Design of Efficient Query Interfaces for Web Sources

Ramana Yerneni
Stanford University
yerneni@cs.stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Abstract

Data sources over the Web publish their query interfaces through forms or templates. In order to keep the query interfaces simple and efficient, it is desirable to design concise template sets for data sources. In this paper, we study the problem of minimizing the number of templates required to represent the query interface of a Web source. We show that the problem is intractable in general. However, we develop efficient minimization algorithms for problem instances that occur often in practice. We also present techniques that yield approximate solutions to the general case of the problem.

1 Introduction

With the advent of the Web, data sources that present limited query interfaces (e.g., search forms) have become common place. These interfaces are often determined both by the capabilities of the underlying databases at the sources and by the query requirements of their target customers. In this paper, we deal with the fundamental problem of designing concise query interfaces for Web data sources.

1.1 Query Templates

Templates are often used ([6, 9, 10, 12]) to describe query interfaces. A template indicates the restrictions on attribute specification in queries to the source (e.g., some attributes are required while others are optional). These restrictions are specified as attribute *adornments* in the templates. We adopt the attribute adornments introduced by [12], as they are well suited for describing the query interfaces of Web sources. The adornments are: **f** (for free) – the attribute may or may not be specified in the query; **u** (for unspecifiable) – the attribute cannot be specified in the query; **b** (for bound) – the attribute must be specified in the query; **c**[S] (for constant) – the attribute must be specified, and in addition, it must be chosen from the set of constants S; **o**[S] (for optional) – the attribute may or may not be specified in the query, and if specified, it must be chosen from the set of constants S.

EXAMPLE 1.1 Consider a Web bookstore that exports the view *BOOKS(author, title, format)*. Suppose that the source supports two query forms, each allowing users to enter certain queries.

When using the first form, the *author* and *format* (e.g., hard cover, soft cover, audio tape, etc.) fields must be specified, along with optional specification of the *title* field. In the second form, the user must enter the required fields *title* and *format*, but there is no place to specify the desired *author*. We can represent this two-form query interface by two templates: *bfb* and *ubb*. (The adornments are given in the same order as the attributes in the *BOOKS* schema.)

To illustrate the use of constant menus, suppose that the *format* attribute can only be specified through a pull down menu with two choices, 'Hard cover', and 'Paperback'. In this case, the templates would be *bfc['Hard cover','Paperback']* and *ubc['Hard cover','Paperback']*. □

1.2 Designing Source Interfaces

The query templates succinctly represent the capabilities and functionality of a source, and can be used to drive the design of Web forms, to aid in query optimization, or to informs users and applications of available services. Our next example illustrates one way in which templates can be used in the design of source interfaces.

EXAMPLE 1.2 Consider the Web bookstore of the previous example that exports the view *BOOKS(author, title, format)*. Customer requirements are gathered by interviewing potential users and determining the queries they are most likely to submit. Each prototypical query leads to one template that should be offered by the source. For example, if users want to search for books by only giving the *author* name, then this leads to the template *ubu*.

Overall, suppose that the expected customer needs lead to the following seven templates: $\{buu, ubu, bbu, uub, ubb, bub, bbb\}$. We could then meet these needs with seven search forms, one for each template. However, note that all the expected customer needs can also be satisfied with three templates: $\{bff, fbf, ffb\}$. The first template *bff* allows all queries permitted by the four templates *buu*, *bbu*, *bub* and *bbb*. The second new template *fbf* also covers four templates in the original design, but two of them are the same as those covered by the *bff* template. Finally, the new *ffb* template also covers four templates, only one of which is not covered by the previous two new templates, namely *uub*.

The *BOOKS* interface can support either design, the one with the seven templates or the one with the three templates. However, as we argue below in more detail, the three-template design is more compact, easier to implement, and easier to maintain. Thus, we believe it is important to minimize template sets to arrive at compact interface descriptions. The algorithms we present in this paper perform such minimization in an efficient fashion. □

Example 1.2 illustrates a "top down" interface design, where we start with user requirements and develop a concise set of query forms to offer. The interface design can also be driven in a "bottom up" fashion, driven by what the underlying data engines can provide. For example, consider a mediator that gets its information from other Web sources. Based on the templates offered by its sources, and the processing the mediator does, one can derive a set of mediator-supported templates [12]. Again, these templates can be minimized into a compact set, using the techniques we develop in this paper.

1.3 Concise Source Interfaces

Given a set of templates, why is it important to find a smaller set with the same functionality? There are several reasons, including:

- *User Satisfaction.* A customer wishing to ask a query needs to figure out which search form to use. Obviously, having more than a handful of forms may lead to customer aggravation and consequent loss of business.

- *Query Optimization.* A mediator or application that queries a Web source must develop a good execution plan. Each template offered by the source represents additional potential queries that must be considered in developing a good plan. Thus, the more succinctly the available queries can be described, the more efficient the mediator's query optimization process will be.

- *Maintenance.* Each form supported by an interface must be maintained as software and system evolve (e.g., the form must present the latest graphics, must support the latest browsers). The more forms a site supports, the larger its expense for constructing the corresponding Web pages and upgrading them on a continual basis.

Currently, template minimization is done manually and in an ad-hoc fashion. For example, given a set of templates that describe expected user queries, a human studies the templates and designs forms that can satisfy more than one query type at once. Unfortunately, manual solutions suffer from three main drawbacks. First, they are error prone (particularly when many attributes are involved and the adornments have menus of choices associated with them). An erroneous interface description can easily lead to customer aggravation, loss of business, inefficient query planning and execution. Second, manual solutions do not scale well with the increasing complexity of rich interfaces. Third, the interface requirements change often over the life of a Web source. It is not unusual for a service to change significantly the set of answerable queries every few weeks. Incrementally evolving the interfaces by hand is clearly undesirable.

