AUTOMATED CREATION OF DATABASE FORMS

by

Magesh Jayapandian

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

ACKNOWLEDGEMENTS

I am forever in debt to my advisor, Prof. H. V. Jagadish for making this thesis even possible. His belief in me often far exceeded my belief in myself. His intelligence, creativity, steadfastness, understanding, mentorship, leadership, support, encouragement, work-ethic, availability, humor and compassion went a long way in getting me passionate about research and sustaining my effort and self-confidence until this point. He has been the perfect research advisor and I could not have asked for anyone better to lead me through this challenging but ultimately worthwhile journey.

I am also grateful to my colleagues in the database lab who have provided much support and friendship during my graduate career. I thank the department staff for helping me through various procedures during my pursuit of this degree and for always being there to share a laugh and provide some much needed distraction every once in a while.

I am of course, indebted to my family for without them, I would not even be in this part of the world, nor would I have been ready and able to take on the challenges of living, learning and contributing in a foreign country.

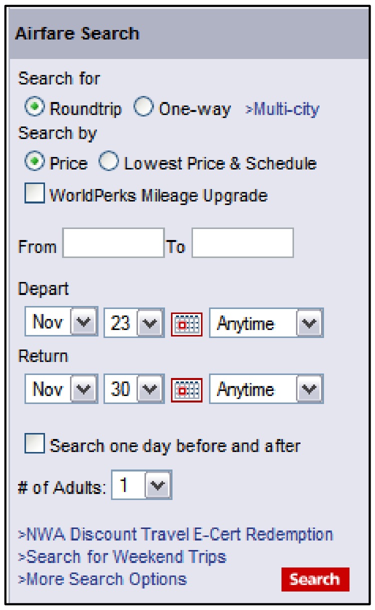
TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A form-based query interface is usually the preferred means to provide an unsophisticated user access to a database. Not only is such an interface easy to use, requiring no technical training, but it also requires little or no knowledge of how the data is structured in the database. Consider the form shown in Fig. 1.1 (taken from NorthwestAirlines.com).



Airfare Search

Search for: [✓ Roundtrip] One-way >Multi-city

Search by: [✓ Price] Lowest Price & Schedule

☐ WorldPerks Mileage Upgrade

From: ____ To: ____

Depart: Nov 23 Anytime

Return: Nov 30 Anytime

☐ Search one day before and after

# of Adults: 1

>NWA Discount Travel E-Cert Redemption
>Search for Weekend Trips
>More Search Options
Search

Figure 1.1: A Simple Form

To search for a flight, all a user needs to do is fill out the form specifying the origin, destination and dates of travel. While the underlying query can be fairly complex requiring the joining of multiple tables in the database, the form is intuitive and easy to use. A user can immediately see what it means and how to specify a query using it. No expertise in a query language is needed. Nor does the user need to know how the data is organized in tables in the database. This abstraction is made possible by appropriate translations inside the form. Each form acts as a wrapper for a carefully selected set of queries that an interface developer expects to be of value to the customers of the data service. Underneath any form lies the query and data logic needed to transform a filled form into the corresponding instructions to the database system to execute the user's query and return the desired results back to the user. To be able to develop this logic, interface developers must have knowledge of how the data is organized in the database and how it can be manipulated using the appropriate language constructs. In addition, they must also design the form's external appearance, which includes content and presentation issues critical to the form's intuitiveness.

1.1 The Problem

While forms are one of the simplest querying mechanisms to use, they have drawbacks. Firstly, forms are highly restrictive, i.e., they only support a precise set of queries for which they have been designed and disallow all others. Interfaces sometimes have multiple forms to allow users to ask different types of queries. If a user wishes to ask a query that is not allowed by the forms provided, he or she may leave unsatisfied. Secondly, the form developer is required to have complete knowledge of the database schema (which can be very complex) and also have expertise in a system-dependent query language in order to create useful forms. Thirdly, generating forms is often a repetitive, tedious and largely manual task. A forms-based interface often requires several forms to satisfy the diverse querying needs of the targeted group of users. Since a developer can only make educated guesses of what users of the database will be interested in asking it, the creation of a good set of forms is a black art. Interfaces might have to go through multiple design iterations, with or without feedback, before achieving the desired level of user satisfaction.

1.2 Our Contributions

Our work in this thesis attempts to address these shortcomings of forms-based interfaces without compromising their simplicity and ease of use. Further, we attempt to make this task almost automatic, while still effective. Anyone who sets up a database should be able to “push a button” to create a good forms-based interface for users of that database. While we do not realize this vision in its entirety in this thesis, we do come close, and address all the major issues.

We first make use of schema information and, if available, the content of the database itself to identify query foci and build forms centered at these entities. Choosing the entities, attributes and relationships of most interest to users as well as the types of queries on each of them to support via forms is the major challenge of this interface design approach. We do so by analyzing the database – its schema as well as its content to identify zones of potential interest and then generate a set of forms that highlight these parts of the data and support as many and as diverse queries as possible to these areas. Without any knowledge of querying needs, the schema and content of the database provide a reasonable staring point to estimate these needs. For instance, a movie database is likely to have movies as the most important and highest populated entity. It should be no surprise that users accessing such a database will most likely look for information about movies (as opposed to information about producers, distributors, directors, writers, actors, etc. which are important but less frequently queried). Hence a form that focuses on movie-related information is likely to be used extensively and satisfy a significant fraction of user queries. Of course, this is a very simplistic example. Another example of a fairly simple schema is shown in Fig. 1.2 which is taken from the XMark benchmark [15, 71]. Here, one can see that the primary entities are person, open_auction, closed_auction and item. Intuitively these are the entities of most interest to users whose queries may involve one or more of those entities. Most current databases, however, have much more complex schemas with numerous entities, attributes and relationships. Many of these databases

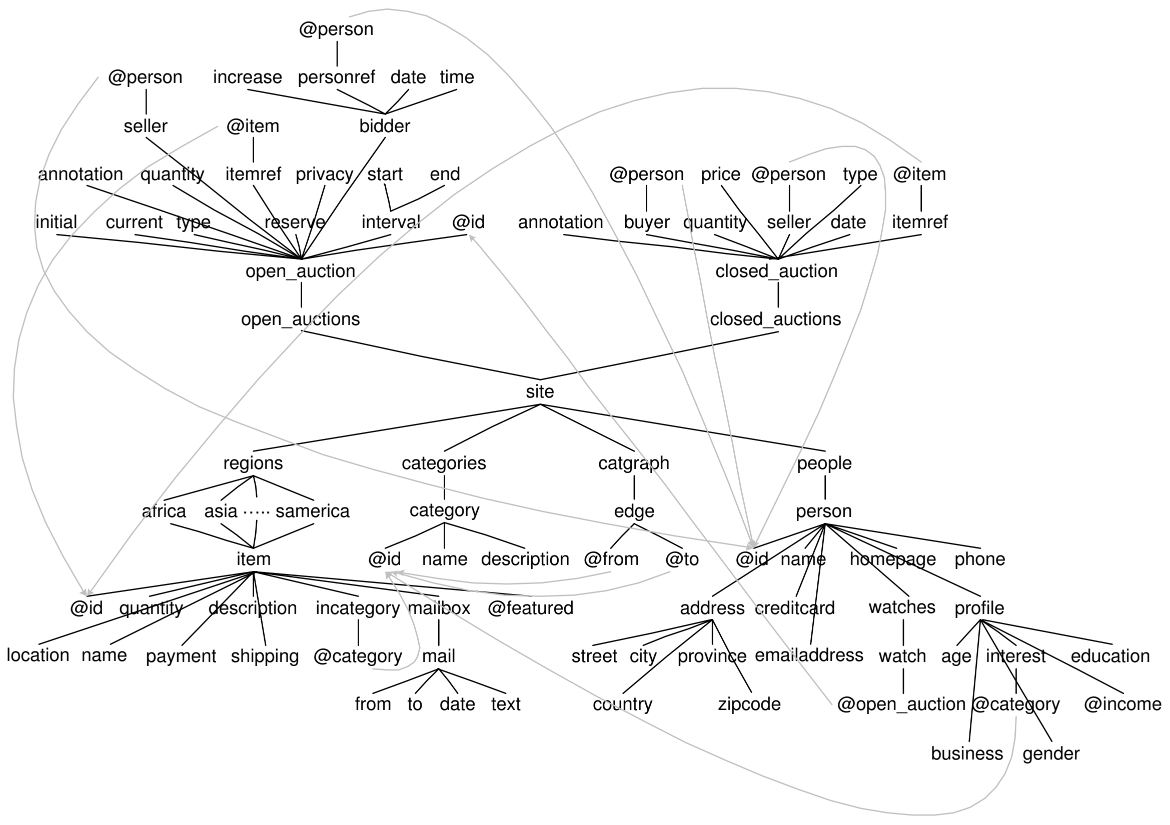


Figure 1.2: XMark Schema

are normalized to a large extent which increases the number of entities needed to capture the data of interest. In most cases the schema complexity is simply due to the richness of the data. This complexity is reflected in the queries to the database, many of which will involve more than one entity of interest. The larger the schema, the harder it is to discern which entities, attributes and relationships are the most important and of most interest to querying users. In Chapter IV, we describe our approach to determine exactly that. Through experiments, we have observed that the forms we generate for a given database satisfy a reasonable fraction of user queries. This fraction can be improved by making use of other inputs available to the system such as query log analysis and domain knowledge.

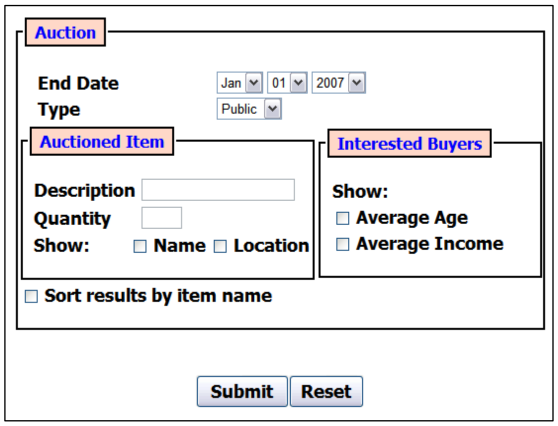
If some queries posed by users to a database are known, these are a direct indication of what questions users would like to ask the database, and if forms are provided, these are the very queries those forms should allow users to express. If form-design can be query-driven, these can be expected to do better than just using the schema and the data distribution. If these queries can drive an automatic form-generation process, we can overcome the two main challenges of creating a forms-based interface, namely user satisfaction and minimality of developer effort. However, queries specified declaratively and the forms that support them are fundamentally different – query languages were developed to specify to a database engine how a query should be evaluated while forms convey to a human user how to specify a desired query. In other words, the latter are meant to be human-readable while the former must more importantly be machine-readable. There exists a semantic gap between these two querying mechanisms. Given a query in a declarative language, it is not always obvious how to create a corresponding form that allows it to be posed while also being meaningful to a casual database user. For example, consider the following query to the XMark [15, 71] dataset (schema shown in Fig. 1.2):

Q: *Display the names and locations of all one-of-a-kind sports memorabilia on public auction ending today and the average age and income of people who are interested in them, sorted by item name.*

The XQuery expression for this query is:

```
for $o in document("auction.xml")//open_auction
for $i in document("auction.xml")//item
let $p := for $pi in document("auction.xml")//person
              where $pi//interest/@category=$i//@category
                return $pi
where $i/description = "sports memorabilia" and $i/quantity = 1
    and $o/privacy = "public" and $o/interval/end = today()
and $o/itemref = $i/@id
order by $i/name
return {$i/name} {$i/location}
        {AVERAGE($p//age)} {AVERAGE($p//income)}
```

Although the selection predicates and projected attributes are constant and dictated by the query, they do not directly specify the structural relationships between the attributes involved, nor do they suggest how they can be presented to a user in a meaningful way. It is easy for a person conversant with this simple query and the underlying data structure to design a form to express it, and to map the input fields to the appropriate query predicates. One such form is shown in Fig. 1.3. While this form does the job very nicely for the query at hand, it is not extensible and cannot support even simple variations of the query such as displaying the current price of the item or the names of potential buyers. As queries and schema become more complex, manual form generation becomes harder as well.



End Date Jan 01 2007
Type Public

Figure 1.3: A Manually-generated Form

In Chapter V, we present a mechanism we have developed to automatically create a form using a single declarative query such that the form can then be used to express that query as well as a set of queries similar to it. More importantly, we extend this technique to generate a form for multiple queries that are similar to each other. Ultimately, given a trace history of queries to a database we can then generate a set of forms that can be used by subsequent users to express any query in that entire set of queries. However, it is both inefficient and cumbersome to create one form per unique query in the set. Instead, our mechanism clusters the queries by structural similarity and generates only one form per cluster which can then be used to express any query in that cluster. The challenges in this effort include minimizing the number of forms generated while simultaneously ensuring that no single form is so complex or confusing that it overwhelms or deters users.

Sometimes, in spite of the best efforts of the interface developer or an automated form generator like ours, a user might have a query that still cannot be expressed using the available forms. In fact, it is unreasonable to expect the interface developer to be clairvoyant of every single user query. Moreover, the more query types a form supports, the more difficult it becomes for users to comprehend and use. This makes interface generators hesitant about adding support for queries that are of little interest to the majority of users. Simplicity and expressivity are conflicting goals for any query interface and trade-offs must often be made. However, sometimes even highly complex query forms can be inadequate to some users. To illustrate, consider a database of flight listings at the website of a popular airline, Northwest Airlines. We have already seen the quick search form provided at this website to book a flight (Fig. 1.1). But consider the more advanced form (Fig. 1.4) it also provides which allows more query conditions to be specified. In spite of the added complexity, the advanced search form, which can still only express conjunctive selection queries, might be insufficient for some users.

For instance, consider the following queries to this airline database:

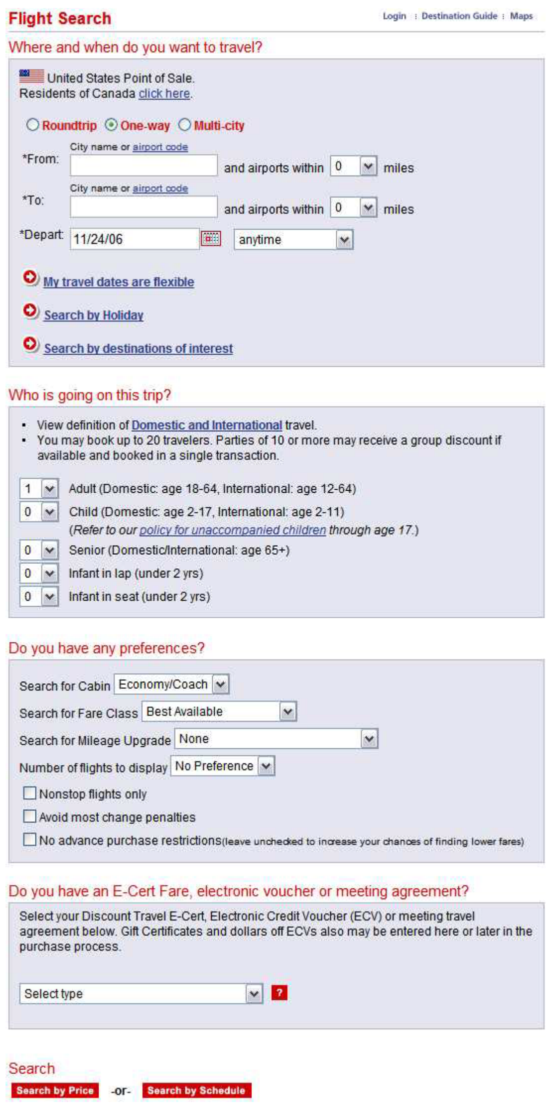
Q1. Display flights from Memphis to Chicago ordered by travel time, shortest to longest.

Q2. Show me flights to an airport in or around Austin (within 30 miles) that depart in the next three days.

Q3. List all such flights that cost less than $400.

None of these queries can be expressed precisely using this form. It only allows result sorting by price and departure times and nothing else. While nearby city alternatives and date ranges are supported, the two cannot be used simultaneously. Finally, the interface does not allow users to specify an upper bound on the price of the ticket.

These drawbacks lead us to ask the following questions: How can one retain the simplicity of form-filling as the querying mechanism yet provide high expressive power so that queries such as the ones above can be specified by the user as desired? Secondly, how can we tune the complexity of a form to be no less than what any user needs and



Flight Search Login | Destination Guide | Maps

Where and when do you want to travel?

United States Point of Sale.
Residents of Canada click here.

Roundtrip [✓ One-way] Multi-city

*From: City name or airport code ______ and airports within 0 miles

*To: City name or airport code ______ and airports within 0 miles

*Depart: 11/24/06 anytime

My travel dates are flexible

Search by Holiday

Search by destinations of interest

Who is going on this trip?

- View definition of Domestic and International travel.
- You may book up to 20 travelers. Parties of 10 or more may receive a group discount if available and booked in a single transaction.

1	Adult (Domestic: age 18-64, International: age 12-64)
0	Child (Domestic: age 2-17, International: age 2-11) *(Refer to our policy for unaccompanied children through age 17.)*
0	Senior (Domestic/International: age 65+)
0	Infant in lap (under 2 yrs)
0	Infant in seat (under 2 yrs)

Do you have any preferences?

Search for Cabin: Economy/Coach
Search for Fare Class: Best Available
Search for Mileage Upgrade: None
Number of flights to display: No Preference

☐ Nonstop flights only
☐ Avoid most change penalties
☐ No advance purchase restrictions(leave unchecked to increase your chances of finding lower fares)

Do you have an E-Cert Fare, electronic voucher or meeting agreement?

Select your Discount Travel E-Cert, Electronic Credit Voucher (ECV) or meeting travel agreement below. Gift Certificates and dollars off ECVs also may be entered here or later in the purchase process.

Select type ?

Search
Search by Price -or- Search by Schedule

Figure 1.4: An Advanced Form

no more than what he or she is comfortable with, without having to resort to extremes (Fig. 1.1 and Fig. 1.4)? In Chapter VI, we propose to address these challenges by means of form customization. The user starts with a form that approximates the desired query and then modifies it to obtain a customized form that precisely captures the desired query. For this scheme to make sense, of course, it is necessary that form modification not require any special knowledge on the part of the user, whether in formal query specification or in database schema. Our proposal is that form modification itself be carried out as a form-filling process. Specifically, we develop a form manipulation language that comprises a small set of operators each of which takes one or more forms as input, and produces a single form as output. The desired form edit can then be written as an expression in this language. We have developed a form editor that implements this language. If we are able to provide a form-filling interface that supports construction of such form expressions, then we have the desired form customization process. Unfortunately, the form manipulation language itself is so highly expressive that it is hard to see how one could create a single form to specify any expression of choice. Instead, we propose that the expression be constructed iteratively. Simple form-based mechanisms can be used to specify the next element in the expression one at a time. After each such step the user sees a new query form, which is a modification of the original, and is hopefully closer to the desired goal.

Our study of form creation would be incomplete if we only focused on form structure and form function. We also need to study form usability, i.e., how easy it is for a user to express a query using a form that is theoretically capable of expressing that query. In Chapter VII, we look at the limitations of auto-generated forms (when compared to forms created by hand) and various ways to make these forms easier to use. Issues affecting form usability range from presentation aspects such as form-element grouping and layout, availability of helper text, etc. to form-filling assistance that lets a user browse valid attribute values and select one of them rather than having the user type in the dark with no feedback until the form is submitted. We explore these avenues of form usability enhancement and devise techniques to implement them.

CHAPTER II

RELATED WORK

User interfaces, particularly database query interfaces are almost always created with significant input from a human domain expert. Although several tools and technologies have been developed over the years to simplify the design process, these interfaces still require human effort, be it on the developer's side or in some cases, on the part of end-users.

In this chapter, we describe some of these efforts and highlight the similarities and differences in methodology and philosophy with our own work on goals for automated form generation. We believe that generating forms automatically is of tremendous use to database creators who may not possess the skills and experience needed to design useful and usable interfaces and who may not want to incur the expense of hiring such an expert.

In Sec. 2.1, we describe related work involving semi-automatic tools that enable interface creation and query specification. While the types of queries that users can ask are dictated by the capabilities of the interface, some research efforts have attempted to increase the expressive power of an interface by increasing the burden of query expression on users. We describe one such class of tools, graphical query languages, in Sec. 2.2.

One of our proposed approaches to automated form generation (which we describe in more detail in Chapter V), is to analyze a given query workload and create forms to support all the queries in that workload. Such a technique requires the ability to identify similarities across queries to minimize the number of forms needed. Other research efforts have had a similar need, i.e. identifying query similarity (albeit for a different purpose), that we cite in Sec. 2.3.

Stepping away from query interfaces for a moment, we surveyed the work done in automatic interface generation in web frameworks. While these can also include query specification capabilities, including query forms, to allow users to query back-end databases, their focus is on website design. We introduce some of the technologies available in this domain in Sec. 2.4. While all of the work we listed thus far can serve to assist interface developers, they cannot operate without a human with with domain expertise and decision-making capabilities. Our goal is to generate good interfaces even without such a person who may not always be available or affordable.

Finally, there is a significant aspect of our work that pertains to human-computer interaction. Although HCI is an entirely separate domain of research, we examined the work done by the HCI community in interface design as it relates to our own work and we cite relevant endeavors in Sec. 2.5.

2.1 Form Builders

While we have not seen much work on automatic or even semi-automatic form construction, there have been several efforts to simplify manual form design. The key difference between these and our approach to interface design is the human element. We attempt to minimize, if not completely eliminate, the need for significant time and expertise of a domain expert. Non-automatic and semi-automatic interface design tools, on the other hand, simplify the task but are still useless without the decision-making capabilities and efforts of a human expert.

Even with standard form building tools like Cold Fusion [34, 4], Visual Basic [39], CGI [38], ASP [33], JSP [40], etc. having come a long way, the task of form creation is still largely manual. An exception is [42] in which forms are generated dynamically based on metadata embedded in the relational tables. This system however, only creates survey forms whose purpose is data-entry rather than data-retrieval. It essentially supports only a single insertion query rather than a wide range of retrieval queries.

Early work on form-based querying mechanisms include [27, 28, 31] which provided users visual tools to frame their queries as well as to perform other operations such as database design and view definition. The GRIDS system [70] generated forms that allowed users to pose queries in a semi-IR, semi-declarative fashion. In the relational world, the Acuity Project [75] developed form generation techniques for data-entry operations such as updating tables in a relational database. An important difference between this project and ours is that we target data-retrieval queries while their aim is to generate data-entry forms which target data acquisition (which is a significantly simpler problem since the "queries" have almost a fixed pre-determined format).

In querying XML data, much work has been done to shield users from details of the XQuery syntax, as well as from the textual representation of XML. FoXQ [16] is a system that helps users build queries incrementally by navigating through layers of forms, and view results in the same way. EquiX [29] is another such language that helps users build queries in steps using a form. Semi-automatic form generation tools have previously been presented in QURSED [63, 67, 65]. These provide form-developers visual tools to generate forms manually, but the task is significantly simpler than traditional form-building tools. This editor uses the schema of the database as a starting point (like us) but it is up to the developer to select the desired schema elements and attributes to use on the form (see Fig. 2.1). It expects developers to annotate the schema beforehand so that every time an element is selected for use, the right set of form-controls and the right query-fragment (for translation to declarative queries) are automatically put in place. Another effort in UI design is DRIVE [55], a runtime user-interface development environment for object-oriented databases. This system used the NOODL data model which enables context-sensitive interface editing. [77] proposes the use of XML to represent forms rather than HTML making forms more reusable, scalable, machine-readable and easier to standardize. Our system too has a textual human-readable form representation, but we still render forms in HTML/JavaScript. However, the use of an internal form representation that is different from its external representation is an important similarity. We describe our own model for form representation in Chapter III which although is not XML, is still hierarchical. Their work however, like Acuity, targets data-entry forms rather than data-retrieval forms which are significantly more difficult to design.

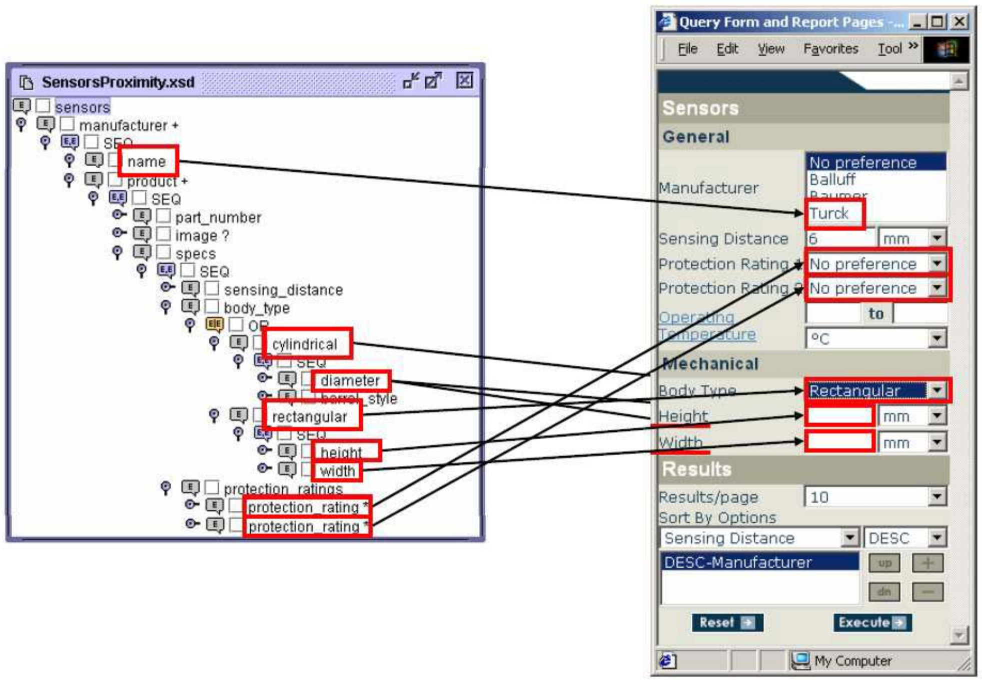
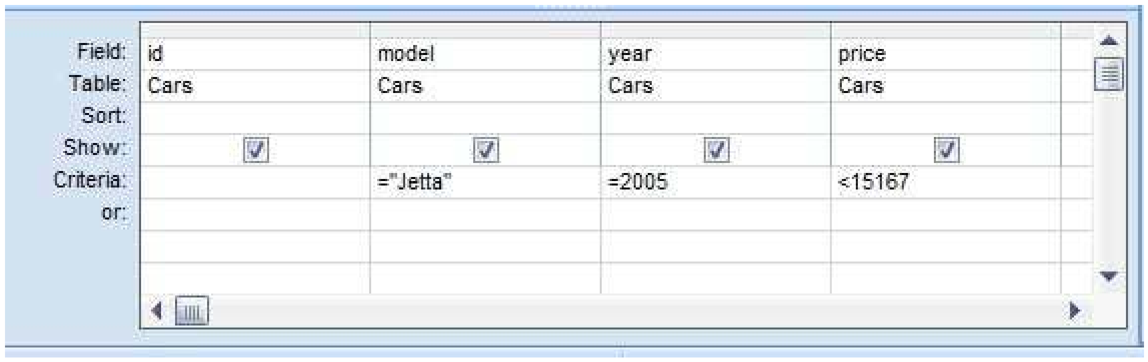


Figure 2.1: QURSED Form Editor

2.2 Graphical Query Languages

There have been several endeavors in non-form-based visual querying mechanisms that simply the task of query specification at the end-user level. These forms of query expression make interfaces more generic but place an extra burden on users. Unlike form builders which we discussed in the previous section, these approaches do limit the need for domain expertise in an interface developer, but this need is simply transferred to the end-users of the database. Specifically, users of these interfaces need to understand how the data is structured and where to look for the information they desire. Additionally many of these tools have a learning curve that users must get over to be comfortable using them. These hurdles are not present in form-based querying. Form-based query interfaces do not expect any schema knowledge or querying language expertise to pose queries. Hence automated form generation attempts to build query interfaces that are easier to use than graphical query languages and close to them in terms of expressive power.

