Evaluating the Cost of Boolean Query Mapping *

Chen-Chuan K. Chang   Héctor García-Molina

Electrical Engineering Department   Computer Science Department

Stanford University
Stanford, CA 94305-9040, USA
{kevin,hector}@db.stanford.edu

Abstract

Non-uniform query languages make searching over heterogeneous information sources difficult. 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 results. This post-filtering approach may involve significant cost because the documents that the users will not see may have to be retrieved and filtered. There are generally two ways to implement post-filtering: batch post-filtering and incremental post-filtering. In this paper we evaluate the costs of both methods for different search features such as proximity operators. The experimental results show that in many cases incremental post-filtering cost may be acceptable, while the batch post-filtering cost may sometimes be extremely large.

1 Introduction

Emerging Digital Libraries can provide a wealth of information. However, there are also a wealth of search engines behind these libraries, each with a different document model and query language. Our goal is to provide a front-end to a collection of Digital Libraries that hides, as much as possible, this heterogeneity. As a first step, we have focused on supporting Boolean queries [1, 2] because they are used by most current commercial systems. We are also planning to extend our work to the vector space model – see Section 6.

Our approach supports integrated access to heterogeneous systems through an intelligent front-end system responsible for query mapping and post-filtering. The front-end provides a powerful query language that may not be fully supported by the underlying systems. Users do not access the underlying services directly; instead they submit queries to the front-end. The front-end translates the user queries into *subsuming queries* that are supported by the target systems but that may return extra documents. This translation allows the queries to be evaluated by multiple services in parallel. Because the preliminary results may contain extra documents that the users did not ask for, the front-end also generates *filter queries* and processes the preliminary results locally against these filter queries to produce the final answers.

Our approach requires post-filtering to compensate for the functionality differences between the front-end and the underlying services. Because the post-filtering cost is of major concern in our approach, we performed experiments to quantify the overhead. This paper presents some of the experimental results. Due to space limitation, this paper does not provide the details of query translation; readers can refer to [3, 4, 5] for them. 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*

User Query: $Q =$ Title contains multiprocessor AND distributed (W) system.

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

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

Subsuming Query: $Q^S =$ Find Title multiprocessor AND distributed AND system.

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

Filter Query: $F =$ Title contains distributed (W) system.

The post-filtering step does produce the matching documents for Q because $Q \equiv Q^S \wedge F$. However, this post-filtering requires more work at the front-end. This paper evaluates the post-filtering overhead of query translation. Specifically, we quantify the total cost, which is proportional to the result size of Q^S. In addition, we also evaluate how this number compares to the result size of Q, which indicates the per-document cost.

Figure 1 shows the main components of the proposed front-end system. The user submits (lower left) a query in a powerful lan-

*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. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the other sponsors.

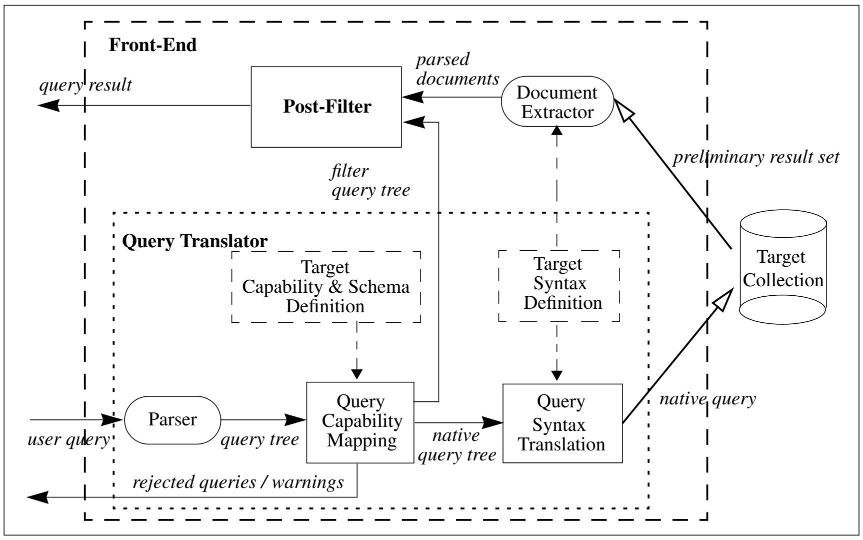


Figure 1: The architecture of the front-end system illustrating query translation and post-filtering. The dashed boxes are target-specific metadata defining the target's syntax and capabilities.

guage 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 into the native query tree, which subsumes the user query, and the filter query tree. Using the syntax of the target, the native query tree is translated into a native query and sent to the source. After the documents are received and parsed according to the syntax for source documents, they are processed against the filter query tree, yielding the final answer. Notice that currently we do not consider semantic mapping issues for the query terms (*e.g.*, mapping "fault tolerant sysetms" to "reliable systems").

Even though heterogeneous search engines have existed for over 20 years, the approach we advocate here, full search power at the front-end with appropriate query transformations, has not been studied in detail. The main reason is that our approach has a significant cost, *i.e.*, documents that the end users will not see have 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. Reference [3] provides more details of the related work. In summary, they generally fall into four categories:

1. Eliminate the heterogeneity by standardization of the query languages [6, 7, 8].
2. Present inconsistent query capabilities specific to the target systems with no intention to hide the heterogeneity and have the end users write queries specifically for each [9, 10]. The service *All-In-One-Search* (accessible at `http://www.albany.net/allinone/`) uses this approach, for example.
3. Provide a "least common denominator" front-end query language that can be supported by all sources [11, 12, 13].
4. Copy all the document collections that users may be interested to a single system that uses one search engine and one language. For example, Knight-Ridder's *Dialog* (accessible at `http://www.dialog.com/`) uses this approach.

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. Compared to the first solution, our approach does not enforce any standard that requires the corporations of the underlying services and thus maintain their autonomy. Compared to the second solution, our approach of supporting a uniform query language provides transparent accesses to multiple services. As for the third solution, because our front-end language is powerful, the unification of query languages does not mask the rich capability that a service may provide. Finally, our approach does not require the front-end to actually manage data, which certainly helps the front-end to scale, in contrast to the fourth solution.

