its first operand must precede the second and no more than n words apart. The "(W)" operator is used when the distance is implicitly zero. If the order does not matter, operators "(nN)" and "(N)" may be used instead. However, these operators may not available in other systems, e.g., Folio supports none of these. Other features where systems differ include truncation, stemming [16][9], stopwords, etc. [18][6][1].

To illustrate, Table 1 provides feature comparison from our survey of several Boolean query languages. For example, all the systems define their own sets of stopwords, except Alta Vista in which all words are indexed. For systems having stopwords, if given a query containing stopwords, the systems may reject the query, ignore the stopwords, or simply return no hits. There are also languages that provide some way to override stopwords and make them searchable.

In this paper 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 most commercial Boolean search engines and found this to be true, with one exception: Most systems do not support the proper but degenerate query *True*. (We discuss the implications of this exception in Section V.)

IV. Query Capability Mapping

As discussed in the introduction, 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 language. The following definitions formalize these concepts. Notice that the notation $<Q>$ represents the result set of a query Q.

Definition 4.1. (Query Subsumption). A query Q' *subsumes* query Q ($Q' \supseteq Q$) if $<Q'> \supseteq <Q>$ regardless of the contents of the collection. If $<Q'> \supset <Q>$, then Q' *properly subsumes* Q ($Q' \supset Q$). ❑

Definition 4.2. (Minimal Subsuming Query). A query Q^S is the *minimal subsuming query* of query Q, or Q^S *minimally subsumes* Q, w.r.t. the 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. ❑

We will use the symbol Q^S to represent the minimal subsuming query of Q w.r.t. some target system that is clear from the context. After retrieving the results of a native query, we need a filter for locally removing the unnecessary answers.

Definition 4.3. (Filter). A query F is a *filter* for a query Q given its subsuming query Q', if $Q \equiv Q' \wedge F$. ❑

A filter always exists given that the native query subsumes the user query. To see this, note that the user query itself is *always* a correct filter. At the other extreme, F=*True* is also a possible filter when $Q \equiv Q^s$. (In this case no filtering is necessary.) In general, there may be more than one filter possible, and we are interested in one that requires the least processing effort.

Definition 4.4. (Optimal Filter). A query F is the *optimal filter*, w.r.t some processing cost definition, for a query Q with subsuming query Q', if F is a filter for Q and Q', and there is no query F' which is also a filter for Q and Q', and costs less than F under the cost definition. ❑

A. Overview of the Capability Mapping Process

The main steps for transforming a query (Query Capability Mapping box in Fig. 1) into its native query and filter are as follows. These steps will be described in more detail in the following subsections.

1. *Predicate atomization.* Starting from the query tree Q, this step outputs a logically equivalent query tree Q^a where all predicates are atomic. Tree Q^a is obtained by decomposing non-atomic predicates using the distributive law.
2. *Query normalization*. The tree Q^a is transformed into disjunctive normal form (DNF), Q^d, so that it is ready for Step 4 below. (Notice that this and the previous step are target-independent and therefore need only be done once if translations to multiple target languages are requested.)

3. *Predicate rewriting*: For each predicate P in Q^d, we rewrite it into its negative or positive subsuming form (or both) depending on whether it is negative, positive, or mixed in Q^d.
4. *Logic mapping*: Given the DNF of the query, Q^d, and the subsuming forms of the predicates, this step constructs the minimal native query and derives the optimal filter.

B. Step 1- Predicate Atomization

Predicates are the basic constructs of queries and hence the basis of query mapping. Sometimes a predicate contains logical conjunctions or disjunctions within it, and it is more effective to break it into simpler *atomic* predicates. For example, consider the predicate Contains(Title, multiprocessor AND distributed (W) system). It is equivalent to the conjunction of the following two predicates: Contains(Title, multiprocessor) AND Contains(Title, distributed (W) system). This atomization lets us separate predicates that may be unsupported at a target from those that are and hence leads to better native queries. Also, it makes it easy to determine the filtering that is required for each predicate in a query: if the atomic predicate is supported at the target, then no filtering is needed; if it is not, then the predicate itself (in its entirety) must be the filter.

A predicate $P=\Phi(F, E)$, where Φ is a predicate operator (e.g., Equals or Contains), F is a field designation, and E is a value expression, is atomic if there are no AND or OR operators in E that can be “pulled out” of the predicate. To decompose a predicate into its atomic terms, we apply the distributive law to the operator tree that defines the predicate. However, we have to be careful because AND cannot be distributed over certain operators (OR always can) [13]. For example, the predicate Contains(Title, multiprocessor (W) (distributed AND system)) is not equivalent to Contains(Title, multiprocessor (W) distributed) AND Contains(Title, multiprocessor (W) system). For additional details, see [1][2].

C. Step 2- Query Normalization

The next step is to transform the query into Disjunctive Normal Form (DNF) (see Fig. 2(c)). A Boolean expression is in DNF if it is the logical OR of clauses, which are the logical AND of normal or negated predicates. That is, $Q=\sum_{i=1..m} C_i$, where $C_i=\prod \tilde{P}_j$, and $\tilde{P}_j \in \{P_j, \bar{P}_j\}$. ($\bar{P}_j$ is the negation of P_j, i.e., $\neg P_j$.) C_i's are *conjunction terms* of some but not necessarily all (atomic) predicates P_j's defined in Q. Notice that the DNF representation is not canonical, that is, there may be more than one DNF's for a given query. As there are well-known algorithms for DNF transformations [12], we will not discuss the transformation here.

The normalization is in preparation for the ensuing steps. First, for Step 3, it is required that we know whether a predicate is negative, positive, or mixed in a query. In a DNF query (Fig. 2(c)) it is clear what predicates are negated. Second, for Step 4, it is required that the query be expressed in DNF trees so we can guarantee the minimality of the native query.

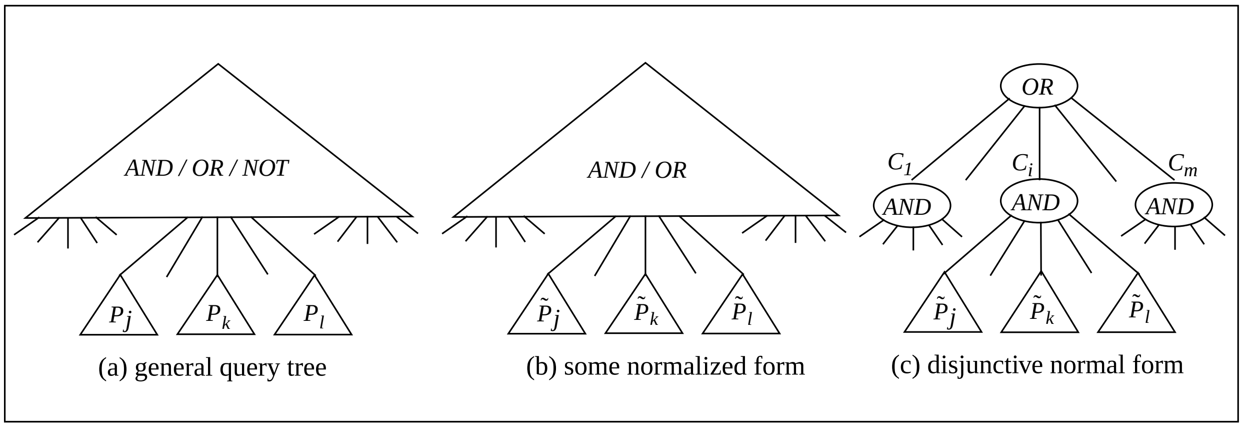

Fig. 2. Representations of a Boolean query. The P's represent predicate subtrees and $\tilde{P} \in \{P, \bar{P}\}$.

D. Step 3- Predicate Rewriting

The mapping of single predicates is the basis for query mappings as it ensures that the translated queries are at least doable by the target systems. A predicate is unsupported by a target system T if there are any operators not supported by T appearing in the predicate subtree. Considering individual predicates as simple queries, for each unsupported predicate, we must find its subsuming queries and the filter. As stated earlier, finding the filters for atomic predicates is straightforward.

The rewriting of unsupported predicates is a systematic procedure of replacing unsupported operators by supported ones. The proper substitutes are those supported operators that are weaker or stronger in the sense of selectivity and are as close to the unsupported operators as possible. The readers may refer to [2] and [1] for the details of predicate rewriting. Due to the space limitation, we illustrate the idea by the following example.