Visual querying began as early as in 1975 with QBE [81]. Since then, there have been several graphical query languages designed to make declarative querying more intuitive. Some examples of these graphical query languages include XQBE [17, 20], BBQ [57], MIX [56], Xing [32], XML-GL [25], VISIONARY [18], Kaleidoquery [58], QBT (Query By Templates) [72], Marmotta [22] and GRIP [41]. The SEWASIE project includes an intelligent query interface [23] that uses ontology to overcome the vocabulary ignorance of users. Microsoft Access offers QBE-based tools to allow database users to construct and issue queries to a relational database (see Fig. 2.2). IBM has its own query builder called the Visual XQuery Builder distributed with its Developer Workbench. All of these tools are different from traditional forms where the mapping from human-readable queries (form exterior) to declarative language expressions (form interior) is done by the developer rather than the user. Also, for the sake of completeness, tools based on graphical query languages need to expose the entire database schema to the user. Forms, on the other hand, only contain a small fragment of the schema at any one time. Automated form generation tools need to be able to identify which fragment of the schema users would most like to query since such decisions cannot otherwise be made in the absence of a human expert.



Field:	id	model	year	price
Table:	Cars	Cars	Cars	Cars
Sort:				
Show:	✓	✓	✓	✓
Criteria:		="Jetta"	=2005	<15167
or:				

Figure 2.2: Query By Example (Microsoft Access)

2.3 Form and Query Analysis

To generate and manipulate, query forms, we need to model them appropriately and use this model to identify form components. To prevent redundancy if the forms generated, we must be able to compare one form with another and identify similarities and differences. A measure for inter-form dissimilarity is useful in this regard. Since forms ultimately express queries, and queries can be used to design forms, the same needs apply to queries as well. A metric to identify inter-query distance can help limit form redundancy, and thus, interface size and complexity.

There are other research areas, such as query optimization, where there is a need to compare query expressions and identify common fragments. Measuring structural similarity between queries is a problem that has been explored in recent years. In the relational context, [37] analyzed queries written in SQL for a purpose somewhat different from ours, i.e., visual query analysis. They define the structure of a query and propose a distance metric to measure pair-wise query similarity. They make use of graph visualization techniques to display the similarity across a given collection of declarative queries to simplify analysis of such a collection. This can be helpful to database administrators in deciding what views and indexes to build to maximize the performance of their database. Similar work has also been done for XPath queries in [54], which also focuses on query-optimization as opposed to our motivation of generating forms. In this work, the authors focus on identifying query containment as well as query equivalence.

In the area of data integration, systems such as MetaQuerier [26, 80] analyze forms from different data sources and build integrated query interfaces to provide a single query point for users. They also enable a query on one form S (based on one schema) to be issued to another form T (which can be based on an entirely different schema). Identifying the similarities between the two forms is a critical step in this process. An example of such a query translation across forms is shown in Fig. 2.3. Using our automatic form generator all forms created conform to a single structural model although they can correspond to any given database schema. This makes it easy to identifying structural similarity between forms but not necessarily semantic similarity.

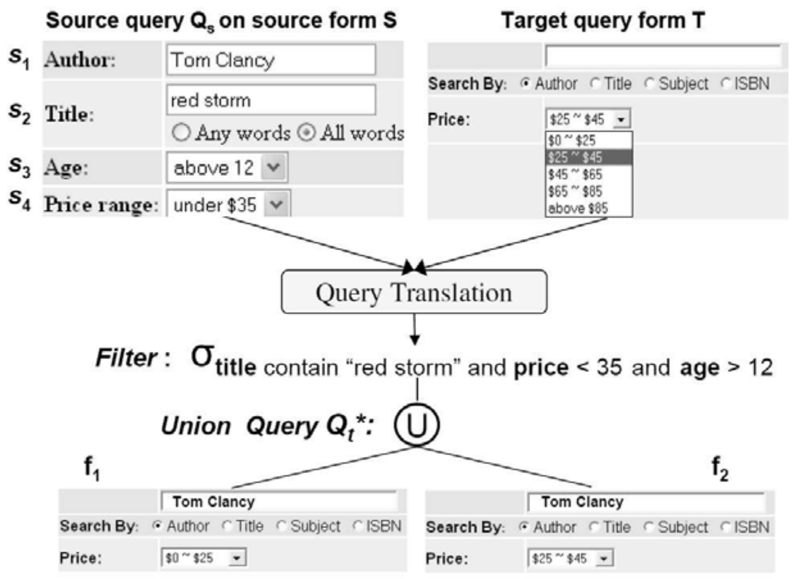


Figure 2.3: Query Translation from one Form to another

2.4 Web UI Design Frameworks

There is an increasing number of new websites being created around the world for various purposes. To simplify the process of developing complex data-backed websites, tools such as Ruby on Rails [11], Django [5], TurboGears [13], WebObjects [14] and Struts [2], are available which are all based on the MVC (Model-View-Controller) paradigm. The benefit of these tools is that they automate many of the tasks that are common across most of these websites. One of these tasks is database access all of these tools enable easy access to back-end databases both for update and for search.

Django, in particular, has a data-browse feature that dynamically creates a rich, browsable Website by introspecting models created by the data owner. Such a website is useful if end-users just want to browse the underlying database rather than ask specific queries. However, this and other web frameworks only provide support for simple queries via the interfaces that they create. Our work aims at creating such interfaces (even without any available models) that allow users to ask specific and reasonably complex queries of interest.

2.5 Human-Computer Interaction

Although not specific to databases, there has been a lot of work in the HCI community in the area of automatic interface creation, computer-aided UI design, model-based interfaces and automated UI evaluation, each of which we touch upon in this thesis.

2.5.1 Automated Interface Generation

XWeb [51], for instance, is a user interface architecture and system that automatically creates platform-independent interfaces to services (including those that access a database) for a variety of different hardware and software platforms. It allows these clients to read from as well as write to the databases on the server. In [61], Nichols et al describe an automated technique of creating remote control interfaces to devices that are consistent with one another. When multiple devices need to be used together, [62] shows how their individual user interfaces can be combined automatically to build aggregate or flow-based user interfaces which can be difficult even for human interface designers to create.

While human generated interfaces tend to be better, the performance of auto generated interfaces can be improved by the user of *smart templates* that model device behavior in a standard way [60]. As with database query forms, different human-created interfaces adopt different conventions which requires a user who is familiar with one of them to become familiar with a different one even if the task is the same. Automated interface generators, like automated form generators, use consistency to minimize the learning curve for new users [59]. SUPPLE [35] is a popular automatic interface generator that automatically generates layouts for user interfaces using an optimization approach to choose controls and their arrangement.

2.5.2 Model-based Interfaces

An interface model represents all the relevant aspects of a user interface in some type of interface modeling language. Model-based systems provide the software tools that build and refine the interface model to produce a user interface [69, 68]. No model is universally applicable and sometimes mappings in user interface design also need to be generated. This is analogous to a form not being universally applicable to query a database although it is difficult to create a mapping from a form to the set of queries it allows to be expressed. But enabling a user to find the right form for his or her query is critical for the form-based interface to be successful.

2.5.3 Usability Evaluation

The usefulness of an interface ultimately depends on the experience it allows end users to have. Hence it is imperative that user interfaces, both human and machine generated be evaluated with the help of real users (test subjects). However, this can be extremely expensive and time consuming to carry out. We present a cost model to evaluate the usability of automatically generated forms in Chapter VII. In the HCI community as well, work has been done to enable analytical evaluation of user interfaces. A survey of these methods can be found in [43]. The two main techniques are: automated capture of use data, and automated analysis of these data according to some metrics or a model.

CHAPTER III

FORM MODEL

We begin by presenting the structure and function of the forms we generate automatically. We describe the various form components, how they are presented to the user and how they communicate the query to the underlying database. In Sec. 3.1, we describe a canonical form representation that defines the structure of our forms. Then, in Sec. 3.2 we show how given a filled form, we can generate an equivalent declarative query expression which can then be executed by a database system.

3.1 Form Definition

A form is a visual representation of a set of user-specifiable query operations laid out in a meaningful way. Forms allow users to place conditions on certain attribute values, or to choose which attributes of the query's result are to be displayed. These user-specifications are then translated into corresponding query operations to be executed by the database. While these atomic data manipulation operations can be combined in several ways, the arrangement of their visual representations on the form dictates the manner of their composition. A form's layout is crucial to the user's understanding of the semantics of the query, what each operation means and how they relate to one another. The visual representation of an operation can be thought of as a *form-element* which can be represented using *form-controls* such as textboxes, checkboxes, drop-down menus and radio buttons. Form-elements also have a logical representation that corresponds with their visual representation both of which we describe in this section. The layout of these form-elements involves organizing them into collections spatially (on the form) and appropriately labeling them to tell users what they represent. We term these collections as *form-groups*. Form-groups may nest so that smaller groups combine, possibly with additional form-elements, to form larger form-groups. Thus a form-element is just a set of form-controls and a form-group is simply a set of form-elements and smaller form-groups. Typically the construction of elements and groups and their subsequent arrangement and labeling are done manually by the form developer as a one-time exercise.

3.1.1 Form Elements

A form as a whole issues a query to the underlying database. Each element in the form, a *form-element*, corresponds to a single data manipulation operation in that query. The form-element also contains references to one or more schema entities (or attributes) that become the operands of that operation. We present a few common types of form-elements and the data manipulation operations they perform. This is not an exhaustive list. It is merely a reasonable set of data manipulation operators (or equivalently, form-element types) to work with that we have used in our implementation. The expressive power of a form is directly impacted by the set of data manipulation operators chosen as the basic form-elements. This set of operators is sufficient to express relational algebra with aggregates, but without set or arithmetic operations. It is also enough to express XQuery core excluding output structuring, quantification, set and arithmetic operations. For concreteness, we will use only this set of operators in all our examples. However, by choosing an appropriately rich set of form-elements, one can generate arbitrarily expressive forms.

Constraint Specification

A constraint-specification operator c denotes a selection predicate on an entity set E which can either be a set of tuples of a relation or a set of identically structured elements in XML data. The form-element is represented as $c(E : n)$ where n is the attribute (of E) whose values are used to perform the selection. A selection condition also needs a relational operator and a data value but these are not included in the form-element representation since they are user-specified and thus undefined at form creation time. Multiple such operators can be combined conjunctively to construct more complex query predicates.

Result Display

A result-display operator corresponds to a query projection and is represented as $r(E : n)$, where E is the entity set operated on and n is the name of the projected attribute (of E). If selected by the user, this attribute is displayed in the results of the query.

Aggregate Computation

An aggregate-computation operator operates on two schema entity-sets/attributes and is represented as $a(E_n : n, E_b : b)$. Here E_n and n together denote the entity-set/attribute being aggregated, and E_b and b together reference the entity-set/attribute that forms the scope (grouping basis) of the aggregation. There are two types of aggregate-computation operators (based on what action is taken on the aggregated result): an *aggregate-constraint-specification operator*, that places a selection condition on the aggregated value, and an *aggregate-result-display operator*, that projects out the aggregate values and displays them in the result.

Result Ordering

A result-ordering operator operates on a single entity set and specifies a sort order for the results of the query. It is expressed as $s(E : n, o)$ where E is the entity-set whose tuples are sorted, n is the attribute to sort by, and o is the sort order (*ascending* or *descending*). A query's result can be sorted using one or more such operators in sequence.

Disjunction

A disjunction operator operates on a single entity set and is a combination of two or more selection predicates (at least one of which needs to be satisfied by a tuple for it to occur in the result). This operator can be expressed as $d(c_1, c_2, ..., c_n)$ where c_i is a constraint-specification operator.

Join Specification

A join-specification operator operates on two entity sets (or the same entity set with itself). A form containing this operator issues a join query to the underlying database. The operator is expressed as $j(E_1 : n_1, E_2 : n_2)$, where n_1 and n_2 correspond to the attributes in the join condition (between the joined entities E_1 and E_2). The relationship itself (equijoin or non-equijoin) is user-specifiable.

3.1.2 Element Organization

Elements in a form are grouped hierarchically and the placement of elements is governed largely by the way they are grouped. Grouping also helps resolve ambiguities between like-named elements that may co-occur in a single query. On the form, related form-elements are juxtaposed to the maximum extent possible (by putting them in the same group, for instance) to aid users in understanding what they mean in the context of the query. Form-elements can be related to one another in one of two ways that necessitate grouping:

- They reference attributes of the same entity (*schema-based grouping*)
- The query containing them specifies a relationship between their referenced entities (*query-based grouping*)

A labeled rectangular box is created around each group to enhance clarity. Grouping is recursive. In other words, form-groups may comprise form-elements or other form-groups.

3.1.3 Canonical Structure

Each field on a form has a specific role with respect to the query the form evaluates. In most cases, a field is used to specify a search criterion, making it an *input* to the underlying query. If a field is used to specify the content or display order of the query's result, it manipulates the *output* of the query. It is useful to think of form specifications as having three main components: inputs, outputs and relationships. The first two are completely independent of each other. Relationships, by definition, relate the entities associated with one or more (input or output) form-elements. We find it convenient to separate these three components of a form. In fact, we recommend doing so in a user visible manner, dividing a form into three distinct panes. We label them *Criteria*, *Output* and *Advanced* respectively. Each component can be represented as a tree owing to the hierarchical nature of grouping. We name these components the *input tree*, *output tree* and *relationship tree*, respectively. We say such forms are *canonically structured*.

Definition 1. (CANONICAL FORM STRUCTURE) *The* **canonical structure** *of a form is a 3-tuple consisting of three logical trees, each containing one or more form-elements and form-groups. Formally,* $F = \langle IT, OT, RT \rangle$*, where:* IT *denotes the* **input tree**, OT *denotes the* **output tree** *and* RT *denotes the* **relationship tree**.

Next we introduce each of these form trees — how they are structured, what their semantics are and finally, their functions, both with respect to a user and to the database system.

Definition 2. (INPUT TREE) *The* **input tree** *of a form is a tree,* $IT = \langle \xi_i, \Gamma_i, \varepsilon_i \rangle$*,where:*
– ξ_i *is a finite set of form-elements that will serve as inputs to the query of interest;*
– Γ_i *is a finite set of form-groups;*
– $\varepsilon_i = \{(\gamma, g) | (\gamma \in (\xi_i \cup \Gamma_i)) \wedge (g \in \Gamma_i) \wedge (\gamma \notin A_i(g))\}$ *where* $A_i(g)$ *is the set of ancestor groups of* g *in* IT.

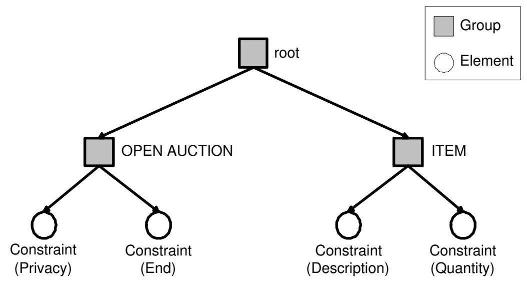


Figure 3.1: Canonical Structure of a Form (Input Tree)

In this tree, form-elements are the leaves and groups make up the internal nodes. Input form-elements include constraint-specification and aggregate-constraint elements whose parameter values can be viewed as *inputs* from a user for query evaluation. An example is shown in Fig. 3.1, which has multiple constraint-specification elements grouped by their respective schema-entities: open_auction and item. It is displayed in the first pane (called the "Criteria" pane) of the form (Fig. 3.2).

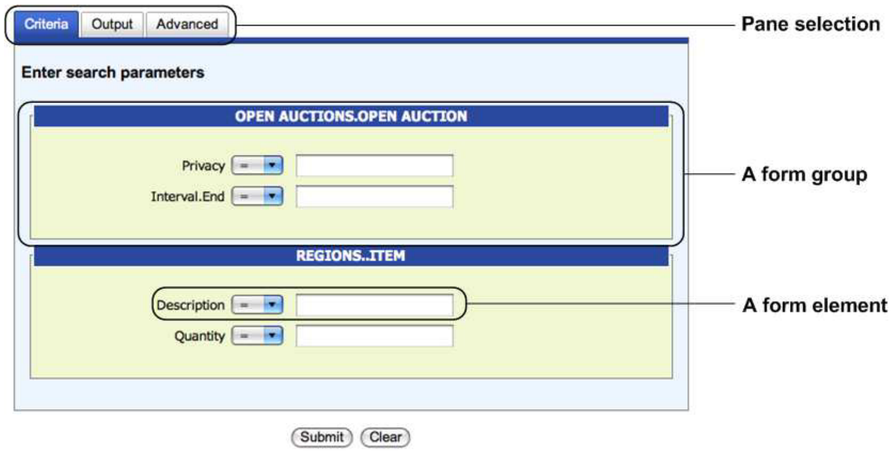


Figure 3.2: Visual Representation of a Form (Criteria Pane)

Definition 3. (OUTPUT TREE) *The* **output tree** *of a form is a tree,* $OT = \langle \xi_o, \Gamma_o, \varepsilon_o \rangle$*,where:*
– ξ_o *is a finite set of form-elements that will control the content and format of the output of a user's query, i.e., its result data;*
– Γ_o *is a finite set of form-groups;*
– $\varepsilon_o = \{(\gamma, g) | (\gamma \in (\xi_o \cup \Gamma_o)) \wedge (g \in \Gamma_o) \wedge (\gamma \notin A_o(g))\}$ *where* $A_o(g)$ *is the set of ancestor groups of* g *in* OT.

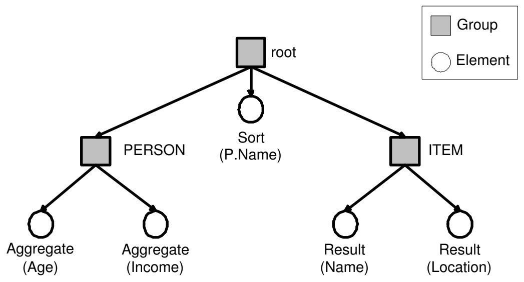


Figure 3.3: Canonical Structure of a Form (Output Tree)

Again, the leaf nodes of the tree correspond to form-elements, while internal nodes represent the groups they belong to. As described in Sec. 3.1.1, result specification operators are of two main types: result-display and result-ordering. These two types have an important difference in the context of output trees. For presentation reasons, we prefer to group display elements by schema entity, but place all ordering elements in a single group globally. If results of a query are sorted on more than one field, the order between these fields is easier for a user to interpret if the fields are placed together on the form (in order of precedence) than if they are grouped by entity (possibly tagged with a precedence value). Hence all result-ordering elements are placed in the root group of the output tree (order-preserved) and ignored during form-element grouping. Fig. 3.3 shows such a tree (for our example form). The second pane of the form, called the "Output" pane (Fig. 3.4), is organized into two groups—one for form-elements that are concerned with persons (labeled 'PEOPLE.PERSON') and another for item-related parameters (labeled 'REGIONS..ITEM').

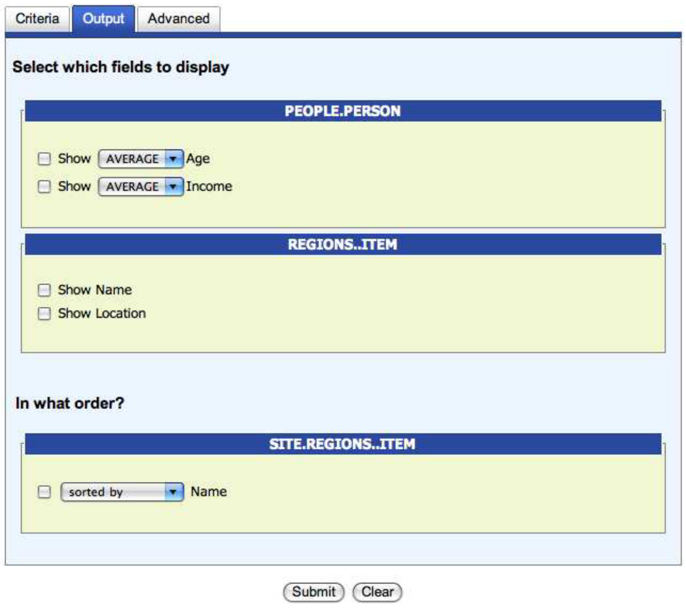


Figure 3.4: Visual Representation of a Form (Output Pane)

There is a third component of our form representation called the relationship tree which is not meant to be viewed by typical end-users. Its primary purpose is to capture and preserve inter-entity relationships for query evaluation purposes.

Definition 4. (RELATIONSHIP TREE) *The* **relationship tree** *of a form is a tree,* $RT = \langle \xi_r, \Gamma_r, \varepsilon_r \rangle$*,where:*

– ξ_r is a finite set of form-elements that are related to one another by the query (including one or more join-specification elements), where:
– Γ_r is a finite set of form-groups;
– $\varepsilon_r = \{(\gamma, g)|(\gamma \in (\xi_r \cup \Gamma_r)) \wedge (g \in \Gamma_r) \wedge (\gamma \notin A_r(g))\}$ where $A_r(g)$ is the set of ancestor groups of g in RT;
– $(\xi_i \bigcup \xi_o) \subset \xi_r$.

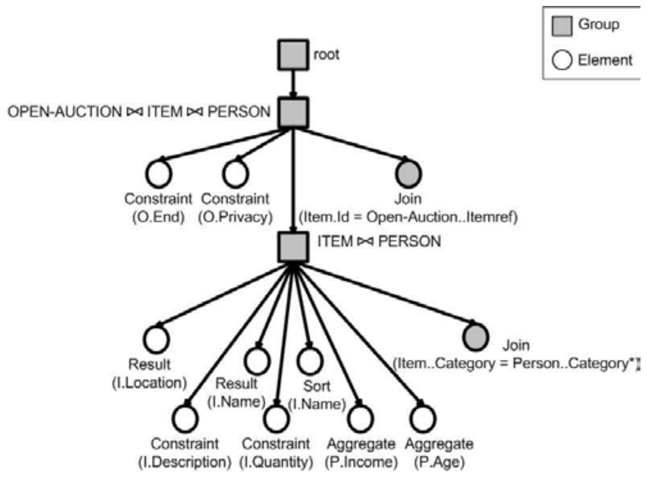
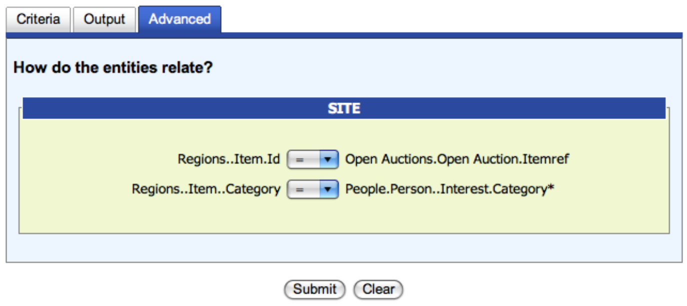


Figure 3.5: Canonical Structure of a Form (Relationship Tree)

Non-leaf nodes in this tree denote groups based on relationships between entities in the schema that are referenced by the form-elements. Sometimes a query defines relationships between two or more fields in the form, as in the case of join queries or queries with nested subqueries. Form inter-relationships are captured in the relationship tree (Fig. 3.5). The relationship tree contains all form-elements used in the form. Apart from join-specification elements, all form-elements present in this tree are also present in either the input or output tree depending on their role in the query. Only the join-specification elements are placed in a pane devoted to inter-entity relationships called the “Advanced” pane (Fig. 3.6). We do so because specification of join conditions requires more technical capability from the user than merely specifying inputs and outputs. In our example form, the third pane contains the relationships between the entities open_auction, item and person. This pane indicates how the entities corresponding to these form-groups relate to one another and how these relationships will translate to joins or nested subqueries in the queries that users will ask the database using this form.



How do the entities relate?

SITE

Regions..Item.Id = Open Auctions.Open Auction.Itemref

Regions..Item..Category = People.Person..Interest.Category*

Figure 3.6: Visual Representation of a Form (Advanced Pane)

3.1.4 Form Notation

We can represent a form textually using curly braces to show grouping. Each form-group is an unordered comma-separated list of form-elements and form-groups present in it within curly braces. For ease of reference, we create a unique numeric identifier for each form-group and attach this as a subscript to its closing curly brace. A form is always denoted by three pairs of curly braces representing the root-level form-group in each of the three form-trees. An empty form is denoted as $F = \{\}_{\mathbf{I}}\{\}_{\mathbf{O}}\{\}_{\mathbf{R}}$, where $\{\}_{\mathbf{I}}$, $\{\}_{\mathbf{O}}$ and $\{\}_{\mathbf{R}}$ denote the input, output and relationship trees respectively. We also use braces to mark the extents of a form-group and use a numeric subscript to uniquely identify it. Form-elements are represented using the notation presented in Sec. 3.1.1. Under this scheme, our example form is expressed as follows.

$$\{\{c(\text{open\_auction:privacy}), c(\text{open\_auction:end})\}_1,$$
$$\{c(\text{item:description}), c(\text{item:quantity})\}_2\}_{\mathbf{I}},$$
$$\{\{r(\text{item:name}), r(\text{item:location})\}_3,$$
$$\{a(\text{person:age}), a(\text{person:income})\}_4, s(\text{item:name})\}_{\mathbf{O}}$$

$$\{\{c(\text{open\_auction:privacy}), c(\text{open\_auction:end}),$$
$$j(\text{open\_auction:itemref}, \text{item:id}),$$
$$\{c(\text{item:description}), c(\text{item:quantity}), r(\text{item:name}),$$
$$r(\text{item:location}), s(\text{item:name}), a(\text{person:income}),$$
$$a(\text{person:age}), j(\text{item:category}, \text{person:category})\}_5\}_6\}_{\mathbf{R}}$$

For the remainder of this thesis, we focus solely on canonically structured forms.

3.2 Query Generation

Once a form has been filled, the desired query needs to be issued to a database engine that can evaluate it. We now discuss how a form generated automatically can interact with a database to evaluate a posed query. Unlike traditional forms, there is no hard-coded query that only needs to be parameterized to be executed. This query has to be generated automatically by our system. In this section, we describe the translation scheme that we use to generate queries from filled forms based upon a systematic conversion of the *relationship* tree (as alluded to earlier) into a query statement. Our system is based on an XML data environment, hence our declarative language of choice is XQuery. We first present a translation scheme for each data manipulation operator defined in Sec. 3.1.1. Then we consider the major classes of user queries and explain how the corresponding form is mapped to a query language expression that can be subsequently evaluated. Our algorithm for query generation is shown in Algorithm 1. To show how declarative expressions are generated, we first classify queries into two categories.