In this paper we present algorithms for *automatically* finding concise representations for sets of query templates. Notice that "finding concise representations" may involve identifying and eliminating templates that are redundant in a given set of templates. It may also involve "coalescing" two or more templates in a set into a more powerful one. Or it may involve finding a new set of templates that is smaller and as a whole covers the original set. In particular, we make the following contributions:

- We define the template-set minimization problem (TSMP), as well as related problems like template subsumption, template coalescing, and template redundancy.
- We study the complexity of the problems, showing in particular that TSMP and the template redundancy problem are NP-hard, while the subsumption and coalescing problems can be solved efficiently (in polynomial time).
- We identify many practical scenarios of interface design for Web sources that can be mapped to well defined classes of instances of the TSMP problem, and develop efficient algorithms that yield minimum template sets for these classes.
- We present approximation algorithms for the general class of arbitrarily complex templates, i.e., the general case of the TSMP problem.
- We also present simple efficient algorithms for detecting template subsumption and for coalescing templates to form more powerful templates.

1.4 Overview of the Paper

In Section 2 we review relevant existing literature. In the next section, we formally define the template-set minimization problem (TSMP) and discuss its complexity. In Section 4, we present an efficient algorithm to detect template subsumption. Based on the subsumption detection algorithm we develop an efficient algorithm for TSMP. This algorithm yields optimal results for well-defined classes of templates that occur often in practical design scenarios in contemporary systems. In the general case, this algorithm becomes an efficient approximation algorithm. In Section 5, we extend this algorithm to arrive at another algorithm for TSMP that has performance guarantees for the general case also. Finally, in Section 6, we present an efficient algorithm to coalesce a given pair of templates. Template coalescing can be used in conjunction with our earlier algorithms for template-set minimization to improve upon their ability to yield near optimal designs.

2 Related Work

Query templates using attribute adornments have been employed to describe sets of answerable queries in many research prototypes [2,5,6,7,8,11], and recent literature [4,10,13]. The main emphasis of this body of research is on issues surrounding query planning, as opposed to interface design. By focusing on the problem of efficient interface design for templates, our paper complements this body of research. The solutions developed here can be employed by systems like TSIMMIS [6] and GARLIC [5] to design powerful interfaces to their data sources and use the concise template sets so obtained to efficiently plan the query execution over these sources. For instance, in Section 4 we develop a very efficient algorithm that yields the most-concise template sets for data sources in TSIMMIS [6].

The sets of interface adornments used in most previous systems are very limited. For instance, [4] and [13] use only b and f adornments. Data sources on the Web have a wide range of query-processing capabilities and interface requirements from their target customers. This scenario is evidenced by the large number and variety of search forms on the Web (e.g., [14, 15, 16]). To describe this variety of query interfaces, extended sets of attribute adornments, and templates based on these adornments, are proposed in recent literature [12]. In studying the problem of obtaining concise query interfaces, we adopt the extended set of adornments of [12] instead of the simpler sets of adornments of frameworks like [4] and [13].

Mediation systems extend the limited capabilities of data sources they work with [1, 2, 5, 6]. Essentially, the templates of mediators will be more flexible than the corresponding data sources, depending on the abilities of such systems to extend the source capabilities. Recent work [12] has shown how to compute the capabilities of mediators, based on the capabilities of their sources. During the computation of the set of templates for a mediator, we may end up with large template sets when the mediator is integrating information over many sources. Large template sets lead to undesirable consequences like inefficient query processing and cumbersome user interfaces. The techniques we develop here are very useful in alleviating the problems with complex interfaces of large-scale mediators.

3 Problem Definition

The main problem we are studying in this paper, which is at the core of a good interface design, is formally defined as follows:

A template indicates a set of queries answerable by the source. We call this set of queries the set of *supported queries* of the template. We define the set of supported queries of a set of templates as the union of the sets of supported queries of each template in the set. A template set is *equivalent* to another template set if they both have the same set of supported queries.

Template-Set-Minimization Problem (TSMP). Given a template set T, what is the smallest set of templates M such that M and T are equivalent?

Note that in TSMP, the smallest equivalent set is not restricted to be a subset of the given set of templates (i.e., M does not have to be a subset of T).

The following theorem establishes the complexity of TSMP.

Theorem 3.1 *TSMP is NP-hard.* □

Proof: We show that TSMP is intractable by reducing the CNF-SAT problem [3] to it. Given an instance of the CNF-SAT problem, we create a template corresponding to each clause in the CNF expression. Specifically, we define an attribute corresponding to each variable in the CNF expression. The templates specify adornments over this attribute set. When constructing a template corresponding to a clause, the adornment of an attribute is f if the corresponding variable does not appear in the clause, the adornment is b if the variable appears as a positive literal and the adornment is u if the variable appears as a negative literal. Thus, we construct a set of templates T corresponding to the set of clauses in the CNF expression. We now argue that the CNF expression is satisfiable if and only if T can not be minimized to the singleton set containing the template with f adornment for all the attributes (i.e., $T \neq \{fff...f\}$).

If T cannot be minimized to $\{fff...f\}$, there must exist a query Q that is not supported by T. Based on Q, let us construct an assignment A such that A assigns 1 to variables whose corresponding attributes are not specified by Q and A assigns 0 to variables whose corresponding attributes are specified by Q. Now, consider a clause c_i in the given CNF expression. Since Q is not supported by the corresponding template t_i, it must be the case that t_i has a u adornment for an attribute that Q specifies or t_i has a b adornment for an attribute that Q does not specify. In the former case, c_i has a negative literal whose variable is assigned a value of 0 in A. In the latter case, c_i has a positive literal whose variable is assigned a value of 1 in A. In other words, A satisfies c_i. Thus, we conclude that if T cannot be minimized to $\{fff...f\}$, the CNF expression is satisfiable.