Example 4.1 (Predicate Rewriting): Consider the predicate P = Contains(Title, color (5W) printer) which means the Title must have the two words appearing no more than 5 words apart and in that order. Assume the only proximity operator available in the target system is the immediate adjacency operator "(W)" in which the distance is always implicitly zero. In this case, we would substitute "(5W)" by "AND" since "AND" is its 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 $\neg P$ by $\neg P^S$ 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 suggests that the unsupported operators in negated predicates should be replaced by its closest *stronger* substitute, i.e., "(5W)" should be replaced by "(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 by $\neg P^-$ and get a broader result set. ❑

As suggested in Example 4.1, we need different subsuming forms for positive or negative predicates. Formally, a query P^S is the *positive subsuming form* of the predicate P w.r.t. the target system T, if P^S minimally subsumes P w.r.t. T. Similarly, a query P^- is the *negative subsuming form* of the predicate P w.r.t. the target system T, if $\neg P^-$ minimally subsumes $\neg P$ w.r.t. T. Notice that we have $P^S \supseteq P \supseteq P^-$. In some extreme cases, it is possible that there is no *non-trivial* rewriting for either the positive or negative subsuming forms, in which case $P^S = True$ or $P^- = False$. 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...).

Notice that in some cases (non-trivial) subsuming predicates may be hard to obtain or may be unwieldy. For example, say the front-end query requests a "soundex" search for documents with the word "right." (This is a search for terms that sound like "right," e.g., "write," "wrt.") If the target source does not support this feature, the subsuming predicate must include a disjunction of all words that sound alike and the source *might* have. If the front end does not have access to the source's vocabulary, then this cannot be done, so the subsuming predicate will be *True*. Even if the source's vocabulary is available, the list of like-sounding terms may be large.

V. Logic Mapping Algorithms

A. Minimal Native Query Construction

By replacing predicates by their positive or negative subsuming forms (depending on whether they are negated or not) we obtain a native query that is executable by the target source. (By construction, the subsuming predicates are executable at the target. By our Section III assumption, the target can execute Boolean expressions of the supported predicates.) We can also show that the native query is correct (it subsumes the original query). This is because the AND/OR operators are *monotonic* in the sense that successively increasing operands yield non-decreasing results [21]. (The fact that NOT operators have been pushed below the top level of the query by some sort of normalization, Fig. 2(b)(c), is critical here. Otherwise, they could cause the subsumption relationship to be "reversed" when we consider the full query.)

In general, a constructed native query could be correct but not minimal. We illustrate this non-minimality in Example 5.1.

Example 5.1 (Native Query Construction): Consider the queries $Q_1 = (P_1 + P_2)(\bar{P}_1 + P_3)$ and $Q_2 = P_1 P_3 + \bar{P}_1 P_2$ where P_1, P_2, and P_3 are predicates. Notice that Q_1 and Q_2 are logically equivalent to each other and both of them are normalized, i.e., Q_1 is in CNF and Q_2 is in DNF. Now assume that P_2 and P_3 are supported in the target, while P_1 is not. Suppose that P_1^S is some arbitrary query and $P_1^- = False$. Substitution of the unsupported predicates yields $Q_1^S = P_1^S + P_2$, and $Q_2^S = P_1^S P_3 + P_2$. Clearly Q_1^S is not minimal because it at least subsumes Q_2^S. ❑

The reason that Q_1^S fails to be minimal is that, being a conjunction term, it does not satisfy the property of *inferential completeness*. Intuitively, notice that any answers satisfying Q_1 must also satisfy (P_2+P_3), a condition we denote as X. That is, from Q_1 one can infer X. Because P_2 and P_3 are supported by T, so is X. Moreover, although X is implied by Q_1, it is not implied by Q_1^S, which means that X can be conjuncted with Q_1^S to yield a smaller native query. That is, $Q_1^X = Q_1^S X$ is smaller than Q_1^S. In fact, it can be shown that Q_1^X is logically equivalent to Q_2^S. In summary, the existence of the condition X makes Q_1 fail to be inferentially complete, as defined below.

Definition 5.1. (Inferential Completeness of Conjunction). A conjunctive query $Q = Q_1 Q_2 ... Q_n$ is *inferentially complete* w.r.t. the target system T if, for any query X that can be inferred from Q (in which case $X \supseteq Q_1 Q_2 ... Q_n$), the minimal subsuming query of X w.r.t. T can also be inferred from the conjunction of the minimal subsuming queries of Q_i's w.r.t. T (in which case $X^S \supseteq Q_1^S Q_2^S ... Q_n^S$). ❑

The importance of inferential completeness is that it is both a necessary and sufficient condition for minimality to be preserved over AND operators. We formally state this in Theorem 5.1. For OR operators, this property is not required, as stated in Theorem 5.2. The reader may refer to [1] for the proofs.

Theorem 5.1. (Minimality Preserving over Conjunction): For a conjunctive query $Q = Q_1 Q_2 ... Q_n$, where Q_i's are the sub-queries of Q, the minimal subsuming query of Q w.r.t. the target system T is the conjunction of the minimal subsuming queries of Q_i's w.r.t. T, i.e., $Q^S = Q_1^S Q_2^S ... Q_n^S$, *if and only if* $Q_1 Q_2 ... Q_n$ is inferentially complete w.r.t. T.❑.

Theorem 5.2. (Minimality Preserving over Disjunction): For a disjunctive query $Q = Q_1 + Q_2 + ... + Q_n$, where Q_i's are sub-queries of Q, the minimal subsuming query of Q is the disjunction of the minimal subsuming queries of Q_i's, i.e., $Q^S = Q_1^S + Q_2^S + ... + Q_n^S$. ❑

These results motivate our use of DNF to represent queries (Fig. 2(c)). If the a query Q is in DNF, and we substitute each predicate by its minimal subsuming form, we obtain the minimal subsuming query for Q, provided that each conjunction term is inferentially complete. Since conjunction terms are made up of atomic predicates, we argue that this holds in the vast majority of cases. For completeness not to hold, the predicates would need to be "interrelated" and this is hard to achieve with atomic predicates. As a matter of fact, the only examples we can come up with are ones that no reasonable user would pose. (Example: $Q = P_1 P_2$, where P_1 is Equals(Title,"Disributed System") and P_2 is Equals(Title, "Color Printer"). Given that a document can have only one title, the minimal subsuming query is

False, which may not be obtained by $P_1^S P_2^S$.) Checking inferential completeness of conjunction terms depends not only on the semantics of the predicates but also on the target system under consideration. Thus, we doubt there is a computationally feasible way of checking, and even if there were, it would not be worth the effort since cases where it does not hold are so rare.

To summarize our discussion, we present our algorithm that generates native queries.

Algorithm 1. (DNF-Based Minimal Native Query Construction): Given a front-end query Q in DNF, with respect to the target system T, find the minimal subsuming query, $RESULT$.

- Initially, $RESULT=Q$.
- For each conjunction term C_i in $RESULT$ and for each normal or negated predicate $\tilde{P}_j$ in C_i,
  1. if $\tilde{P}_j = P_j$, i.e., P_j is positive in C_i, substitute P_j by P_j^S, the positive subsuming form of P_j w.r.t. T; otherwise,
  2. if $\tilde{P}_j = \bar{P}_j$, i.e., P_j is negative in C_i, substitute P_j by $P_j^{\bar{S}}$, the negative subsuming form of P_j w.r.t. T. ❑

Notice that if we apply Algorithm 1 to the query of Example 5.1, we obtain Q_2^S, the minimal subsuming query. Given our earlier results, we can now state the conditions under which the algorithm yields an optimal result:

Theorem 5.3. (Minimality of Algorithm 1): For any query Q, the native query given by Algorithm 1 is the minimal subsuming query of Q w.r.t. the target system, provided that every conjunction term in the DNF of Q satisfies inferential completeness. ❑

B. Optimal Filter Derivation

The logic mapping of the front-end queries is not complete without the derivation of the filters. Given a query Q and its native query Q^S, any query F satisfying Definition 4.3 is a correct filter. There may be more than one such filters that are not logically equivalent. Thus, we wish to choose the “best” one. At first glance one may think that the broadest filter, i.e., the one that subsumes the others, would be the best. However, since all valid filters will produce exactly the same result set (from the result of the native query), this is not the right metric to focus on. Instead, we would like the filter with the simplest Boolean expression, which will involve the smallest computational effort.