We believe our approach is a fundamental solution to address the heterogeneity. End users really require powerful query languages to describe their information needs, and they do require access to information that is stored in different systems. At the same time, increasing computer power and network bandwidths are making the full front-end query power approach more acceptable. Furthermore, many commercial sources are opting for easy-to-manage broad agreements with customers that provide unlimited access. Thus, in many cases it may not be that expensive to retrieve additional documents for front-end post filtering. Even if there is a higher cost, it may be worth paying the cost to get the users the required documents with less effort on their part. This paper reports on these extra costs, and as we will see, in many cases the overhead is reasonable. Due to space limitation, we do not report on all the experiments we have performed; instead we describe three representative experiments and briefly sample some of the results we have obtained.

We start by introducing the experimental framework and formalizing our evaluation criteria. Then each of Section 3, 4, and 5 presents the experiments designed specifically for a specific query feature that may not be uniformly supported, namely the proximity search, the stopwords, and the equality operator.

2 Experimental Criteria and Methodology

There are generally two ways to implement post-filtering: either in a *batch* or *incrementally*. With batch processing, the front-end retrieves all the documents in the preliminary result set, evaluates each with the filter query, and produces the final answers. With the second approach, the front-end processes the documents incrementally when a matching document is requested. In other words, if a user wishes to see, say a screenful of documents, only some of the source's documents must be retrieved and filtered. As the user requests more, more are processed.

Our experiments evaluated the costs for both batch and incremental processing. The factors contributing to post-filtering cost include the access to documents in the target service, their transportation across the network, and finally their evaluation by the post-filtering engine at the front-end. In batch processing, the cost is proportional to the size of the preliminary result set, because the entire set must be processed. Therefore, we use

$$\mathsf{Size}(\langle Q^S \rangle)$$

as the metric for batch processing, where Q^S represents the subsuming query generated by the query translator for a user query Q, and $\langle Q^S \rangle$ represents the result set of Q^S. With incremental processing documents are retrieved on demand, and the interesting cost metric is the per-document cost, *i.e.*, the batch cost amortized by the number of documents matching Q. We call this metric the *selectivity ratio*, denoted by $\mathsf{SR}(Q^S, Q)$, because it indicates the reduction of selectivity from Q to Q^S; *i.e.*,

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

Both batch and incremental processing are adequate in certain cases. The advantage of batch processing is that the front-end can present users the list of matching documents and report the number of hits at once. However the cost can be significant, because the preliminary result set can be extremely large. Furthermore, because users may not want to see all the matching documents, some processing time and cost may be simply unnecessary. In comparison, incremental processing only performs the necessary effort on demand. However, with incremental processing the exact number of matching documents is not known initially. If the user requires this number, it must be estimated, as discussed in Section 6.

It is misleading to interpret either $\mathsf{SR}(Q^S, Q)$ or $\mathsf{Size}(\langle Q^S \rangle)$ as the "quality" or "effectiveness" of query translation. In principle, the query translator always produces the best subsuming queries according to the targets' capability constraints. Therefore, the reduced query selectivity results directly from the inherent limitations of the underlying services. The metrics do evaluate the cost of supporting heterogeneous search through query translation.

To evaluate the cost metrics, the experiments measured the result sizes for a set of sample query Q's and their subsuming query Q^S's. First, each sample query Q is converted to its subsuming query Q^S with respect to some source capability. Both Q and Q^S are then evaluated over the same document collection (possibly by different search engines) to obtain $\mathsf{Size}(\langle Q \rangle)$ and $\mathsf{Size}(\langle Q^S \rangle)$. Finally, $\mathsf{SR}(Q^S, Q)$ is computed.

There are two ways to drive the experiments: from query traces or through query generation. We used the latter approach, *i.e.*, we use feature-specific synthesized queries automatically generated for the query features of interest. For example, to explore the proximity feature, our experiments generate queries that use proximity operators to search for phrases, which is a typical usage of the feature. Specifically, first, we need to select a publicly available phrase vocabulary compiled for the subject domains of the target search service. Then, the experiments randomly pick phrases from the vocabulary and generate sample queries accordingly. For instance, if the selected phrase is "digital libraries", we can generate a sample proximity query as "Title digital (W) libraries".

We did not use query traces because of the difficulty in obtaining relevant traces. For instance, we do have access to the query traces from Stanford's *Folio*; however, *Folio* does not support interesting features such as proximity. At the same time, commercial search services that support powerful features are reluctant to provide their query traces. It is important to notice that while traces may be more "realistic" than synthetic queries, they may not be as good at identifying the effects of specific features. The synthetic approach allows us to design sample queries specific to a particular feature, and the results can give us a better understanding of the overhead of translating that particular feature.

3 Proximity Experiments

Proximity operators are used to specify the positional adjacency or logical grouping constraints of search words. Among some possible variants, the most commonly supported are the word proximity operators, with which users can specify the adjacency constraints in the unit of words. The operator nW, where n is a positive integer, can be used to specify the maximum distance (number of words) within which the search terms must occur in the specified order, *e.g.*, "integrated (2W) systems." The immediate adjacency 0W can be abbreviated as W. If the order does not matter, nN and N can be used instead.

Not all systems support the word proximity operators. They are standard features supported by common command languages such as ISO-8777 [8] and Z39.58 [7], and some commercial systems such as *Dialog*. Other systems either do not support the proximity operators (*e.g.*, Stanford's *Folio*), or support them only partially, *e.g.*, the distance and/or order parameters may not be specified freely. For instance, *WebCrawler* (accessible at `http://www.webcrawler.com/`) does not support nW for arbitrary n, although it does support 0W (written as ADJ) and the unordered proximity operator nN (written as NEAR/n). As another example, *AltaVista* (accessible `http://altavista.digital.com/`) has only the operator NEAR, which means 10N.

Example 3.1 (Translation for Proximity Queries) *If a query Q is* "Title contains digital (W) libraries"*, and the operator* W *is not available, we can replace it with a semantically weaker operator to form a subsuming query. For example, we can use the* AND *operator and generate Q^S as* "Title contains digital AND libraries."

However, a system may support better substitutes than AND*. For instance, the* NEAR *operator (meaning* 10N*) is a better choice in AltaVista. In this case, the subsuming query is Q^S =* "Title contains digital (10N) libraries." *Readers can refer to [3, 4, 5] for the details of the general translation algorithms.*

For this experiment, we focused on one common usage of proximity operators that searches for a phrase like "digital libraries", as illustrated in Example 3.1. Typically, a service builds indices for particular fields either with the individual words or the complete values. Consequently, to search for a (partial) phrase contained in some field (instead of its complete value), users must express the phrase as a proximity expression using the operator W. Of course, this is not the only way proximity operators are used, *i.e.*, there may be no phrase behind a proximity query. (For instance, the query "Title contains car (10N) lawsuit" simply looks for the documents in which the words "car" and "lawsuit" appear nearby.) Nevertheless, for our experiment we do focus on the scenario where there is a phrase behind the query.