If the CNF expression is satisfiable, there must be a satisfying assignment A'. Based on A, let us construct a query Q' such that Q' specifies a value for all the attributes whose corresponding variables are assigned a value of 0 in A and Q' leaves all other attributes unspecified. Now, consider a template t_j in T. Since A' is a satisfies the corresponding clause c_j, it must be the case that c_j has a positive literal whose variable is assigned a value of 1 in A' or c_j has a negative literal whose variable is assigned a value of 0 in A'. In the former case, Q' leaves unspecified an attribute that has the b adornment in t_j. In the latter case, Q' specifies an attribute that has the u adornment in t_j. That is, Q' is not supported by t_j. Thus, we see that Q' is not supported by any of the templates in T and so T cannot be equivalent to $\{fff...f\}$. In other words, if the CNF expression is satisfiable, T cannot be minimized to $\{fff...f\}$. ∎

Since TSMP is intractable, the next logical step is to investigate if it remains intractable for special cases of TSMP arising from restrictions on the kinds of templates involved. One particularly well-known class of templates to consider is the one that does not use menus of choices to restrict the constants that must be specified. That is, we consider templates that use only the b, f and the u adornment. TSMP remains intractable for this class of templates. However, it turns out that for some other restricted classes of templates, TSMP can be solved efficiently. Moreover, these "good" classes happen to be used often in practice and in many query-processing systems dealing with Web sources. In the next section, we develop an efficient algorithm to solve TSMP for these restricted classes of templates.

4 Template Subsumption

One approach to obtaining concise source designs is to look for redundant templates in a given design. Each template has a set of supported queries. If all the queries supported by a template are also supported by another template, then the first template can be eliminated from any set that contains the second one. For instance, all the queries supported by the template *ubfb* are also supported by the template *fbff*. So, if we are given the design $\{ubfb, fbff\}$ we can come up with the more concise design $\{fbff\}$.

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	yes	no	no	no	no
o$[S_{11}]$	yes	yes if $S_{11} \subseteq S_{21}$	no	no	no
b	yes	no	yes	no	no
c$[S_{12}]$	yes	yes if $S_{12} \subseteq S_{21}$	yes	yes if $S_{12} \subseteq S_{22}$	no
u	yes	yes	no	no	yes

Table 1: Adornment restrictiveness checking.

We formalize the notion of identifying redundant templates as described above. We say that a template t subsumes another template t' if every query supported by t' is also supported by t. Based on this notion of template subsumption, we develop a very efficient algorithm for minimizing query interfaces. It turns out that this simple algorithm can yield optimal designs in many contexts involving restricted classes of templates.

We need to first construct an algorithm to test for template subsumption. We can ascertain if a template t subsumes another template t', based on the notion of the degree of restrictiveness of the attribute adornments, which in turn is specified by Table 1. The adornment on the left side of the table is at least as restrictive as the adornment on the top of the table if the corresponding entry in the table says "yes." For instance, b is more restrictive than f and c is more restrictive than b.

Intuitively, a template that has more restrictive adornments is subsumed by a template that has less restrictive adornments. This intuition is formalized in the following lemma.

Lemma 4.1 *A template is subsumed by another if and only if for each attribute, the adornment in the former is at least as restrictive as the adornment in the latter.* □

Proof: Consider two templates t and t'. If for each attribute the adornment in t is at least as restrictive as the adornment in t', it is clear that every query supported by t is also supported by t'. If there is an attribute (say A) for which the adornment in t is not at least as restrictive as the adornment in t', we can construct a query that is supported by t and not supported by t'. The query specifies the A attribute in such a way that the specification satisfies the restriction of the A adornment in t and does not satisfy the restriction of in t'.

Thus, t is subsumed by t' exactly when each attribute is at least as restrictive in t as it is in t'. ■

Figure 1 shows an efficient algorithm called SUBSUME for testing if a given template t is subsumed by another given template t'. SUBSUME simply checks if for each attribute t is at least as restrictive as t', by looking up Table 1. The number of lookups to this table is k, where k is the number of attributes in the templates. Each lookup may take $O(m)$ time where m is the maximum size of a menu in the two templates.[1] Thus, we see that algorithm SUBSUME ascertains

[1]We assume for simplicity that operations like checking for equality of two menus and constructing the union of two menus take time that is proportional to the size of the menus.

Algorithm 4.1 *SUBSUME*
Input: Two templates t and t'
Output: 1 if t is subsumed by t'; else 0
Method:

```
For each attribute A in t and t'
    Let x be the adornment of A in t
    Let y be the adornment of A in t'
    If (Table1[x, y] is not yes)
        Return(0)
Return(1)
```

Figure 1: Algorithm *SUBSUME*

if a template subsumes another template in $O(km)$ time.

4.1 Algorithm SIMPLE

Based on the notion of template subsumption, we devise an algorithm called SIMPLE, for computing the most concise design equivalent to a given design. As shown in Figure 2, given a set of templates, algorithm SIMPLE identifies and eliminates templates that are subsumed by other templates in the set.

We illustrate algorithm SIMPLE by way of an example.

EXAMPLE 4.1 Given the set of templates $\{bfu, ubf, bbu\}$, algorithm SIMPLE identifies that the bbu template is subsumed by the bfu template. Consequently, it eliminates the bbu template and arrives at $\{bfu, ubf\}$, as neither of the remaining templates is subsumed by any template in the given set.

Now, consider the set of templates $\{bfu, ubf, fbu\}$. As before, the first two templates are not subsumed by any template in the set. The third template fbu is also not subsumed by either of the other two templates. So, algorithm SIMPLE returns the original set of three templates. However, we can see that every query supported by the fbu template is also supported by one of bfu or ubf templates. That is, the fbu template is actually redundant with respect to the other templates. So, there is a set of two templates $\{bfu, ubf\}$ that is equivalent to the original set. □

Algorithm SIMPLE conducts $O(n^2)$ checks of template subsumption, where n is the number of templates in its input set. It then filter out the subsumed templates in $O(n)$ time. Each check for template subsumption takes $O(km)$ time. Thus, SIMPLE runs in $O(n^2km)$time.