Example 5.2 (Filters): Consider the query $Q = P_1 + P_2\bar{P}_3$. Suppose P_1 and P_2 are supported by the target, and $P_3^{\bar{S}} = False$. Algorithm 1 gives $Q^S = P_1 + P_2$. Given Q and Q^S, the correct filters include $F_1 = P_1 + \bar{P}_2 + \bar{P}_3$, and $F_2 = P_1 + \bar{P}_3$. Both filters are valid since $Q = Q^S F_1$ and $Q = Q^S F_2$. Filter F_1 is broader than F_2, as it subsumes F_2. However, it is clear that F_2 is a better choice because it has a simpler expression which implies less processing cost under any normal cost definition. ❑

Correct filters are not difficult to derive. Intuitively, given a query Q and Q^S, if we can find all the necessary conditions that Q must imply, a filter can be composed as the conjunction of those necessary conditions that are not implied by Q^S. We refer to the not-implied conditions as the *residue* conditions. One way to find the necessary conditions that Q implies is to transform it into Conjunctive Normal Form (CNF). Written in CNF, $Q = \prod_{i = 1..m} D_i$, where $D_i = \sum \tilde{P}_j$, and $\tilde{P}_j \in \{P_j, \bar{P}_j\}$. D_i's are *disjunction terms* of some predicates Pj's defined in Q. Since the D_i's are conjuncted in Q, they are the necessary conditions that Q must satisfy. Any D_i containing unsupported predicates is a necessary condition that cannot be implied by Q^S, and therefore a residue condition. Consequently, a filter can be composed as the conjunction of those residue D_i's.

To illustrate this procedure, consider query Q of Example 5.2. Written in CNF, $Q = (P_1 + P_2)(P_1 + \bar{P}_3)$. Since $P_3^- = False$, the second disjunction term is a residue (the only one), and hence the filter is $F_2 = P_1 + \bar{P}_3$. In this case we obtained the optimal filter, but this is not always the case, as the following example illustrates.

Example 5.3 (Filter Derivation): Consider the query $Q = P_1\bar{P}_2P_3 + P_2\bar{P}_3$, and $Q^S = P_1P_3 + \bar{P}_3$ which is resulted from Algorithm 1 by assuming $P_2^S = True$ and $P_2^- = False$. Writing Q in the (minimal) CNF as $(P_2 + P_3)(\bar{P}_2 + \bar{P}_3)(P_1 + P_2)$, we find that the three disjunction terms all contain the unsupported predicate P_2. Therefore, the filter composed from the residue conditions is $F_1 = (P_2 + P_3)(\bar{P}_2 + \bar{P}_3)(P_1 + P_2)$. However, F_1 is not optimal; the reader may easily verify that $F_2 = (P_2 + P_3)(\bar{P}_2 + \bar{P}_3)$ is also correct, and is simpler than F_1. ❑

By comparing a query Q to its subsuming query Q^S, one can find a unique but incomplete specification for the family of all the correct filters. If further constrained by the cost definition, the optimal filter can be decided uniquely. Due to the space limitations, we are not able to present the algorithm here. The interested readers may refer to [1].

Our final theorem below combines the results we have presented. Theorem 5.3 has shown that a front-end query can be transformed into a minimal, subsuming native query. Furthermore, given a query Q, it is not hard to see that its minimal subsuming query is unique (if Q_1 and Q_2 are both minimal w.r.t. the target system T, then their conjunction, $Q' = Q_1Q_2$, must also be executable by T and is still smaller than both Q_1 and Q_2, a contradiction). For the filters, we presented a simple approach to derive correct filters. As we just mentioned, the optimal filter is also unique under some cost definition.

Theorem 5.4. (Query Capability Mapping): Given a query Q, the target system T, and the cost definition for post-filtering, the minimal subsuming query of Q w.r.t. T and the optimal filter w.r.t. the cost definition can always be found uniquely. ❑

Keep in mind that Theorem 5.4 only addresses the existence of a native query and filter, not its practicality or cost.

As a matter of fact, we will be unable to run some native queries because of “quirks” of real systems. For instance, we mentioned earlier that some systems do not accept the valid query *True*. This means that if our algorithm generates that as the minimal subsuming query for some user query *Q*, then it will be impossible to answer *Q*. Similarly, some systems may not export certain fields in their documents. So if a filter wishes to search locally one of those “hidden” fields, it will fail. In all these cases, as well as those where the native query is too expensive, the user will have to reformulate his query or access the source directly.

VI. Conclusion

This paper gave an overview of the query translation process and focused on the logic mapping algorithms. As pointed out earlier, there are situations in which our approach has a number of drawbacks or even fails. For one, in some cases translation can turn out to be too expensive. For example, the approach may create too much network traffic, or they may create queries that contain too many terms. The latter can happen, for instance, when truncation is approximated by enumerating terms over a source’s vocabulary. Another failure mode can be that a source simply does not provide the information the algorithms need. For example, it may not be possible to obtain a source’s vocabulary in order to provide approximations to truncation. Similarly, a source with a large corpus is generally not able to return all of its contents which the algorithms can in some cases call for when query translation produces *True* as the native query. In this case, the query cannot be executed.

In general, our algorithms for rewriting predicates, as briefly discussed in Section IV, require the following from the underlying search engines to perform a complete translation (this is the contents of the “Target Capability & Schema Definition” box in Fig. 1.):

1. the schema definition: the set of searchable fields, and the indexing scheme of each field.
2. the supported operators in addition to Boolean operators.
3. the stopword list.
4. the vocabulary: the word vocabulary, and the phrase vocabularies for phrase-indexed fields.
5. the details of expansion features, e.g., for stemming: the algorithm used; for truncation, the supported truncation patterns.

Among these, items 1, 2, 3, and the truncation patterns are usually documented for the end users to search the sources. The others are presently harder to obtain.

There are several factors which mitigate the drawbacks we have discussed. To reduce the cost for retrieving and post-filtering the entire preliminary result set, we may instead process the result *incrementally.* That is, if the user wishes to see, say a screenfull of documents, only some of the source’s documents must be retrieved and filtered. As the user requests more documents, more are processed.

We found that in many practically interesting cases a translation is possible. For example, our study shows that predicate rewrites to *Tru*e are actually unlikely in practice. Moreover, notice that trivial rewriting of a predicate does not necessarily imply that the entire native query becomes *True*. For example, when a query contains conjunctions of which one translates to *True*, the remaining terms are used to maintain reasonable selectivity. The native query degenerates to *True* only when there are no other components left. To be precise, the query translator generates *True* as the native query for a user query if and only if all predicates forming an *implicant* (a conjunction of predicates that implies the query) become *True*. For example, query $Q=P_1P_2+P_2P_3$ translates to *True* only when both P_1 and P_2, or P_2 and P_3 are rewritten to *True*.

For cases where translations produce excessively large queries, approximations can be applied. For the truncation emulation example, an implementation could decide not to expand to all possible terms. This would not yield a complete result. But depending on the user's task, a partial solution may be acceptable if it still produces "enough" documents. If some required metadata is not available, we believe that approximations can also help out. For example, a common vocabulary, such as the set of words from a dictionary can be used to approximate the emulation of truncation. Although this does not guarantee a precise translation, it might not be a fatal drawback given the inherent uncertainty in information retrieval. The approximation of using a common dictionary also greatly helps our approach to scale.

When a translation is truly impossible, our approach still provides a benefit. As shown in Fig. 1, we provide a feedback loop to the user. When a translation failure is detected, the user can be informed about precisely which part of the query is problematic. The user can then reformulate just that part.

As discussed in Section II, query unification has been attempted in various forms over the years. We believe that the increased power of search engines and machines available locally to users, combined with increased network bandwidth and changing information access economics, call for a re-examination of this area. In many cases it is now feasible to compensate for lacking retrieval features by extracting more information, and filtering it locally. This paper sketched our efforts in beginning such a re-examination.

Our initial prototype implementations are very encouraging. We have transformed the kinds of queries shown in this paper and have successfully executed them on very different search engines. Future work will involve schema unification (which we did not discuss here) and extensions for non-Boolean queries.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] K.C.-C. Chang, H. Garcia-Molina, and A. Paepcke, *Boolean Query Mapping Across Heterogeneous Information Sources*, Technical Report, in preparation. Dept. of Computer Science, Stanford Univ., 1996.

[2] K.C.-C. Chang, H. Garcia-Molina, and A. Paepcke, ”Predicate Rewriting for Translating Boolean Queries in a Heterogeneous Information System,” submitted for publication. URL: http://www-db.stanford.edu/pub/chang/1996/pred_rewriting.ps.

[3] W. Cross, *All-In-One Search Page*, URL: http://www.albany.net/allinone/.

[4] M.I. Crystal and G.E. Jakobson, “FRED, A Front End for Databases,” *Online*, vol. 6, no. 5, pp. 27-30, Sep. 1982.

[5] D. Dreilinger, *SavvySearch Home Page*, URL: http://www.cs.colostate.edu/~dreiling/smartform.html.