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

Figure 2: Configurations of the proximity experiments.

The experiments automatically generated sample queries of the parametric form

$$Q_W = \textsf{F contains } w_1 \textsf{ (W) } w_2 \textsf{ (W) } \cdots \textsf{ (W) } w_n,$$

where F is a field designation like Title, and w_i's are words in a phrase. For the field parameter, we selected three common fields representing different typical lengths: Title, Abstract, and Text (the body of text of documents). These three fields roughly represent lengths on the order of 10, 100, and 1000 words, respectively. We expect that the typical lengths of the fields may impact the selectivity of proximity queries. For instance, for a typically short field such as Title, the operator AND is likely to approximate 10W, while this approximation may not hold for a longer field such as Text.

To generate the proximity expression "w_1 (W) w_2 (W) $\cdots$ (W) w_n", we first selected some phrase vocabularies, then randomly picked a phrase from the vocabularies, and finally extracted words $w_1, w_2, \cdots, w_n$ from the phrase. In particular, we chose as vocabularies the Free On-line Dictionary of Computing (VOC_{Foldoc}) [14] and the *Inspec* Thesaurus (VOC_{Inspec}) [15]. VOC_{Foldoc} is an evolving dictionary of computing-related terms, and VOC_{Inspec} is a set of controlled subject terms that IEE compiles to categorize the documents in the *Inspec* collection. The experiments used only phrases of length 2,3, and 4, because longer phrases are extremely infrequent (less than 2% in both vocabularies). Furthermore, we also removed entries containing stopwords (*e.g.*, "of", "the", *etc.*) because we investigate queries with stopwords separately in Section 4.

Among the search services we have access to, we chose to run the queries on *Dialog* because its search engine can handle the proximity queries and thus saves the effort of post-filtering. Furthermore, among the many collections *Dialog* provides, we chose to use *Computer Database* (file 275, designated *Dialog-275*) and *Inspec* (file 2, designated *Dialog-2*) based on the fields we wanted to search and the vocabularies we have. First, for the search fields, *Dialog-2* supports Title and Abstract, while *Dialog-275* supports Title and Text. Second, VOC_{Inspec} was designed specifically for *Dialog-2*, and VOC_{Foldoc} is appropriate for *Dialog-275*, because both are related to computing.

Figure 2 shows the four different configurations used for our experiment, where each configuration defines a search field, a phrase vocabulary, and a source collection. We performed two sets of experiments, each sharing these configurations, but with different subsuming queries. The first set of experiments evaluated the costs when the W operator is replaced by AND, which represents the worst-case substitution. In the second set of experiments, we investigated how progressively weaker operators impact query selectivities.

3.1 AND-queries versus W-queries

Figure 3 sketches the results of the first set of experiments, where a user query Q_{W} with the W operator is replaced with a native query Q_{AND} that uses the AND operator instead, for our four configurations (Figure 2). Specifically, the figures plot the pairs

$$[\ \mathsf{Size}(\langle Q_{\mathsf{W}} \rangle),\ \mathsf{Size}(\langle Q_{\mathsf{AND}} \rangle)\ ]$$

for the queries of the corresponding configurations. That is, they illustrate the distribution of $\mathsf{Size}(\langle Q_{\mathsf{AND}} \rangle)$ with respect to $\mathsf{Size}(\langle Q_{\mathsf{W}} \rangle)$.

To illustrate the ranges of $\mathsf{SR}(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ values in the figures, the diagonal dotted lines (representing the $y/x = m$ axes, where m is labeled on the $y-$axis) partition the space into different bands, each representing a range of $\mathsf{SR}(Q_{AND}, Q_W)$ (*i.e.*, y/x). For instance, for all the data points falling in the lowest band, (y/x between 1 and 10), the subsuming query Q_{AND} fetches between 1 and 10 times as many documents as the original query Q_{W} would have fetched, were it supported by the source.

Note that all the points fall on the upper-left side of the $y/x = 1$ axis because Q_{AND}'s subsume Q_{W}'s. In other words, the closer the points accumulate to the $y/x = 1$ axis, the better the Q_{AND}'s approximate the Q_{W}'s.

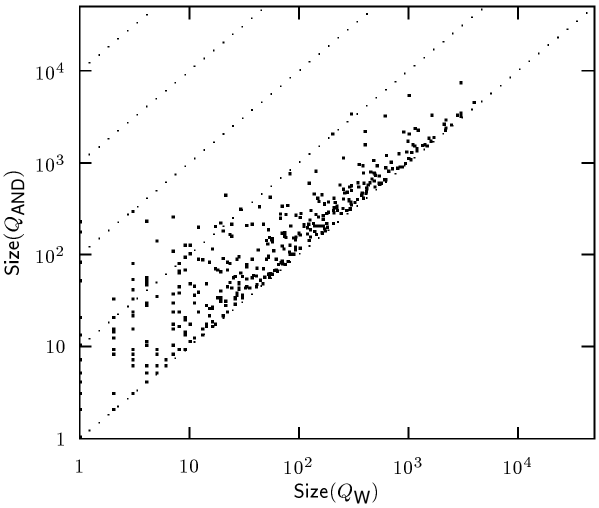
The cost of batch post-filtering depends on $\mathsf{Size}(\langle Q_{\mathsf{AND}} \rangle)$, which varies greatly, from as little as 1 to on the order of 10^3. In principle, the only upper bound is the size of the queried collection. Given this significant variation, batch post-filtering may not always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

In contrast, the results in terms of selectivity ratios indicate that incremental post-filtering is feasible in most cases. The middle column of Figure 4 summarizes the average results. (The right column will be explained later.) As expected, for shorter fields, Q_{AND}'s tend to approximate Q_{W}'s better. In summary, Figure 4 shows that the average $\mathsf{SR}(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ values for the configurations Inspec-Ti and Comput-Ti are less than 10, while those of longer fields (the configurations Inspec-Ab and Comput-Tx) are about an order of magnitude larger.