Algorithm 4.2 *SIMPLE*
Input: A set of templates T
Output: An equivalent set of templates S
Method:

$S \leftarrow T$
For each $t \in S$
    For each $t' \in T$
        If (SUBSUME(t, t'))
            $S \leftarrow S - \{t\}$
Return(S)

Figure 2: Algorithm *SIMPLE*

Although algorithm SIMPLE is efficient, it has the drawback of not being able to obtain optimal solutions to the template-set minimization problem (TSMP) in all cases, as illustrated by the above example. However, there are well-defined classes of templates for which SIMPLE is guaranteed to produce optimal solutions. Moreover, some of these classes of templates are very useful in the sense that they occur in practical systems dealing with query interfaces of Web sources.

The first class of restricted templates we consider is the class of *atomic* templates. Templates in this class are restricted to use only the b, u and c adornments. Moreover, the c adornments can only have single constants in their menus. There are two reasons why we study the class of atomic templates. First, atomic templates are fundamental – a combination of atomic templates can express any template over the set of all adornments; an atomic template cannot be expressed as a combination of any other templates. Second, atomic templates are used in many practical systems, like the TSIMMIS system at Stanford University [6] [2].

We first identify an interesting property of atomic templates and then use this property to assert that algorithm SIMPLE yields *optimal* solutions to TSMP for the class of atomic templates.

Lemma 4.2 *Given an atomic template t and a set of atomic templates T, the set of supported queries of t is a subset of the set of supported queries of T if and only if t is subsumed by some template in T.* □

Proof: It is obvious that if t is subsumed by some template in T, the set of supported queries of t is a subset of the set of supported queries of T.

[2] In TSIMMIS, access to data sources is through *wrappers* that translate user queries to those supported by the underlying sources. Wrappers can sometimes perform special postprocessing steps on the results of source queries and hence convert the atomic templates of the source query interfaces into more powerful templates

If t is not subsumed by any template in T, we construct a query Q that is supported by t and not by T. Query Q specifies a unique value (which does not appear in any of the constant menus of the templates in T) for all the attributes that have the b adornment in t. Attributes that have the u adornment in t are left unspecified by Q. In the case of attributes with c adornments in t, Q specifies the required constant values. Obviously, Q is supported by t.

If a template in T were to support Q, it must have the u adornment for attributes left unspecified by Q, b adornment for the attributes that have the unique value specified by Q and either a b adornment or a c adornment (with matching constants) for other attributes (that had the c adornments in t). Clearly, such a template subsumes t. Since T must have a template that supports Q in order for Q to be in the set of supported queries of T, we conclude that the set of supported queries of T is not a superset of the set of supported queries of t if T does not contain a template that subsumes t. ■.

Theorem 4.1 *Algorithm SIMPLE yields optimal solutions to TSMP for atomic templates.* □

Proof: The set of templates obtained by SIMPLE can not support more queries than the original set because the former is a subset of the latter. It is also easy to see that the set obtained by SIMPLE supports all the queries that the original set supports because every template that is dropped by SIMPLE must have a subsuming template in the remaining set obtained by SIMPLE (note that template subsumption is a transitive property). Thus, SIMPLE returns a set of templates that is equivalent to the original set.

Next, for the class of atomic templates, we show that the set returned by SIMPLE is the smallest set equivalent to the original set. Given a set of templates T, let S be the set of templates obtained by SIMPLE and let M be a smallest set of templates equivalent to T. Consider a template t in S that is not in M. We know that every query supported by t must also be supported by M (because S and M are equivalent). Therefore, based on Lemma 4.2, there must exist a t' in M that subsumes t. We know that t is not subsumed by any template in T because it is in S. Therefore, t' cannot be in T. Since S is equivalent to M, every query supported by t' is also supported by S. Therefore, once again based on Lemma 4.2, there must exist a t'' in S that subsumes t'. Since subsumption is transitive, we conclude that there is a template t'' in S that subsumes another template t in S. This conclusion is impossible, so we see that there can not be a template t that is in S and that is not in M. That is, M is a superset of S and so the set of templates obtained by algorithm SIMPLE is indeed the smallest set equivalent to the original set of templates. ■

In systems like TSIMMIS, *wrappers* are deployed on top of data sources with limited query-processing capabilities. An important function of the wrappers is to enhance the query-processing capabilities of the underlying data sources by providing additional filtering steps. For instance, if a source does not allow the specification of an attribute value in a query (i.e., it has the u adornment), the wrapper may enhance the capability by allowing the attribute value to be specified. When a user submits such a query, the wrapper can send a corresponding query to the data source by not specifying values for attributes with the u adornment and then postprocess the results of the source query by applying filtering conditions with respect to the values for these attributes specified in the user query. Essentially, the set of templates exported by the wrapper are just those that it obtains from the data sources, with the u adornments replaced by the f adornments. That is, wrappers with postprocessing capabilities use templates with b, f and c adornments (with singleton menus) when the corresponding sources use atomic templates. We call this new class of templates the *filter* templates. Like in the case of the atomic templates, algorithm SIMPLE is guaranteed to produce optimal results for filter templates.

Theorem 4.2 *Algorithm SIMPLE yields optimal solutions to TSMP for filter templates.* □

Proof: The proof is similar to that of Theorem 4.1 which established the ability of SIMPLE to yield optimal solutions to TSMP for atomic templates. ■

The above result is important for two reasons. First, we noted that systems employing wrappers to postprocess the results of source queries use filter templates to express the enhanced query interfaces. For such systems, we can use SIMPLE to efficiently design concise query interfaces. Second, many systems and query-processing frameworks – traditional ([2,8]) as well contemporary ([4,13]) – use templates from a subclass of filter templates. In particular, they use only b and f adornments. In such cases, algorithm SIMPLE can efficiently obtain concise interfaces.