[6] W.B. Frakes and R. Baeza-Yates, *Information Retrieval Data Structures & Algorithm*. Englewood Cliffs, N.J.: Prentice Hall, 1992.

[7] D.T. Hawkins and L.R. Levy, “Front End Software for Online Database Searching Part1: Definitions, System Features, and Evaluation,” *Online* 9(6):30-37, Nov. 1985.

[8] ISO, *ISO 8777:1993 Information and Documentation -- Commands for Interactive Text Searching*. Geneva: Int’l Organization for Standardization.

[9] J.B. Lovins, “Development of a Stemming Algorithm”, *Mechanical Translation and Computational Linguistics*, vol. 11, no. 1-2, pp. 22-31, 1968.

[10] R.S. Marcus, “User Assistance in Bibliographic Retrieval Networks Through a Computer Intermediary,” *IEEE Trans. on Systems, Man, and Cybernetics*, vol. smc-12, no. 2, pp. 116-133, 1982.

[11] T.H. Martin, *A Feature Analysis of Interactive Retrieval Systems*, Report SU-COMM-ICR-74-1. Stanford, C.A.: Institute of Communication Research, Stanford University, Sep. 1974.

[12] E.J. McCluskey, *Logic Design Principles*. Englewood Cliffs, N.J.: Prentice Hall, 1986.

[13] P.C. Mitchell, “A Note about the Proximity Operators in Information Retrieval,” *Proc. of ACM SIGPLAN-SIGIR Interface Mtg.*, pp. 177-180, Nov. 1973.

[14] National Information Standards Organization, *Z39.58-1992 Common Command Language for Online Interactive Information Retrieval*. Bethesda, M.D.: NISO Press.

[15] A.E. Negus, “Development of the Euronet-Diane Common Command Language,” *Proc. 3rd Int’l Online Information Mtg.*, pp. 95-98, 1979.

[16] M.F. Porter, “An Algorithm for Suffix Stripping”, *Program*, vol.14, no. 3, 130-137, 1980.

[17] S.E. Preece and M.E. Williams, “Software for the Searcher’s Workbench,” *Proc. of the 43rd American Society for Information Science Annual Mtg.*, vol. 17, pp. 403-405, 1980.

[18] G. Salton, *Automatic Text Processing*. Reading, Mass.: Addison-Wesley, 1989.

[19] E. Selberg and O. Etzioni, “Multi-Service Search and Comparison using the MetaCrawler”, *Proc. of the 4th Int’l WWW Conf.*, URL:http://metacrawler.cs.washington.edu:8080/papers/www4/html/Overview.html.

[20] D.E. Toliver, “OL’SAM: An Intelligent Front-End for Bibliographic Information Retrieval,” *Information, Technology and Libraries*, vol. 1, no. 4, pp. 317-326, 1982.

[21] J.D. Ullman, *Database and Knowledge-Base Systems*, vol. 1. Rockville, M.D.: Computer Science Press, 1988, ch. 3, pp. 121-122.

[22] M.E. Williams, “Transparent Information Systems Through Gateways, Front Ends, Intermediaries, and Interfaces,” *Journal of the American Society for Information Science*, vol. 37, no. 4, pp. 204-214, Jul. 1986.

[23] S. Zinn, M. Sellers, and D. Bohli, “OCLC’s Intelligent Gateway Service: Online Information Access for Libraries”, *Library Hi Tech*, vol. 4, no. 3, pp. 25-29, 1986.

Boolean Query Mapping Across Heterogeneous Information Sources

Kevin Chen-Chuan Chang, Hector Garcia-Molina, Andreas Paepcke[*]

***Abstract*---Searching over heterogeneous information sources is difficult because of the non-uniform query languages. Our approach is to allow a user 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 result. In this paper we introduce the architecture and associated algorithms for generating the supported subsuming queries and filters. We show that generated subsuming queries return a minimal number of documents; we also discuss how minimal cost filters can be obtained. We have implemented prototype versions of these algorithms and demonstrated them on heterogeneous Boolean systems.**

***Index Terms*---Boolean queries, query translation, information retrieval, heterogeneity, digital libraries, query subsumption, filtering.**

I. 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. As a first step, in this paper we focus on translating Boolean queries [18][6], from a generalized form, into queries that only use the functionality and syntax provided by a particular target search engine. We initially look at Boolean queries because they are used by most current commercial systems; eventually we will incorporate other types of queries such as vector space and probabilistic-model ones [18][6]. The following example illustrates our approach.

Example 1.1 Suppose that a user is interested in documents discussing multiprocessors and distributed systems. Say the user's query is originally formulated as follows:

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 word "distributed" must immediately precede "system."

Now assume the user wishes to query the INSPEC database managed by the Stanford University Folio system.

*. K.C.-C. Chang is with the Dept. of Electrical Engineering, Stanford University, Stanford, CA 94305; e-mail: changcc@cs.stanford.edu.
H. Garcia-Molina and A. Paepcke are with the Dept. of Computer Science, Stanford University, Stanford, CA 94305; e-mail: {hector, paepcke}@cs.stanford.edu.

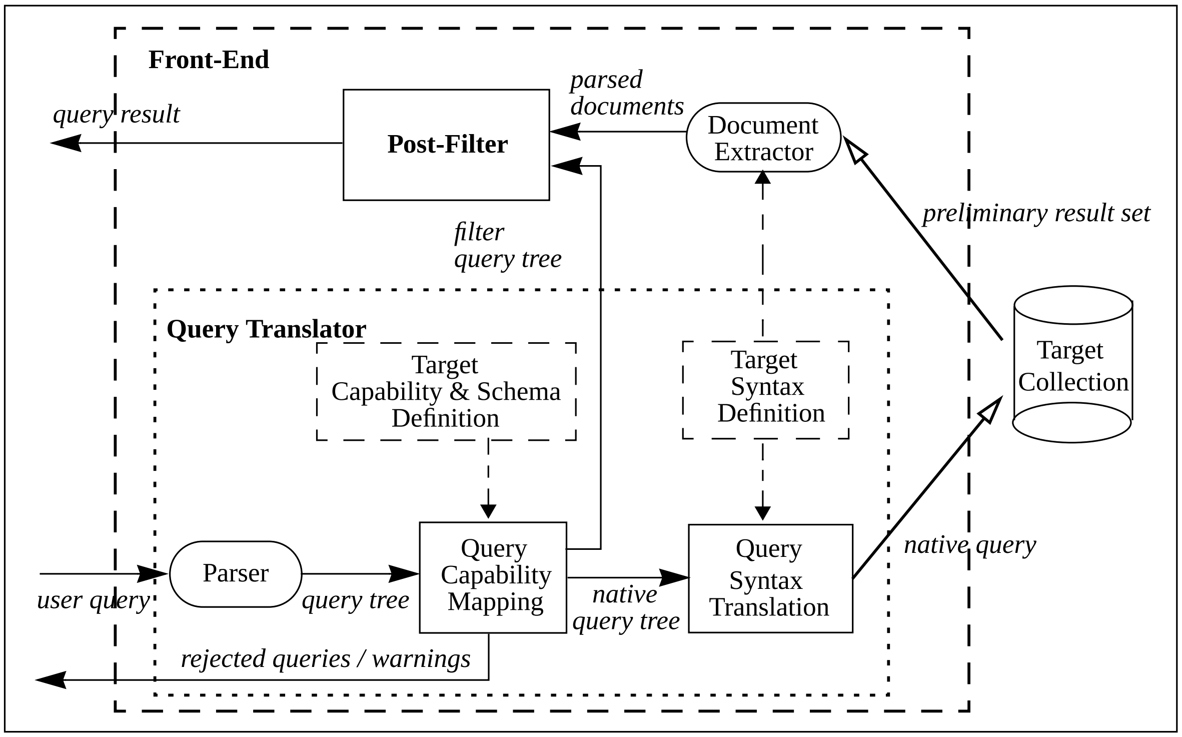

Fig. 1. The architecture of the front-end system illustrating query translation and post-filtering. The dashed boxes are target-specific metadata defining the target's syntax and capabilities.