3.2.1 Simple Queries

We define simple queries as those that do not have a join relationship. In simple queries, there is only one group (the root group) and no query-specified relationships between the fields selected. In such cases, the main entity is discovered by finding the lowest common ancestor of the attributes selected and assigning a binding variable to it. This forms the for-clause of the XQuery. All constraint-specification elements are used to generate the where-clause and finally, all result-display elements are mapped to the return-clause. If there is an aggregate-computation element, it provides both the attribute to be grouped as well as the scope of grouping. Instead of the attribute itself, its scope is used to generate the binding variable, and the scope also forms the let-clause in the query expression. The aggregated attribute is now treated like a simple constraint-specification or a result-display and enters either the where-clause or the return-clause of the query.

3.2.2 Complex Queries

Unlike simple queries, forms that generate complex queries such as join queries and queries with nested subqueries have more than one group in their relationship trees. At each group, the join condition is considered and all descendent elements are partitioned into two groups, one corresponding to the left-hand side of the join condition and the other corresponding to its right-hand side. Following this partitioning the lowest common ancestor of all elements within each partition becomes the binding variable and the rest of the elements in that partition become selections, projections or aggregate selections or projections associated with that partition. The join condition is added to the where-clause of the query. If multiple query blocks are created (for nested queries), the nesting condition ties the query blocks together and is added as a condition in the where-clause of the subquery.

As an example of a nested query, consider the filled form in Fig. 3.7. Using this form and our query generation algorithm, the system generates the following XQuery that can be evaluated by an XML database system:

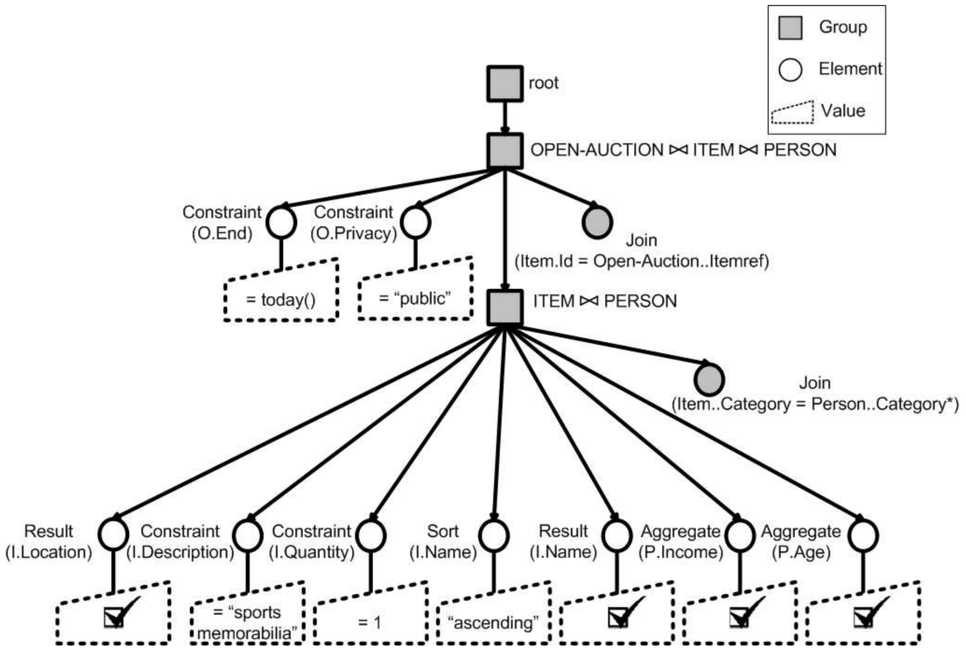


Figure 3.7: A filled form

```
for $o in document(“auction.xml”)//open_auctions/open_auction
for $i in document(“auction.xml”)//regions//item
let $p := for $pi in document(“auction.xml”)//people/person
          where $pi//interest/category=$i//category
          return $pi
where $i/description = “sports memorabilia” and $i/quantity = 1
    and $o/privacy = “public” and $o/interval/end = today()
    and $o/itemref = $i/id
order by $i/name
return {$i/name} {$i/location}
       {AVERAGE($p//age)} {AVERAGE($p//income)}
```

Algorithm 1: Algorithm `GenerateQuery`

```
Input: A filled form $F$
Output: An XQuery expression $X$
// Create binding variables and assign them to form-groups
foreach join-element $j \in T$, the relationship tree of $F$ do
    Create a new binding variable $v_1$ for the entity referenced by the left-hand side of the
    join condition (if it does not already exist);
    Create a binding variable $v_2$ for the entity referenced by the right-hand side of the join
    condition (if it does not already exist);
    Assign $v_1$, $v_2$ to the group containing $j$;
    [Note:  A binding variable assigned to a group may also be
    assigned to other groups in its subtree if the same entity
    is referenced in multiple join conditions.]
    if $j$ denotes a nested relationship then
        Construct a new query block $b$ and record the current query block as its parent;
        Add $b$ to $B$, the set of query blocks;
if no join-element found then
    Create a single binding variable $v$ and assign it to the root group;
// Assign form-elements to binding variables
foreach form-group $g \in T$ do
    Partition the form-elements in $g$ between the two binding variables by schematic
    similarity;
// Determine schema entities to bind variables to
foreach binding variable $v$ do
    Assign $v$ to the lowest common ancestor of schema entities (referenced by elements) that
    are assigned to it;
foreach query block $b \in B$ do
    Assign a unique variable name to $b$ and denote its associated schematic entity as its
    scope;
    Create a for clause using the variable name and scope;
    [Note:  The scope may be relative to another binding
    variable if the query block is nested.]
    Create a predicate for each constraint-specification or join-specification element and add
    it to the where clause;
    Create a projection for each result-display element and add it to the return clause;
    Create an orderby attribute for each result-ordering element and add it to the orderby
    clause;
foreach query block $b \in B$ do
    Construct a let-clause that connects $b$ to its parent block;
Construct the XQuery expression $X$ recursively using a DFS traversal of the query blocks;
```

CHAPTER IV

FORM GENERATION

Creating a forms-based interface for an existing database requires careful analysis of its data content and user requirements. To design structured forms, an interface developer must have a clear understanding of what data is available, its structure and semantics, in addition to predicting user needs. The task of form design, creation and deployment can be very difficult if the schema is large and complex and if the querying needs are diverse. In such scenarios it is worth exploring the extent to which form creation can be automated. Thus our goal is to automate the task of form generation and significantly reduce, if not eliminate, the developer's role in the process.

Our first approach to automated form generation is to make use of what is available to any database, even one that is freshly compiled: its schema and its data content. While a careful analysis of real or expected query workloads are useful in designing the interface, these query sets are often unavailable or hard to obtain prior to the database even being deployed. Hence generating a good set of forms just using the database itself is a challenging yet important problem. In this chapter, we discuss our data analysis approach, the heuristics and metrics used to evaluate query targets, the algorithm to generate the forms that focus on them and finally extensions to improve users' satisfaction with the interface.

The first challenge to address is determining the schema fragment(s) most likely to be of interest to a querying user. Schemas can be extremely complex in real-world databases, but actual queries issued to a database typically focus on a small subset of its schema (as we have observed and believe). In Sec. 4.1, we introduce a metric, called *queriability*, to measure the likelihood of various elements of a schema being queried. We develop techniques to estimate queriability based solely on the database schema and content, and use it to discard schema elements that are less likely to be queried.

The second challenge in automated form design is to partition the filtered collection of schema elements into groups (not necessarily distinct) such that the entities, attributes and relationships present in a single group can meaningfully interrelate on a form to express user queries. A random collection of highly queriable schema elements may not make sense together in a single form. We present our approach in Sec. 4.2. Thresholds on form complexity and form set size are input parameters to this procedure. An interface that has too many forms or numerous parameters per form could overwhelm users. The complexity and size thresholds guard against this.

A very large number of queries can potentially be composed from a given set of related schema elements. Not all of these queries can be supported by a single form. The third challenge in our form generation process is to convert each of these groups of schema elements into a form that a user can employ to express a desired query. We address this challenge in Sec. 4.3.

Since we are predicting likely user queries based solely on schema and data, our techniques cannot have associated analytic guarantees. As such, a careful performance evaluation is essential. We evaluate the performance of our system with the help of real database queries and present our results and observations in Sec. 4.4. Our main result is that, using only the schema and the data, it is possible to create a small set (e.g. < 4) of simple forms (e.g. < 12 fields each) that can express most queries users ask (e.g. $> 80\%$).

4.1 Database Analysis

Any form that can express a query of interest must include one or more entities, attributes of those entities and optionally one or more relationships between the entities. To build a forms-based interface we must select which entities to employ and in what combinations in order to cover all queries of interest to users. This problem may have a deterministic solution if the queries are known a priori. Since that is not typically the case, the best we can do is to use heuristics for entity selection. We define a set of postulates that we use to compute the queriabilities based on observations of the schema and the data. In this section we describe the database analysis we perform to obtain this queriability score. In the next section we show how we can use this score to construct forms.

4.1.1 Schema Analysis

The schema of a database defines its structure. It is a set of entities along with their attributes and the relationships they have with one another. These relationships may be structural links or referential links between the respective entities. A schema is thus a graph whose nodes denote entities and attributes and whose edges represent links between them. An entity in our data model corresponds to an entity set in the ER Model. Our notion of an attribute includes not only simple and multi-valued attributes (as defined in the ER Model), but also complex-typed attributes (which are modeled as entities in the ER Model). Also unlike the ER Model, our data model does not support relationship attributes nor does it distinguish between strong and weak entities. Classes in a class hierarchy are all viewed as separate entities. A formal definition of a schema in our data model is as follows.

Definition 5. *(SCHEMA)* The ***schema*** of a database is a directed graph $\langle E, A, L\rangle$,where:
– E is a finite set of ***entities***;
– A is a finite set of ***attributes***, each belonging to a single entity;
– L is a finite set of ***links*** between nodes $($entities or attributes$)$ in the graph, i.e., L is a subset of $(E \cup A) \times (E \cup A)$.

Entities differ from one another structurally in terms of the number of attributes they possess and how well connected they are in the schema. We exploit these differences to identify the ones more likely to be queried by a user. A starting point is the definition of schema element importance [79] used to choose entities to summarize a schema. But in addition, we also need to analyze attributes and relationships, which are not of much significance in schema summarization. This brings us to the first of the postulates on which our form generation approach is based.

Postulate 1. The query relevance of an entity depends on how well-connected it is to other parts of the schema.

By *connectedness* we mean the number of attributes and other entities to which this entity is connected via structural or referential links. In the next subsection, we assign a strength to each connection based on how often it occurs in the data. Connectedness of an entity depends on its neighborhood in the schema. In a sense, it is a measure of an entity's centrality to the database.

4.1.2 Data Analysis

If the given database is populated and its content is available at interface design time, we can analyze how the data is distributed to infer the relative importance of each schema entity, attribute and relationship. Specifically, we at for the number of times each node in the schema graph (element or attribute) is instantiated in the data. We call this its *absolute cardinality*. The higher the number of occurrences of an entity node, the higher the probability that it is an entity that a user may be interested in. For instance, a movie database is likely to have many movie entities in it, but very few production companies. This brings us to the second postulate we use to estimate entity importance.

Postulate 2. The query relevance of an entity depends on how many instances (records) of it occur in the database.

Absolute cardinality, as a measure of entity importance, may not always tell the whole story. For instance, a movie database typically has more actors than movies. This does not mean that they are more important either to the data provider or the database users. To avoid ranking actors ahead of movies while designing an interface for such a database, we need to look at a second property of an entity, its *relative cardinality* with respect to its neighboring nodes (via structural or referential links) [79]. This is a measure of its connectedness as observed in the data rather than in the schema. It measures the relative importance of all nodes in the schema.

Formula 1. *(RELATIVE CARDINALITY)* The relative cardinality of a node with respect to a neighboring node is computed as the cardinality of the link instances between them (in the data), normalized by the absolute cardinality of the node.

$$RC(n_i \rightarrow n) = \frac{C(n_i \rightarrow n)}{C(n)}$$

Here, $C(n_i \rightarrow n)$ denotes the link cardinality between the nodes n_i and n in the database, while $C(n)$ is the ***absolute cardinality*** of node n.

4.1.3 Queriability

Given a database, our goal is to determine which entities, entity collections and attributes are likely to be queried most often. We do this by computing their *queriability*, an estimate of their likelihood of being used in a query. We compute a probabilistic estimate for every entity, entity collection or attribute and call this its queriability. If an entity has a queriability of 1, we expect it to occur in every single query to the database (this is, in fact, possible if it is the only entity in the schema). A queriability of 0 means that it is unlikely any query to the database will include this entity (which is possible if, for example, the entity is deprecated and/or has no data instances).

Queriability of Entities

The queriability of an entity depends on its schema connectedness and its data cardinality. While connectedness of an entity is independent of the connectedness of other entities, an entity ought to be more queriable if it is connected to other highly queriable entities than if it were connected to the same number of lowly queriable entities[1]. Hence we use a recursive formula to compute queriability making use of postulates P1 and P2 (like how schema element importance was computed in [79]).

[1]This heuristic is inspired by the approach taken by several search engines to rank web documents. A document is considered "important" if it is connected (linked) to other "important" documents.

Formula 2. *(ENTITY QUERIABILITY)* The queriability of an entity $e \in E$ is computed in two steps. First we perform an iterative computation of the ***importance*** I of each node n $($entity or attribute$)$ in the schema graph until convergence[2] is reached.

$$I_n^r = p * I_n^{r-1} + (1-p) * \sum_i W_{n_i \to n} * I_{n_i}^{r-1}$$

$$\text{where } W_{n_i \to n} = \frac{RC(n_i \to n)}{\sum_k RC(n_i \to n_k)}\,,\ \ (n_i \to n) \in L$$

Here, p is a tuning parameter that takes values between 0 and 1, r is the iteration counter and RC denotes the ***relative cardinality*** of a node with another node. W, which we call the ***neighbor weight***, weighs the importance contribution of each neighbor node (i.e., a node that is linked to this node in the schema graph) by its relative cardinality with this node. The initial importance of any node (I_n^0) is simply its ***absolute cardinality*** in the data. After the values converge, we normalize the final importance of each entity by the sum of the ***absolute cardinalities*** of all nodes in the schema and assign the resulting value to the queriability of the entity.

$$Q(e) = \frac{I_e^c}{\sum_i C_{n_i}}$$

Here, I_e^c denotes the importance of entity e at ***convergence*** and C_{n_i} is the ***absolute cardinality*** of node n_i.

In the above formula, schema connectedness is factored in by the summation over all neighbor nodes and the data cardinality is captured by the neighbor weights W of these nodes. Due to the interdependence of a node's importance on the importance of other nodes, a recursive formula is necessary. The contribution of one node to the importance of another is weighted by the strength of the link between them, measured by their relative cardinality.

Queriability of Related Entities

Entities in a schema can have relationships with one another and related entities are often the focus of queries to the database. Forms querying a single entity can be limiting. On the other hand, creating forms for all pairs, triples, etc. of queriable entities can lead to too

[2]Proof of convergence can be seen in an analogous problem in [53].

many forms that do not make sense. What we need is a measure of queriability of related entities indicating how likely a pair (and eventually, a collection) of entities will be queried together.

Postulate 3. The queriability of a collection of related entities depends on the queriabilities of individual entities in it.

Postulate 4. The queriability of a collection of related entities depends on the data cardinality of all pair-wise relationships between the entities in it.

We posit that the queriability of a collection of related entities is directly proportional to their individual queriabilities and is also proportional to the strength of the relationship (measured by its data cardinality). If multiple relationships exist between two entities, each relationship contributes to the queriability of the collection.

Formula 3. *(RELATED ENTITIES QUERIABILITY)* We calculate the queriability of two related entities $e_1, e_2 \in E$ as the product of their individual queriabilities weighted by the average of their ratios of participation in the relationship between them.

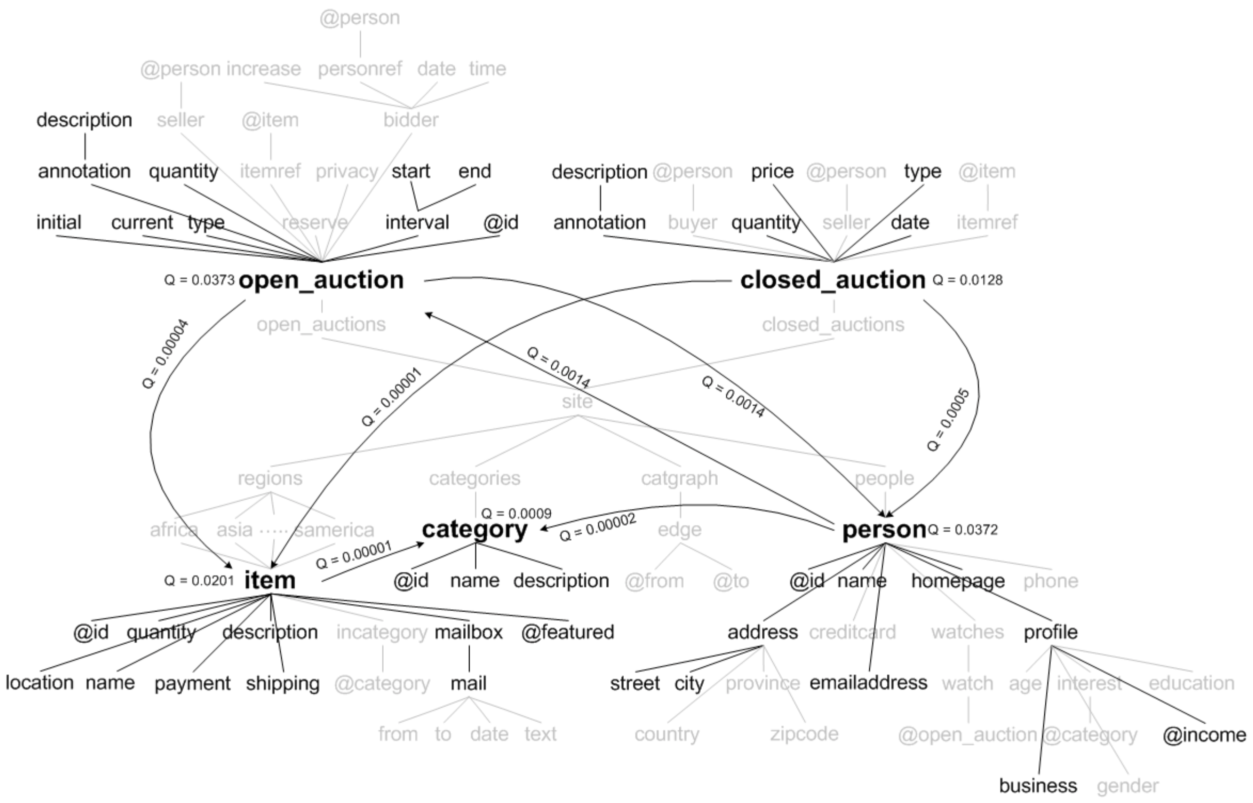
$$Q(e_1 \wedge e_2) = Q(e_1) * Q(e_2) * \left(\frac{R(e_1 \to e_2) + R(e_2 \to e_1)}{2} \right)$$

$$\text{where } R(e_i \to e_j) = \frac{N(e_i \to e_j)}{C(e_i)}$$

Here $R(e_i \to e_j)$ denotes the ***participation ratio*** of e_i in the relationship between e_i and e_j. $N(e_i \to e_j)$ is the number of instances of entity e_i connected to some instance of entity e_j while $C(e_i)$ refers to the ***absolute cardinality*** of entity e_i in the data[3]. The participation ratio of an entity in a relationship is 0 if no instance of that entity participates in the relationship, and it is 1 if every instance of the entity is related to at least one instance of the other entity via the relationship.

An example of related entity queriability can be seen in Fig. 4.1 where the closed_auction entity is related to the person entity (a closed auction has a seller who is a person). Since its participation in the relationship is total, (every closed auction has a

[3]Note the subtle difference between *participation ratio* and *relative cardinality*. If a link exists between entities e_i and e_j in the schema, relative cardinality measures the average number of e_i instances per instance of e_j. Participation ratio, on the other hand, is the fraction of e_i instances connected to at least one instance of e_j. While participation ratio lies between 0 and 1, relative cardinality has no upper bound.



@person
@person increase personref date time
description seller @item bidder
annotation quantity itemref privacy start end
description @person price @person type @item
initial current type reserve interval @id
annotation buyer quantity seller date itemref
Q = 0.0373 open_auction
closed_auction Q = 0.0128
open_auctions
closed_auctions
Q = 0.00004
Q = 0.00001
Q = 0.0014
Q = 0.0014
Q = 0.0005
site
regions categories catgraph people
Q = 0.0009
africa asia ····· samerica
category
Q = 0.00002
edge
person Q = 0.0372
Q = 0.00001
Q = 0.0201 item
@id name description
@from @to
@id name homepage phone
@id quantity description incategory mailbox @featured
address creditcard watches profile
location name payment shipping @category mail
street city province emailaddress watch age interest education
from to date text
country zipcode @open_auction @category @income
business gender

Figure 4.1: XMark schema with the most queriable components highlighted.

seller), the queriability of the related entity pair is simply the product of queriabilities of the two entities.

$$
\begin{aligned}
Q(\text{closed\_auction} \wedge \text{person}) &= Q(\text{closed\_auction})Q(\text{person}) \\
&= 0.0128 * 0.0372 = 0.0005
\end{aligned}
$$

If there are three or more related entities, the formula changes slightly. We start by enumerating all possible permutations of these entities. If there are m related entities, the number of permutations will be $N_\pi(e_1, .., e_m) = m!$ provided there is at most one relationship between any two entities. But if there are multiple relationships between any two entities, each additional relationship will create additional permutations. The total number of permutations then becomes

$$N_\pi(e_1, .., e_m) = m! * \prod_{e_i, e_j \in E} N_r(e_i \rightarrow e_j)$$

where $N_r(e_i \rightarrow e_j)$ is the number of different relationships between entity e_i and entity e_j. For each permutation we first compute the participation ratio of the first and second entities, then the second and third entities, and so on until the last two entities in the permutation. We find the product of these $(m-1)$ ratios, each between 0 and 1, and compute the average of this product across all permutations (note that if the participation ratio of any two entities in the permutation is zero, then that permutation has a zero product). Finally, this average is multiplied by the queriability of each entity. If m = 3, for example, related entity queriability is computed as follows.

$$Q(e_1 \wedge e_2 \wedge e_3) = Q(e_1) * Q(e_2) * Q(e_3) * \frac{1}{N_\pi(e_1, e_2, e_3)} *$$
$$\left(\sum_{\substack{i,j,k \in [1,3] \\ i \neq j \neq k \\ u,v}} R(e_i \xrightarrow{u} e_j) * R(e_j \xrightarrow{v} e_k) \right)$$

For $m \geq 3$ entities, the general formula is:

$$Q(e_1 \wedge e_2 \wedge ... \wedge e_m) = Q(e_1) * Q(e_2) * ... * Q(e_m) *$$
$$\frac{\sum_{\substack{a,b,...,z \in [1,m] \\ a \neq b \neq ... \neq z \\ u,v}} R(e_a \xrightarrow{u} e_b) * R(e_b \xrightarrow{v} e_c) * ... * R(e_y \rightarrow e_z)}{N_\pi(e_a, .., e_z)}$$

Here u and v iterate over all relationships between the entities.

Queriability of Attributes

An attribute is a property of an entity and it is represented in the schema graph by a node connected to its parent entity node. In relational databases, attributes are stored as columns in a table or as separate tables (complex attributes). In XML, attributes can either be XML attributes or non-repeatable sub-elements. Unlike entities, which are meaningful by themselves, attributes are of little use without the entities they describe. Very few queries request an attribute without referencing its parent entity. While entities are ranked relative to other entities in the database (based on queriability), attributes are only compared locally (with other attributes of that entity), not globally. We only consider attributes of entities. Attributes of relationships can be incorporated as a simple extension but are beyond the scope of this thesis.

Postulate 5. The queriability of an attribute depends on its necessity, i.e., how frequently it appears in the data relative to its parent entity.

Necessity of an attribute is a measure of its importance to the entity it describes. We define it as follows.

Formula 4. *(ATTRIBUTE NECESSITY)* The necessity of an attribute is defined as the number of times it is instantiated in the data for each occurrence of its parent entity. The necessity of an attribute a of an entity e is computed as follows.

$$N(a) = \frac{C(e \to a)}{C(e)}$$

Here $C(e)$ is the ***absolute cardinality*** of the entity e, i.e., the number of times it occurs in the database, and $C(e \to a)$ is the number of instances of e with at least one non-null instance of attribute a.

If an attribute appears at least once for every occurrence of the entity, it' *necessity* will be 1. Required attributes thus have higher (or equal) necessity than optional attributes. In the relational context, a null value in a column is treated as the absence of that attribute. We define an attribute's queriability to just be its necessity.

$$Q(a) = N(a)$$

4.2 Form Composition

Having computed the queriabilities of entities, collections of related entities, and attributes, we then select the most queriable of them to compose forms. When we consider these three types of schema components in turn, we see that they are not co-equal: they do not relate to one another in a symmetric way. For example, we will often find it useful to query an entity even with no related entities included in the form. In contrast, querying a collection of related entities always means querying the constituent entities. Similarly, attributes can be present in forms only as constituent components of their parent entities. This leads us naturally to a three step selection process to compose forms. First, we select an entity and create a form for it. Next we choose other entities related to it and create additional forms, one for each unique relationship and its participating entities. Finally we select attributes for each entity and place them in each form containing that entity.

4.2.1 Choosing Form Components

We first sort the entities in decreasing order of queriability and select the top-k_e of them to compose forms. k_e is a pre-specified threshold chosen by the interface designer to control the size of interface generated by the system. Each of the selected (top-k_e) entities will be the query-focus of one form in the generated interface. Next, we take into consideration related entities. We compute the queriability of every collection of entities (related by a schema-defined relationship) containing at least one entity ranked in the top-k_e most queriable. We then consider each of the top-k_e entities, rank its related entity collections by queriability (Formula 3) and select its top-k_r collections. k_r is a threshold that serves as an upper bound on the number of forms created for related entity collections. We then create a new form for each collection. In practice, however, we only consider direct binary relationships and indirect binary relationships, i.e., relationships in which the two entities are not related directly but are both related a common third entity. Binary relationships between entities that are distantly related, i.e., connected by a path of length greater than two in the schema graph, as well as ternary and higher arity relationships are not considered because they incur considerably higher enumeration and computation costs without a high likelihood of reaping commensurate benefits—the queriabilities of these related entity pairs are hard to predict even with heuristics and incorporating them can lead to forms of little interest to users. Finally, we consider the attributes of each entity to place in these forms. We compute the queriability of each attribute and rank them by queriability. We then select the top-k_a attributes of each entity for inclusion in each form containing that entity. k_a is a pre-specified system threshold that controls the complexity of each individual form. In Fig. 4.1, we show the result of our queriability computations for the XMark schema. The XMark dataset we used was generated using the benchmark by setting the scaling factor to 1. The thresholds were set at 5 entities (k_e), 10 attributes per entity (k_a) and 5 related entity pairs per entity (k_r).