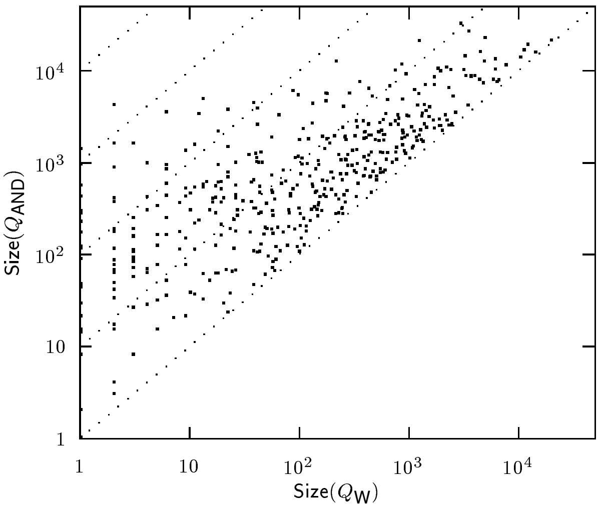
Interestingly, in all the four configurations, $\mathsf{SR}(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ values tend to decrease as $\mathsf{Size}(\langle Q_{\mathsf{W}} \rangle)$ increases. This means that incremental processing complements batch processing well. That is, if batch processing does not work well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

In addition, notice that some points with very small $\mathsf{Size}(\langle Q_{\mathsf{W}} \rangle)$ (say, smaller than 5) have surprisingly large $\mathsf{SR}(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$, as illustrated by the points near the y-axis. Incremental post-filtering is expensive for this kind of query although it is relatively infrequent. However, notice that these queries represent searches on phrases that *rarely* appear in the queried collections. In other words, those phrases may not be considered appropriate for the subject domains of the collections, or vice versa. If users rely on good source selection tools such as GlOSS [16], then we would expect such queries to be rare. Furthermore, if users do submit these expensive queries, it may be possible for the front-end to dynamically estimate the post-filtering cost and advise the user that he should either reformulate the query of pay a high price. We discuss these issues further in Section 6.

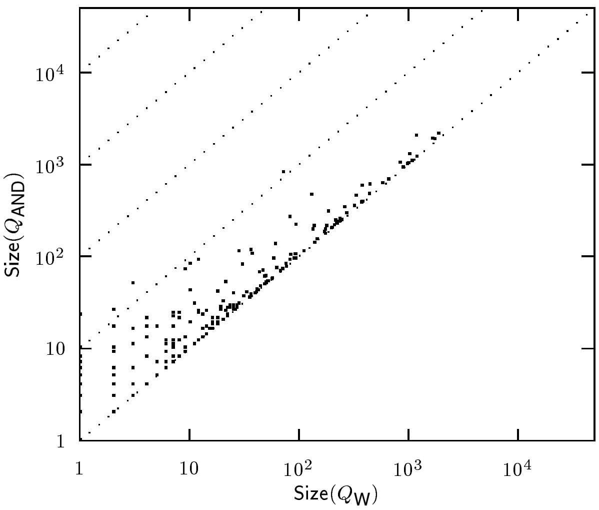
To show the overhead of translation if these "odd" queries are avoided, the right column of Figure 4 provides the averages for those samples with $\mathsf{Size}(\langle Q_{\mathsf{W}} \rangle)$ greater than 5. Note that the average ratios decrease dramatically, especially for the configurations with long fields, such as Comput-Tx.



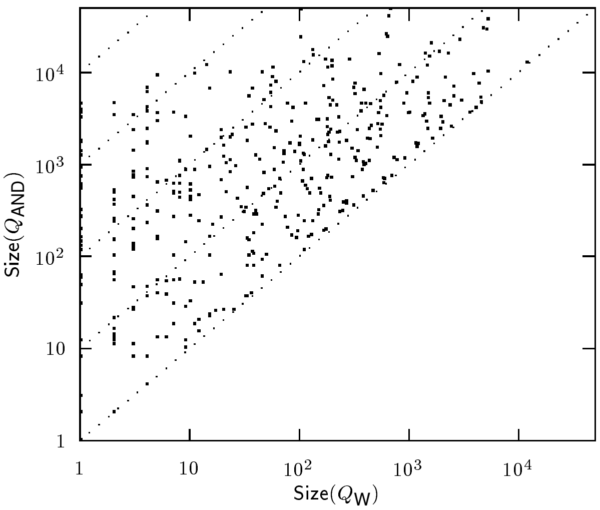
(a) Configuration Inspec-Ti



(b) Configuration Inspec-Ab



(c) Configuration Comput-Ti



(d) Configuration Comput-Tx

Figure 3: Results of AND-queries versus W-queries.

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

Figure 4: Average selectivity ratios of AND-queries versus W-queries.

3.2 Other Operators

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

For each configuration, Figure 5 gives the average $\mathsf{SR}(Q_{\mathsf{op}}, Q_{\mathsf{W}})$ values, when a query Q_{op}, with operator op, is compared to a query Q_{W} with the most selective operator W. For each configuration we plot two curves: the first curve consists of a series of pairs

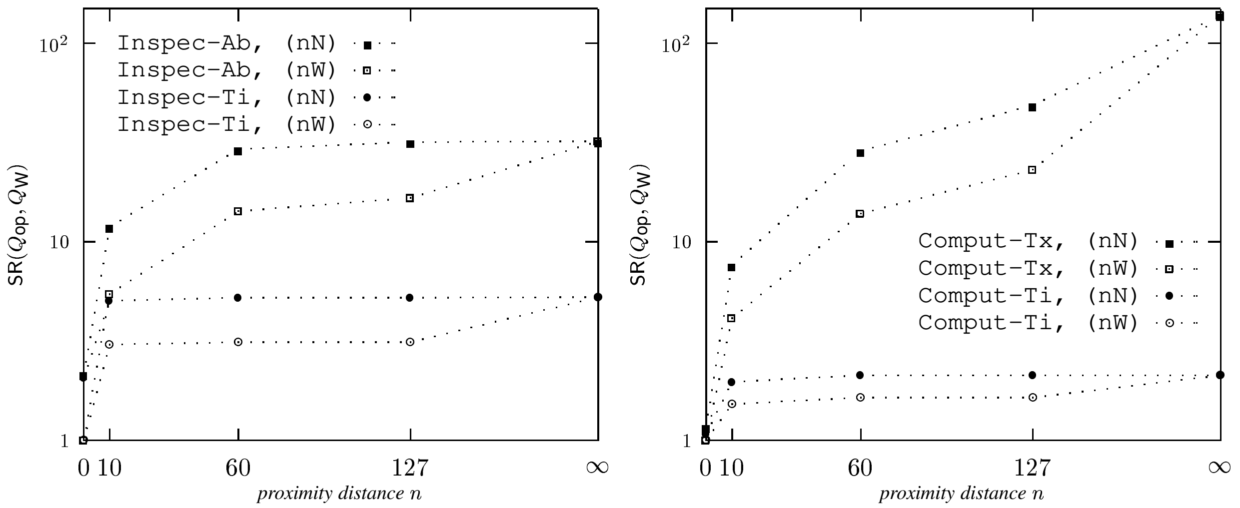
$$[\ n,\ \mathsf{SR}(Q_{\mathsf{nW}}, Q_{\mathsf{W}})\ ],$$