Unfortunately, this source does not understand the (W) operator. In this case, our approach will be to approximate 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 native query that is sent to Folio-INSPEC is

Native Query: Find Title multiprocessor AND distributed AND system

Notice that now this query is expressed in the syntax understood by Folio. The native query will return a preliminary result set that is a super-set of what the user expects. Therefore, an additional post-filtering step is required at the front-end to eliminate from the preliminary result documents that do not have words "distributed" and "system" occurring next to each other. In particular, the filter query that is required is:

Filter Query: Title Contains distributed (W) system ❑

Fig. 1 shows the main components of the proposed front-end system. The user submits (lower left) a query in a powerful language that provides the combined functionality of the underlying sources. The figure shows how the query is then processed before sending to a target source; if the query is intended for multiple sources, the process can be repeated. First, the incoming 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 (by a process we will describe here) into the native query tree 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 user will not see have 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 three categories:

(1) Present inconsistent query capabilities specific to the target systems with no intention to hide the heterogeneity and have the end user write queries specifically for each;

(2) Provide a “least common denominator” front-end query language that can be supported by all sources;

(3) Copy all the document collections that a user may be interested 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. And even if there is a higher cost, it may be worth paying it to get the user the required documents with less effort on his 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, since there are many different operators provided by Boolean systems, and it is challenging to determine what other weaker operators can provide a super-set of results. Furthermore, as we will see, the transformation process also needs to consider the structure of the query tree, not just the individual operators.

Due to space limitations, in this short paper we only study the central query transformation algorithms (Query Capability Mapping box in Figure 1), and furthermore, leave some of the details for an extended technical report [1]. Also, there are several important issues that are not covered here. First, we only focus on the *feasibility* of the translations, not their cost. Some feasible translations may be too expensive to execute, so a system component (not discussed here) must inform the user that their query cannot be translated with a reasonable cost. (In such cases, the user will have to reformulate the query.) Second, we do not consider semantic mapping issues (e.g., how to know whether "author" on one system is really the same as "author" on another.) Here we simply assume we are given tables (and possible transformation functions) that specify how fields or attributes map to each other. Third, we do not discuss the implementation of the algorithms. However, we do note that the algorithms presented here have been implemented and used to transform queries for three systems, Knight-Ridder's DIALOG, Stanford's Folio, and Alta Vista (Digital Equipment Corporation), each with different Boolean query syntax and functionality. We are in the process of extending our query transformation system to other Boolean sources.

We start by briefly reviewing the alternative approaches suggested for access to heterogeneous search engines. In Section III we provide a brief overview of the Boolean query languages, while in Section IV we discuss the preliminary steps that are required for query transformation. Section V then describes the central algorithms that yield the query for the target source and the filter query.

II. 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 [11]. 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 [15], Z39.58 [14], and ISO 8777 [8]. 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 what we advocate. Reference [22] and [7] 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 previous section, their approaches generally fall into three 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 [17], OCLC's Intelligent Gateway Service [23], and the more recent internet search services such as the All-in-One Search [3]. 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 [10], OL'SAM [20], and FRED [4]. 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 the query in the "pass-through" mode, in which the query is sent untranslated. This again compromises transparency.

Finally, there are systems that actually manage numbers of collections and do the search by themselves. For example, Knight-Ridder'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.

The closest works to ours are the recent development of *meta-searcher* on the internet such as MetaCrawler [19] and SavvySearch [5]. 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 other existing search services (e.g., WebCrawler, Lycos) to provide information necessary to fulfill 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.

III. Boolean Query Languages

In Boolean retrieval systems, queries are Boolean expressions consisting of *predicates* connected by the Boolean operators OR, AND, and NOT. A document is in the *result set* of a query if and only if the query evaluates to *True* for the document.

In Boolean systems, a document consists of a set of fields, each representing a particular kind of information such as Title, Author, and Abstract. In general a predicate consists of three components: a *predicate operator,* a *field designation*, and a *value expression*. For example, the predicate Contains(Title, cat$*$) evaluates to *True* for a document if it contains a word starting with the letters "cat" in its Title field. The predicate Equals(Author, "Joe Doe") is satisfied if the Author field is exactly equal to the string "Joe Doe." As seen in Example 1.1, value expressions can be compound, formed by connecting expressions by AND, OR, and proximity operators. For processing, we represent a predicate as a syntax tree, where the root is the predicate operator, the left child is the field designation, and the right child is a subtree representing the value expression. The predicates of a query are then combined into a query tree with the appropriate AND, OR, NOT operators; see Figure 2(a).

Boolean systems mainly differ in how they process predicates. First, they may have different fields in their documents, and may disallow searches over some fields (e.g., because they have not built an index). Second, they may support different types of operators and value expressions. For example, systems may support various kinds of proximity

Features	Z39.58 CCL	Stanfod Folio	U.C. Melvyl	Knight-Ridder's DIALOG	DEC's Alta Vista	CNIDR's freeWAIS
Boolean operators	√	√	√	√	√	√
fielded search	√	√	√	√	√	√
proximity operators:						
-(nN)	√	×	×	√	only (10N)	×
-(nW)	√	×	×	√	×	×
-same field	×	×	×	√	×	×
-sentence/paragraph	×	×	×	×	×	×
truncation						
-open right (wom*)	√	√	√	√	√	√
-controlled right (wom??)	×	×	×	√	×	×
-internal (wom?n)	×	×	×	√	×	×
stemming	×	×	×	×	×	√
synonym expansion	×	×	×	×	×	√
Soundex/Phonix	×	×	×	×	×	√
stopwords	overrid-able	ignore	reject	no hits	no stop-words	no hits

Table 1. Feature comparison of query languages supported at various information sources.

expressions and operators for them. In the DIALOG language, the "(nW)" proximity operator specifies that its first operand must precede the second and no more than n words apart. The "(W)" operator is used when the distance is implicitly zero. If the order does not matter, operators "(nN)" and "(N)" may be used instead. However, these operators may not available in other systems, e.g., Folio supports none of these. Other features where systems differ include truncation, stemming [16][9], stopwords, etc. [18][6][1].

To illustrate, Table 1 provides feature comparison from our survey of several Boolean query languages. For example, all the systems define their own sets of stopwords, except Alta Vista in which all words are indexed. For systems having stopwords, if given a query containing stopwords, the systems may reject the query, ignore the stopwords, or simply return no hits. There are also languages that provide some way to override stopwords and make them searchable.

In this paper 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 most commercial Boolean search engines and found this to be true, with one exception: Most systems do not support the proper but degenerate query *True*. (We discuss the implications of this exception in Section V.)

IV. Query Capability Mapping

As discussed in the introduction, 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 language. The following definitions formalize these concepts. Notice that the notation $<Q>$ represents the result set of a query Q.

Definition 4.1. (Query Subsumption). A query Q' *subsumes* query Q ($Q' \supseteq Q$) if $<Q'> \supseteq <Q>$ regardless of the contents of the collection. If $<Q'> \supset <Q>$, then Q' *properly subsumes* Q ($Q' \supset Q$). ❏

Definition 4.2. (Minimal Subsuming Query). A query Q^S is the *minimal subsuming query* of query Q, or Q^S *minimally subsumes* Q, w.r.t. the 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. ❏

We will use the symbol Q^S to represent the minimal subsuming query of Q w.r.t. some target system that is clear from the context. After retrieving the results of a native query, we need a filter for locally removing the unnecessary answers.

Definition 4.3. (Filter). A query F is a *filter* for a query Q given its subsuming query Q', if $Q \equiv Q' \wedge F$. ❏

A filter always exists given that the native query subsumes the user query. To see this, note that the user query itself is *always* a correct filter. At the other extreme, $F=True$ is also a possible filter when $Q \equiv Q^s$. (In this case no filtering is necessary.) In general, there may be more than one filter possible, and we are interested in one that requires the least processing effort.

Definition 4.4. (Optimal Filter). A query F is the *optimal filter*, w.r.t some processing cost definition, for a query Q with subsuming query Q', if F is a filter for Q and Q', and there is no query F' which is also a filter for Q and Q', and costs less than F under the cost definition. ❏

A. Overview of the Capability Mapping Process

The main steps for transforming a query (Query Capability Mapping box in Fig. 1) into its native query and filter are as follows. These steps will be described in more detail in the following subsections.

1. *Predicate atomization.* Starting from the query tree Q, this step outputs a logically equivalent query tree Q^a where all predicates are atomic. Tree Q^a is obtained by decomposing non-atomic predicates using the distributive law.
2. *Query normalization*. The tree Q^a is transformed into disjunctive normal form (DNF), Q^d, so that it is ready for Step 4 below. (Notice that this and the previous step are target-independent and therefore need only be done once if translations to multiple target languages are requested.)

3. *Predicate rewriting*: For each predicate P in Q^d, we rewrite it into its negative or positive subsuming form (or both) depending on whether it is negative, positive, or mixed in Q^d.
4. *Logic mapping*: Given the DNF of the query, Q^d, and the subsuming forms of the predicates, this step constructs the minimal native query and derives the optimal filter.