4.3 Form Query Definition

Entities, attributes and relationships together form the skeleton of any form that we generate. But forms at this stage are not yet complete. We still need to select query operations that can be performed on these schema elements. These operations ultimately decide what queries a form can express. Our system generates forms that can support a large fraction of queries expressible using a declarative query language. These include select-project-join queries with sorting as well as aggregation. To control computation costs, the system only allows one join or a single nested sub-query. The most common query operation that forms support is *selection*. Filling in a text field in a form enables a user to issue a filtering condition (selection) on the database resulting in only the desired rows. However, there are various other operations that we should also support. For instance, one could create a default form that has joins capturing the relationships between the form's entities, selection predicates applied to all attributes and projections to return all attributes of those entities. Such a form may be unnecessarily complex. But choosing which operation(s) to associate with each attribute is non-trivial. In this section we present the notion of *operator-specific attribute queriability* which quantifies the likelihood that an attribute in the context of a specific query operation being desirable to a user. Our basic idea is to start with a more complex form that permits not just selection on each attribute, but also projection (allowing each attribute to be retained in the result), aggregation (enabling grouping and aggregate computation on any attribute), and sorting (based on attribute values). We then prune this more complex form using this notion of operator-specific attribute queriability.

Formula 5. (*OPERATOR-SPECIFIC ATTRIBUTE QUERIABILITY*) The operator-specific queriability of an attribute is defined as the queriability of the attribute when coupled with a query operation. It is computed as the product of the attribute's necessity and an operation-dependent function of the attribute.

$$Q_q(a) = w_q(a) * N(a)$$

Here, q denotes the query-operation, $w_q(a)$ is a function specific to the query operation (which we introduce in this section) and $N(a)$ is necessity of the attribute.

4.3.1 Operator-Specific Queriability of Attributes

We now define the operator-specific function w_q for some important operators, to compute operator-specific attribute queriability.

Selection

To determine which attributes are suited for selection conditions we consider their range of values in the database.

Postulate 6. The more an attribute distinguishes its parent entity from other entities, the more likely it is to be used as a filter. In other words, since users tend to ask very selective queries, the wider the range of attribute values, the higher the likelihood that attribute is used in a selection predicate.

Continuing with the movie database example, a user looking for information about a specific movie would most likely query the database using its title which is highly selective and distinguishes the movie from most if not all other movies in the database. Users are less likely to search for a movie by year of release, which is less selective since several movies are released in a single year.

Formula 6. *(SELECTION-ATTRIBUTE QUERIABILITY)* The queriability of an attribute for selection depends on both its selectivity and its necessity to its parent entity.

$$Q_\sigma(a) = w_\sigma(a) * N(a) \qquad \left[w_\sigma(a) = \frac{r_a}{C(a)} \right]$$

Here e refers to the parent entity of attribute a, $N(a)$ is the ***necessity*** of a and $w_\sigma(a)$ is the ***selectivity*** of attribute a computed as the range of a (number of unique values a can take) normalized by $C(a)$, the number of occurrences of a in the database (i.e., its ***absolute cardinality***).

Projection

Attributes that are projected compose the output of a query. It is difficult to estimate which attributes of the entity (or entities) involved with the query would be of most interest to users. Using *necessity* as a metric to choose the most desirable attribute(s) may be the best we can do. But sometimes entities have complex attributes, i.e., attributes that are not simple single-valued fields. These include repeatable sub-elements in XML and dependent entities in relational databases. If present, these attributes should be given higher priority than simpler fields because they convey more information.

Postulate 7. Complex attributes are more queriable than simple attributes.

We propose a measure called *attribute size* that counts the number of non-null text-fields in an attribute. This is averaged over all its data instances. The *attribute size* of single-valued attributes is 1.

Formula 7. *(PROJECTION-ATTRIBUTE QUERIABILITY)* The queriability of an attribute for projection depends on its necessity and its size measured as the number of text-fields it contains.

$$Q_\pi(a) = w_\pi(a) * N(a) \quad \left[w_\pi(a) = \frac{C(a \to f)}{\sum_{e \to a_i} C(a_i \to f_i)} \right]$$

Here, $w_\pi(a)$ refers to the size of attribute a (number of text-fields, f) normalized by the sum of sizes of all attributes of the entity, e.

Sorting

"Order-by" attributes have an output role and determine the display order of the query's result. Query results can be sorted by different field types, such as numeric, alphanumeric or purely textual. But these must be simple and preferably required (not null).

Postulate 8. Single-valued and mandatory attributes are more often used to order query results than optional or complex attributes.

We measure their queriability using the following formula.

Formula 8. *(SORT-ATTRIBUTE QUERIABILITY)* The queriability of an attribute for sorting is its necessity multiplied by an indicator function denoting whether the attribute is both single-valued and a required attribute or not.

$$Q_\psi(a) = w_\psi(a) * N(a) \quad \left[w_\psi(a) = \begin{cases} 1 & a \text{ is single-valued} \\ & \text{and required;} \\ 0 & \text{otherwise.} \end{cases} \right]$$

Aggregation

While any attribute in a schema can be aggregated as the result of a grouping, aggregation on numeric-valued attributes (for e.g., to find a sum, an average, a maximum value, etc.) is more common than others. Also, repeatable attributes are more likely to be aggregated. For example, the number of authors of a book, the number of available seats in an airline, etc. In relational schemas, repeatable attributes are represented as separate tables with a 1-to-many or a many-to-many relationship with the entity tables. These notions are captured in the following postulate.

Postulate 9. Repeatable and numeric attributes have a greater likelihood of being aggregated in a query than other types of attributes.

Formula 9. *(AGGREGATION-ATTRIBUTE QUERIABILITY)* The queriability of an attribute for aggregation is its necessity multiplied by an indicator function denoting whether or not the attribute is both numeric and repeatable.

$$Q_\gamma(a) = w_\gamma(a) * N(a) \quad \left[w_\gamma(a) = \begin{cases} 1 & a \text{ is numeric and} \\ & \text{repeatable;} \\ 0 & \text{otherwise.} \end{cases} \right]$$

4.3.2 Choosing Form Fields

Using the modified measures of attribute queriability, we can determine which form-fields (for each entity) to place on each form to maximize the likelihood that it is desirable to a user. We start by creating all possible form fields for each of the k_a most queriable attributes that were chosen in the third step of our form generation process. These include selection, projection, aggregation and sort fields. Note that join fields are determined by the relationships between pairs of related entities chosen in the second step. We now compute the operator-specific queriabilities of each operator-attribute pair, i.e., each attribute paired up with a query operator (selection, projection, etc.) and for each operator type, we use this score to rank all fields of that type.

Next we need to determine how many fields of each type to include in the final form. We define a threshold k_f on the total number of fields (of any type) per entity in a form. Such a threshold ensures that no form is too complex. While increasing the number of form-fields also increases the range of queries a form can support, it also increases form complexity. We use k_f to choose how many form fields to keep. We next have to divide k_f among the various operators. We define operator-specific thresholds: k_σ, k_π, k_ψ and k_γ (which sum to k_f) to limit the number of fields of each type. These thresholds again are system thresholds, but may be specified relatively (as fractions of k_q) rather than in absolute terms. Each form is thus composed of the top-k_f fields (operator-specific attributes) of any top-k_e entity and may also include the top-k_f fields of one of its top-k_r related entities. A summary of the three-step form generation process is provided in Algorithms 2 through 5. A sample form (created for the Geoquery schema [6]) can be seen at `http://www.eecs.umich.edu/~mjayapan/vldb2008/GeoqueryForm.html`.

Algorithm 2: Algorithm `GenerateForms`

```
Input: A Database $D$ with a schema $S$
Input: Complexity thresholds: $k_e$ (for entities), $k_a$ (for attributes), $k_\sigma$, $k_\pi$, $k_\psi$, $k_\gamma$ (for
       operator-specific attributes) and $k_r$ (for related entity collections)
Output: A set of forms $F$
$G$ = AnalyzeSchema($D$, $S$);
$F$ = AssignSchemaCompntsToForms($G$, $k_e$, $k_a$, $k_r$);
$F$ = CreateFormComponents($F$, $k_\sigma$, $k_\pi$, $k_\psi$, $k_\gamma$);
```

Query Generation

The purpose of a form is to convey a user-specified query to the underlying database for execution. However, unlike human-designed forms, our forms cannot have machine-readable queries (in SQL or XQuery, for example) hard-coded in them. Since form generation is automatic, query generation must also be automatic. Furthermore, since the

Algorithm 3: Algorithm `AnalyzeSchema`

```
Input: A Database $D$ with a schema $S$
Output: An annotated schema graph $G$
Create an empty graph $G$;
foreach node $n_s \in S$ do
    // $n_s$ : element/attribute/table/column
    Create a corresponding node $n$ in $G$;
foreach edge $e_s \in S$ do
    Create a corresponding edge $e$ in $G$;
foreach relationship $r_s \in S$ do
    // $r_s$ : key-keyref/PK-FK
    Create an edge connecting the nodes participating in $r_s$;
    Record $r_s$ in each of the nodes in $G$ that participate in it;
Identify which nodes in $G$ are entities and which are attributes;
foreach node $n \in D$ do
    if $n$ represents an entity $e$ then
        Find its node $n_e$ in $G$ and increment its cardinality;
    else if $n$ represents an attribute $a$ then
        Find its node $n_a$ in $G$ and increment its cardinality;
        Annotate $n_a$ and its entity node $n_e$ with each other's cardinality;
        if $a$ is a simple attribute then
            Add its current value to its value range;
        if $a$ is part of a relationship specification $r$ then
            Increment the cardinality of $r$ and record it in $n_e$;
        Compute the attribute-size of $a$ and annotate $n_a$;
foreach node $n \in G$ do
    if $n$ represents an entity $e$ then
        Compute its queriability $Q(e)$;
        foreach entity $e_r$ related to $e$ do
            Compute the queriability $Q(e \wedge e_r)$;
    else if $n$ represents an attribute $a$ then
        Compute its operator-specific queriability $Q(a)$ for all operators (selection, projection, sort
        and aggregation);
```

number of different queries that a single form can produce is exponential in the number of fields it contains, instead of generating these queries at form-creation time, we generate them at runtime, i.e., after any user has specified exactly which fields he or she requires. We have a translation mechanism in place that can convert a filled form into a query in a standard declarative language that any standard database system can use to evaluate queries.

Algorithm 4: Algorithm `AssignSchemaCompntsToForms`

```
Input: An annotated schema graph $G$
Input: Complexity thresholds: $k_e$, $k_a$ and $k_r$
Output: A set of forms $F$
Rank schema entities in $G$ by queriability and put the top-$k_e$ in the set $E$;
foreach schema entity $e \in E$ do
    Rank attributes of $e$ and put the top-$k_a$ in the set $A_e$;
    Rank entity collections that $e$ is a part of and put the top-$k_r$ in the set $R_e$;
    Create a blank form $f$ and assign entity $e$ to it;
    Include the attributes in $A_e$ in $f$;
    Add $f$ to $F$;
    foreach entity collection $r \in R_e$ do
        Create a blank form $f_r$ and assign $r$ to it;
        Assign all entities in $r$ to $f_r$;
        Include the top-$k_a$ attributes of each entity in $r$ in $f_r$;
        Add $f_r$ to $F$;
```

Algorithm 5: Algorithm `CreateFormComponents`

```
Input: A set of forms $F$
Input: Complexity thresholds: $k_\sigma$, $k_\pi$, $k_\psi$, $k_\gamma$
Output: A set of forms $F$
foreach form $f \in F$ do
    if $f$ has one assigned entity then
        Let $e$ be the assigned entity;
        Create a panel $p$ in $f$;
        foreach attribute $a$ of $e$ do
            Create a selection form field $sf$ for $a$;
            Create a projection form field $pf$ for $a$;
            Create a sort form field $tf$ for $a$;
            Create an aggregation form field $af$ for $a$;
            Add $sf$, $pf$, $tf$ and $af$ to the panel $p$;
        Prune out all but the top-$k_\sigma$ selection form fields, the top-$k_\pi$ projection form fields, the
        top-$k_\psi$ sort form fields and the top-$k_\gamma$ aggregation form fields;
    else if $f$ has a collection of entities assigned to it then
        Let $E_f$ be the set of entities in $f$;
        Let $R_f$ be the set of relationships between them;
        foreach $e \in E_f$ do
            Create a panel $p_e$ in $f$;
            foreach attribute $a$ of $e$ do
                Create a selection form field $sf$ for $a$;
                Create a projection form field $pf$ for $a$;
                Create a sort form field $tf$ for $a$;
                Create an aggregation form field $af$ for $a$;
                Add $sf$, $pf$, $tf$ and $af$ to the panel $p$;
            Prune out all but the top-$k_\sigma$ selection form fields, the top-$k_\pi$ projection form fields, the
            top-$k_\psi$ sort form fields and the top-$k_\gamma$ aggregation form fields;
        Build a join form field $jf$ for each pair-wise relationship $r \in R_f$;
        Create a panel $p_r$ on $f$ and add each $jf$ to it;
```

4.4 Evaluation

We implemented our system on top of the TIMBER [45] database system which uses the XML data model. Schemas are defined as XML Schema Definitions (XSD) and queries are represented using XQuery. To evaluate our system on a particular dataset, we need its schema, data and query log. Since query logs are not readily available for many XML databases, we had to work hard to find a diverse collection of data sets to work with. Note that while we described relationships between entities that could be n-ary, our current implementation only allows binary relationships.

4.4.1 Experimental Methodology

In this chapter, we propose a new technique to generate user interfaces to databases. The best way to evaluate a user interface is to measure how useful it is (or can be) in satisfying the needs of real users. In this section, we suggest a way to quantify the *usefulness* of a forms-based interface which we then use to demonstrate the effectiveness of our proposed techniques. We obtained three datasets (two publicly available and one in-house) for which we also had access to real user queries. For each of these databases, we generated forms automatically using their schema and content. We then used the accompanying queries to evaluate the forms generated for each dataset. We define *usefulness* as the fraction of queries in each query set that can be expressed using the forms we create. In addition, for one of the datasets, we compared the forms generated by our system with forms created by a human expert for a database in the same domain. We present our results and observations in this section. To understand interface usefulness better, we also evaluated the effect system thresholds have on form query support. Experiments were also conducted to evaluate the effects of our design heuristics, both individually and collectively. The usefulness experiments serve to judge our form generation techniques while the threshold variation experiments serve to help data providers develop an intuition for good threshold values for their own databases.

4.4.2 Data Sets and Query Workloads

The system was evaluated using three datasets: a real-world database, MiMI [8], and two natural language querying benchmarks, Geoquery [6, 76] and Jobsquery [7, 21, 76]. These datasets differ from each other in terms of schema-size, schema-complexity, data-size and data-organization. Our goal in choosing these diverse sources is to understand the usefulness of our form generator in different real world environments.

MiMI

MiMI (Michigan Molecular Interactions) [8] is a real-world molecular interactions database used by biologists and bioinformaticians. MiMI has a complex schema with close to 100 XML elements. This schema was used to hand-generate the initial set of forms which were built for the key entities including molecule (e.g. proteins, genes, DNA, etc.), interaction (e.g. protein-protein interactions) and organism (species of origin for molecules, interactions, etc.) and key relationships between these entities. The web-interface to the MiMI database allows query specification using either a form, direct XQuery or by a special querying tool that allows users to navigate the schema, click on the entities and attributes of interest and specify query parameters in a form that is generated dynamically based on their selections [46].

Query Workload: We obtained a log of queries posed against the MiMI database by real users (many from the biology and bioinformatics domains) using any of the above methods. Each query was logged as an XQuery statement regardless of which querying method was actually used. The query log comprised a total of 3,856 queries of varying complexities ranging from simple single-attribute selection queries to complex nested queries with aggregation. A few of the queries in the trace were erroneous, either due to XQuery syntax errors or due to incorrect references to schema entities and/or attributes. We detected and removed 12 such queries leaving 3,844 valid queries in the log.

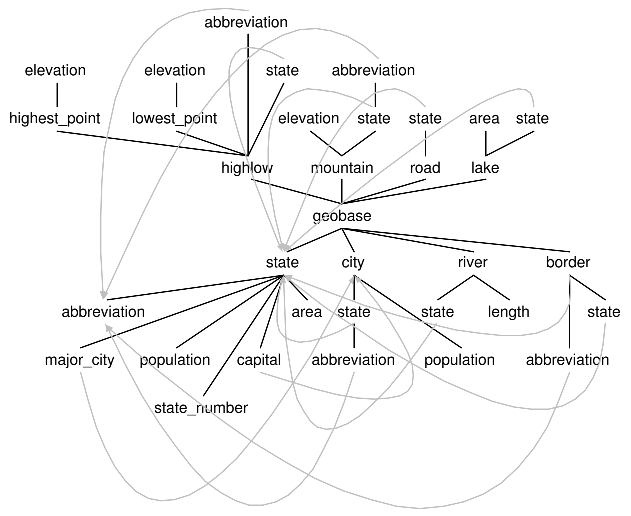


Figure 4.2: Geoquery Schema

Geoquery

The Geoquery database [6] contains geographical information about the United States including states, cities, roads, rivers, etc. (schema shown in Fig. 4.2). This information was compiled and stored as Prolog assertions (close to 900 assertions) primarily to test a natural language query interface. In order to use this dataset for our experiments, we translated the data into XML and extracted a schema definition in XSD. The query set consists of 880 natural language questions posed by real users of the database's publicly accessible web interface and also by undergraduate students in the Computer Science department at the University of Texas (Austin). With the help of the deduced schema, we translated these English queries into XQuery which we then use to evaluate the effectiveness of our system.

Query Workload: The Geoquery query set consists of 880 declarative queries. Although the queries in their original form (natural language) were unique, upon transformation, the resulting set had 301 repeated queries (34%). The queries are of a wide range of complexity, with some queries as simple as *"How high is Mount Mckinley?"* and some as complex as *"How many states have a higher point than the highest point of the state with the largest capital city in the US?"*. While the former is a single-attribute selection query, the latter has two nested sub-queries each with an associated aggregation operation and as many as five join conditions.

Jobsquery

The Jobsquery database [7, 21] consists of a set of 1000 computer-related job postings from the USENET newsgroup *austin.jobs*. Information from these job postings were extracted to create a database which contains specific information about each position available (including job title, company location, salary offered, required skills, experience, etc.). The schema of this dataset is fairly simple and flat. It only has one entity (job) with 18 attributes including title, salary, required_experience and desired_degree. The query set [7] of this benchmark consists of 640 queries, 240 of which were posed by real users and the remaining 400 generated artificially using a simple grammar that generates certain obvious types of questions people may ask.

Query Workload: Here again, we needed to translate the data from Prolog to XML and deduce an XML schema. The queries had to be converged to XQuery and we performed this translation manually. While the natural language queries were all unique, only 338 of the 640 queries were distinct (about 53% of the total). These queries include a wide range of selection queries including simple, conjunctive, disjunctive, quantified and negated selection conditions.

4.4.3 Form Usefulness

We evaluated the query support of our system for three different datasets with respect to their respective query sets.

MiMI

We set thresholds to ensure that the system selected no more than 2 entities, 10 attributes per entity and 1 related entity per entity. Given these constraints, a total of 4 forms were built using the schema and content of the MiMI database. Testing each of the 3,844 valid queries against each form generated we found that as many as 3,150 of them were supported by a form in the automatically created interface (about 82%). When the number of entities allowed (k_e) was raised to 3, an additional 578 queries (15%) were satisfied by the 2 new forms generated (in total 97% of the queries were expressible using 6 forms).

Geoquery

We observed a substantial number of highly complex queries in the query set. Unfortunately, our system only considers SPJ queries with sorting and aggregation containing one join or nested sub-query[4]. Hence queries that have multiple levels of nesting that also require multiple joins are not supported by the forms we generate. This results in performance not as good as the other two sets we used for our evaluation. Even so, the forms generated can still satisfy a majority of the queries posed. If we ignore these complex queries that involve two or more join conditions we find a much higher fraction of queries being supported. For example, if we restricted the interface size to 13 forms and only 7 attributes per entity, we could answer 61% of all user queries, and as many as 91% of the non-complex queries.

Jobsquery

Our system generated a single form for this dataset since it only contained one entity, job. Moreover, the schema does not define any relationship between attributes or between multiple instances of the same entity. The single form generated contains fields for the 14 most queriable attributes of job. We found that this form was enough to satisfy 80% of all queries that its users desired.

COMPARISON WITH HUMAN-GENERATED FORMS

[4]Support for nested sub-queries is important because in XQuery, unlike SQL, an aggregation query involving two related entities may not be possible without a nested sub-query. For such an aggregation, XQuery requires that the aggregated entity either be a sub-element (in the schema) of the first entity (in which case no value-join is necessary), or it must be bound to a set variable after the join. The latter case requires the user of a nested sub-query.

In this section we attempt to measure the performance of a manually generated form and compare that with an automatically generated form for a given schema. We use the Jobsquery query set and a real jobs database with a schema similar. The database chosen for this comparison is the popular employment website, Monster.com. While the exact schema this database conforms to is not known, the data it presents in response to user queries contains all the attributes used by the Jobsquery schema. Hence queries in the Jobsquery query set are all answerable by the Monster.com database. We reproduced the "Advanced form" provided on the website and measured the fraction of queries from the query set that it can express.

We observed that the form, which contains 12 fields, supports 202 of the 640 queries in the set, or 31.56%. In contrast, recall that a machine-generated form using our procedure containing 14 fields supports 80% of the query set. Even if we restrict it to 12 fields (not a good choice as evident from Fig. 4.11), the automatically generated form still supports 52% of the query set. If allowed just 18 fields, the automated procedure supports 100%! To compensate, the form at Monster.com does provide a keyword search box, so a free text search could be performed in addition to fielded search. Thus any of the 438 unsupported queries could still be asked, just not precisely and not directly in one easy step. Our form generation technique can greatly reduce the need to resort to this backup plan.

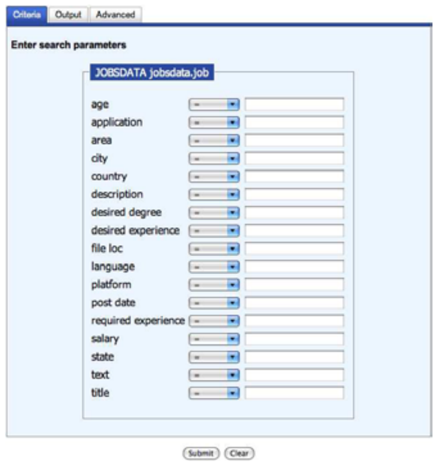
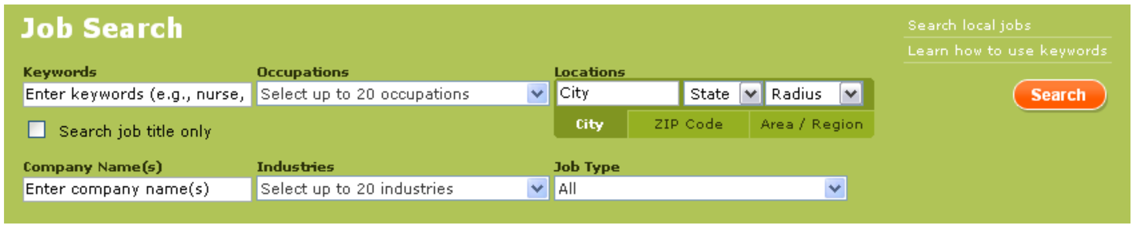


Figure 4.3: Form automatically generated for Jobsquery

Fig. 4.3 shows the automatically generated form while Fig. 4.4 shows the human generated form.


Job Search

Search local jobs
Learn how to use keywords

Keywords: Enter keywords (e.g., nurse,
Occupations: Select up to 20 occupations
Locations: City | State | Radius
City | ZIP Code | Area / Region

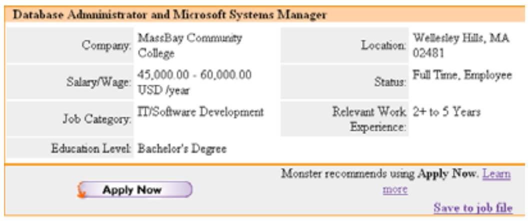
Search

☐ Search job title only

Company Name(s): Enter company name(s)
Industries: Select up to 20 industries
Job Type: All

Figure 4.4: Monster.com Query Form (Advanced)

As one can see, there are some differences in the form fields provided by the two interfaces and these differences essentially dictate the difference in the number of queries of the Jobsquery query log supported by either interface. For instance, fields like desired experience and salary are not queriable using the form provided by Monster.com even though these fields occur in the result page (Fig. 4.5).


Database Administrator and Microsoft Systems Manager

Company:	MassBay Community College	Location:	Wellesley Hills, MA 02481
Salary/Wage:	45,000.00 - 60,000.00 USD /year	Status:	Full Time, Employee
Job Category:	IT/Software Development	Relevant Work Experience:	2+ to 5 Years
Education Level:	Bachelor's Degree		

Apply Now

Monster recommends using **Apply Now**. Learn more

Save to job file

Figure 4.5: Sample Monster.com Query Result

Moreover, the human generated form does not support negation or other non-equality conditions (e.g. city ! = “austin”) unlike the automatically generated form.

Here are some example of queries from the query set that *are* supported by the human generated from:

1. what systems analyst jobs are there in austin?

2. find all network administration jobs in austin?

3. show me the job application for ic design engineer?

4. show me programmer jobs in tulsa?

5. show me all of the software qa jobs in austin?
6. show me positions in web programming?
7. show me jobs in texas?
8. show me all job that are available?
9. are there any project manager positions open?
10. could a senior consulting engineer find work in boston?

On the other hand, here is a list of sample queries (also from the query set) that could *not* be expressed with just the human generated form:

1. which system administrator jobs in dallas require 2 years experience and pay 50000?
2. what jobs as a senior software developer are available in houston but not san antonio?
3. show me the jobs with 30000 salary?
4. show the jobs with the title systems analyst requiring 2 years of experience?
5. show jobs in austin that require a bscs?
6. what are the jobs in washington that require at least 5 years of experience?
7. can i find a job making more than 40000 a year without a degree?
8. i want a job that doesnt use windows?
9. are there any jobs for people in austin that want to program in lisp but do not have a degree?
10. what jobs require a bscs and no experience?

For queries that aren't supported, a user could always issue a more general query that leaves out one or more constraints that cannot be specified, or even put the constraint value in the all-purpose keyword search field, submit the form and then inspect all the results to find the desired result (by ignoring the results that do not satisfy the condition(s) not explicitly specified). However, there is a cost associated with filtering out results manually, often mentally, which some users must pay to overcome the restrictions of the form. Moreover, this is an error-prone task which could have been avoided had the form included the missing attribute fields (or had the form designers made alternate arrangements for these kinds of queries).

4.4.4 Effect of Thresholding

Next, we sought to analyze how system performance depended on the thresholds that controlled the number of forms and the structure and content of each form (using the same three datasets).

MiMI

We varied the entity-threshold (k_e) and observed its effect on the number of user queries supported by the interface. The number of forms created for related entities k_r, was fixed at 1, while the number of attributes made queriable for each selected entity, k_a was kept constant at 10. Our observations are shown in Fig. 4.6. We can see a clear bias in the queries towards the most queriable entity (which happened to be molecule) which accounted for 76% of all queries in the workload. An additional 218 queries (6%) involved the next most queriable entity (interaction) and finally, the third entity (organism) was found in 578 queries posed to the database.