There are other restricted classes of templates for which we can show that algorithm SIMPLE guarantees optimal solutions to TSMP. However, for the general class of unrestricted templates, we have seen that algorithm SIMPLE can yield suboptimal solutions to TSMP. Moreover, in the general case, there is no nontrivial bound on how inferior the solution yielded by SIMPLE can be, when compared with the optimal solution.

5 Irredundant Template Sets

In the previous section, we have seen an efficient algorithm called SIMPLE that guarantees optimal designs if the templates involved are of certain kinds. In the general case, we need to do better. In this section, we will develop an algorithm called SPLITCHECK that guarantees certain quality of output in its computation of concise interfaces. SPLITCHECK achieves this guarantee, based on a stronger notion of template *redundancy* defined as follows.

We say that a template t is redundant in a set of templates T if every query supported by t is also supported by at least one other template in T. The notion of template redundancy is similar to but subtly different from the notion of template subsumption. Specifically, if a template t is subsumed by another template t' in a set of templates T, it is clear that t is redundant in T. However, t can be redundant in T even if T does not have any other template t' that subsumes t.

Algorithm SIMPLE attempted to solve the problem of minimizing a given template set by identifying and eliminating templates in the set that are subsumed by other templates in the set. Consequently, SIMPLE may end up with a template set that has redundant templates in it. Algorithm SPLITCHECK goes beyond SIMPLE to identify and eliminate templates that are redundant in the given set, thus guaranteeing that the yield is an irredundant set of templates. In a sense, we can think of SIMPLE also trying to eliminate redundant templates, but using the simpler notion of subsumption to approximate redundancy. Consequently, SIMPLE is an efficient approximation algorithm to obtain irredundant sets of templates.

5.1 Detecting Template Redundancy

Formally, we define the Template Redundancy Problem (TRP) as follows: given a template t and a set of templates T, is t redundant with respect to T? As indicated by the following theorem, identifying template redundancy (i.e., solving TRP) is much harder than checking for template subsumption.

Theorem 5.1 *TRP is NP-hard.* □

Proof: The proof is by reducing CNF-SAT to TRP. Given a CNF expression, we create a template corresponding to each template in a way that is similar to the proof of Theorem 3.1 – b adornment for positive literal, u adornment for negative literal and f adornment if the variable is absent in the clause. The set of clauses in the CNF expression yield the set of templates T. We create another template t that has the f adornment for all attributes.

Using an argument similar to that in Theorem 3.1, we can show that the CNF expression is satisfiable if and only if t is not redundant with respect to T. ■

Even though in general it is very hard to test for redundancy of templates, for specific classes of templates the problem is tractable. In particular, we have the following variation of Lemma 4.2 that allows us to efficiently test for template redundancy in the case of atomic templates.

Lemma 5.1 *An atomic template is redundant with respect to a set of (atomic and non-atomic) templates if and only if it is subsumed by at least one of those templates.* □

Proof: The proof is similar to that of Lemma 4.2. ■

It turns out that every non-atomic template can be split into an equivalent set of atomic templates (see Lemma 5.2 and Figure 3). Thus, we can verify the redundancy of non-atomic templates by splitting them into their constituent atomic templates and checking for their redundancy easily using Lemma 5.1.

Lemma 5.2 *Every template can be split into a set of atomic templates such that the resulting set of atomic templates supports the same set of queries as the original template.* □

Proof: We prove this lemma constructively. That is, we construct a set of atomic templates starting with a given template and argue that resultant set has the same supported queries as the original template. Whenever the given template has an f adornment for an attribute, we split that into two templates, one with a b adornment and the other with a u adornment for that attribute. The split templates copy the adornments of all the other attributes from the original template. It is obvious that such a replacement of the original template with the two constituent templates preserves the set of supported queries. If the

Algorithm 5.1 *ATOMIZE*

Input: A template t

Output: A set of atomic templates equivalent to t

Method:

$D \leftarrow \{t\}$
While (D changes)
    $t' \leftarrow$ the first nonatomic template of D
    $D \leftarrow D - \{t'\}$
    If (t' has an f adornment)
        Let A be an attribute with f adornment in t'
        Create d_1 as a copy of t' except it has b adornment for A
        Create d_2 as a copy of t' except it has u adornment for A
        $D \leftarrow D + \{d_1\}$
        $D \leftarrow D + \{d_2\}$
    Else If (t' has an o adornment)
        Let A be an attribute with o adornment in t'
        Create d_1 as a copy of t' except it has u adornment for A
        Create d_2 as a copy of t' except it has c adornment with same menu of t' for A
        $D \leftarrow D + \{d_1\}$
        $D \leftarrow D + \{d_2\}$
    Else If (t' has a c adornment with multiple-choice menu)
        Let A be an attribute with a c adornment having a multiple-choice menu M in t'
        For each choice $c_i \in M$
            Create d_i as a copy of t' except with menu of $\{c_i\}$ instead of M for A
            $D \leftarrow (D + \{d_i\}$
Return(D)

Figure 3: Algorithm *ATOMIZE*

Algorithm 5.2 *REDUNDANT*

Input: A template t and a set of templates T

Output: 1 if t is redundant in T, 0 otherwise

Method:

```
D_t ← ATOMIZE(t)
For each d ∈ D_t
    For each t' ∈ T
        If (SUBSUME(d, t'))
            D_t ← D_t − {d}
If (D_t is empty)
    Return(1)
Else
    Return(0)
```