B. Step 1- Predicate Atomization

Predicates are the basic constructs of queries and hence the basis of query mapping. Sometimes a predicate contains logical conjunctions or disjunctions within it, and it is more effective to break it into simpler *atomic* predicates. For example, consider the predicate Contains(Title, multiprocessor AND distributed (W) system). It is equivalent to the conjunction of the following two predicates: Contains(Title, multiprocessor) AND Contains(Title, distributed (W) system). This atomization lets us separate predicates that may be unsupported at a target from those that are and hence leads to better native queries. Also, it makes it easy to determine the filtering that is required for each predicate in a query: if the atomic predicate is supported at the target, then no filtering is needed; if it is not, then the predicate itself (in its entirety) must be the filter.

A predicate $P=\Phi(F, E)$, where Φ is a predicate operator (e.g., Equals or Contains), F is a field designation, and E is a value expression, is atomic if there are no AND or OR operators in E that can be "pulled out" of the predicate. To decompose a predicate into its atomic terms, we apply the distributive law to the operator tree that defines the predicate. However, we have to be careful because AND cannot be distributed over certain operators (OR always can) [13]. For example, the predicate Contains(Title, multiprocessor (W) (distributed AND system)) is not equivalent to Contains(Title, multiprocessor (W) distributed) AND Contains(Title, multiprocessor (W) system). For additional details, see [1][2].

C. Step 2- Query Normalization

The next step is to transform the query into Disjunctive Normal Form (DNF) (see Fig. 2(c)). A Boolean expression is in DNF if it is the logical OR of clauses, which are the logical AND of normal or negated predicates. That is, $Q=\sum_{i=1..m} C_i$, where $C_i=\prod \tilde{P}_j$, and $\tilde{P}_j \in \{P_j, \bar{P}_j\}$. ($\bar{P}_j$ is the negation of P_j, i.e., $\neg P_j$.) C_i's are *conjunction terms* of some but not necessarily all (atomic) predicates P_j's defined in Q. Notice that the DNF representation is not canonical, that is, there may be more than one DNF's for a given query. As there are well-known algorithms for DNF transformations [12], we will not discuss the transformation here.

The normalization is in preparation for the ensuing steps. First, for Step 3, it is required that we know whether a predicate is negative, positive, or mixed in a query. In a DNF query (Fig. 2(c)) it is clear what predicates are negated. Second, for Step 4, it is required that the query be expressed in DNF trees so we can guarantee the minimality of the native query.

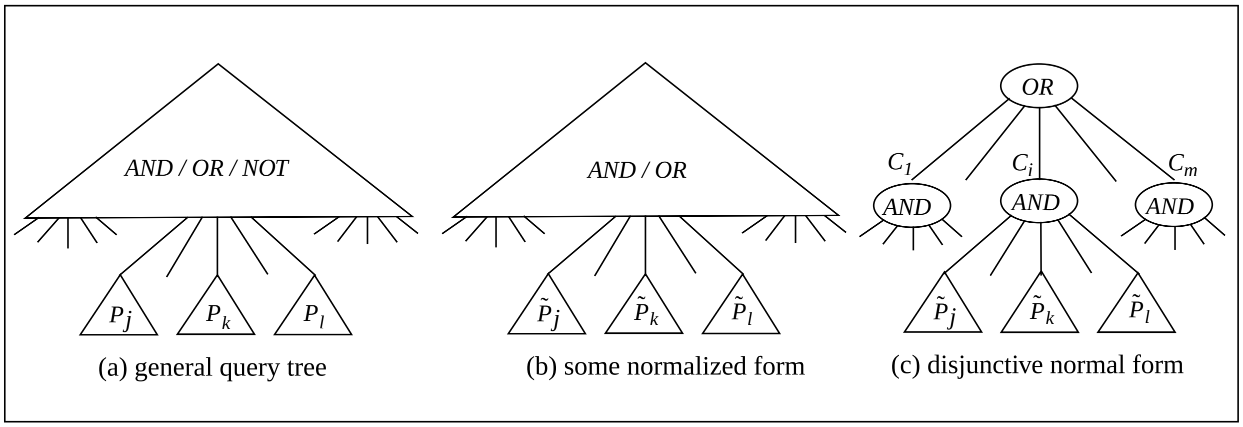

Fig. 2. Representations of a Boolean query. The P's represent predicate subtrees and $\tilde{P} \in \{P, \bar{P}\}$.

D. Step 3- Predicate Rewriting

The mapping of single predicates is the basis for query mappings as it ensures that the translated queries are at least doable by the target systems. A predicate is unsupported by a target system T if there are any operators not supported by T appearing in the predicate subtree. Considering individual predicates as simple queries, for each unsupported predicate, we must find its subsuming queries and the filter. As stated earlier, finding the filters for atomic predicates is straightforward.

The rewriting of unsupported predicates is a systematic procedure of replacing unsupported operators by supported ones. The proper substitutes are those supported operators that are weaker or stronger in the sense of selectivity and are as close to the unsupported operators as possible. The readers may refer to [2] and [1] for the details of predicate rewriting. Due to the space limitation, we illustrate the idea by the following example.

Example 4.1 (Predicate Rewriting): Consider the predicate P = Contains(Title, color (5W) printer) which means the Title must have the two words appearing no more than 5 words apart and in that order. Assume the only proximity operator available in the target system is the immediate adjacency operator "(W)" in which the distance is always implicitly zero. In this case, we would substitute "(5W)" by "AND" since "AND" is its 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 $\neg P$ by $\neg P^S$ 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 suggests that the unsupported operators in negated predicates should be replaced by its closest *stronger* substitute, i.e., "(5W)" should be replaced by "(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 by $\neg P^-$ and get a broader result set. ❑

As suggested in Example 4.1, we need different subsuming forms for positive or negative predicates. Formally, a query P^S is the *positive subsuming form* of the predicate P w.r.t. the target system T, if P^S minimally subsumes P w.r.t. T. Similarly, a query P^- is the *negative subsuming form* of the predicate P w.r.t. the target system T, if $\neg P^-$ minimally subsumes $\neg P$ w.r.t. T. Notice that we have $P^S \supseteq P \supseteq P^-$. In some extreme cases, it is possible that there is no *non-trivial* rewriting for either the positive or negative subsuming forms, in which case P^S=*True* or P^-=*False*. 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...).

Notice that in some cases (non-trivial) subsuming predicates may be hard to obtain or may be unwieldy. For example, say the front-end query requests a "soundex" search for documents with the word "right." (This is a search for terms that sound like "right," e.g., "write," "wrt.") If the target source does not support this feature, the subsuming predicate must include a disjunction of all words that sound alike and the source *might* have. If the front end does not have access to the source's vocabulary, then this cannot be done, so the subsuming predicate will be *True*. Even if the source's vocabulary is available, the list of like-sounding terms may be large.

V. Logic Mapping Algorithms

A. Minimal Native Query Construction

By replacing predicates by their positive or negative subsuming forms (depending on whether they are negated or not) we obtain a native query that is executable by the target source. (By construction, the subsuming predicates are executable at the target. By our Section III assumption, the target can execute Boolean expressions of the supported predicates.) We can also show that the native query is correct (it subsumes the original query). This is because the AND/OR operators are *monotonic* in the sense that successively increasing operands yield non-decreasing results [21]. (The fact that NOT operators have been pushed below the top level of the query by some sort of normalization, Fig. 2(b)(c), is critical here. Otherwise, they could cause the subsumption relationship to be "reversed" when we consider the full query.)

In general, a constructed native query could be correct but not minimal. We illustrate this non-minimality in Example 5.1.

Example 5.1 (Native Query Construction): Consider the queries $Q_1 = (P_1 + P_2)(\bar{P}_1 + P_3)$ and $Q_2 = P_1P_3 + \bar{P}_1P_2$ where P_1, P_2, and P_3 are predicates. Notice that Q_1 and Q_2 are logically equivalent to each other and both of them are normalized, i.e., Q_1 is in CNF and Q_2 is in DNF. Now assume that P_2 and P_3 are supported in the target, while P_1 is not. Suppose that P_1^S is some arbitrary query and $P_1^- = False$. Substitution of the unsupported predicates yields $Q_1^S = P_1^S + P_2$, and $Q_2^S = P_1^S P_3 + P_2$. Clearly Q_1^S is not minimal because it at least subsumes Q_2^S. ❑

The reason that Q_1^S fails to be minimal is that, being a conjunction term, it does not satisfy the property of *inferential completeness*. Intuitively, notice that any answers satisfying Q_1 must also satisfy (P_2+P_3), a condition we denote as X. That is, from Q_1 one can infer X. Because P_2 and P_3 are supported by T, so is X. Moreover, although X is implied by Q_1, it is not implied by Q_1^S, which means that X can be conjuncted with Q_1^S to yield a smaller native query. That is, $Q_1^X = Q_1^S X$ is smaller than Q_1^S. In fact, it can be shown that Q_1^X is logically equivalent to Q_2^S. In summary, the existence of the condition X makes Q_1 fail to be inferentially complete, as defined below.