Next, we varied the attribute-threshold (k_a) and observed the effect this had on the number of issued queries that were supported by the generated interface. Our results are shown in Fig. 4.7. Here we see that except for a very low limit on the number of attributes (per entity), a significant fraction of user queries were expressible using the interface

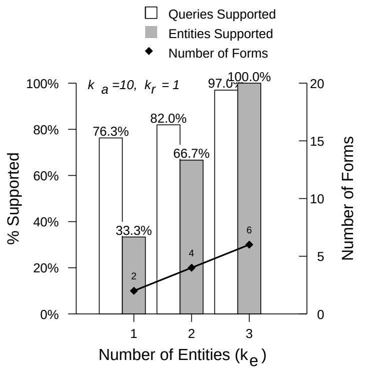


Figure 4.6: Effect of Entity Threshold in MiMI (k_e)

generated automatically. We also observe that for $k_a > 6$, the increase in query support is low even as more and more attributes are made queriable to users. This is because the threshold of 6 includes the most important attributes of each entity, and subsequent increases only add attributes that were less frequently used. We also show in the figure the actual fraction of unique attributes queried by users that our forms actually include. Considering the complexity of the database, it is noteworthy that 54% of queried attributes are ranked in the top 10 by our algorithm. These accounted for 97% of all queries posed to the database.

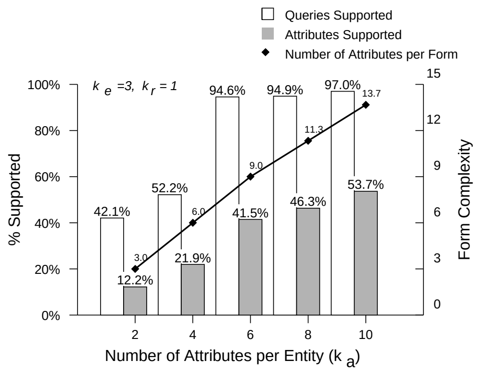
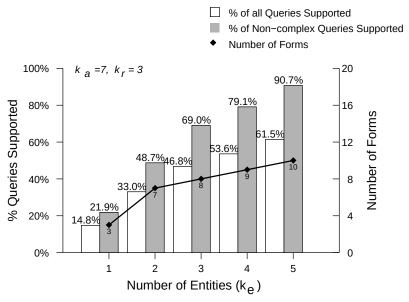


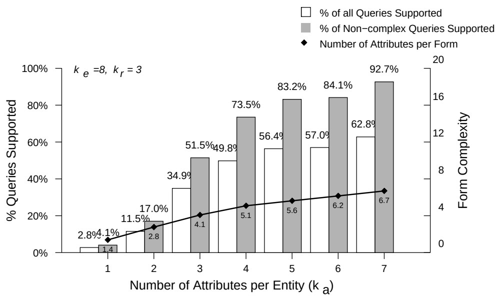
Figure 4.7: Effect of Attribute Threshold in MiMI (k_a)

Geoquery

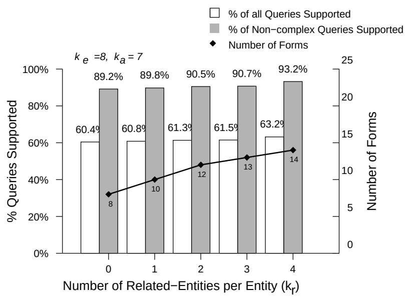
We altered the system thresholds k_e, k_a and k_r one at time, while keeping the other two constant (generating a fresh form set for each combination). We show the fractions of queries supported with and without the multiway join queries (non-complex and complex respectively) in Fig. 4.8.



(a) Geoquery: Dependence on k_e



(b) Geoquery: Dependence on k_a



(c) Geoquery: Dependence on k_r

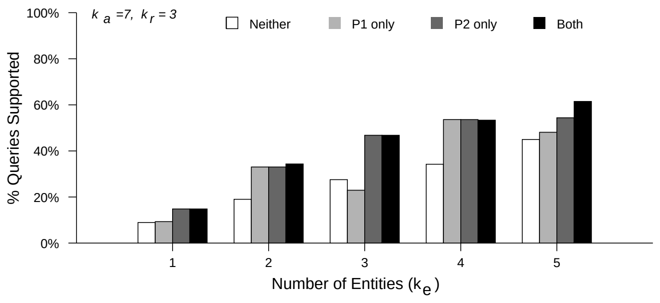
Figure 4.8: Effect of Thresholds on Query Support in Geoquery

We observe that if the number of entities for which forms are generated is varied from 1 to 8 (the total number of entities in the schema) the fraction of queries supported increases for each form set. The rate of increase is high initially but reduces thereafter. This shows that for any given k_e the system generates forms for the most queriable entities within that threshold. This is a desirable property that ensures that the interface does not need to be large to support a large portion of actual queries. The ranking mechanism is simply based on postulates which may not always match the actual queriability of each entity, attribute or related entity pair. As we can see in Fig. 4.8(b), if k_a = 1, i.e, if the system allows only 1 attribute per entity on the form, it does not find the most queriable attribute for each entity. This is evident from the increase in query support being higher from k_a = 2 to k_a = 3 than from k_a = 1 to k_a = 2. However, the figure also shows that the system achieves more or less the desired behavior from k_a = 3 onwards. The effect of threshold k_r, which sets the number of related entity pairs per entity, can be seen in Fig. 4.8(c). It is observed to have less impact than the other thresholds.

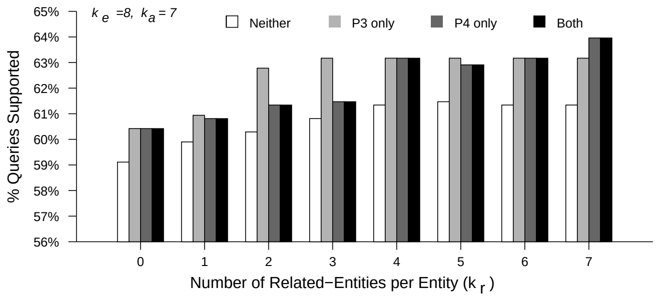
EFFECTIVENESS OF POSTULATES

First, we compared the usefulness of the ranking mechanism with that of random selection (of entities, attributes and related entity pairs) using Monte Carlo simulation. We observed that for 100 random trials, the p-value is 0.11 if only one entity must be selected (k_e = 1) whereas for greater values of the entity threshold (up to k_e = 8) the p-value is less than 10^{-2}. This means the ranking of entities that performed best, i.e. supported the most user queries, could not have been arrived at by chance. Further, we conducted trials in which selected postulates were turned on in isolation (all others were turned off) to evaluate the individual effectiveness of these postulates.

First, since only postulates P1 and P2 are used for entity selection, we generated forms, first using both, then only using P1, next only using P2 and finally using neither (random selection, without using queriability). We measured query support in each case for different entity thresholds and compared them (Fig. 4.9(a)) observing that using both provided the greatest query support while using neither provided the least. Related entity selection is based on the use of postulates P3 and P4. We examined their effectiveness for different values of threshold k_r. Our results are shown in Fig. 4.9(b). Finally, we evaluated the effectiveness of postulates P5 through P7 which are captured in Figures 4.10(a) and 4.10(b). We found that not all postulates in isolation produced an improvement in query support for all values of k_e, k_a, k_r and k_f. But when these threshold values are increased to typical levels each postulate is useful.



(a) Effect of Postulates P1 and P2



(b) Effect of Postulates P3 and P4

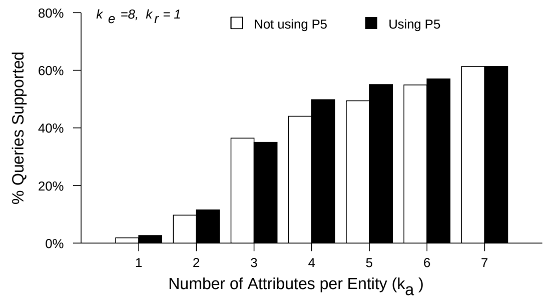
Figure 4.9: Effect of Postulates on Query Support in Geoquery

Jobsquery

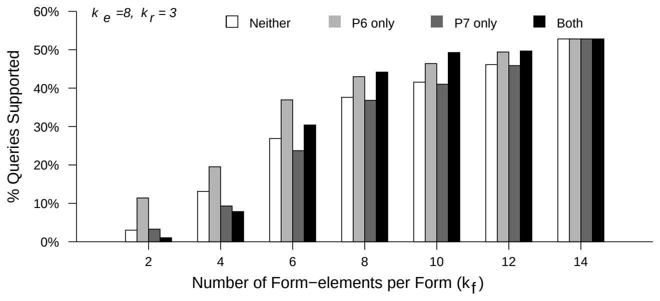
Since this dataset only has one entity and no relationships, we did not analyze the effect of k_e or k_r. We only varied k_a and each time evaluated the form created using the query set. As in the previous experiments, the number of queries satisfied increases sharply for lower values of k_a (> 4), but flattens out towards the end, demonstrating a good initial choice of attributes by the system.

4.4.5 Discussion

Our evaluation shows that our form generation system can indeed produce forms, of manageable number and complexity, that are capable of posing a majority of user queries to a given database, using just its schema and its data content. Notably, our automated form for Jobsquery supported more than twice as many queries as the form for a major commercial website, Monster.com. However, as we saw in the Geoquery experiment, the



(a) Effect of Postulate P5



(b) Effect of Postulates P6 and P7

Figure 4.10: Effect of Postulates on Query Support in Geoquery

system does not satisfy the most complex of queries which can be of interest to users. We have seen that a majority of the queries in any workload are not as complex. Secondly, some datasets have attributes that are not intended to be queried by users, such as metadata or private information. Without expert assistance, the system cannot recognize these and may rank such attributes to be highly queriable (e.g. file_loc in Jobsquery).

Number of Forms

The greater the number of distinct forms, the higher the expressivity of the interface. However, having a large number of forms can be a disadvantage if they make it difficult for a user to find the one he or she wants to use. The number of forms we generate is dependent on the size and complexity of the schema as well as the pre-specified thresholds: k_e and k_r. Many of today's databases have very complex schema necessitating the creation of a large number of forms to obtain a sufficiently expressive interface.

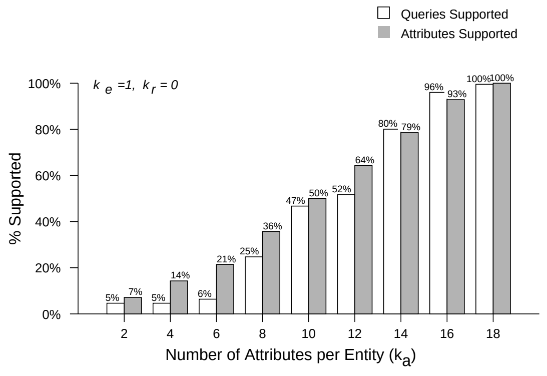


Figure 4.11: Effect of k_a on Query/Attribute Support (Jobsquery)

Many current database-backed websites employ over 10 different forms, some as many as 102 [9]. Ideally, our system should be appropriately tuned to generate interfaces with 5-15 forms to balance query expressivity with interface size. One might argue that greater query expressivity makes the case for a more expressible querying mechanism, such as a declarative language. However, this is not a good option for the vast majority of database users who are programming-averse.

Even if a large number of forms is unavoidable, an automatically generated interface can still be made usable. Instead of reducing the size of the form set, we can try to solve the problem of *form selection*, i.e., choosing the right form from a large set of forms without having to scan many of them. Since each form concentrates on one or more entities, we can identify forms by the names of their corresponding entities. For instance, a form used to find a movie can be named ‘movie’, while a form that queries both movies and actors can be called ‘movie–actor’. If two entities can be related in more than one way, the names of the attributes corresponding to the each relationship can be included in a form’s name. We could then build a two-level menu bar that presents users with the available form choices. The first (top-level) menu, which is always visible, lists the names of the top-k_e entities from left to right in decreasing order of queriability. Once the user selects one of them, a second menu bar appears below (or to the right of) the first and displays the names of its top-k_r related entities. If the user is only interested in querying the first entity, the search stops here. If he or she is interested in querying two entities simultaneously, a second entity selection is required, this time from the second menu bar. In either case, the appropriate form is pulled up from the form repository and presented to the user. Using this selection mechanism, a user needs to look at no more than k_e form names for single entity forms and (k_e + k_r) names for two-entity forms. In the latter case, a scan of the interface could require browsing as many as (k_e * k_r) forms. A third or fourth level menu can also be added if higher arity relationships are captured in the forms generated.

Selection of Thresholds

Our evaluation also shows that the effectiveness of our form generator is reliant on the choice of thresholds k_e, k_a, k_r and k_f. But how does one set these when the interface is first created (without the benefit of hindsight)? Ideally the interface has at least one form for each entity by itself. If the schema has more than 10 entities, this already results in too many forms. Hence the entity threshold k_e ought to be less than this upper bound. Some entities may be left out, but these should be the less important ones. Within a form it is acceptable, and in fact typical, to have up to 20 specifiable conditions. But k_a is a threshold on the number of attributes per entity, not per form. Our experiments show that a majority of user queries involve no more than two entities. This suggests that it is prudent to set k_a between 5-10. k_f can be set to 20. Finally the choice of k_r depends on how entities are connected to each other within the database. A setting of 1-2 is advisable from our observations. In the worst case, these settings will result in 30 forms being generated (if k_e = 10 and k_r = 2 and the 10 most queriable schema entities are each related to 2 other entities). But typically not all thresholds are “maxed out” in every combination. These settings generate approximately 5-15 forms.

4.5 Cost of Form Generation

The steps taken in generating forms automatically were outlined in Algorithms 2 - 5. We can express the total cost of form generation as the sum of costs of these individual steps. First we construct the schema graph which has a node for every entity or attribute in the schema and an edge for every link between nodes. If the number entities in the schema is n_e, the maximum number of attributes per entity is n_{a_e} and the maximum number of relationships per entity is n_{r_e} (represented either by a parent-child link to another entity node in the graph or a link between attribute nodes of the two entities), the cost complexity of graph construction is given by:

$$\begin{aligned} C_{gc} &\in O(n_e + n_e * n_{a_e} + n_e * n_{r_e}) \\ &\in O(n + e) \end{aligned}$$

where n is the total number of nodes in the schema graph (representing entities and attributes) and e is the number of edges (for both attributes and relationships).

The next step is to annotate the schema graph using the content of the database. This involves examining each instance of every node and edge in the schema. If d_e represents the maximum number of instances of any entity, d_{a_e} denotes the maximum number of instances of any attribute of an entity in the data and d_{r_e} is the maximum number of instances of any relationship in which an entity participates, the cost complexity of graph annotation is given by:

$$C_{ga} \in O(n_e * d_e + n_e * n_{a_e} * d_{a_e} + n_e * n_{r_e} * d_{r_e})$$

$$\Rightarrow C_{ga} \in O(n_e(d_e + n_{a_e} * d_{a_e} + n_{r_e} * d_{r_e}))$$

The third step in schema analysis is to compute the importance of each node followed by the queriability of each entity, attribute and collection of related entities. The cost of importance computation is quadratic in the number of graph nodes (n) in the worst case, i.e., if the graph is complete[5]. Queriability computation, on the other hand, is linear in the number of entities, attributes and relationships since it is done only after the importance values converge.

$$\begin{aligned} C_{qc} &\in O(n^2 + n_e(1 + n_{a_e} + n_{r_e})) \\ &\in O(n^2 + n + e) \end{aligned}$$

Next, we sort the entities by queriability and select the top-k_e most queriable entities for which to design forms. The cost of entity selection is thus given by:

$$C_{es} \in O(n_e \log n_e + k_e)$$

For each top-k_e entity, we sort its attributes and related entities (by queriability) and select the top-k_a and top-k_r of them respectively.

$$C_{as} \in O(k_e * (n_{e_a} \log n_{e_a} + k_a))$$

$$C_{rs} \in O(k_e * (n_{e_r} \log n_{e_r} + k_r))$$

If n_q denotes the number of query operators considered while computing operator-specific attribute queriabilities for each of the top-k_a attributes (for each top-k_e entity), the cost of this computation and selecting the top-k_f operator-specific attributes for each entity is given by:

$$C_{os} \in O(k_e(n_q k_a + k_f))$$

For example, if selection, projection, sort and aggregation are the operators considered, then $n_q = 4$ and $k_f = k_\sigma + k_\pi + k_\psi + k_\gamma$.

Finally, we generate forms for each selected entity and pair of related entities such that

[5]Database schemas are typically not complete graphs and so instead of n^2, the importance computation is typically (in practice) bounded by nf where f is the maximum fan-out of any node.

each form contains k_f form-elements per entity. The cost of this form design step is given by:

$$C_{fd} \in O(k_e k_f + k_e k_r k_f)$$

$$\Rightarrow C_{fd} \in O(k_e k_f (1 + k_r))$$

The total cost of form generation is the sum-total of all the above costs. Since k_e, k_a and k_r are by definition less than n_e, n_{e_a} and n_{e_r} respectively, the cost of form generation is asymptotically bounded by:

$$C_{fg} \in O(n^2 + n_e(d_e + n_{a_e} d_{a_e} + n_{r_e} d_{r_e})$$

$$+k_e(n_{e_a} \log n_{e_a} + n_{e_r} \log n_{e_r}))$$

4.6 Extensions

The completely automated form generation process we described can be improved through expert guidance. It may be useful to allow a human expert, such as the database administrator, to provide hints to the system that drive the form generation process. Such a person would not be expected to design actual forms, but use domain knowledge to guide the system in the right general direction. This is a one-time effort and is provided at interface creation time. We outline how this can improve the effectiveness of a forms-based interface with minimal effort.

4.6.1 Schema Annotation

A domain expert who is also conversant with the database schema could have a sense of which parts of the schema and data users will find most useful, i.e., which parts are likely to generate the highest query traffic. There could be differences between these and what the automated system determines to be important, and by annotating these parts of the schema appropriately, the system can place greater emphasis on them while generated forms to cater to users who would be interested in these parts of the schema.

The annotation procedure can simply be the addition of one or more entities to the list of highly queriable ones, or in rare cases a re-ranking of the entity list. Additionally, there could be some attributes in that data that are of a sensitive nature. These can be marked by an expert to be excluded from consideration while generating forms. A second way in which an expert can assist the automated process is by identifying collections of entities that relate to one another which collectively form the basis of a meaningful and perhaps useful query. This is especially useful for relationships with arity greater than 2 (ternary, quaternary, etc.). Although the system does factor entity-relationships while designing forms, queriability and usefulness are hard to gauge for collections larger than two entities.

4.6.2 Useful Query Types

Aside from being able to identify regions of maximum interest in the schema, a domain expert may also have an intuition of the nature of queries (to those regions) that would be of interest to users. For instance, the expert may know the fields for which users would like to ask range queries (rather than simple selections), which fields are ideal to sort all results by, what indirect relationships may be worth joining entities on (these are not explicitly specified in the schema), which numerical fields are sensible to sum up or average and so on. We introduced some postulates for form generation in Sec. 4.3, but it is not always obvious which of these to apply on a given database. Each postulate is implemented in the system and an expert, if available, would only have to turn one or more of them on (or off) at form creation time. For example, in the Jobsquery dataset, we observed two fields file_loc and post_date within job which are used to record a disk location and a posting date for a job listing respectively. While their properties made them seem highly queriable to our form generator, they were never once queried for by actual users. It is in situations like these that an expert can guide the system to neglect fields that are intuitively of little or no interest to users. Had this step been taken, we could have achieved 80% query support with only 12 fields instead of 14.

CHAPTER V

FORM REFINEMENT

We have shown in Chapter IV that the schema and the content of the database can be used to generate a set of forms for users to query the data. This set is generated without any knowledge of actual querying behavior. However, should actual user queries be available to the DBA, not all of which are supported by the forms generated initially, a mechanism to improve the interface by including support for the real queries can have a significant impact on user satisfaction. These queries are a direct indication of what questions users would like to ask the database, and if forms are to be made available, these are the very queries the forms should allow users to express.

In this chapter, we present a framework for generating forms in an automatic and principled way, given a database and a sample query workload. We show in Sec. 5.1 how a single query can be used to generate a form that supports it which includes how to design form layout and how to label individual fields. It is not desirable to create one form for each query of interest. This can make the task of form selection daunting, and can potentially make the interface cumbersome. A single form should be expressive enough to support multiple queries as well as slight variations of each. If there are queries in the set that are similar to each other, these queries should be able to generate a single form that can express all of them. This will enable us to divide the entire query set into clusters of similar queries, allowing us to only have to generate one form per cluster instead of one per query.

In Sec. 5.2, we introduce the notion of a *form query class* which represents a set of similar queries that can be posed using a single form. We formalize this notion of query *similarity* and use this to define a similarity *threshold* that forms the basis of query clustering. Needs of users change over time. In spite of the generalization described above, there will occasionally be queries that are unsupported by our interface. We discuss incremental modifications to the interface in Sec. 5.3. We have implemented our form generation system on a real database and evaluated it on a comprehensive set of query loads and database schemas. The experimental section (Sec. 5.4) that follows is based on the Timber native XML database that we have previously developed. While our implementation is XML-based, we believe it can be easily adapted to a relational environment and SQL-based querying, since the algorithm structure and system architecture are independent of the data model.

5.1 Form Generation

Creating a form for a given declarative query is a two-step process. First, we analyze the query to identify the various query operations which we translate into their corresponding form-elements. In the second step we arrange these elements in groups, label them suitably, and lay them out in a meaningful way on the form. An overview of the steps involved in this process can be obtained from Algorithm 6. Using this algorithm, the form generated for query Q (in Sec. 1.2) is shown in Figs. 3.2, 3.4 and 3.6. We discuss these two steps in detail.

5.1.1 Query Analysis

A standard query language parser is applied to the given query string to identify its operations and create a logical (algebraic) representation of the query. Each operation is then translated to its corresponding form-element (as defined in Sec. 3.1.1) and these elements together form the required form. The translation procedure for each query operation into its corresponding form-element is described below.

Selection

A *selection* operation has a filtering predicate that specifies a constraint to be satisfied by the result of a query issued using the form generated. This predicate can be directly mapped to a simple constraint-specification element (or a conjunctive disjunctive combination of constraint-specification elements) using the schema attribute and its associated entity. A constraint-specification element consists of the following form controls: a static field label denoting the schema attribute, a drop-down list of relational operators ($<$, $\leq$, $=$, $\geq$, $>$ and $\neq$) and a textbox for the user to fill in a value for the attribute. The $\neq$ comparison operator allows users to specify negation conditions in the form. Our query Q (in Sec. 1.2) has three selection predicates—on auction privacy, auction end date and item description. These are translated into constraint-specification elements in the input tree of the form (Fig. 3.1).

Projection

A *projection* operation contains a single schema attribute that is to be returned in a query's result. This attribute can be translated into a result-display element on the form. The result-display element is composed of a field label and a checkbox for users to specify if the desired query's result must include this attribute. In our example, all attributes of Item are projected. Notice the result-display element in the output tree (Fig. 3.3) and in the *Results* pane (Fig. 3.4).

Sorting

An *order-by* operation contains a single schema attribute which the result of a query is to be sorted on. This operation can be converted to a result-ordering element which contains a field label, a drop-down list for a user to choose the sort-order (ascending or descending) and a checkbox that determines whether or not this operation is desired.

Aggregate Function

A *group-by* operation is used to specify an aggregation of data. The resulting groups of tuples are often the basis for an aggregate computation. If the query contains an *aggregate function*, an aggregate-computation element is created with three user-specifiable inputs: the function itself, the schema attribute whose value is to be aggregated and the basis of grouping (scope) for the aggregation. The corresponding aggregate-computation element contains a drop-down list of aggregate functions (COUNT, SUM, MAX, etc.), a field label, a drop-down list of relational operators ($<$, $>$, =, etc.) and a textbox (if it is in the *input tree*) or just a checkbox (if it is in the *output tree*). The label contains the name of the element (or attribute) whose value is aggregated, with the grouping basis highlighted (it will be a prefix of the field label under our labeling scheme). But if the grouping basis is the same as the form-group containing the aggregate-computation element, no highlighting is necessary. In our example form, two such form-elements are created—one for average income of potential buyers, and the other for their average age (Fig. 3.4).

Join

A *join* operation includes the two participating entities, the attributes in their join condition, and the type of join. Once the join condition and the participating entities are determined, they are used to build a join-specification element and if necessary, an associated form-group for the elements associated with the related schema entities (as discussed in Sec. 5.1.2). A join-specification element is displayed as a set of controls that includes the labels of the two attributes that participate in the join condition, and a drop-down list of relational operators for the type of join (equijoin, non-equijoin, etc.). Our example query Q has two join conditions and hence two join-specification elements are created in the relationships tree (Fig. 3.5) and also in the form's *Advanced* pane (Fig. 3.6).

5.1.2 Form Structuring

In the query analysis phase, form-elements are created based on the data operations prescribed by the query. But these form-elements still need to be organized and labeled meaningfully in the form (initial labels are created for form-elements in isolation and can be improved based on how the form is organized). The first step in the form structuring phase is to identify the role each form-element plays and place it in the corresponding tree: *input*, *output* or *relationship* (and its associated pane). In Sec. 3.1.2, we stated that elements can be grouped either on the basis of query-dictated relationships (only in the *relationship* tree) or by schematic closeness (only in *input* and *output* trees). We now describe why this grouping is needed and how it can be accomplished.

Query-based grouping

Query-based grouping is *required* in order to be able to generate a query once the form is submitted. There are many ways in which the multiple form-elements specified can be combined but query-based grouping specifies the right way. Sometimes, it may be possible to determine a “minimal” (or simplest possible) relationship between the query elements for simple forms (for instance, if only one entity is present or if two or more entities are present that allow a natural join between them). In such cases, the system (or an uninitiated human) can *guess* what query the form should evaluate simply based on the individual form-elements. But for more complex forms, with multiple “correct” ways of grouping form-elements, query-based grouping (based on the original query used to generate the form) is needed because it preserves the original relationship(s) between entities enabling the system to generate the “correct” query for the given form.

The relationship tree is constructed using the relationships between entities specified by the query. The logical query representation has a tree-hierarchy and our relationship tree is created to structurally mirror the logical query. This is a design decision we make in order to preserve these entity-relationships for reasons stated above. We traverse the query in a top-down fashion (preorder traversal) and, in parallel, build the relationship tree. For every join operation node in the query tree, we create a new form-group if it involves an entity seen for the first time (if the two entities in the join were observed previously while generating this form, no additional form-group is created since only one form-group is needed per entity referenced in the query). Based on how form-groups are created, if a join j_1 is higher-up in the logical query representation than another join j_2, the form-group corresponding to j_1 is a parent (or ancestor) of the group corresponding to j_2. At the very beginning, while still at the top of the query tree, a root form-group is created that contains all form-elements encountered until a new group is created.

Schema-based grouping

In contrast to query-based grouping which is only done in the *relationship* tree, schema-based grouping occurs in the *input* and *output* trees with the purpose of making forms easier to understand for end-users. Schema-based grouping helps generate more concise labels for both form-elements and form-groups. In this phase of form generation, form-elements (except result-ordering elements) are organized into groups based on common schema ancestry. This means that all form-elements pertaining to a single schema-entity will be placed together on the form in a rectangular panel labeled using the name of that entity. There is a high likelihood that elements located close to each other in the schema of the database are also semantically related. Maintaining this closeness in the form makes it easier for users to relate them in a similar way. For instance, if a form is used to express queries involving two entities movie and actor, then all form-elements associated with the attributes of movie are placed in one panel (form-group), while the others which reference attributes of actor are placed in another panel that is visually separate from the panel for the movie entity.

The exception to this rule are the result-ordering elements which, for presentation reasons, we prefer to place all in one group. If results of a query are sorted on more than one field, the order between these fields is easier to interpret by a user if the fields are placed together in the form (in order of precedence) than if they are grouped by schema entity (even if tagged with a precedence value). Hence all result-ordering elements are placed in the root group of the output tree (order-preserved) and ignored while grouping by schema.