where $n \in \{0, 10, 60, 127\}$. That is, this curve represents the selectivity of the *ordered* proximity operators as the distance constraints are progressively relaxed. Similarly, the second curve is for the *unordered* proximity operators, *i.e.*, it plots the pairs

$$[\ n,\ \mathsf{SR}(Q_{\mathsf{nN}}, Q_{\mathsf{W}})\ ],$$

where $n \in \{0, 10, 60, 127, \infty\}$. Note that ∞N represents the AND operator. For example, looking at the configuration Inspec-Ti, we see in Figure 5(a) that the 60W operator has an overhead of $\mathsf{SR}(Q_{\mathsf{60W}}, Q_{\mathsf{W}}) = 3.1$, while the 60N operator has an overhead of $\mathsf{SR}(Q_{\mathsf{60N}}, Q_{\mathsf{W}}) = 5.2$.

Observe that the results for the nN-queries are relatively close to those of the nW-queries in all the configurations, with the former



(a) Configurations Inspec-Ti and Inspec-Ab

(b) Configurations Comput-Ti and Comput-Tx

Figure 5: Average $\mathsf{SR}(Q_{\mathsf{op}}, Q_{\mathsf{W}})$'s for different operator op's.

being no more than two times greater than the latter. The implication is that, if a system only supports unordered proximity (*e.g.*, *WebCrawler*), replacement of ordered operators requires incremental post-filtering cost of no more than 2.

For systems with even only partial support of the proximity operators, the incremental post-filtering cost decreases significantly compared to those with no support. In other words, in a system that supports some operators stronger than AND, the incremental cost may be greatly reduced with these operators. For example, although *AltaVista* supports only the NEAR operator, which means 10N, it is evident that $\mathsf{SR}(Q_{\mathsf{10N}}, Q_{\mathsf{W}})$ is much smaller than $\mathsf{SR}(Q_{\mathsf{AND}}, Q_{\mathsf{W}})$ for long fields as the configurations Inspec-Ab and Comput-Tx show. Thus the NEAR operator can be very useful.

It is interesting to note that the (unordered) proximity operators nN approximate the AND operator for n greater than some threshold value. The reason is that, when the specified distance is greater than the typical length of a field, the occurrences of words imply their proximity. For instance, for short fields as the configurations Inspec-Ti and Comput-Ti show, the nN operators start to approximate AND for $\mathsf{n} \geq 10$. For longer fields such as Abstract, this threshold is about 60. The implication is that, for queries on short fields (*e.g.*, bibliographic fields), lack of full support of the proximity operators may not be a crucial restriction, because the AND operator can approximate them well.

4 Stopword Experiments

Most systems specify a set of stopwords which are not searchable. Stopwords are common words, such as "the" and "on", occurring so frequently that they supposedly do not distinguish one document from another. If a query contains stopwords, a system may reject the query (*e.g.*, U.C. Berkeley's *Melvyl*), ignore the stopwords (*e.g.*, Stanford's *Folio*), or simply return no matches (*e.g.*, *Dialog*). There are also languages (*e.g.*, Z39.58 CCL [7]) that provide some way to override stopwords and make them searchable.

In a front-end where a user can search multiple underlying services, the user may not be aware of the stopwords specified by each service. To prevent a query containing stopwords from being rejected or returning no matches at a service, it is desirable that the front-end check the query against the stopword definition of the target service, and rewrite the query to remove the stopwords. The following example illustrates this idea.

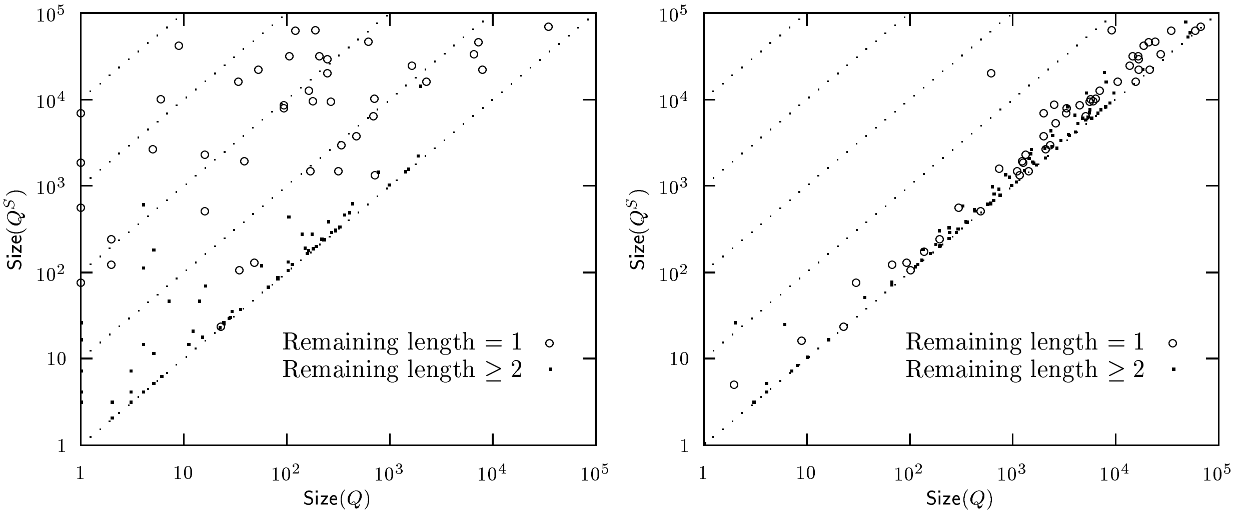
configuration parameter	Config-Prox	Config-Conj
Operator	W	AND
Search field	Text	
Phrase vocabulary	VOC_{Foldoc}	
Source collection	*Dialog-275*	
Source of stopwords	*Britannica Online*	

Figure 6: Configurations of the stopword experiments.