Definition 5.1. (Inferential Completeness of Conjunction). A conjunctive query $Q = Q_1 Q_2 ... Q_n$ is *inferentially complete* w.r.t. the target system T if, for any query X that can be inferred from Q (in which case $X \supseteq Q_1 Q_2 ... Q_n$), the minimal subsuming query of X w.r.t. T can also be inferred from the conjunction of the minimal subsuming queries of Q_i's w.r.t. T (in which case $X^S \supseteq Q_1^S Q_2^S ... Q_n^S$). ❑

The importance of inferential completeness is that it is both a necessary and sufficient condition for minimality to be preserved over AND operators. We formally state this in Theorem 5.1. For OR operators, this property is not required, as stated in Theorem 5.2. The reader may refer to [1] for the proofs.

Theorem 5.1. (Minimality Preserving over Conjunction): For a conjunctive query $Q = Q_1 Q_2 ... Q_n$, where Q_i's are the sub-queries of Q, the minimal subsuming query of Q w.r.t. the target system T is the conjunction of the minimal subsuming queries of Q_i's w.r.t. T, i.e., $Q^S = Q_1^S Q_2^S ... Q_n^S$, *if and only if* $Q_1 Q_2 ... Q_n$ is inferentially complete w.r.t. T.❑.

Theorem 5.2. (Minimality Preserving over Disjunction): For a disjunctive query $Q = Q_1 + Q_2 + ... + Q_n$, where Q_i's are sub-queries of Q, the minimal subsuming query of Q is the disjunction of the minimal subsuming queries of Q_i's, i.e., $Q^S = Q_1^S + Q_2^S + ... + Q_n^S$. ❑

These results motivate our use of DNF to represent queries (Fig. 2(c)). If the a query Q is in DNF, and we substitute each predicate by its minimal subsuming form, we obtain the minimal subsuming query for Q, provided that each conjunction term is inferentially complete. Since conjunction terms are made up of atomic predicates, we argue that this holds in the vast majority of cases. For completeness not to hold, the predicates would need to be “interrelated” and this is hard to achieve with atomic predicates. As a matter of fact, the only examples we can come up with are ones that no reasonable user would pose. (Example: $Q = P_1 P_2$, where P_1 is Equals(Title,”Disributed System”) and P_2 is Equals(Title, “Color Printer”). Given that a document can have only one title, the minimal subsuming query is

False, which may not be obtained by $P_1^S P_2^S$.) Checking inferential completeness of conjunction terms depends not only on the semantics of the predicates but also on the target system under consideration. Thus, we doubt there is a computationally feasible way of checking, and even if there were, it would not be worth the effort since cases where it does not hold are so rare.

To summarize our discussion, we present our algorithm that generates native queries.

Algorithm 1. (DNF-Based Minimal Native Query Construction): Given a front-end query Q in DNF, with respect to the target system T, find the minimal subsuming query, $RESULT$.

- Initially, $RESULT=Q$.
- For each conjunction term C_i in $RESULT$ and for each normal or negated predicate $\tilde{P}_j$ in C_i,
  1. if $\tilde{P}_j = P_j$, i.e., P_j is positive in C_i, substitute P_j by P_j^S, the positive subsuming form of P_j w.r.t. T; otherwise,
  2. if $\tilde{P}_j = \bar{P}_j$, i.e., P_j is negative in C_i, substitute P_j by $P_j^{\bar{S}}$, the negative subsuming form of P_j w.r.t. T. ❑

Notice that if we apply Algorithm 1 to the query of Example 5.1, we obtain Q_2^S, the minimal subsuming query. Given our earlier results, we can now state the conditions under which the algorithm yields an optimal result:

Theorem 5.3. (Minimality of Algorithm 1): For any query Q, the native query given by Algorithm 1 is the minimal subsuming query of Q w.r.t. the target system, provided that every conjunction term in the DNF of Q satisfies inferential completeness. ❑

B. Optimal Filter Derivation

The logic mapping of the front-end queries is not complete without the derivation of the filters. Given a query Q and its native query Q^S, any query F satisfying Definition 4.3 is a correct filter. There may be more than one such filters that are not logically equivalent. Thus, we wish to choose the “best” one. At first glance one may think that the broadest filter, i.e., the one that subsumes the others, would be the best. However, since all valid filters will produce exactly the same result set (from the result of the native query), this is not the right metric to focus on. Instead, we would like the filter with the simplest Boolean expression, which will involve the smallest computational effort.

Example 5.2 (Filters): Consider the query $Q = P_1 + P_2\bar{P}_3$. Suppose P_1 and P_2 are supported by the target, and $P_3^{\bar{S}} = False$. Algorithm 1 gives $Q^S = P_1 + P_2$. Given Q and Q^S, the correct filters include $F_1 = P_1 + \bar{P}_2 + \bar{P}_3$, and $F_2 = P_1 + \bar{P}_3$. Both filters are valid since $Q = Q^S F_1$ and $Q = Q^S F_2$. Filter F_1 is broader than F_2, as it subsumes F_2. However, it is clear that F_2 is a better choice because it has a simpler expression which implies less processing cost under any normal cost definition. ❑

Correct filters are not difficult to derive. Intuitively, given a query Q and Q^S, if we can find all the necessary conditions that Q must imply, a filter can be composed as the conjunction of those necessary conditions that are not implied by Q^S. We refer to the not-implied conditions as the *residue* conditions. One way to find the necessary conditions that Q implies is to transform it into Conjunctive Normal Form (CNF). Written in CNF, $Q = \prod_{i = 1..m} D_i$, where $D_i = \sum \tilde{P}_j$, and $\tilde{P}_j \in \{P_j, \bar{P}_j\}$. D_i's are *disjunction terms* of some predicates Pj's defined in Q. Since the D_i's are conjuncted in Q, they are the necessary conditions that Q must satisfy. Any D_i containing unsupported predicates is a necessary condition that cannot be implied by Q^S, and therefore a residue condition. Consequently, a filter can be composed as the conjunction of those residue D_i's.

To illustrate this procedure, consider query Q of Example 5.2. Written in CNF, $Q = (P_1 + P_2)(P_1 + \bar{P}_3)$. Since $P_3^{\bar{}} = False$, the second disjunction term is a residue (the only one), and hence the filter is $F_2 = P_1 + \bar{P}_3$. In this case we obtained the optimal filter, but this is not always the case, as the following example illustrates.

Example 5.3 (Filter Derivation): Consider the query $Q = P_1\bar{P}_2P_3 + P_2\bar{P}_3$, and $Q^S = P_1P_3 + \bar{P}_3$ which is resulted from Algorithm 1 by assuming $P_2^S = True$ and $P_2^{\bar{}} = False$. Writing Q in the (minimal) CNF as $(P_2 + P_3)(\bar{P}_2 + \bar{P}_3)(P_1 + P_2)$, we find that the three disjunction terms all contain the unsupported predicate P_2. Therefore, the filter composed from the residue conditions is $F_1 = (P_2 + P_3)(\bar{P}_2 + \bar{P}_3)(P_1 + P_2)$. However, F_1 is not optimal; the reader may easily verify that $F_2 = (P_2 + P_3)(\bar{P}_2 + \bar{P}_3)$ is also correct, and is simpler than F_1. ❑

By comparing a query Q to its subsuming query Q^S, one can find a unique but incomplete specification for the family of all the correct filters. If further constrained by the cost definition, the optimal filter can be decided uniquely. Due to the space limitations, we are not able to present the algorithm here. The interested readers may refer to [1].

Our final theorem below combines the results we have presented. Theorem 5.3 has shown that a front-end query can be transformed into a minimal, subsuming native query. Furthermore, given a query Q, it is not hard to see that its minimal subsuming query is unique (if Q_1 and Q_2 are both minimal w.r.t. the target system T, then their conjunction, $Q' = Q_1Q_2$, must also be executable by T and is still smaller than both Q_1 and Q_2, a contradiction). For the filters, we presented a simple approach to derive correct filters. As we just mentioned, the optimal filter is also unique under some cost definition.

Theorem 5.4. (Query Capability Mapping): Given a query Q, the target system T, and the cost definition for post-filtering, the minimal subsuming query of Q w.r.t. T and the optimal filter w.r.t. the cost definition can always be found uniquely. ❑

Keep in mind that Theorem 5.4 only addresses the existence of a native query and filter, not its practicality or cost.