The first step in schema-based grouping is to analyze the schema attributes referenced by form-elements associated with the form's input. The absolute paths of these attributes are used to cluster them around their respective schema entities. After clustering the form-elements, a form-group is created for each cluster with two or more form-elements in it. It bears mentioning that a form-group can also contain other form-groups if these groups are associated with entities in the subtree of the entity associated with the first form-group. If a form-element is alone in a cluster, it is associated with a form-group created for its closest ancestor, or the root form-group if no other groups exist. Schema-based grouping of input form-elements results in the *input* tree (Fig. 3.1) and grouping of output form-elements produces the *output* tree (Fig. 3.3).

Labeling of Groups & Elements

Labeling elements and groups properly is important to convey the correct semantics of the form to the user. Labels must be descriptive, but not verbose. They must also be unique across the form to prevent ambiguity. The simplest way to label an element is to use its parent entity (if the data is relational) or its full schema ancestry (if the data is in XML). However, following this approach often leads to unnecessarily long labels. Instead, we use the groups created in the schema-based grouping phase to determine which ancestors of an attribute need to be included or excluded in every its label. Once related form-elements are grouped together in a form, it is enough just to label the top-most form-group fully and then label each child element or group relative to its parent.

In a query language like XQuery, queries may use wildcards that match multiple entities. In SQL on the other hand, each entity used in a query is required to be stated explicitly and unambiguously. Thus if a form is created based on one or more given queries in XQuery, its form-groups can have labels that refer to more than one entity. In such cases, we use the same wildcards in the form labels as well and display (in a non-obtrusive way) the list of all entities that match each such label (as a mouse-over tooltip, for instance).

Algorithm 6: Algorithm `GenerateForm`

```
Algorithm GenerateForm
Input: A query $Q$ (logical representation)
Output: A form $F$ (canonical structure)
Let $T_I$, $T_O$ and $T_R$ denote the input, output and relationship trees of the form respectively;
// Element construction and Grouping
Create a new form-group $g$ and add it to $T_R$;
foreach operation $o \in Q$ when traversed top-down do
    case $o$ is a “selection”
        Create a constraint-specification element using $o$;
        Put this constraint-specification element in $g$;
    case $o$ is a “projection”
        Create a result-display element using each projected attribute;
        Put these result-display elements in $g$;
    case $o$ is a “sort”
        Create a result-ordering element using each order-by attribute;
        Put these result-ordering elements in $g$;
    case $o$ is an “aggregate function”
        Create an aggregate-constraint-specification element or an aggregate-result-display element
        using the the group-by attribute, the grouping-basis and the aggregate function;
        Put this aggregate-computation element in $g$;
    case $o$ is a “disjunction”
        Create constraint-specification elements for each selection;
        Create a disjunction-element, put them in it and put it in $g$;
    case $o$ is a “join”
        Create a join-specification element using the two attributes of the join condition;
        Put this join-specification element in $g$;
        Create a new group $g'$ as a child of $g$ in $T_R$;
        Set $g \leftarrow g'$;
Let $E_I$ be the set of form-elements in $T_I$;
Let $E_O$ be the set of form-elements in $T_O$;
foreach form-element $e \in T_R$ do
    if $e$ is a “constraint-specification element”
    or $e$ is an “aggregate-constraint-specification element”
    or $e$ is a “disjunction element” then Put a copy of $e$ in $E_I$;
    else if $e$ is a “result-display element”
    or $e$ is a “result-ordering element”
    or $e$ is an “aggregate-result-display element” then
        Put a copy of $e$ in $E_O$;
Analyze the schema entity and attribute referenced by each form-element $e \in E_I$ and $E_O$ and create
form-groups (in $T_I$ and $T_O$ respectively) for each entity referenced more than once;
Within each tree, place each form-element in the form-group corresponding to its schema entity and
place each form-group in the form-group corresponding to its closest ancestor in the schema;
// Element and Group Labeling
foreach form-group $g \in T_I$ or $T_O$ do
    Label $g$ relative to its parent group (use absolute path if $g$ is root);
    foreach form-element $e \in g$ do Label $e$ relative to $g$;
```

5.2 Form Query Class

We now extend the form generation technique to design forms for an entire set of queries. Given a set of interesting queries, the naïve approach is to build one form for each of them. In this section, we describe our notion of similarity and how queries are classified. In the last section we showed how a form can be generated given a single query. But if we are provided an entire workload of queries, creating one form per query would produce a cumbersome interface. Instead, we can take advantage of the similarity between queries to produce a smaller set of forms that can still "cover" all queries in the workload. In this section we show how a set of queries can be partitioned into clusters, called *query classes* so that only one form needs to be generated per class. We define a notion of *similarity* between queries and use it to formulate a distance metric that can be used to cluster queries incrementally.

5.2.1 Similarity Analysis

Analyzing similarities across the queries (that expressed in a declarative language) directly is difficult because the same query can often be written in many different ways. Once we obtain their canonical form structures, inferring structural similarity between them can be done more efficiently.[1] We define two queries to be *structurally equivalent* if their canonical form representations are identical. Two queries are *structurally similar* if their *relationship* trees have the same form-groups and the same join-specification elements. Structurally dissimilar queries are never placed in the same query class as they cannot be expressed using the same form. This property can be used to screen queries from classes they cannot belong to.

A simple but effective measure of the similarity between two queries is the number of

[1]Determining query equivalence is a known hard problem, but our problem is simpler since we focus only on query structure.

form-elements that are shared between their canonical form structures, normalized by the total number of distinct form-elements in the two queries together. We define the distance between two queries based on this notion of similarity. The distance (or dissimilarity) between two queries is computed as one minus the number of common form-elements in their form representations divided by the total number of distinct form-elements in both combined. This measure is better known as the Jaccard distance [44] which is a metric (see [52] for proof) used to measure the dissimilarity between sample sets such as text documents.

Formula 10. (QUERY DISTANCE) *The* **distance** *from one query Q_1 to another, Q_2, which we denote by $d(Q_1, Q_2)$, is computed as one minus the number of form-elements in the canonical form structure of Q_1 (F_{Q_1}) that are also present in that of Q_2 (F_{Q_2}), divided by the total number of distinct form-elements in F_{Q_1} and F_{Q_2}.*

$$d(Q_1, Q_2) = 1 - \frac{|F_{Q_1} \cap F_{Q_2}|_e}{|F_{Q_1} \cup F_{Q_2}|_e}$$

Here $|F|_e$ is the number of form-elements found in the set F. The above computation results in a distance value that lies between 0 and 1. Other, more sophisticated, similarity measures may be defined without affecting any other technique described in this chapter. However, for concreteness, we restrict our discussion to the specific similarity score defined above.

5.2.2 Clustering

Queries to the same database, more often than not, have similarities between them that can be exploited to minimize redundancy. We define a *form query class* to be a set of similar queries that are covered by a single form. Given a technique for computing the distance between any pair of objects in a data set, it is straightforward to use any standard clustering algorithm to group together similar objects. We use a simple incremental algorithm that compares the canonical form structure of each query in turn with the canonical form structure of each query class created thus far. Each query is put in the class that is the most similar to it (i.e., at minimum distance). If all existing classes are at a distance greater than

Algorithm 7: Algorithm `ClusterQueries`

```
Algorithm ClusterQueries
Input: A set of queries $Q$
Input: Distance threshold $d_{threshold}$
Output: A set of query classes $C$
$C \leftarrow \{\}$;
foreach $q \in Q$ do
    $d_{min} \leftarrow 1$;
    Convert $q$ to its Canonical Form Structure $F_q$;
    foreach $c \in C$ do
        if q and c are structurally different then continue;
        if $d(F_q, F_c) < d_{min}$ then
            $d_{min} \leftarrow d(F_q, F_c)$;
            $c_{min} \leftarrow c$;
    if $(d_{min} > d_{threshold})$ or $(|F_q \cup F_{c_{min}}|_e > FCT)$ then
        Create a new class $c'$;
        Add $q$ to $c'$;
        $F_{c'} \leftarrow F_q$;
        Add $c'$ to $C$;
    else
        Add $q$ to $c_{min}$;
        Regenerate $F_{c_{min}}$;
```

a specified *distance threshold* from the given query, a new class is created and its canonical form structure is simply the canonical form structure of that query.

The computational complexity of our clustering algorithm (Algorithm 7) is $O(|Q|^2)$ where $|Q|$ is the cardinality of the query set Q. Queries are analyzed one at a time and each query is compared with all query classes created before its arrival (for previously analyzed queries). Since the number of such classes can be, at worst, linear in the number of queries analyzed, the running time of our algorithm is quadratic in the total number of queries. The order in which queries are analyzed also matters. Incremental clustering algorithms, in general, are affected by the order in which objects are processed [74]. Each such algorithm proceeds typically in a hill-climbing fashion which yields local minima rather than global minima [30]. The choice of the distance threshold parameter impacts the sizes of the classes created, and thus the form complexity. See Sec. 5.2.4 for a more detailed discussion of this topic. Query clustering produces a set of query classes each of which can be used to build a form that can express all its queries.

An Example

A simple example of a query set and the form generated using it is shown below. The XQuery expressions for three simple queries (to the XMark schema) are as follows.

Q1: *List the names and ages of male bidders in all auctions with a current highest bid greater than 10.*

```
for $p in document(“auction.xml”)//person
for $o in document(“auction.xml”)//open_auction
where $o/bidder//@person = $p/@id
    and $o/current > 10 and $p//gender = “male”
return {$p/name} {$p//age}
```

Q2: *List the start and end times of all auctions with a bidder under the age of 18.*

```
for $a in document(“auction.xml”)//open_auction
for $b in document(“auction.xml”)//person
where $a/bidder//@person = $b/@id
    and $b//age < 18
return {$a//start} {$a//end}
```

Q3: *What are the incomes of people bidding in auctions with an initial bid of 1000?*

```
for $x in document(“auction.xml”)//open_auction
for $y in document(“auction.xml”)//person
where $x/bidder//@person = $y/@id
    and $x/initial = 1000
return $y//@income
```

Their respective canonical form structures, given by F1, F2 and F3, are as follows.

F1 $= \{c(\text{OpenAuction:Current}), c(\text{Person:Gender})\}_{\mathbf{I}},$
$\{r(\text{Person:Name}), r(\text{Person:Age})\}_{\mathbf{O}}$

$\{\{c(\text{OpenAuction:Current}), c(\text{Person:Gender}),$

$r(\text{Person:Name}), r(\text{Person:Age}),$

$j(\text{OpenAuction:Person}, \text{Person:Id})\}_1\}_{\mathbf{R}}$

F2 $= \{c(\text{Person:Age})\}_{\mathbf{I}}$

$\{r(\text{OpenAuction:Start}), r(\text{OpenAuction:End})\}_{\mathbf{O}}$

$\{\{r(\text{OpenAuction:Start}), r(\text{OpenAuction:End}),$

$c(\text{Person:Age}), j(\text{OpenAuction:Person}, \text{Person:Id})\}_1\}_{\mathbf{R}}$

F3 $= \{c(\text{OpenAuction:Initial})\}_{\mathbf{I}}, \{r(\text{Person:Income})\}_{\mathbf{O}}$

$\{\{c(\text{OpenAuction:Initial}), r(\text{Person:Income}),$

$j(\text{OpenAuction:Person}, \text{Person:Id})\}_1\}_{\mathbf{R}}$

The distances between queries are $d(Q2, Q1)$ = 0.875, $d(Q3, Q1)$ = 0.857 and $d(Q3, Q2)$ = 0.833. For a high threshold (say 0.9), the three queries are similar enough that they collectively produce just one form, F_R (Fig. 5.1):

$$F_R = (F1 \cup F2) \cup F3$$

FR $= \{c(\text{OpenAuction:Current}), c(\text{OpenAuction:Initial}),$

$c(\text{Person:Gender}), c(\text{Person:Age})\}_{\mathbf{I}},$

$\{r(\text{OpenAuction:Start}), r(\text{OpenAuction:End}),$

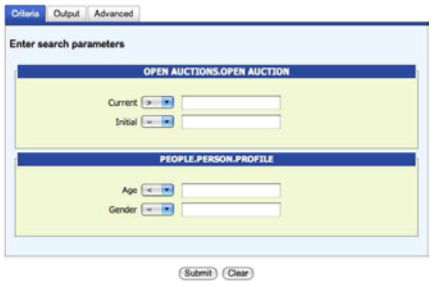
$r(\text{Person:Name}), r(\text{Person:Age}), r(\text{Person:Income})\}_{\mathbf{O}},$

$\{\{c(\text{OpenAuction:Current}), c(\text{OpenAuction:Initial}),$

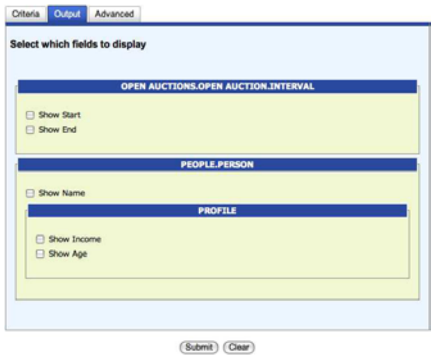
$c(\text{Person:Gender}), c(\text{Person:Age}), r(\text{OpenAuction:Start}),$

$r(\text{OpenAuction:End}), r(\text{Person:Name}), r(\text{Person:Age}),$

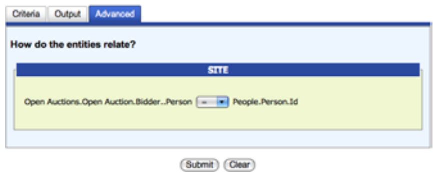
$r(\text{Person:Income}), j(\text{OpenAuction:Person}, \text{Person:Id})\}_1\}_{\mathbf{R}}$



(a) Criteria Pane



(b) Output Pane


Criteria
Output
Advanced
How do the entities relate?
SITE
Open Auctions.Open Auction.Bidder..Person
=
People.Person.Id
Submit
Clear

(c) Advanced Pane

Figure 5.1: Form generated for Queries Q1-Q3

5.2.3 Expressivity of a Form

We define the *expressivity* of a form to be the number of distinct queries that can be expressed using it[2]. It takes only one query to generate a form, but the expressivity of that form need not be limited to that one query. The query dictates what parameters are available to the user but it is entirely up to the user to decide which and how many to use. This property holds true regardless of data model.

Consider, for instance, the following query which can also be expressed using the form shown in Figs. 3.2, 3.4 and 3.6, generated for query Q in Sec. 1.2.

Q′: *Name all privately auctioned items and the age of the youngest prospective buyer for each of them.*

This new query can be posed simply by ignoring the constraints on *end date*, *description*

[2]Two queries are equivalent (not distinct), for purposes of this count, if they differ only in the values of constants used. Two queries that are structurally distinct may still be semantically equivalent. Determining such equivalence is hard and we do not attempt to solve this problem. Rather we will consider them to be distinct queries. In practice, since such equivalence is not frequent, we just accept that our count here is a slight over-estimate.

and *quantity*, the aggregation based on *income* and the result ordering. The new XQuery expression is:

```
for $o in document("auction.xml")//open_auction
for $i in document("auction.xml")//item
let $p := for $pi in document("auction.xml")//person
          where $pi//interest/@category=$i//@category
          return $pi
where $o/itemref = $i/@id and $o/privacy = "private"
return {$i/name} {MINIMUM($p//age)}
```

The form also gives us a choice of aggregate functions to compute on *age*, and here we opt for the function MINIMUM(). This simple example shows how a potentially large number of queries can be expressed using a relatively small set of forms, even if those queries did not belong to the base query set (i.e., the set of queries that were used to generate the forms).

The expressivity of a form can be computed mathematically. The number of queries that can be expressed using a form is a function of the number and type of its elements.

A constraint-specification element can either be used or ignored. If used, it is either an equality or an inequality. It can involve a complex function such as a string (e.g. $contains()$, $startsWith()$, etc.) or a date function (e.g. $daysBetween()$) that is supported by the query execution engine. If R denotes the set of available relational operators and/or complex functions ($R = \{<, \leq, =, \geq, >, contains(), ...\}$), then the dependence of a form's expressivity E_f on its n_c constraint-specification elements is given by:

$$E_f \propto (|R| + 1)^{n_c}$$

Similarly, a result element (result-display or result-ordering) can either be selected or unselected. The dependence of a form's expressivity on its n_r result elements is given by

$$E_f \propto 2^{n_r}$$

An aggregate-computation element has an additional variable: the aggregate function. If A is the set of functions (A = {COUNT(), SUM(), AVERAGE(), MAXIMUM(), MINIMUM(), ...}), the dependence of a form's expressivity on its n_a aggregate-computation elements can be expressed as:

$$E_f \propto (2|A|(|R|+1))^{n_a}$$

A disjunction element is a set of constraint-specification elements, hence its effect on the form's expressivity is simply

$$E_f \propto \prod_{i=1}^{n_d} n_{dc_i}|R|+1$$

where n_d is the number of disjunction elements and n_{dc_i} is the number of constraint-specification elements in the ith disjunction element. If a form contains a join-specification element, the join will be present in the resulting query. It is needed to specify the relationship between the entities it references. However, the possible types of joins (equijoin, non-equijoins) is subset of R (the set of relational operators) and is variable. We denote this subset as J = $\{<, \leq, =, \geq, >\}$. If n_j is the number of join-specification elements in a form,

$$E_f \propto |J|^{n_j}$$

Combining the above relationships,

$$E_f = (|R|+1)^{n_c} \cdot 2^{n_r} \cdot (2|A|(|R|+1))^{n_a} \cdot (\prod_{i=1}^{n_d} n_{dc_i}|R|+1) \cdot |J|^{n_j} - 1$$

We subtract 1 just to account for the disallowed case where none of the form-elements is selected. The expressivity of a set of forms is equal to the sum of the expressivities of the forms that belong to it, adjusted for overlaps, if any.

5.2.4 Form Complexity

As the number of queries assigned to a class grows, the number of form-elements and form-groups in its canonical representation also grows. This can make the form generated for that class very large and complex. A user must be able to view any form, be able to decide if it fulfills his or her information need, and finally use the form to specify the desired query. The complexity of these tasks depends on the complexity of the form. Hence we would like to limit the complexity of generated forms so that they remains usable.

Formula 11. (FORM COMPLEXITY) *We define the* **form complexity** *of a form as the number of form-elements in it.*

$$C_F = |F|_e$$

More complex measures may be used such as weighting form-elements by their type instead of treating all elements the same. For instance, we could weight a join-specification element higher than a constraint-specification element since the former is harder to comprehend for a casual user. But for the sake of concreteness, we adopt simple element-cardinality as our form-complexity metric in this paper.

Keeping forms simple may suggest that we force all forms to have very low complexity. However, there are two disadvantages to this approach. Form complexity affects the number of forms generated which affects the ease of form selection for a user. The simpler the forms, the more of them there are likely to be (to still be able to cover all queries in the given workload). This can make it very hard for the user to choose the right one to express a desired query. Second, an interface having several simple forms is much less expressive than one that has a few complex forms (if both are generated using the same query set). This means that the former is likely to have much better query coverage (of queries outside the base query set) than the latter. To see this, consider a complexity threshold of 5 on a form-set F that has 10 result elements. In this case, there would be 2 forms each having 5 result elements. Their combined expressivity would be

$$E_F = \sum_{f \in F} E_f = (2^5 - 1) + (2^5 - 1) = 62$$

On the other hand, had the threshold been 10 or greater, there would only be 1 form. Its expressivity would be

$$E_F = (2^{10} - 1) = 1023$$

With these effects in mind, we would like to be able to generate forms that are neither too simple nor too complex. So we set a trade-off threshold called the *form complexity threshold* (FCT) that no form may exceed. Note that this threshold may be unenforceable if set too low, or if some of the queries involved are too complex. If there is a single query with complexity that exceeds the FCT, then no clustering can satisfy the FCT. Barring this scenario, FCT is satisfied by splitting a query cluster covered by a complex form into smaller clusters covered by simpler forms.

5.3 Incremental Form Maintenance

In Chapter IV, we created a set of forms using the schema and content of the database. This interface can be refined using a list of queries posed by users. Initially, our *base query set* is empty. Each query observed in chronological order is treated as an *addition* to this base set. A query can only alter one existing form, or cause the creation of a new form. Hence we can associate every query to exactly one form. This is necessary if we wish modify existing forms over time if certain forms become too complex due to the number of queries associated with them or if it makes sense to prune or remove them because their associated queries show decreased or zero usage. This allows the system to evolve with changing query behavior throughout its lifetime.

It is essential to be able to modify the base set incrementally, without having to start over. Changes include addition and removal of one or more queries. Modification of an existing query is treated as deleting the existing query from the base set and inserting the modified query into it. We now discuss these scenarios and their ramifications.

5.3.1 Query Addition

When a query is added to the base set, the system must identify which class it belongs to. This is not different from initial query classification to generate the first set of forms. Once the appropriate class is determined, the query is added and the form associated with that class is re-generated, if necessary. Following a large number of query additions, a form might become too complex. If this happens, we split the class into smaller subclasses.

Form Splitting

A complex form is typically replaced by two or more simpler forms by re-classifying only those queries belonging to the associated form class. If the number of form-elements is greater than the pre-specified complexity threshold, the associated queries are re-examined after increasing the similarity threshold for that class alone. This threshold is raised until the queries can be split into two or more classes. Forms are then generated for the new classes, and the threshold is recorded.

5.3.2 Query Deletion

If the need is felt to remove a query from the base set, only the class and form associated with the deleted query are affected. Form maintenance includes maintaining statistics of form usage that can be periodically examined. These can help determine if the system can be made more efficient by removing unused or scarcely used queries. Upon deletion of one or more queries in a single class, the form associated with that class is re-generated.

Form Merging

Following a query deletion, the opportunity arises to reduce the number of forms by merging two or more of them without violating threshold constraints. In the event of a query deletion, the cluster to which the deleted query belonged is completely deleted, and the other queries in it are re-classified. If the queries can now merge with other forms without violating complexity thresholds, we will have accomplished a form merge. Otherwise, the original form, possibly with some simplification, is regenerated.

5.4 Evaluation

We used the system described above to evaluate our form generation methodology experimentally. In our evaluation, we did not use the techniques proposed in Chapter IV to generate an initial set of forms. Forms were generated using queries alone.

We decided on three metrics to perform this evaluation:

Size of form-set: This denotes the number of forms a user has to choose from in order to pose a query to the database. A central hypothesis for our effort is that this number is quite small in practice since many users issue similar queries.

Complexity of individual forms: Form complexity indicates how quickly or easily a user can understand and use it.

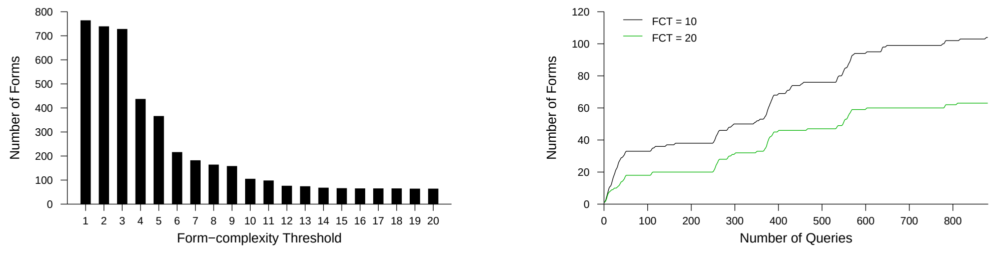
Expressivity of the interface: While a complex form might overwhelm a user, a simple form might not be expressive enough to pose that user's query at all. We measure the combined expressivity of all forms generated.

To evaluate system performance we conducted experiments with three datasets. First, we used queries found in the Geoquery training corpus [6][76] which are posed against a geographical database. Second, we used a trace history of queries posed to web databases hosted on our EECS department's database server. Finally, we also made use of queries from the XMark benchmark, which are fairly disjoint (disobeying our central hypothesis), and built a set of forms to express those queries.

5.4.1 Geoquery Training Corpus

The Geoquery database [6] contains geographical information about the United States such as states, cities, roads, rivers, etc. The information is stored as Prolog assertions (about 800 assertions) and was designed and populated primarily to test a natural language query interface. In order to use this dataset for our experiments, we translated the data into XML and extracted a schema definition in XSD. The query set consists of 880 natural language questions to the dataset posed by real users of the database's publicly accessible web interface and also by undergraduate students in the Computer Science department at the University of Texas at Austin. With the help of the deduced schema, we translated these English queries into XQuery so that we could use them to generate forms.

Observations: We analyzed the effect of Form-Complexity Threshold (FCT) on the number of forms generated. We see in Fig. 5.2(a) that for a threshold of 10 form-elements per form, just over 100 forms can cover all 880 queries.

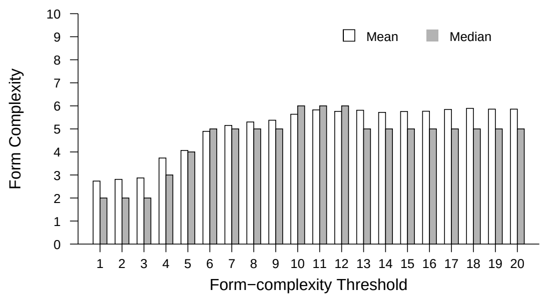


(a) Effect of Threshold on Form-set Size (b) Incremental Form Generation

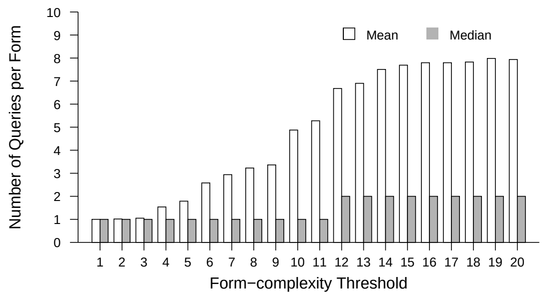
Figure 5.2: Forms for the Geoquery Training Corpus

Given the full workload, it is more interesting to understand the dynamics of form creation. Therefore, we created the forms incrementally, starting from scratch. Fig. 5.2(b) shows the growth in the size of the form-set with each new query analyzed. We observed an initial learning period in which a large number of forms are created, but following this, the rate of form creation dropped significantly. With a complexity-threshold of 10, the first 375 queries required the creation of 60 forms, while the next 505 queries only required 44 more forms. With a threshold of 20, only 40 forms were needed to express the first

375 queries, and only 23 more for the remaining 505. As we can see from the figure, the curve flattens towards the end of the workload, as fewer and fewer new queries are found substantially dissimilar to ones seen before. The form generation cost is fairly low, ranging from 0.8 seconds to 12.3 seconds (for all 880 queries). It varies depending on the complexity-threshold because the threshold determines the number of forms generated and materialized (disk I/O). Finally, we show, in Fig. 5.3, the complexity of individual forms and their query coverage as the complexity-threshold is raised. We observe that the query distribution among forms is close to Zipfian, i.e., most of the queries are covered by a few forms keeping the median coverage low.