Example 4.1 (Translation for Queries with Stopwords) *Suppose a query Q is* "Title contains gone AND with AND the AND wind." *If Q is for* Dialog, *the result is nothing because the words* "with" *and* "the" *are not indexed, i.e., they are stopwords. To avoid this situation, the query translator can remove the two words from the query, which generates Q^S as* "Title contains gone AND wind." *Notice that Q^S subsumes Q because it requires only two of the four words to appear. Consequently the post-filtering must check if the returned documents actually contain the removed words, i.e., the filter query is* "Title contains with AND the."

As another example, suppose the query Q is "Title contains gone (W) with (W) the (W) wind", *i.e., it uses the proximity operator* W *instead of* AND. *In this case, the query translator needs to modify the proximity expression to reflect the occurrence of the removed words. The translated Q^S is* "Title contains gone (2W) wind", *and the filter is* "Title contains gone (W) with (W) the (W) wind." *For more details of the general translation algorithms, readers can refer to [3, 4, 5].*

Our experiments compare the result size of a sample query Q containing stopwords to that of its subsuming query Q^S, with stopwords removed, as Example 4.1 illustrates. The experimental procedure is similar to that of the proximity experiments except that different sample and subsuming queries are used. We performed two sets of experiments; Figure 6 shows the corresponding configurations.



(a) Configuration Config-Prox

(b) Configuration Config-Conj

Figure 7: Results of the stopword experiments.

For the experiments, we considered the sample queries of the parametric form

$$Q = \textsf{F contains } w_1 \textsf{ op } w_2 \textsf{ op} \cdots \textsf{op } w_n,$$

where F is a search field, w_i's are words in a phrase (*e.g.*, "video on demand") containing one or more stopwords (*e.g.*, the word "on"), and op is an operator connecting the words (*e.g.*, AND or W).

The two configurations differ only in the operators op used in the sample queries. To see how the operator connecting the words may affect the results, the configuration Config-Prox uses W as the operator, while Config-Conj uses AND. The other parameters are shared by the two configurations. In particular, we used the set of stopwords (over 900 words) specified by the information service *Britannica Online* (`http://www.eb.com/`) because of its large size. We selected from the vocabulary VOC_{Foldoc} all the phrases that contain at least one stopword defined by *Britannica Online*. We used Text as the search field. (We believe that the results should be similar for other fields.) Finally, both the sample queries and the subsuming queries are evaluated in the source collection *Dialog-275*.

Figure 7 plots the pairs [Size($\langle Q \rangle$), Size($\langle Q^S \rangle$)], and Figure 8 shows the average SR(Q^S, Q) values. We found that the results depend not only on the connecting operators (*i.e.*, AND or W), but also on the *remaining lengths* of the subsuming queries, which refers to the numbers of search words remaining in the subsuming queries after the removal of stopwords. We therefore categorize the results of each configuration by the remaining lengths.

In all cases except when the remaining length is 1 in the configuration Config-Prox, the incremental costs are small, because the subsuming queries closely approximate the sample queries. First, for the configuration Config-Conj, Figure 7(b) shows that most of the pairs accumulate around the $y/x = 1$ axis. Because the AND operator tests only the occurrence of search words, which is almost guaranteed for stopwords, removing them from conjunctive expressions does not make much difference. Second, for the configuration Config-Prox, as long as there is more than one non-stop word, the Q^S's still approximate Q's.

In summary, unless a query contains mostly stopwords, a subsuming query with stopwords removed closely approximates the original query. Also notice that stopword removal from conjunctive expressions does not reduce selectivities as much as was the case with proximity expressions.

Remaining Length	Average SR(Q^S, Q)	
	Config-Prox	Config-Conj
1	442.48	2.36
2	8.42	1.61
3	1.15	1.09
≥ 4	1.13	1.06

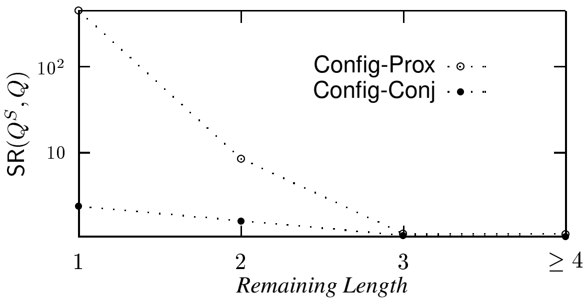


Figure 8: Average selectivity ratios of the stopword experiments, categorized by the remaining lengths.

5 Equality Experiments

Many search services, especially library systems, support the comparison operator equals for testing the complete value of typically short fields such as Title and Author. This feature is useful when users have particular bibliographic citations in mind. However, for the same fields, other systems may support only the contains comparison operator to test the fields' partial values based on some search words because they have not built indexes of complete values for these fields.

configuration parameter	Config-Full	Config-Trun
Truncation	no (full phrase)	yes (the first 5 words)
Phrase vocabulary	*Inspec* Title Phrase	
Source collection	*Inspec* (a subset since 1988)	
Search service for equals-queries	*Folio*	
Search service for contains-queries	*Dialog*	
Search field	Title	

Figure 9: Configurations of the equality experiments.

Example 5.1 (Translation for Queries with equals Operator) *Suppose a user is interested in a citation with title* "Boolean Query Mapping Across Heterogeneous Information Sources." *In this case, the user may simply issue a query* "Title equals 'Boolean Query Mapping Across Heterogeneous Information Sources'."

If the equals *operator is not supported at a service, the query translator needs to transform the query to use the* contains *operator instead. Therefore, the subsuming query is* "Title contains Boolean (W) Query (W) Mapping (W) Across (W) Heterogeneous (W) Information (W) Sources."