Figure 4: Algorithm *REDUNDANT*

template has an o adornment, we split it into two templates with the u and the c adornments. The menu of the o adornment in the original template is copied over to the c adornment in the split template. Once again, the original template is equivalent to the combination of the two templates it is split into. Continuing in this manner, we end up with a set of templates that do not have any f and o adornments. Then, we replace each template in this set that has a c adornment for an attribute with a set of templates that have c adornments with singleton menus for that attribute. Thus, we construct a set of atomic templates that are equivalent to a given template. ∎

Based on Lemma 5.1 and Lemma 5.2, we construct algorithm REDUNDANT (see Figure 4) to test for the redundancy of a template t with respect to a set of templates T. REDUNDANT uses algorithm ATOMIZE (see Figure 3) to enumerate the set of constituent atomic templates of t and checks their subsumption by the templates in T. If all the atomic constituents of t are subsumed by templates in T, REDUNDANT declares that t is redundant with respect to T. The worst-case running time of REDUNDANT is $O((m+2)^k nkm)$ where m is the maximum size of any menu in t, n is the number of templates in T and k is the number of attributes over which the templates are defined. The complexity expression is derived by noting that ATOMIZE finds $O((m+2)^k)$ constituent atomic templates for t and for each of these constituents REDUNDANT tests for subsumption with respect to some template in T (which has n templates and each subsumption test takes $O(km)$ time).

Algorithm 5.3 *SPLITCHECK*
Input: A set of templates T
Output: An irredundant subset S of T
Method:

```
S ← T
For each t ∈ S
    For each t' ∈ T
        If (SUBSUME(t, t'))
            S ← S − {t}
For each t ∈ S
    If (REDUNDANT(t, S − {t})
        S ← S − {t}
Return(S)
```

Figure 5: Algorithm *SPLITCHECK*

5.2 Eliminating Redundant Templates

Given a set of templates, algorithm SPLITCHECK (see Figure 5) first mimics algorithm SIMPLE to eliminate subsumed templates. Then, it uses algorithm REDUNDANT to identify and eliminate redundant templates that are present among the remaining templates. Note that if SPLITCHECK does not eliminate all the subsumed templates first, it is possible for SPLITCHECK to end up with an equivalent set of templates that is larger (with more templates) than the one obtained by SIMPLE! This possibility is due to the choice of different sequences of eliminating redundant templates, about which algorithm SPLITCHECK is arbitrary.

EXAMPLE 5.1 As in Example 4.1, consider the set of templates $\{bfu, ubf, fbu\}$. Recall that algorithm SIMPLE could not identify that fbu is redundant in this set of templates because it is not subsumed by either of the other two templates. That is, SIMPLE returns the given set as its result.

Algorithm SPLITCHECK first finds that none of the three templates is subsumed by others. Then, it examines each template for redundancy. Template bfu is not redundant because one of its atomic constituents, buu, is not subsumed by either ubf or fbu. Similarly, ubf is not redundant because one of its atomic constituents ubb is not subsumed by either bfu or fbu. However, after splitting the third template fbu into $\{bbu, ubu\}$, we can ascertain that both its atomic constituents are subsumed (bbu by bfu and ubu by ubf). Therefore, SPLITCHECK correctly identifies that fbu is redundant and eliminates it. In the next iteration, neither of the remaining two templates is redundant and hence SPLITCHECK returns the irredundant subset $\{bfu, ubf\}$.

On a closer examination, we can see that the solution obtained by SPLITCHECK in this case indeed turns out to be the smallest set equivalent to the given set (i.e., there cannot a singleton set of templates that is equivalent to the given set). □

Given a set of templates T, algorithm SPLITCHECK runs in time that is polynomial in n, the cardinality of T, and exponential in k, the number of attributes in t. The first step of mimicking SIMPLE to eliminate subsumed templates takes $O(n^2km)$ time. The number of iterations to check and eliminate redundant templates is bounded by n. Each iteration is dominated by the call to REDUNDANT which runs in $O((m+2)^k nkm)$ time. Thus, the worst-case running time of SPLITCHECK is $O(n^2km + n(m+2)^k nkm)$.

Even though SPLITCHECK has a worst-case time complexity that is exponential in its input size, it can be very useful in practice. First, the exponent is the number of attributes (typical small), not the number of templates or the number of choices in menus (typically large). Second, eliminating redundant templates is done in an "offline" manner, at the time of design (not at query execution time). The benefits of a good design far outweigh the potentially long time SPLITCHECK takes to run.

6 Template Coalescing

Algorithm SPLITCHECK yields irredundant designs. Nevertheless, there may still be opportunities to find more concise designs. For instance, consider the set: $\{bbuf, bbbf, buff, bfbb\}$. The only template that is redundant in this set is $bfbb$ and so SPLITCHECK will yield the irredundant design: $\{bbuf, bbbf, buff\}$. However, a more concise equivalent design is: $\{bfff\}$. This design can be obtained by recognizing that we can *coalesce* the templates of the earlier design, two at a time. For example, $bbuf$ and $bbbf$ coalesce to form $bbff$, which in turn coalesces with $buff$ to form $bfff$. In a sense, the process of coalescing templates is inverse to the process of splitting templates employed by SPLITCHECK (to identify the redundancy of "composite" templates).

If it were true that we can always coalesce two templates to form a single equivalent template, we can minimize any given set of templates into a singleton set. However, it is not always possible to coalesce a pair of templates, as illustrated by the following example.

EXAMPLE 6.1 Consider two templates: bub and ubb. These two templates cannot be coalesced into a single template. That is, there does not exist any template that is equivalent to the combination of these two templates. Note that it is quite easy to find a template that subsumes the two given templates, but such a template will allow queries not allowed by either of the two given templates. For instance, the template *ffb* subsumes both the templates, but it allows a query that specifies only the third attribute. This query is not allowed by either of the two given templates we started with. □

How can we determine if two templates can be coalesced? Fortunately, as with template subsumption, we can ascertain efficiently the ability to coalesce a given pair of templates. The following lemmas specify precisely the conditions for a pair of templates to be coalesced.