(a) Effect of Threshold on Form Complexity



(b) Query Distribution among Forms

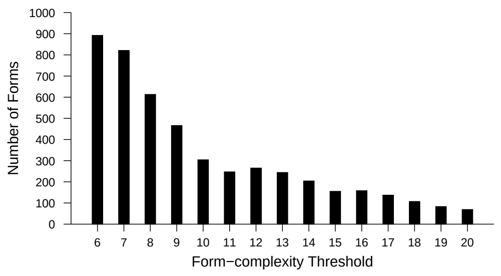
Figure 5.3: Effect of Threshold on Forms for the Geoquery Training Corpus

5.4.2 EECS Query Trace

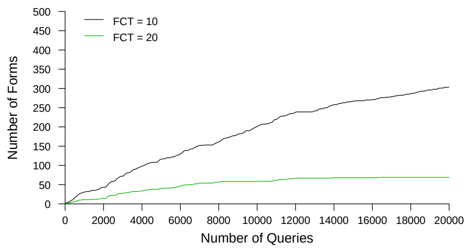
We now report on experiments run with another real workload, obtained from a log of all queries posed to web databases hosted on a MySQL server in our department in a 2-day period. We converted these SQL queries to XQuery expressions, creating an appropriate XML Schema Definition (XSD) corresponding to the relational schema of the databases accessed. (We had to go through this trouble for both this and Geoquery datasets since we could not obtain an XQuery query log from anywhere and our system only handles XML/XQuery).

Workload Characterization: We analyzed the first 20,000 queries in the workload which involved 17 different databases (115 different tables) that were accessed by users over the approximately 2-day period.

Observations: Fig. 5.4(a) shows the effect of the threshold (FCT) on the size of the generated form-set. The results do not differ much from those for the previous experiment — fewer than 70 forms are enough to cover as many as 20,000 queries if the threshold on form-complexity is set to 20 elements. We analyzed the incremental growth of the form-set for this dataset as well and, as before, performed this experiment with two different complexity-thresholds: 10 form-elements-per-form, and 20 form-elements-per-form. We observe a steep curve at the early iterations followed by a a more stabilized set of forms (Fig. 5.4(b)). With a complexity-threshold of 10, the first 10,000 queries required the creation of 202 forms, while the next 10,000 queries only required 102 more forms. With a threshold of 20, only 59 forms were needed to express the first 10,000 queries, and only 10 more for the other 10,000.



(a) Effect of Threshold on Form-set Size



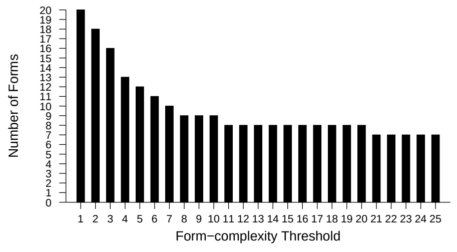
(b) Incremental Form Generation

Figure 5.4: Forms for the EECS Query Trace

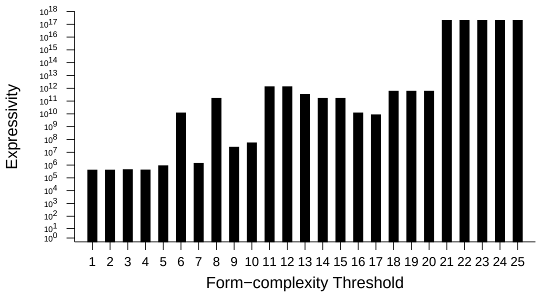
5.4.3 XMark Benchmark

The XMark benchmark was designed to test the various capabilities of an XQuery implementation. The queries are broad in scope and bear little similarity to one another. Such a setup is far from ideal for our system, as we expect that users of a real database often have interests similar to other users, and tend to pose queries that have parts in common with queries posed by them. However, we saw this diverse query set as an opportunity to test the worst case performance of our interface-generator.

Observations: The benchmark has 20 queries in total, fairly tiny compared to a real query log. All 20 queries were posed to the same schema, that of an auction-house database, consisting of data pertaining to live and completed auctions, people who sell or bid on auctions, items that are auctioned, etc., and their inter-relationships. We ran our form-generator on this query set for varying thresholds on form-complexity, and our results are shown in Fig. 5.5 and Fig. 5.6.



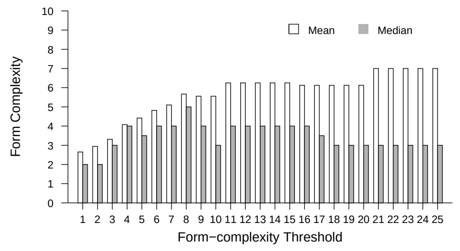
(a) Effect of Threshold on Form-set Size



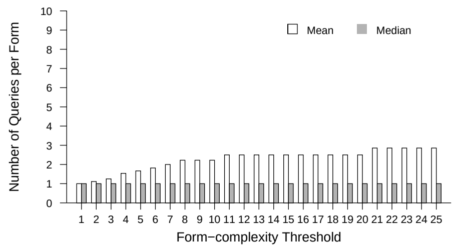
(b) Expressivity of Forms

Figure 5.5: Forms for the XMark Benchmark

Despite the high variance between queries, we only needed to create 10 forms for a reasonable upper bound of 7 elements per form. We computed the expressivity of the set of forms generated for the XMark query-set for varying thresholds on form-complexity and observed, as expected, that the number of queries expressible using the form-set increased when the threshold was raised. It was interesting to note that on placing a reasonable bound of 10 elements-per-form, a set of 9 forms that were generated using only



(a) Effect of Threshold on Form Complexity



(b) Query Distribution among Forms

Figure 5.6: Forms for the XMark Benchmark

20 queries could be used to express close to 50 million different queries. The expressivity does not go up smoothly as the FCT is increased. Depending on precisely which forms get retired and which new forms get pressed into service, there may even be a small drop in expressivity as FCT is increased by one. However, the overall trend in the relationship between the number of form-elements and expressivity is exponential, in accordance with our theoretical analysis.

5.4.4 Discussion

Both the Geoquery Training Corpus (GTC) and EECS Query Trace (EQT) are both collections of queries posed by real users. However, there are some interesting differences in querying behavior of users in the two scenarios that are evident from the forms our system generates to express them. First, the first 100 queries of GTC required as many as 33 forms at FCT=10 (104 forms were needed for all 880 queries). On the other hand, the first 100 queries of EQT, only required 4 forms with the same complexity threshold (304 forms were needed for all 20,000 queries). This difference indicates that the GTC queries are more diverse than the EQT queries because they require more forms to express the same number of user queries. In fact, at FCT=20, both query sets can be completely supported using an almost identical number of forms (63 for GTC, 69 for EQT) in spite of the large difference in the number of queries (880 for GTC, 20,000 for EQT).

Second, we observe that the curves corresponding to FCT=10 and FCT=20 are almost parallel for the GTC query set while they diverge over time for EQT queries. This observation suggests that EQT queries have greater skew, i.e., there are a larger number of queries that are similar to each other (even though they cannot all be expressed using a single form) than in GTC queries. A single complex form can often replace several (similar) simpler forms and an increase in FCT (from 10 to 20) enables more such complex forms to be generated (replacing simpler forms). The extent to which simpler forms are replaced is much higher in the EQT query set. This is evident from the bigger drop in the number of forms generated once the complexity threshold (FCT) is raised.

While our experiments show how complexity thresholds affect the size and expressivity of the generated form-set, their results are of little use if they don't provide any meaningful insight. The asymptotic nature of Figs. 5.2(a), 5.4(a), and 5.5(a) helps us choose ideal values for this threshold in the 3 scenarios to best balance complexity versus size. Obviously, this is a design choice that depends heavily on the distribution of queries, but having analyzed several disparate workloads, we recommend a DBA to set the threshold at about 12-15. Beyond this range, increased form complexity shows little gain in terms of form set size. Expressivity also shows a similar trend beyond a certain level of form complexity (Fig. 5.5(b)) further strengthening the case for this trade-off point.

CHAPTER VI

FORM CUSTOMIZATION

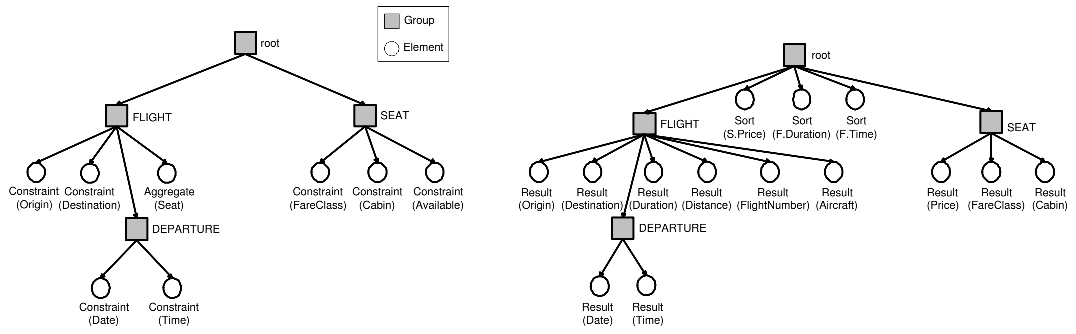
A form-based interface generated using the schema, the data, and any query set available, may still have users who cannot express a desired query using it. Instead of another preemptive interface improvement step, we decided to give users the ability to customize forms in the interface to suit their needs.

Unlike other form editing tools and guided query generators, we do not expect our users to create forms from scratch, i.e., a null starting point. The user starts with a form that approximates the desired query. He or she then edits the form to obtain a modified form that precisely captures the desired query. To draw an analogy, forms in an interface can be viewed like pizzas at a pizzeria. Just as the pizzeria may offer several pizzas with predefined sets of toppings (based on expectations of customer preferences), each form has a predefined set of query parameters decided by the data provider (based on expectations of user needs). It is possible that a customer would like to customize a pizza (by adding, removing or replacing one or more toppings) as it is possible that a user would like to customize a form (by adding, removing or replacing one or more form fields). In either scenario, given the available list of options (toppings in a menu or attributes in a schema), the range of possible combinations (pizzas or forms) is extremely large. But with a reasonably good set of starting points, the ability to customize helps satisfy all users without knowing, in advance, exactly what they want.

For this scheme to make sense, it is necessary that form editing not require any special knowledge on the part of the user, either about formal query specification or about the database schema. Our proposal is that form editing itself also be carried out as a form filling process. Specifically, we develop a form manipulation language comprising a small set of operators that take one or more forms as input, and produce corresponding forms as output. The desired form edit can then be written as an expression in this language. If we are able to provide a form filling interface that supports construction of such form edit expressions, then we will have the desired form editing process.

The expressive power of a form modification language should be measured not in terms of what the resulting forms can express, but in terms of how much modification can be expressed using this language. The actual query expressivity and specification process depend on an external data manipulation algebra that is treated as an argument to the form manipulation model that we propose. The more expressive the operators in this data model are, the higher the complexity of the queries that the forms can express. Since the addition and deletion of any data manipulation operator in any position are allowed, the form modification language can express any desired change within the bounds of the expressive power of the underlying data manipulation language. It is even possible to "modify" an empty form to define a desired form from scratch. However, the engineering design of this technique makes it best suited for small changes to complex query forms.

We introduce our form manipulation language in Sec. 6.1, which describes the operations that can be performed to customize forms. Based on this, we present, in Sec. 6.2, our form editor built on top of this manipulation language. We discuss briefly, in Sec. 6.3, the provision for advanced users to modify forms to a greater extent than typical form users to demonstrate the expressivity of our interface. The experimental section that follows (Sec. 6.4), presents the results of a controlled user study we conducted to evaluate the system's effectiveness. While our current implementation is XML-based, we believe our technique can be easily adapted to a relational environment and SQL-based querying, since the form expression language, the form generation algorithm, and system architecture are all independent of the data model.



(a) Input Tree (b) Output Tree

Figure 6.1: Logical Representation of a Form

6.1 Form Manipulation

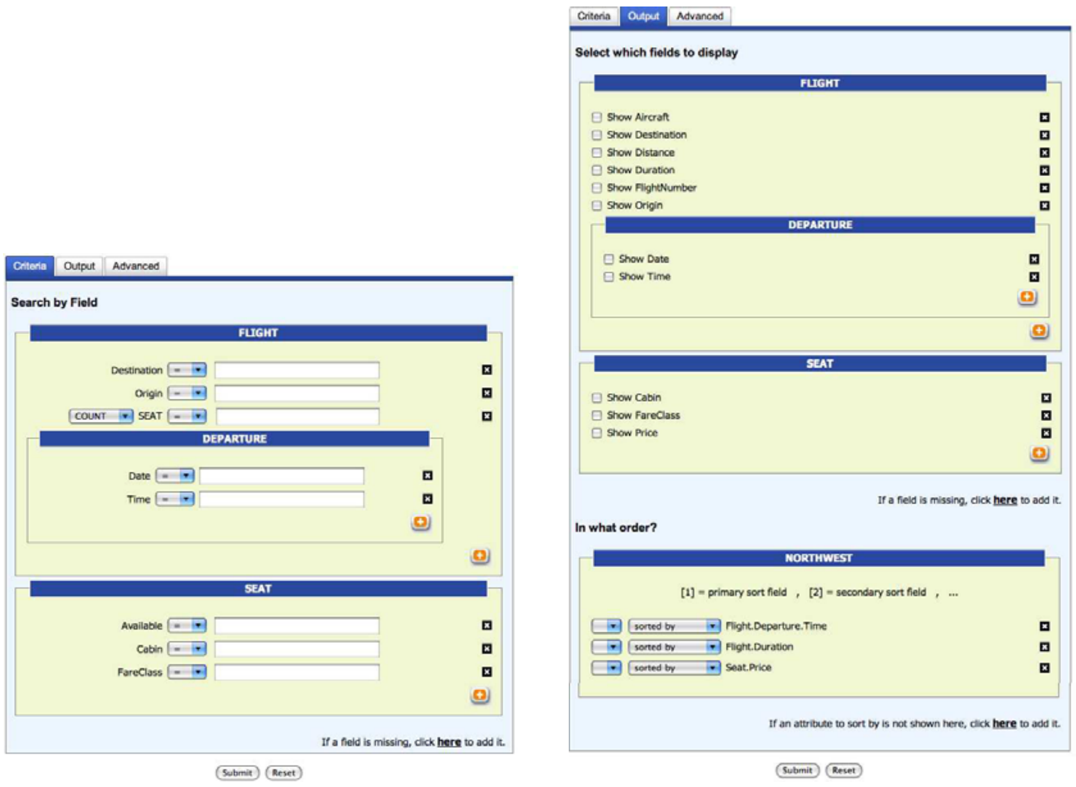
We present a set of possible form manipulation operators such that the result of any operation involving one or more forms is a new form, i.e., the language is closed with respect to any form operator. This form manipulation language is independent of the data manipulation operators that correspond to form elements. We present these operations in abstract form here, and show how they are realized in Sec. 6.2.

6.1.1 Form Operators

We use the notation F^t to denote the t tree of the form F, where $t \in \{\mathbf{I}, \mathbf{O}, \mathbf{R}\}$. As an example, we start with the form shown in Figs. 6.1 and 6.2 which is based on the advanced search form found at NorthwestAirlines.com for one-way flights (Fig. 1.4). The input and output trees in our version of this form are:

$$F_0^{\mathbf{I}} = \{\{c(\textsf{F.o}), c(\textsf{F.de}), a(\textsf{F.s}, \textsf{N.f}), \{c(\textsf{D.d}), c(\textsf{D.t})\}_2\}_1, \{c(\textsf{S.c}),$$
$$c(\textsf{S.fc}), c(\textsf{S.a})\}_3\}_{\mathbf{I}}$$
$$F_0^{\mathbf{O}} = \{\{r(\textsf{F.o}), r(\textsf{F.de}), r(\textsf{F.du}), r(\textsf{F.di}), r(\textsf{F.fn}), r(\textsf{F.a})\}_1, \{r(\textsf{S.c}),$$
$$r(\textsf{S.fc}), r(\textsf{S.p}), s(\textsf{S.p})\}_2, s(\textsf{F.du}), s(\textsf{D.t})\}_{\mathbf{O}}$$

We have used initials to identify elements in the above expressions. For example, looking at Fig. 6.1(a), we can recognize that F.o is Flight:Origin, S.fc is Seat:FareClass, etc.


Criteria | Output | Advanced

Search by Field

FLIGHT

Destination =

Origin =

COUNT SEAT =

DEPARTURE

Date =

Time =

SEAT

Available =

Cabin =

FareClass =

If a field is missing, click **here** to add it.

Submit | Reset

Criteria | Output | Advanced

Select which fields to display

FLIGHT

- Show Aircraft
- Show Destination
- Show Distance
- Show Duration
- Show FlightNumber
- Show Origin

DEPARTURE

- Show Date
- Show Time

SEAT

- Show Cabin
- Show FareClass
- Show Price

If a field is missing, click **here** to add it.

In what order?

NORTHWEST

[1] = primary sort field , [2] = secondary sort field , ...

sorted by Flight.Departure.Time

sorted by Flight.Duration

sorted by Seat.Price

If an attribute to sort by is not shown here, click **here** to add it.

Submit | Reset

(a) Criteria Pane (b) Results Pane

Figure 6.2: Visual Representation of a Form

Form-element Insertion (λ)

This operator is used to add a new form-element to an existing form. It requires as input a reference to the group under which the element is to be added and the element itself. Symbolically we can express this operation as $\lambda_{(e,g)}(F^t)$.

$$\begin{aligned} F_1^{\mathbf{O}} &= \lambda_{(a(\mathsf{S},\mathsf{S.p}),2)}(F_0^{\mathbf{O}}) \\ &= \{\{r(\mathsf{F.o}), r(\mathsf{F.de}), r(\mathsf{F.du}), r(\mathsf{F.di}), r(\mathsf{F.fn}), r(\mathsf{F.a})\}_1, r(\mathsf{S.c}), \\ &\quad \{r(\mathsf{S.fc}), r(\mathsf{S.p}), s(\mathsf{S.p}), a(\mathsf{S}, \mathsf{S.p})\}_2, s(\mathsf{F.du}), s(\mathsf{D.t})\}_{\mathbf{O}} \end{aligned}$$

Form-element Deletion (ϕ)

This operator is used to remove an existing form-element from a given form. Unlike element insertion, this does not require a group reference if the element can be uniquely identified. This operation is written as $\phi_e(F^t)$. Continuing our example,

$$\begin{aligned}F_1^{\mathbf{I}} &= \phi_{c(\mathsf{D.t})}(F_0^{\mathbf{I}}) \\ &= \{\{c(\mathsf{F.o}), c(\mathsf{F.de}), a(\mathsf{F.s}, \mathsf{N.f}), \{c(\mathsf{D.d})\}_2\}_1, \{c(\mathsf{S.c}), \\ &\quad c(\mathsf{S.fc}), c(\mathsf{S.a})\}_3\}_{\mathbf{I}}\end{aligned}$$

Form-element Move (ψ)

This operator is used to move an existing form-element from one group to another in a given form. The element and the target group must be specified and the operation is written as $\psi_{e,g}(F)$.

$$\begin{aligned}F_2^{\mathbf{I}} &= \psi_{(c(\mathsf{D.d}),1)}(F_1^{\mathbf{I}}) \\ &= \{\{c(\mathsf{F.o}), c(\mathsf{F.de}), a(\mathsf{F.s}, \mathsf{N.f}), c(\mathsf{D.d}), \{\}_2\}_1, \{c(\mathsf{S.c}), \\ &\quad c(\mathsf{S.fc}), c(\mathsf{S.a})\}_3\}_{\mathbf{I}}\end{aligned}$$

Form-group Insertion (Λ)

As its name suggests, this operator inserts a form-group into a given form and it requires specifying the group and optionally, its parent group which must already exist on the form. If the parent group is not specified, the group is inserted into the root (outermost) group of the form. For example, if $F_0^t = \{\{e_1\}_{g_1}\}_t$, adding a new group g_2 (to the root group) would create the new form:

$$F_1^t = \Lambda_{(g_2)}(F_0^t) = \{\{e_1\}_{g_1}, \{\}_{g_2}\}_t$$

Form-group Deletion (Φ)

A form-group can be removed from a form much like a form-element, but concern arises when the group still contains elements. We choose to allow the deletion of non-empty groups in the language, and leave it to the implementation to decide whether a group-delete command automatically triggers element-delete commands for each element present in it or simply relocates them.

$$\begin{aligned} F_3^{\mathbf{I}} &= \Phi_2(F_2^{\mathbf{I}}) \\ &= \{\{c(\mathsf{F.o}), c(\mathsf{F.de}), a(\mathsf{F.s}, \mathsf{N.f}), c(\mathsf{D.d})\}_1, \{c(\mathsf{S.c}), c(\mathsf{S.fc}), \\ &\quad c(\mathsf{S.a})\}_3\}_{\mathbf{I}} \end{aligned}$$

Form-group Move (Ψ)

A form-group can be moved from one group to another much like a form-element, but like deletion, if the group is non-empty, all its elements must be moved with it. Only the target group and the group itself are specified. For example, if $F_1^t = \{\{e_1\}_{g_1}, \{\}_{g_2}\}_t$, moving group g_2 into group g_1 would create the new form:

$$F_2^t = \Psi_{(g_2,g_3)}(F_1^t) = \{\{e_1, \{\}_{g_2}\}_{g_1}\}_t$$

Form Merge ($\bowtie$)

Forms can be combined using the binary form-merge operator. It performs a shallow merge that results is a single form with all the elements and groups of the operand forms heaped together. For example, if $F_1^t = \{\{e_1\}_{g_1}\}_t$ and $F_2^t = \{\{e_2\}_{g_2}\}_t$,

$$F_1^t \bowtie F_2^t = \{\{e_1\}_{g_1}, \{e_2\}_{g_2}\}_t$$

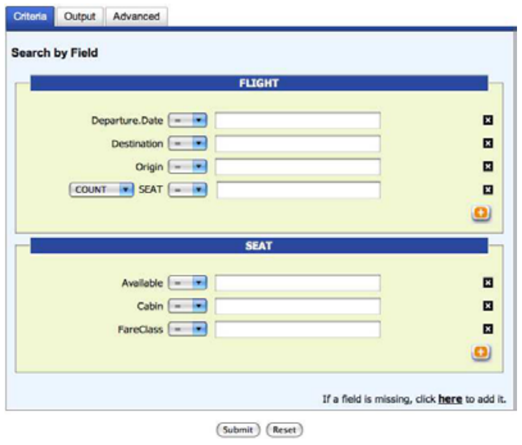
Having a merge operator allows the system to combine existing forms which enhances their reusability and minimizes duplication of effort on the part of a user, or even multiple users with similar querying interests.

Operator Composition

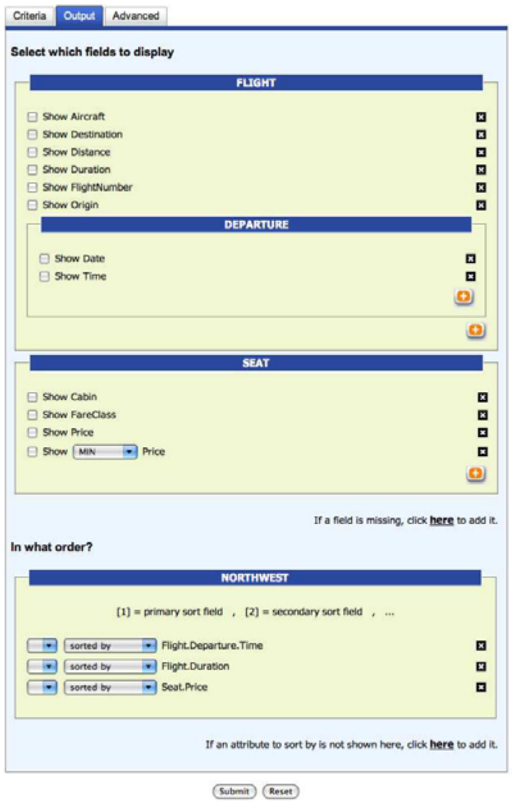
Form operators in the language may be composed. For example, we could write

$$F_3^{\mathbf{I}} = \Phi_2 \psi_{c(\mathsf{D.d}),1} \phi_{c(\mathsf{D.t}),2}(F_0^{\mathbf{I}})$$

obtaining the tree $F_3^{\mathbf{I}}$ obtained in three steps above. This composition property permits the user to make incremental changes to a form, one operator at a time. The resultant form of



(a) Criteria Pane


Criteria
Output
Advanced
Select which fields to display
FLIGHT
Show Aircraft
Show Destination
Show Distance
Show Duration
Show FlightNumber
Show Origin
DEPARTURE
Show Date
Show Time
SEAT
Show Cabin
Show FareClass
Show Price
Show MIN Price
If a field is missing, click here to add it.
In what order?
NORTHWEST
[1] = primary sort field , [2] = secondary sort field , ...
sorted by Flight.Departure.Time
sorted by Flight.Duration
sorted by Seat.Price
If an attribute to sort by is not shown here, click here to add it.
Submit
Reset

(b) Results Pane

Figure 6.3: The Modified Form

the operations we just performed is shown in Fig. 6.3.

6.2 Form Generation

Starting from an existing form a user can edit it using our form editor in multiple iterations until the desired form is obtained. We discussed the supported operations in Sec. 6.1. Here we describe how each form operator is realized.

The form editor provides users a canvas that holds the currently specified form, and button-activated operations that modify the form iteratively.

6.2.1 Form-Element Insertion

Inserting a form-element into a form involves selecting a form-pane, and one or two fields depending on the operator desired.

Pane Selection

The first choice is the type of query operation it involves. As shown in Fig. 6.2(a), the "Criteria" pane contains the simple and aggregate constraints involving user-specified values. Fields to be returned in the result as well as the ordering of the result are dictated by the contents of the "Results" pane (Fig. 6.2(b)). The desired pane can be made active by clicking on its tab at the top of the form.

Field Selection

Having activated the pane of choice, a user must then select a schema attribute (data field) associated with the form-element to be inserted. If the form has been partially built (typically the case in form modification), it can be examined to identify other elements whose schema attributes are related to the one to be added. If two or more attributes belong to the same entity, they will be located within the same group on the form. By simply clicking within this group, the fragment of the schema centered at this entity is presented to the user via a graphical schema browser to pick the desired attribute. Thus the user need not examine the entire schema to locate it. However, in the rare event that no similar attributes can be found already on the form, the user must traverse the schema from the root (or from an arbitrary entity) and drill down to the desired attribute. Only nodes along the selected path are expanded so as to not overwhelm the user if the schema is complex. Once a new group is created for that entity, all subsequent additions only require exploring the portion of the schema rooted at this entity. Some predicates require a second field to be specified by the user: the grouping basis (scope) of an aggregation and the right-hand-side of a join-condition. Here the user is prompted to re-visit the schema or a fragment of it, and select the desired field.

6.2.2 Form-Element Deletion

If the form contains elements that are irrelevant to the user's query or if the user inserted one or more elements in error, these may be removed from the form being edited. This can be done simply by clicking on the remove button next to these elements.

6.2.3 Form-Group Insertion

While users can add or remove form-elements themselves, only the system may add or remove form-groups. This makes the interface simpler for the user and the forms better organized. Grouping of two or more form-elements is based on one of two properties: (1) schema closeness and (2) query-imposed relationships. If two elements reference sibling attributes of a single entity, they will be placed in a single group (schema closeness). Secondly, if two elements reference attributes of two different entities that are joined in the query, they will be placed in a single group. The former grouping basis is observed in the input and output trees of a form, while the latter can only be seen in the relationship tree. When an element is inserted into the form, a group is created in the input or output panes if there is at least one other element on that pane that references the same schema entity (in which case the minimum size of a group is two elements, but this threshold it tunable).