Notice that, following this first step, other translation steps may be necessary. For example, the query translator may identify that the proximity operator is not supported or the word "across" *is a stopword, and rewrite the query as discussed in the preceding sections.*

In practice, users usually relies on truncation instead of giving the entire phrase. For example, the query truncated to the first five words is "Title equals 'Boolean Query Mapping Across Heterogeneous #' ", *where "#" is a truncation symbol. In this case, the query translator generates the subsuming query as above, but only with the provided words.*

To study the effects of truncation, we configured the two sets of experiments shown in Figure 9. The two configurations differ only in the use of truncation. The configuration Config-Full uses un-truncated sample queries of the form

$$Q = \mathsf{F}\ \mathsf{equals}\ \text{“}w_1\ w_2\ \cdots\ w_n\text{”},$$

where F is a search field, and w_i's are words in a phrase representing a complete value of the field. In contrast, the configuration Config-Trun truncates the search phrase to the first five words; the sample queries are of the form

$$Q = \mathsf{F}\ \mathsf{equals}\ \text{“}w_1\ w_2\ \cdots\ w_5\ \#\text{”}.$$

Of course, if a phrase is shorter than 5 words, it is used as is without truncation.

To generate the sample queries, we need a set of "phrases" representing complete values of a search field. Because we used *Inspec* as the source collection, we obtained the phrase vocabulary for the Title field of *Inspec* provided by the Stanford library.

For each sample query, the experiments compared it with two subsuming queries using the contains comparison operator of the form:

$$Q_{\mathsf{op}} = \mathsf{F}\ \mathsf{contains}\ w_1\ \mathsf{op}\ w_2\ \mathsf{op} \cdots \mathsf{op}\ w_n,$$

where n is no greater than 5 for Config-Trun. Specifically, the first subsuming query Q_{W} uses the operator W to connect the words, which represents the best-case translation. The second subsuming query Q_{AND}, however, assumes that the underlying service does not support the proximity operators, and therefore uses AND as the connecting operator.

Because we have no access to a search service that can handle both the equals-queries and contains-queries, we evaluated the queries with two services:*Folio* and *Dialog*, respectively. However, while *Dialog-Inspec* has full coverage of the *Inspec* documents (since 1968), *Folio-Inspec* maintains only those published since January 1988. Therefore, we added additional constraint restricting the publication years of documents, PY $\geq$ 1988, for the queries submitted to *Dialog* so that both *Dialog* and *Folio* search the same subset of *Inspec*.

Surprisingly, the results (not shown fully here) demonstrate high consistency for all the sample queries. Because both the equals–queries and the contains-queries are extremely selective, almost all the equals-queries return only one hit, and the contains-queries closely approximate the equals-queries with the hits ranging from 1 to 10. Because of the small result sizes, batch post-filtering is always feasible, while in the previous two sets of experiments there are cases when it is not.

For the incremental cost, Figure 10 shows the corresponding metric, the average $\mathsf{SR}(Q^S, Q)$'s. Overall, the selectivity reduction is about 2, which indicates that incremental costs may also be acceptable.

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

Figure 10: Average selectivity ratios of the Equality experiments.

As expected, the costs increase for truncated phrases, and for systems that do not support proximity operators. First, comparing Config-Trun with Config-Full, if the query phrase is truncated, the longer the truncation length (5 in our experiments), the better contains-queries approximate equals-queries. Second, if proximity operators are available, the contains-queries can enforce the adjacency of the extracted words and thus better approximate the equals-queries.

6 Concluding Discussion

The emergence of a networked information environment has not only made information more accessible and available, but also forced users to deal with the heterogeneous services. Although many search services can help users query and retrieve information, they all support different query languages and varying capabilities. To address this heterogeneity, we proposed the notion of front-end to unify the search facilities and provide the illusion of uniform capabilities across the underlying services. A front-end does not internally manage data of its own; instead it relies on external services to provide information necessary to fulfill user queries. First, to unify the search facilities, the front-end must be able to translate user queries in a unified language to those supported natively by the underlying services. It is important to note that this translation also makes possible *multi-search* with a single user query. Second, to provide the illusion of uniform search capabilities, it is desirable that the front-end do post-filtering to locally implement the missing functionalities.

Probably the closest work to ours, in terms of the shared goals, is the recent development of *meta-searchers* on the Internet, such as *SavvySearch* (accessible at `http://www.cs.colostate.-edu/~dreiling/smartform.html`) and *MetaCrawler* [17]. Those meta-searchers, similar to our notion of front-end, provide a single, central interface for Web document searching, and query other existing services to provide necessary information. 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.

For a front-end to really support sophisticated search features, post-filtering must be used to compensate for the capability difference between the underlying services and the front-end. In this paper we presented experimental results illustrating the batch and incremental post-filtering costs, with the metrics defined in Section 2. We did not report on all the experiments we have performed, but we believe that the results we have shown here represent the range of costs. Our results show that in some cases post-filtering has a significant cost. However, notice that post-filtering is required only when users wish to see the exact answers to their queries, which is usually during the final stage of information searching when the results are of manageable sizes.

Moreover, notice that in some cases it may be possible to skip the post-filtering at the front-end. That is, the user query is still translated according to our algorithms, and all of the results of the subsuming query are given to the user. Notice that our translation algorithms guarantee that the numbers of extra documents returned by the native queries are minimal [3, 4]. We still have to pay the overhead of fetching additional documents, but there is no filtering work at the front-end. The user does get some documents that do not match the query, but the costs at the front-end are reduced. For example, *MetaCrawler* gives users the option to eliminate post-filtering: it can operate in either normal or verification mode, where only the latter performs post-filtering for phrase queries.

The batch post-filtering cost is given by $\mathsf{Size}(\langle Q^S \rangle)$; in principle, the upper bound is the size of the underlying collection and the lower bound is $\mathsf{Size}(\langle Q \rangle)$. Our experiments show that the batch post-filtering cost can be extremely large in some cases, *e.g.*, if the user query is not selective enough to begin with. On the other hand, batch post-filtering is feasible when the subsuming queries still maintain the high selectivities of the user queries, *e.g.*, as was the case for the equality experiments. In effect, because $\mathsf{Size}(\langle Q^S \rangle)$ (the number of hits of the native query) can be known before doing post-filtering, the front-end can decide and advise users of the feasibility of batch post-filtering.

The incremental post-filtering cost measures the per-document cost, *i.e.*, $\mathsf{SR}(Q^S, Q)$, which ranges from slightly more than 1 to less than 100 in most cases. Notice that incremental post-filtering processes preliminary results only as users request matching documents, and hence users have control over the costs. With fast search engines and networks, combined with changing information access economics, the additional processing time and cost may be acceptable for users to retrieve desired information with less effort on their part.