Lemma 6.1 *Two templates can be coalesced if they differ in at most one attribute adornment.* □

Proof: If two templates do not differ in any attribute adornment, then they are indeed the same templates. These two templates can trivially be coalesced into a single template, which has the same attribute adornments as the given templates. Now, consider two different templates t and t' that differ in the adornment of exactly one attribute. We show that t and t' can be coalesced into a single template s as follows. Let A be the attribute in whose adornment t and t' differ. We construct s by copying over the adornment of t for all attributes other than A. For A, the adornment in s is the combination of the adornments in t and t' according to Table 2. By examining the entries in Table 2, one can see that the adornment of the combined template is not more restrictive than the adornments of the templates being coalesced. For instance, when t has the b adornment and t' has the u adornment, s ends up with the f adornment, one that is less restrictive than both the b and the u adornments. Since s will have an adornment for each attribute that is not more restrictive than either of the two adornments in t and t' for that attribute, we conclude that s subsumes both t and t'. It remains to be shown that s does not allow any query that is not allowed by t and t'. Once again, a close examination of the entries in Table 2 reveals that there cannot be a query which satisfies the adornment of A in s, while at the same time does not satisfy either of the adornments of A in t and t'. For instance, the $o[S_{12}]$ entry for $c[S_{12}]$ and u indicates that every query that satisfies the $o[S_{12}]$ adornment of s either satisfies the $c[S_{12}]$ adornment of t or the u adornment of t'. Since s subsumes both t and t' and does not allow any query that is not allowed by either t or t', s is equivalent to the combination of t and t'. Thus, t and t' are coalesced to form s. ■

Lemma 6.2 *Two templates that differ in more than one attribute can be coalesced if and only if one subsumes the other.* □

Proof: It is obvious that if a template subsumes another, we can trivially coalesce the two templates to form a copy of the subsuming template. We now show that subsumption is a necessary condition for coalescing two templates that differ in more than one attribute adornment. Consider two templates t and t' that differ in more than one attribute adornment, such that neither t nor t' subsumes the other. Let s be the template obtained by coalescing t and t'. We will show that s cannot exist.

We consider two cases for the attribute-adornment difference in t and t'. The first case is when there is an attribute A for which t has an adornment that is more restrictive than the adornment in t' or vice versa. The second case is when no such attribute exists. In the first case, without loss of generality, let the adornment of A in t be more restrictive. Then, the adornment of A in s must be the same as the adornment in t' (in order for s to cover t' it cannot be more restrictive, and in order for s to be equivalent to the combination of t and t' it cannot be less restrictive). Since t and t' differ by more than one attribute and since t' does not subsume t, there must be another attribute B for which t has an adornment that is not more (same or less) restrictive than the adornment in t'. In other words, there must be a query whose specification for B

Algorithm 6.1 *COALESCE*
Input: Two templates t and t'
Output: 1 if t and t' can be coalesced, 0 otherwise
Method:

Let t and t' differ in X attribute adornments
If $((X \leq 1)$ or SUBSUME(t, t') or SUBSUME$(t', t))$
    Return(0)
Else
    Return(1)

Figure 6: Algorithm *COALESCE*

is such that the query is allowed by t and not by t'. Since s subsumes t, this query must be allowed by s. Now, we can modify this query to specify A in such a way that t cannot allow the modified query and yet s allows the modified query (because s has an adornment for A that is strictly less restrictive than t). Because of the specification of the B attribute, the modified query is disallowed by t' too. Thus, we see that there is a query allowed by s that is not allowed by t and t'.

In the second case, there is no attribute for which t is more restrictive than t' or vice versa. Let A and B be two attributes on which t and t' differ. We can construct a query that specifies A in such a way that t allows the query and t' disallows the query. Since s must subsume t, this query has to be allowed by s. We can then modify this query by specifying B so that t disallows the modified query. The modification to the specification of B can be done in such a way that the new specification satisfies the adornment of B in t' and consequently s. In this way, we again construct a query that is allowed by s and not allowed by both t and t'.

Thus, we have shown that there cannot be an s that subsumes both t and t' while not allowing any query disallowed by both t and t'. Hence, two templates that differ in more than one attribute adornments can only be coalesced if one subsumes the other. ∎

Based on the above two lemmas, we develop an efficient algorithm to determine the possibility of coalescing a pair of templates. Algorithm COALESCE, presented in Figure 6, Algorithm COALESCE runs in $O(km)$ time, where k is the number of attributes and m is the size of the largest menu of choices in the given pair of templates.

Once we determine that a pair of templates can be coalesced, the actual process of coalescing the two templates is fairly simple. The resultant template constructs the adornment of each attribute by consulting Table 2 with the adornments of the attribute in the given two templates.

Now that we know how to combine two templates, we can devise an algorithm that can incrementally condense a set of templates. Algorithm CONDENSE, shown in Figure 7, considers pairs of templates to be coalesced in an iterative manner. That is, in each iteration, CONDENSE finds a

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	**f**	**f**	**f**	**f**	**f**
o$[S_{11}]$	**f**	**o**$[S_{11} \cup S_{21}]$	**f**	**o**$[S_{11} \cup S_{22}]$	**o**$[S_{11}]$
b	**f**	**f**	**b**	**b**	**f**
c$[S_{12}]$	**f**	**o**$[S_{12} \cup S_{21}]$	**b**	**c**$[S_{12} \cup S_{22}]$	**o**$[S_{12}]$
u	**f**	**o**$[S_{21}]$	**f**	**o**$[S_{22}]$	**u**

Table 2: Combining adornments from coalesced templates.