As a matter of fact, we will be unable to run some native queries because of “quirks” of real systems. For instance, we mentioned earlier that some systems do not accept the valid query *True*. This means that if our algorithm generates that as the minimal subsuming query for some user query *Q*, then it will be impossible to answer *Q*. Similarly, some systems may not export certain fields in their documents. So if a filter wishes to search locally one of those “hidden” fields, it will fail. In all these cases, as well as those where the native query is too expensive, the user will have to reformulate his query or access the source directly.

VI. Conclusion

This paper gave an overview of the query translation process and focused on the logic mapping algorithms. As pointed out earlier, there are situations in which our approach has a number of drawbacks or even fails. For one, in some cases translation can turn out to be too expensive. For example, the approach may create too much network traffic, or they may create queries that contain too many terms. The latter can happen, for instance, when truncation is approximated by enumerating terms over a source’s vocabulary. Another failure mode can be that a source simply does not provide the information the algorithms need. For example, it may not be possible to obtain a source’s vocabulary in order to provide approximations to truncation. Similarly, a source with a large corpus is generally not able to return all of its contents which the algorithms can in some cases call for when query translation produces *True* as the native query. In this case, the query cannot be executed.

In general, our algorithms for rewriting predicates, as briefly discussed in Section IV, require the following from the underlying search engines to perform a complete translation (this is the contents of the “Target Capability & Schema Definition” box in Fig. 1.):

1. the schema definition: the set of searchable fields, and the indexing scheme of each field.
2. the supported operators in addition to Boolean operators.
3. the stopword list.
4. the vocabulary: the word vocabulary, and the phrase vocabularies for phrase-indexed fields.
5. the details of expansion features, e.g., for stemming: the algorithm used; for truncation, the supported truncation patterns.

Among these, items 1, 2, 3, and the truncation patterns are usually documented for the end users to search the sources. The others are presently harder to obtain.

There are several factors which mitigate the drawbacks we have discussed. To reduce the cost for retrieving and post-filtering the entire preliminary result set, we may instead process the result *incrementally*. That is, if the user wishes to see, say a screenfull of documents, only some of the source’s documents must be retrieved and filtered. As the user requests more documents, more are processed.

We found that in many practically interesting cases a translation is possible. For example, our study shows that predicate rewrites to *Tru*e are actually unlikely in practice. Moreover, notice that trivial rewriting of a predicate does not necessarily imply that the entire native query becomes *True*. For example, when a query contains conjunctions of which one translates to *True*, the remaining terms are used to maintain reasonable selectivity. The native query degenerates to *True* only when there are no other components left. To be precise, the query translator generates *True* as the native query for a user query if and only if all predicates forming an *implicant* (a conjunction of predicates that implies the query) become *True*. For example, query $Q=P_1P_2+P_2P_3$ translates to *True* only when both P_1 and P_2, or P_2 and P_3 are rewritten to *True*.

For cases where translations produce excessively large queries, approximations can be applied. For the truncation emulation example, an implementation could decide not to expand to all possible terms. This would not yield a complete result. But depending on the user's task, a partial solution may be acceptable if it still produces "enough" documents. If some required metadata is not available, we believe that approximations can also help out. For example, a common vocabulary, such as the set of words from a dictionary can be used to approximate the emulation of truncation. Although this does not guarantee a precise translation, it might not be a fatal drawback given the inherent uncertainty in information retrieval. The approximation of using a common dictionary also greatly helps our approach to scale.

When a translation is truly impossible, our approach still provides a benefit. As shown in Fig. 1, we provide a feedback loop to the user. When a translation failure is detected, the user can be informed about precisely which part of the query is problematic. The user can then reformulate just that part.

As discussed in Section II, query unification has been attempted in various forms over the years. We believe that the increased power of search engines and machines available locally to users, combined with increased network bandwidth and changing information access economics, call for a re-examination of this area. In many cases it is now feasible to compensate for lacking retrieval features by extracting more information, and filtering it locally. This paper sketched our efforts in beginning such a re-examination.

Our initial prototype implementations are very encouraging. We have transformed the kinds of queries shown in this paper and have successfully executed them on very different search engines. Future work will involve schema unification (which we did not discuss here) and extensions for non-Boolean queries.

ACKNOWLEDGMENTS

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.

References

[1] K.C.-C. Chang, H. Garcia-Molina, and A. Paepcke, *Boolean Query Mapping Across Heterogeneous Information Sources*, Technical Report, in preparation. Dept. of Computer Science, Stanford Univ., 1996.

[2] K.C.-C. Chang, H. Garcia-Molina, and A. Paepcke, ”Predicate Rewriting for Translating Boolean Queries in a Heterogeneous Information System,” submitted for publication. URL: http://www-db.stanford.edu/pub/chang/1996/pred_rewriting.ps.

[3] W. Cross, *All-In-One Search Page*, URL: http://www.albany.net/allinone/.

[4] M.I. Crystal and G.E. Jakobson, “FRED, A Front End for Databases,” *Online*, vol. 6, no. 5, pp. 27-30, Sep. 1982.

[5] D. Dreilinger, *SavvySearch Home Page*, URL: http://www.cs.colostate.edu/~dreiling/smartform.html.

[6] W.B. Frakes and R. Baeza-Yates, *Information Retrieval Data Structures & Algorithm*. Englewood Cliffs, N.J.: Prentice Hall, 1992.

[7] D.T. Hawkins and L.R. Levy, “Front End Software for Online Database Searching Part1: Definitions, System Features, and Evaluation,” *Online* 9(6):30-37, Nov. 1985.

[8] ISO, *ISO 8777:1993 Information and Documentation -- Commands for Interactive Text Searching*. Geneva: Int’l Organization for Standardization.

[9] J.B. Lovins, “Development of a Stemming Algorithm”, *Mechanical Translation and Computational Linguistics*, vol. 11, no. 1-2, pp. 22-31, 1968.

[10] R.S. Marcus, “User Assistance in Bibliographic Retrieval Networks Through a Computer Intermediary,” *IEEE Trans. on Systems, Man, and Cybernetics*, vol. smc-12, no. 2, pp. 116-133, 1982.

[11] T.H. Martin, *A Feature Analysis of Interactive Retrieval Systems*, Report SU-COMM-ICR-74-1. Stanford, C.A.: Institute of Communication Research, Stanford University, Sep. 1974.

[12] E.J. McCluskey, *Logic Design Principles*. Englewood Cliffs, N.J.: Prentice Hall, 1986.

[13] P.C. Mitchell, “A Note about the Proximity Operators in Information Retrieval,” *Proc. of ACM SIGPLAN-SIGIR Interface Mtg.*, pp. 177-180, Nov. 1973.

[14] National Information Standards Organization, *Z39.58-1992 Common Command Language for Online Interactive Information Retrieval*. Bethesda, M.D.: NISO Press.

[15] A.E. Negus, “Development of the Euronet-Diane Common Command Language,” *Proc. 3rd Int’l Online Information Mtg.*, pp. 95-98, 1979.

[16] M.F. Porter, “An Algorithm for Suffix Stripping”, *Program*, vol.14, no. 3, 130-137, 1980.

[17] S.E. Preece and M.E. Williams, “Software for the Searcher’s Workbench,” *Proc. of the 43rd American Society for Information Science Annual Mtg.*, vol. 17, pp. 403-405, 1980.

[18] G. Salton, *Automatic Text Processing*. Reading, Mass.: Addison-Wesley, 1989.

[19] E. Selberg and O. Etzioni, “Multi-Service Search and Comparison using the MetaCrawler”, *Proc. of the 4th Int’l WWW Conf.*, URL:http://metacrawler.cs.washington.edu:8080/papers/www4/html/Overview.html.

[20] D.E. Toliver, “OL’SAM: An Intelligent Front-End for Bibliographic Information Retrieval,” *Information, Technology and Libraries*, vol. 1, no. 4, pp. 317-326, 1982.

[21] J.D. Ullman, *Database and Knowledge-Base Systems*, vol. 1. Rockville, M.D.: Computer Science Press, 1988, ch. 3, pp. 121-122.

[22] M.E. Williams, “Transparent Information Systems Through Gateways, Front Ends, Intermediaries, and Interfaces,” *Journal of the American Society for Information Science*, vol. 37, no. 4, pp. 204-214, Jul. 1986.

[23] S. Zinn, M. Sellers, and D. Bohli, “OCLC’s Intelligent Gateway Service: Online Information Access for Libraries”, *Library Hi Tech*, vol. 4, no. 3, pp. 25-29, 1986.