Furthermore, during incremental post-filtering, the front-end can dynamically report the processing costs and the expected final result sizes so that users can decide whether to retrieve more documents or to reformulate the queries. Suppose the front-end processes the first M documents in the preliminary results and finds N qualified documents; thus, the average processing cost is $\frac{M}{N}$. Assuming a uniform distribution of qualified documents over the preliminary results, the final result size can be extrapolated as $\mathsf{Size}(\langle Q^S \rangle) \times \frac{N}{M}$. The estimation accuracy increases as more documents are processed.

In general, our approach of query translation and post-filtering can provide users feedback at various stages of processing, given that information searching is highly interactive in nature. First, when translating a user query Q to the native query Q^S, the front-end can estimate the expected post-filtering cost (*e.g.*, in terms of $\mathsf{SR}(Q^S, Q)$) based on some standard cost functions associated with the translation rules. For instance, the average results of our experiments can be used as standard costs for various translation patterns. Specifically, if $Q = P_1 \wedge P_2$ (conjunction of two atomic predicates), we can estimate $\mathsf{SR}(Q^S, Q)$ as $\mathsf{SR}(P_1^S, P_1) \times \mathsf{SR}(P_2^S, P_2)$, where $\mathsf{SR}(P_i^S, P_i)$ depends on the translation rules applied to P_i. Therefore, the front-end can inform the user about the expected costs and even precisely which part of the query is problematic. The user can then reformulate just that part to reduce the cost.

Furthermore, after the native query is evaluated, the front-end can report the size of the preliminary results. If the size is small, the user may decide to do batch post-filtering directly, which yields all the final results at once. Otherwise, the front-end may instead post-filter incrementally and estimate the final result size dynamically, as just described. At any point, based on the execution status, the user can continue to refine the query until the results are manageable.

Although our experiments focused on simple queries with isolated features, we believe the results can be easily extended to complex queries as well. In a Boolean system, complex queries are built by combining simple queries (predicates) with operators like AND and OR. By isolating features, our experiments evaluate the costs associated with particular translation patterns, such as proximity operator substitution and stopword removal. By construction, and assuming predicates are independent (which seems reasonable for our purposes) the cost of complex queries can be estimated as just illustrated. We plan to evaluate this strategy carefully in the future.

In this paper we have focused on the Boolean query model, because it is used by most commercial search service and library systems. However, there are also other kinds of popular query models, the most prominent being the vector-space query model, and we have started to do some work on it. In the vector space model, documents are retrieved and ranked based on their "similarity" with the queries. If a front-end decides to support the vector-space model, translation to an underlying service that also supports vector-space queries is relatively straightforward, because there is no strict syntax, there are typically no sophisticated features, and there are not many dialects as with Boolean queries. However, the major challenge is to collate results returned from different services,*i.e.*, to merge the different rankings. The problem is hard because all the search engines have their proprietary algorithms to compute similarities and rankings, and the details are not publicly available. To help meta-searchers perform the rank-merging properly, it is desirable that the services return some ranking information along with the results, as suggested by STARTS [18] (accessible at `http://www-db.stanford.edu/people/-gravano/starts_home.html`), an emerging protocol for Internet retrieval and search. We have also studied how much data must be retrieved from a ranked source in order to do meaningful merging [19].

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

Acknowledgments

Many useful comments on our work were provided by Andreas Paepcke, Matt Jacobson, Luis Gravano, and other members of the Stanford Digital Library Project.

References

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

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

[3] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, August 1996.

[4] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources (extended version). Technical Report SIDL-WP-1996-0044, Dept. of Computer Science, Stanford Univ., Stanford, California, September 1996. Accessible at `http://www-diglib.stanford.edu`.

[5] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Predicate rewriting for translating boolean queries in a heterogeneous information system. Technical Report SIDL-WP-1996-0028, Dept. of Computer Science, Stanford Univ., Stanford, California, January 1996. Accessible at `http://www-diglib.stanford.edu`.

[6] A. E. Negus. Development of the Euronet-Diane Common Command Language. In *Proc. 3rd Int'l Online Information Meeting*, pages 95–98, 1979.

[7] National Information Standards Organization. *Z39.58-1992 Common Command Language for Online Interactive Information Retrieval*. NISO Press, Bethesda, M.D., 1993.

[8] ISO. *ISO 8777:1993 Information and Documentation – Commands for Interactive Text Searching*. Int'l Organization for Standardization, Geneva, Switzerland, first edition, 1993.

[9] S.E. Preece and M.E. Williams. Software for the searcher's workbench. In *Proc. of the 43rd American Society for Information Science Annual Meeting*, volume 17, pages 403–405, 1980.

[10] S. Zinn, M. Sellers, and D. Bohli. OCLC's intelligent gateway service: Online information access for libraries. *Library Hi Tech*, 4(3):25–29, 1986.

[11] R. S. Marcus. User assistance in bibliographic retrieval networks through a computer intermediary. *IEEE Trans. on Systems, Man, and Cybernetics*, smc-12(2):116–133, 1982.

[12] D. E. Toliver. OL'SAM: An intelligent front-end for bibliographic information retrieval. *Information, Technology and Libraries*, 1(4), 1982.

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

[14] Denis Howe. The free on-line dictionary of computing. Accessible at `http://wombat.doc.ic.ac.uk/`.

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

[16] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The effectiveness of *GlOSS* for the text-database discovery problem. In *Proceedings of the 1994 ACM SIGMOD Conference*, May 1994.

[17] Erik Selberg and Oren Etzioni. Multi-service search and comparison using the MetaCrawler. In *Proc. of the 4th Int'l WWW Conf.*, 1995. Accessible at `http://-metacrawler.cs.washington.edu:8080/-papers/www4/html/Overview.html`.

[18] Luis Gravano, Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. STARTS: Stanford proposal for Internet meta-searching. In *Proceedings of the 1997 ACM SIGMOD Conference*, 1997.

[19] Luis Gravano and Héctor García-Molina. Merging ranks from heterogeneous internet sources. Technical Report SIDL-WP-1997-0063, Dept. of Computer Science, Stanford Univ., Stanford, California, February 1997. Accessible at `http://-www-diglib.stanford.edu`.