Algorithm 6.2 *CONDENSE*
Input: A set of templates T
Output: An equivalent set of templates S
Method:

$S \leftarrow T$
While (COALESCE(t, t') for some $t, t' \in S$)
    $S \leftarrow S - \{t\}$
    $S \leftarrow S - \{t\}$
    Add to S the result of
        coalescing t and t'
Return(S)

Figure 7: Algorithm *CONDENSE*

pair of templates that can be coalesced and replaces the two templates by the template that results from coalescing the two templates. When no pair of templates can be coalesced, CONDENSE returns the remaining set of templates.

EXAMPLE 6.2 Consider the set of templates: $\{bbb, bub, ubb, uub\}$. Algorithm CONDENSE can first identify that the bbb and bub templates can be coalesced and then shrink the set to $\{bfb, ubb, uub\}$. In the next iteration, it can coalesce ubb and uub to arrive at $\{bfb, ufb\}$. Finally, it can then coalesce these templates to arrive at the concise singleton set: $\{ffb\}$.

Consider another set of templates: $\{bub, ubb, bbb, ufb\}$. Let CONDENSE first consider the pair ubb and bbb and coalesce the two to obtain $\{bub, fbb, ufb\}$. At this stage, every pair of templates differs by at least 2 attribute adornments and no template subsumes any other. So, we cannot coalesce any more pairs of templates and CONDENSE returns the design of 3 templates.

However, the given set $\{bub, ubb, bbb, ufb\}$ is again equivalent to the singleton set: $\{ffb\}$. To see this, note that the *ffb* template subsumes each of the four templates in the starting set. Can this singleton set be too powerful? It can be split into its equivalent set of atomic templates:

$\{bub, ubb, bbb, uub\}$. Now, we can see that the first three templates here are also in the starting set and the fourth one here is covered by the fourth template in the starting set. Therefore, we have a concise singleton equivalent to the starting set, while CONDENSE ended up with a design of three templates.

It is true that, starting with $\{bub, ubb, bbb, ufb\}$, if CONDENSE first coalesced bub and bbb, then coalesced ubb and ufb, and then coalesced the resulting two powerful templates bfb and ufb, it would have arrived at the singleton set $\{ffb\}$. In summary, CONDENSE can sometimes choose the "wrong" pairs of templates to coalesce and end up with suboptimal solutions. □

As illustrated by Example 6.2, algorithm CONDENSE is not guaranteed to find the smallest equivalent set. This fact is not surprising since CONDENSE runs in $O(n^3km)$ time while the problem it is trying to solve, namely TSMP, is intractable.

7 Conclusion

In this paper, we studied the problem of obtaining concise query interfaces. In particular, we formally defined the problem of minimizing template sets (TSMP); showed that TSMP is intractable; developed an efficient approximation algorithm, called SIMPLE, that guarantees optimal solutions in many practical instances of TSMP; presented another algorithm, called SPLITCHECK, that guarantees the generation of irredundant template sets in all cases; discussed a technique of combining templates that can be used to postprocess the output of SPLITCHECK to obtain more concise template sets.

Our future work will continue the search for other approximation algorithms that have practical performance guarantees in solving the general case of TSMP. We are also investigating many problems related to TSMP in the arena of template-set design and how the techniques developed in this paper can be applied to solve them. For instance, an important problem to be solved, when designing interfaces, is to compare alternative designs. Specifically, given two sets of templates we need to know if their sets of supported queries are the same or if one set is a subset of the other. These problems can be reduced easily to the Template Redundancy Problem we discussed in Section 5. Consequently, for the restricted classes of templates, namely atomic and filter templates, we can answer the design-comparison questions efficiently, based on algorithm SUBSUME (see Section 4). For the general class of templates, the design-comparison questions are NP-hard and can be answered using the techniques of Section 5 in time that is exponential in the number of attributes and polynomial in the number of templates in the two designs being compared.

References

[1] C. Chang, H. Garcia-Molina. Mind Your Vocabulary: Query Mapping across Heterogeneous Information Sources. *Data Engineering Bulletin*, 10(4):52-62, 1987.

[2] D. Chimenti, et al. An Overview of the LDL System. *Data Engineering Bulletin*, 10(4):52-62, 1987.

[3] M. Garey, D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. *W.H.Freeman and Company*, 1979.

[4] D. Florescu, et al. Query Optimization in the Presence of Limited Access Patterns. *Proc. SIGMOD Conference*, 1999. Also available as Technical Report, INRIA, 1998.

[5] L. Haas, et al. Optimizing Queries across Diverse Data Sources. *Proc. VLDB Conference* , 1997.

[6] J. Hammer, et al. Template-based Wrappers in the TSIMMIS System. *Proc. SIGMOD Conference* , 1997.

[7] A. Levy, A. Rajaraman, J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. *Proc. VLDB Conference*, 1996.

[8] K. Morris, J. Ullman, A. Gelder. Design Overview of the NAIL! System. *Proc. ICLP*, 1986.

[9] Y. Papakonstantinou, A. Gupta, L. Haas. Capabilities-based Query Rewriting in Mediator Systems. *Proc. PDIS Conference*, 1996.

[10] A. Rajaraman, Y. Sagiv, J. Ullman. Answering Queries Using Templates With Binding Patterns. *Proc. PODS Conference*, 1995.

[11] A. Tomasic, L. Raschid, P. Valduriez. Dealing with Discrepancies in Wrapper Functionality. *Proc. ICDCS Conference*, 1996.

[12] R. Yerneni, et al. Computing Capabilities of Mediators. *Proc. SIGMOD Conference*, 1999.

[13] R. Yerneni, et al. Optimizing Large Join Queries in Mediation Systems. *Proc. ICDT Conference*, 1999.

[14] `http://www.amazon.com/exec/obidos/ats-query-page` Search Form for Amazon.com Bookstore.

[15] `http://shop.barnesandnoble.com/booksearch/search.asp` Search Form for BarnesAndNoble.com Bookstore.

[16] `http://shopping.yahoo.com/books` Yahoo Shopping Guide for Books.