6.2.4 Form-Group Deletion

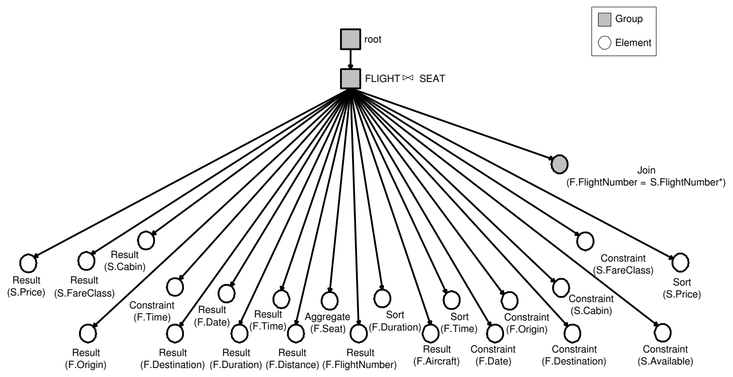
Following the deletion of form-elements, removal of the associated form-groups might be necessary. For instance, if the number of form-elements drops back below the threshold, the presence of the group is no longer necessary, and it is automatically removed. Similarly, if a join condition or a disjunction set is removed, the associated group is subsequently deleted. This re-organization of the form simplifies its appearance and maintains its consistency and correctness. Like form-group insertion, this operation is automatically performed by the system when needed.

Similar to this system, the QURSED Form Editor [63, 66] allows form-elements to be added and removed using a WYSIWYG editor that makes form manipulation easier than traditional form design tools. The main difference from our work is that the QURSED form editor is intended to be used by interface developers and forms are typically created from scratch. Every form edit requires direct interaction with the entire schema. While the schema display is graphical and navigable, making it user-friendly, complex schemas may still be difficult to browse, especially for end-users. Another example is the iTunes Smart Playlist [3] used to query a user's own music database and place desired songs in a single playlist. It allows the insertion of "form-elements" each of which specifies a selection operation on the music metadata (other query operations are not supported). Here too, selecting an attribute requires scanning the complete list of recorded attributes, this time as a drop down list. This is acceptable for simple schemas such as that used by iTunes, but does not scale well to complex schemas.

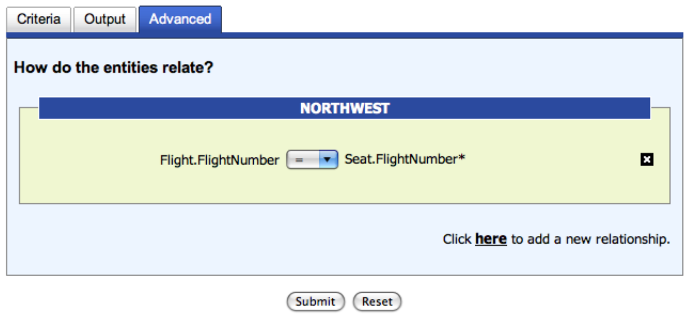
6.3 Advanced Manipulation

In this section, we now describe how experienced users can take advantage of form modification tools to express highly complex queries previously unsupported by the form in question.

Customizations of a form typically involve adding, deleting or modifying atomic query parameters in the first two panes of the form. If however, the relationships between queried entities require changing, the third pane of the form allows users to change them. We describe how these changes can be made in this section. It bears noting, however, that changes like these fundamentally alter the structure of the query and are not typical. Relationships between entities in a form are captured by a *join element*. By adding and/or removing these form-elements, relationships can be added, removed or modified (deletion followed by insertion). Inserting a join element requires specifying the entities to be joined and the exact fields within each entity that participate in the relationship. Internally, form groups are created, removed or modified to reflect the new relationships. A form-group



(a) Relationships Tree (Logical Representation)



How do the entities relate?

NORTHWEST

Flight.FlightNumber = Seat.FlightNumber*

Click **here** to add a new relationship.

(b) Advanced Pane (Visual Representation)

Figure 6.4: Entity-Relationships in Forms

in the relationship tree is defined by the join element it contains. All other form-elements that involve the entities related by this join relationship are contained in the same group. The group is named to reflect this relationship. If a join element is removed, the group is deleted, but the remaining form-elements are reassigned to other groups, typically the parent group according to the hierarchy of the relationship tree[1]. If a user wishes to add a new relationship to a form, the system prompts the user to specify the entities (two-at-a-time) that participate in this relationship. If a schema is available and it contains natural relationships between the chosen entities (defined by primary-foreign key pairs (relational) or key-keyref pairs (XML)) these are shown to the user who can then chose from among them. A user can also specify a relationship outside of these, if desired. In

[1]If the join element is part of an n-ary relationship ($n > 2$) then each form-element is moved to the lowest ancestor group that contains the entity of the form-element. If no such ancestor groups exist, the element is placed in the root form-group.

the example, the related entities are Flight and Seat. The relationship between them is the flight having the same flight number as the seat which means that the seat corresponds to a particular flight. A perceivable customization would be to query only those seats whose fare-class matches the highest class available on a flight. This change would require creating a new join element between these two entities and adding it to the relationship tree (Fig. 6.5).

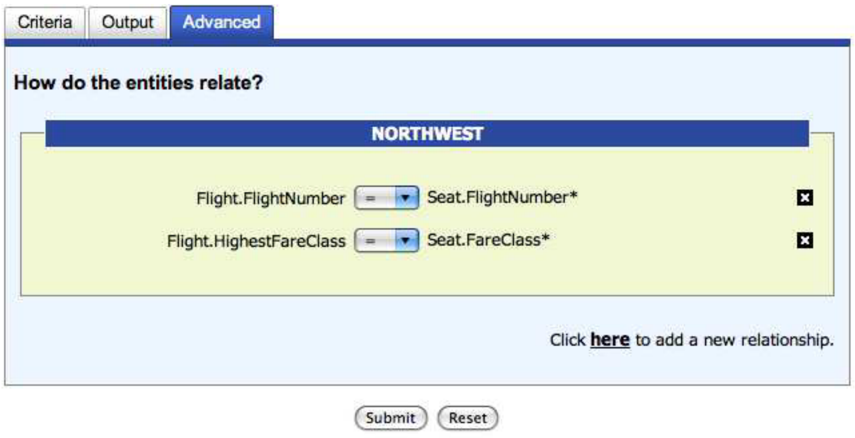

Criteria
Output
Advanced
How do the entities relate?
NORTHWEST
Flight.FlightNumber = Seat.FlightNumber*
Flight.HighestFareClass = Seat.FareClass*
Click here to add a new relationship.
Submit
Reset

Figure 6.5: The Modified Form (Advanced Pane)

A more complex modification could involve adding a new entity to the form that can be related either to Flight or Seat. For instance, if the airline database also had information about partner car rental companies, an advanced user could in fact extend the query to search for all such companies at the destination airport for each matching flight. This modification could be performed by first adding a new form-element in the relationship pane that relates the entity Flight with CarRental, which would be new to this form. Once this relationship is established, the user can modify the output tree to display details of rental companies for each flight.

6.3.1 Merging and Form re-use

The combination of multiple forms becomes useful when a user, who has previously created one or more forms involving different schema entities, would like to re-use them without having to start over. This saves the user some effort and improves the reusability of the forms generated even by other users.

Form Merge Operator

An advanced operation supported by our form editor is the merge form operator that allows a user to combine two or more forms, two at a time. While merging two forms, if the associated entities do not have a relationship between them already specified, the user is prompted to establish this relationship, and can do so by creating a new form-element containing an appropriate join-condition. The merge option is provided on the “Advanced” pane of the form and the user is taken through a series of steps starting with selection of the existing form to merge with the current form, followed by selection of the related entity or entities from each form and finally relationship specification. The result is a merged form whose input, output and relationship trees are concatenations of those from the two participating forms.

6.4 Evaluation

We implemented the form construction and modification ideas described above as a front end to [45], an XML database system. We conducted a user study to evaluate the effectiveness and usefulness of our querying interface whose results we present in this section.

Computing Environment

The study was taken over the internet by the subjects at their homes / labs. The server used was Apache’s Tomcat 4.1 and the interface was coded in Java/JSP. The server ran on a Windows XP workstation with a 3.1 GHz Pentium 4 processor, 1 GB of memory and a 120 GB disk. On the client side, since the study was taken remotely, multiple browser types were used which included: Microsoft Internet Explorer, Mozilla Firefox and Opera.

6.4.1 Casual User Study

To measure the quality of our querying mechanism, we ran an experiment with 10 technically unsophisticated users, knowledgeable neither in formal querying techniques nor in the data domain. Following a short tutorial to gain familiarity with the tools at their disposal, we presented 8 querying tasks to two different schemas, and measured their response times for the form-modifications that each task entailed. Responses were stored in files and evaluated offline.

Query Tasks

The queries were posed to two schemas, both best-effort replications of real-world online databases: the popular Yahoo! Movies database (http://movies.yahoo.com) and the well-known real-estate site, Realtor.com. The respective starting forms in individual tasks were fragments of the advanced search form provided at the websites. The tasks themselves were of a wide range of complexity across multiple dimensions as shown in Table 6.1.

Task	Schema Complexity	Query Complexity	Modification Complexity
1	Simple	Simple	Simple
2	Simple	Complex	Simple
3	Simple	Simple	Complex
4	Simple	Complex	Complex
5	Complex	Simple	Simple
6	Complex	Complex	Simple
7	Complex	Simple	Complex
8	Complex	Complex	Complex

Table 6.1: Task complexities in casual user study

Independent Variables

There are three factors affecting the performance measures of the subjects:

Schema Complexity: This denotes the schema corresponding to the database to which each query was posed. Half the tasks were to a schema that was simple while the other half were to a more complex schema. We define complexity in terms of the number of schema elements — the Realtor schema ($simple$) had 33 elements, while the Yahoo! Movies schema ($complex$) consisted of 63 elements.

Query Complexity: This denotes the difficulty level of each query in the task set. We define query complexity in terms of the number of form-elements needed by the query — a simple query is one involving less than 10 elements (lowest was 3, highest was 8), while a complex query requires 10 or more elements to be specified in order for it to be posed to the database (actual range was between 11 and 15).

Modification Complexity: Each task requires that a form be modified and the extent of modification is certainly a factor affecting user response time. Again the tasks were divided evenly between simple and complex modifications which we define again in terms of the form-elements involved. If a modification involves several simple form-elements or one or more complex form-elements (a join-specification or an aggregate-computation, each of which require multiple fields in the schema to be specified and require the use of the “Advanced” pane of a form) then, the modification is said to be complex. If not, it is a simple modification.

Dependent Variables

There is only one measure of interest that we use to evaluate the form-based interface:

Efficiency: This denotes the amount of time taken to solve each querying task using the interface. If a correct solution could not be obtained, a constant time of 600 seconds (10 minutes) was assigned as the time taken for that task.

Incorrect responses were penalized appropriately. The subjects were first instructed on how to use the form interface, via one sample task that was already solved. Users were strongly urged to take the tutorial at the very beginning for response time measurements to be accurate. However, this tutorial could also be returned to at any point during the study, should the subject need to. For each task, the subjects were simply asked to enter their solutions (modified forms) and proceed to the next one.

Results & Discussion

Efficiency: We observed that, on average, a user took just under 4 minutes per task, while individual task times ranged from a low as 1 minute to as high as 6 minutes for tasks that were completed correctly. 8 of the 10 subjects completed all 8 tasks successfully, while the remaining two erred on 3 tasks each. We also compared the average response time (over all tasks and subjects) for simple schemas vs. complex schemas and similarly for query and modification complexity (Fig. 6.6).

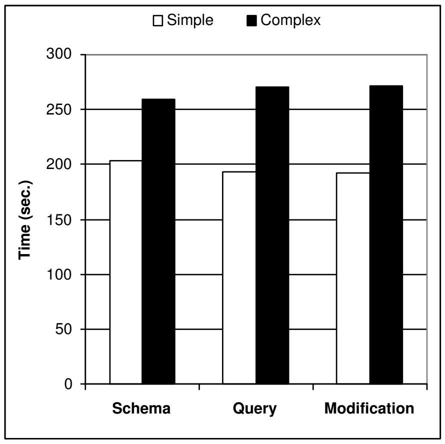


Figure 6.6: Factors affecting response time in casual user study

Response times for tasks involving complex schema show an increase of only 28% over those involving simple schema if other dimensions stay the same. The increase from simple to complex queries brought about a 40% rise in task times. Finally, with query and schema complexity unchanged, users took 42% more time on tasks where the form modification itself was complex as compared to forms needing minor changes. With the average time to complete a task being approximately 4 minutes (Fig. 6.7) these moderate increases in response time validate the system's effectiveness even when schemas, queries and modifications are complex.

Impact: It is very encouraging that our interface allowed non-experts the level of flexibility in query specification required by the tasks in our study (regardless of schema and query complexity).

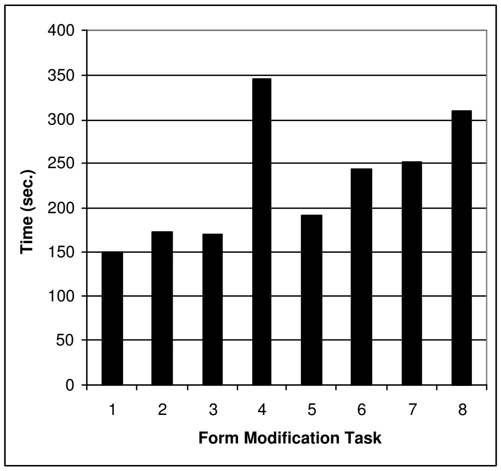


Figure 6.7: Time spent on each query task in casual user study

6.4.2 Expert User Study

To put these response times in perspective, we ran an experiment with 10 technically sophisticated users, each of whom would consider themselves proficient in XQuery, and knowledgeable about the domain underlying the experimental database. We chose to use datasets that were different from those used in the casual user study, but were from a domain that these expert users had familiarity with (more so than even with common-knowledge datasets like movies and real-estate that we used for casual users). In this expert-user study, we made the schemas of the relevant datasets available to them and measured how long it took them to make needed modifications to an initial XQuery statement provided. We also measured how long it took them to make the same modifications using our form modification software. A total of 10 tasks were presented to the user (to be solved both ways), but the first 2 were purely instructional and had the correct answers revealed. Moreover these were queries to a database distinct from the remaining 8 that counted. Responses were stored in files and evaluated offline.

Independent Variables

There is only one factor affecting the performance measures of the subjects:

Interface: This denotes the interface/querying mechanism used. Each user had to attempt each querying task using both mechanisms one after the other, in random order, one task at a time.

Dependent Variables

There are two measures of interest that we use to compare the interfaces:

Efficiency: This denotes the amount of time taken to solve each querying task using either interface. If a correct solution could not be obtained, a constant time of 600 seconds (10 minutes) was assigned as the time taken for that task.

Correctness: We qualitatively assess the two interfaces, and identify strengths and weaknesses in terms of the types of errors.

Subjects

Subjects were 10 students who volunteered to participate in the research. These are specifically students who are taking or have taken the advanced database systems course at our university which qualifies them as conversant in the XQuery language.

Data Collection

We recorded the time taken by each subject to solve each task and also recorded their responses to the queries.

Procedure

The subjects were first instructed on how to use the form interface, via two sample queries that were solved. More help was made available on a separate webpage that could be accessed at any time during the study. For each task, the subjects were simply asked to enter their solution and move on to the next one. For the part of the tasks involving

XQuery, the subjects had access to sample data and an XML schema definition to help them write queries.

Query Tasks

The queries were posed to the XMark [15, 71] dataset and a biological database [8], equally distributed: 4 each. The required modifications ranged from simple constraint insertions, to introducing new entities and relationships via joins.

Results & Discussion:

Efficiency: We recorded the time taken for each of the 8 tasks by each of the 10 subjects and charted average time taken per task for each querying mechanism. These can be seen in Fig. 6.8. We observe that the efficiency of the form-based interface is higher in all but one of the querying tasks, task 3. This was probably because the task was so simple that even in XQuery it required little time. We also observe a slight downward trend in the time taken for form modification from one task to the next. We attribute this to increased familiarity with the working of the interface. In contrast, the time taken for XQuery specification shows no such trend.
Impact: We have reported above results of experiments with users expert in XQuery as the worst case scenario for us. Even with these expert users we find that form-modification is faster than XQuery modification, and results in fewer errors.

6.4.3 Query Builder Comparison

Simplifying the task of querying a database has been the focus of many research efforts, some of which have resulted in the creation of visual query builders. These are tools that allow a user to visually construct a declarative query in an iterative manner. One such tool is the DB2 Visual XQuery Builder packaged with the IBM DB2 Developer Workbench. Since forms and query builders are both approaches to visual query specification, we conducted an experiment to compare our system with the DB2 VXB.

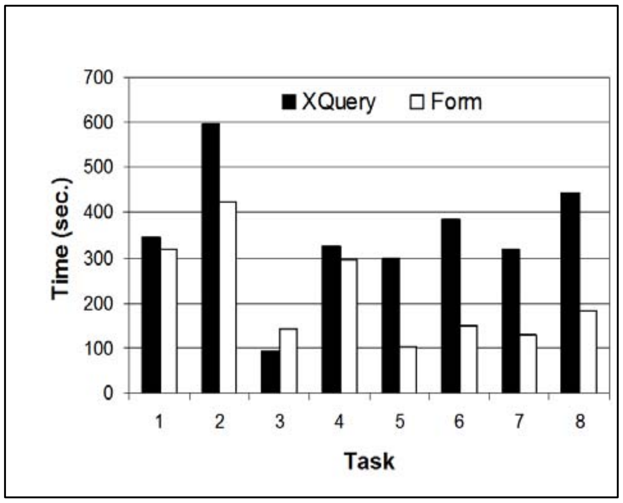


Figure 6.8: Time spent on each query task expert-user study

In this experiment, an XQuery-proficient subject was asked to use this query builder to specify the same 8 queries as that of our expert-user study, but this time with no starting point. A time limit of 20 minutes was set for each task, and we found that the subject was unable to finish any of the tasks in the time allotted. Since this subject was a senior graduate student at the university with intimate knowledge of XQuery, we did not continue this study with other subjects. While we expect that there may be other such query builders that perform differently, we believe that the results have more to do with the fact that the subject had to start each query from scratch than with the specific query-building tool.

Our approach to query specification is to leverage pre-existing complex queries, with the complexity packaged behind a simple form. Having chosen a form to modify, it is reasonable to expect that a portion of the desired query can already be expressed by the form prior to any modification performed by the user. The burden on the user is hence considerably less than that of users who use a graphical query builder to completely specify the desired query. Query builders precisely define how each operation that the user performs alters the resultant query. In our case, these structural modifications do not completely specify how the underlying query changes as a result of them. Rather, the expressive power and limitations of form modification with respect to query specification depend on an external data manipulation algebra that is treated as an argument to the form manipulation model that we propose. The more expressive the operators in this data model are, the higher the complexity of the queries that the forms can express. The expressive power of a form modification language should be measured not in terms of what the resulting forms can express, but in terms of how much modification can be expressed using this language. Since we allow the addition and deletion of any data manipulation operator in any position, our form modification language can express any desired change within the bounds of the expressive power of the underlying data manipulation language. It is even possible to “modify” an empty form to define a desired form from scratch. However, the engineering design of our technique makes it best suited for small changes to complex query forms.

CHAPTER VII

FORM USABILITY

The main advantage of forms over other query interfaces is their usability. For this reason, usability should be a key concern during the form design process. Auto-generated forms have high expressive power but are not the easiest to use. In this chapter, we devise techniques to make a forms-based interface (more) usable in an automated way. To establish a formal basis for usability enhancement, we develop a cost formula to estimate a form's usability based on its structure and composition. We identify three characteristics of any form that can potentially be improved and describe how a form's usability can be enhanced by altering these characteristics. We also present efficient algorithms to implement these enhancements and show how they can make forms more usable.

7.1 Form Usability

In this thesis, we introduced model-based automatic form generation which can be significantly more expressive than traditional form-based query interfaces that are static and restrictive. However, the increased expressive power comes at the cost of somewhat reduced usability (traditionally, forms are among the most usable of querying mechanisms). If the schema of the database is large and complex, the interfaces produced by these two approaches can be complex and sometimes overwhelming. Consider for instance, the form shown in Fig. 7.1 produced by our schema-based form generation technique defined in Chapter IV. It allows a user to issue a query pertaining to customers, their brokerage accounts and/or their brokers. While this form might support all possible queries involving these entities, the usability of this form leaves something to be desired. While it is useful to provide users with several fields to search by, it increases the difficulty of finding the one (or more) fields that pertain to the desired query of any particular user in isolation. There are several other shortcomings to this form, from a usability perspective, and we discuss many of these issues in this chapter. We address many of these in an automated fashion to enhance the usability of forms of this nature. However, the problems we observe aren't limited to forms generated artificially. Consider the form shown in Fig. 7.2. This form has been adopted by hundreds of literature datasets in many domains of scientific research (including genetics, ecology, geology, materials science, agriculture, economics, etc.)[10]. One can see here too that the form is large and disorganized making it potentially difficult for users to specify their desired queries. The techniques of form enhancement we propose in this chapter apply to such forms as well.

7.1.1 Usability and Querying

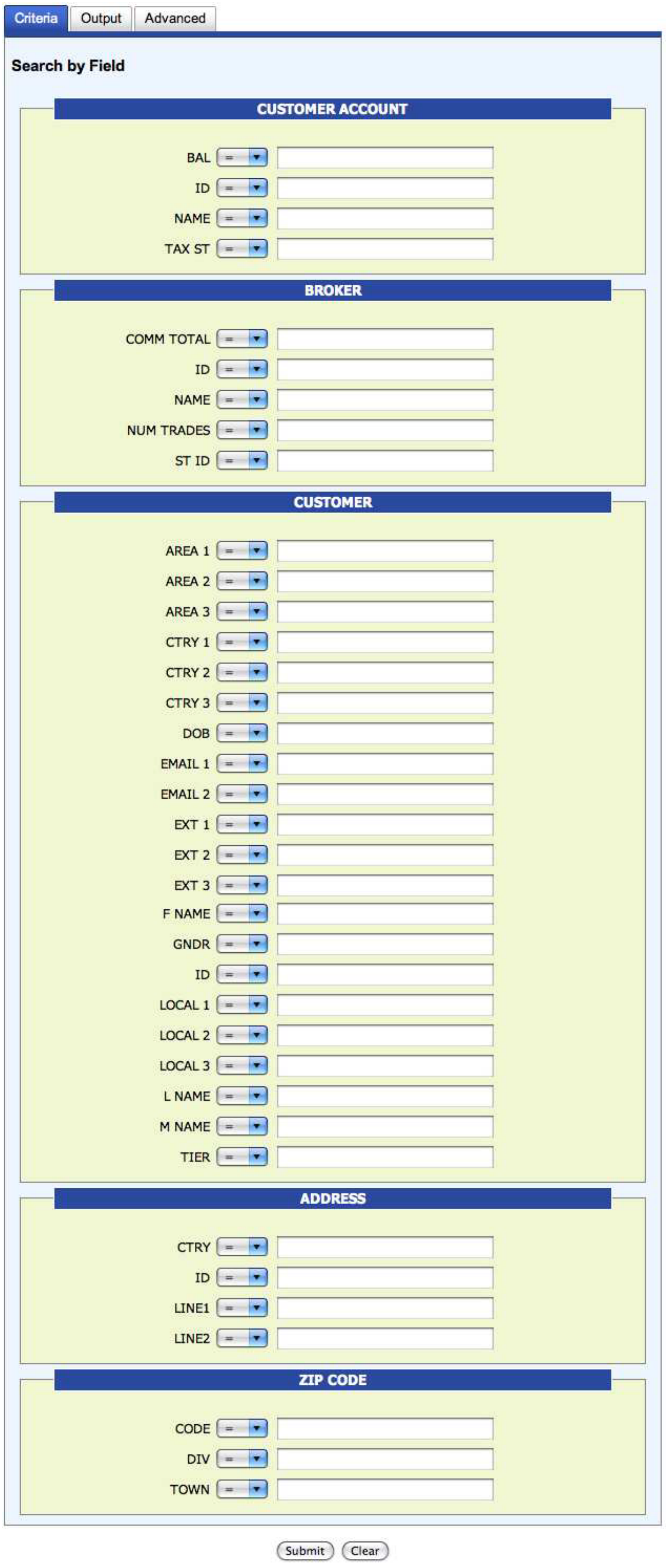
Any information system that requires human interaction must be designed and evaluated from a usability perspective in addition to the functional perspective. The functionality of a system is irrelevant if the system is unusable [36]. The usability of a query interface determines how much of the underlying database can be useful to its users. While usability does depend on the cognitive ability and expertise of users and their familiarity with the system (which can improve over time), these should not be significant design considerations for an interface built for a large and diverse audience. A model of usability based simply on the final appearance of the interface and the functionality it provides is reasonable. We propose such a model in this chapter. The use of analytical models to evaluate user interfaces is not new. In [64], the authors use information theory-based [73] formulae to measure the complexity of a dialog box from a usability perspective. While usability is considered by some to have two components: *objective* usability and *subjective* usability[1] [78], our focus is purely only on objective usability since the latter can vary widely across users based on their familiarity with the system and background knowledge. For database querying specifically, research has shown that the effectiveness of a query interface depends on the class of queries issued as well as the querying experience of users [24].

7.1.2 Our Contributions

In order to improve usability, we need to first be able to determine when one design approach is better than another. For this we need to be able to quantify the usability of a given form and measure the change in usability when it is replaced by a different (better or worse) form that serves the same purpose. This gives us a basis for determining if a particular form enhancement is useful and worth making. To accomplish this goal, we have developed a cost formula that can be used to estimate the usability of a form independent of the query workload. Such a formula may be considered primitive since it does not incorporate learnability and likelihood of user error. However, if the change in design does not affect these properties of the form, the formula still provides us a relative measure of form usability that can be used to evaluate the usefulness of that change. Next, we propose three ways in which query forms can be improved —structurally (layout of form components), textually (labels, descriptions and hints of form fields) and interactionally (form-controls for user input). Using our cost model we can estimate the improvement in form usability as a result of applying one or more of these enhancements. If one approach conflicts with another, we can estimate which one would likely be better and just apply that one to the form in question. The remainder of this chapter is laid out as follows. We present our usability formula in Sec. 7.2. In Sec. 7.3, we describe the techniques developed for three classes of enhancement and algorithms to implement them efficiently. Finally, in

[1]*Objective usability* measures how efficient and error-free a system is in eliciting user preferences. *Subjective usability* measures how satisfied and comfortable users are in interacting with the system.

Sec. 7.4, we show two ways of evaluating our proposed enhancements: analytically and empirically.



Criteria | Output | Advanced

Search by Field

CUSTOMER ACCOUNT

BAL	=	
ID	=	
NAME	=	
TAX ST	=	

BROKER

COMM TOTAL	=	
ID	=	
NAME	=	
NUM TRADES	=	
ST ID	=	

CUSTOMER

AREA 1	=	
AREA 2	=	
AREA 3	=	
CTRY 1	=	
CTRY 2	=	
CTRY 3	=	
DOB	=	
EMAIL 1	=	
EMAIL 2	=	
EXT 1	=	
EXT 2	=	
EXT 3	=	
F NAME	=	
GNDR	=	
ID	=	
LOCAL 1	=	
LOCAL 2	=	
LOCAL 3	=	
L NAME	=	
M NAME	=	
TIER	=	

ADDRESS

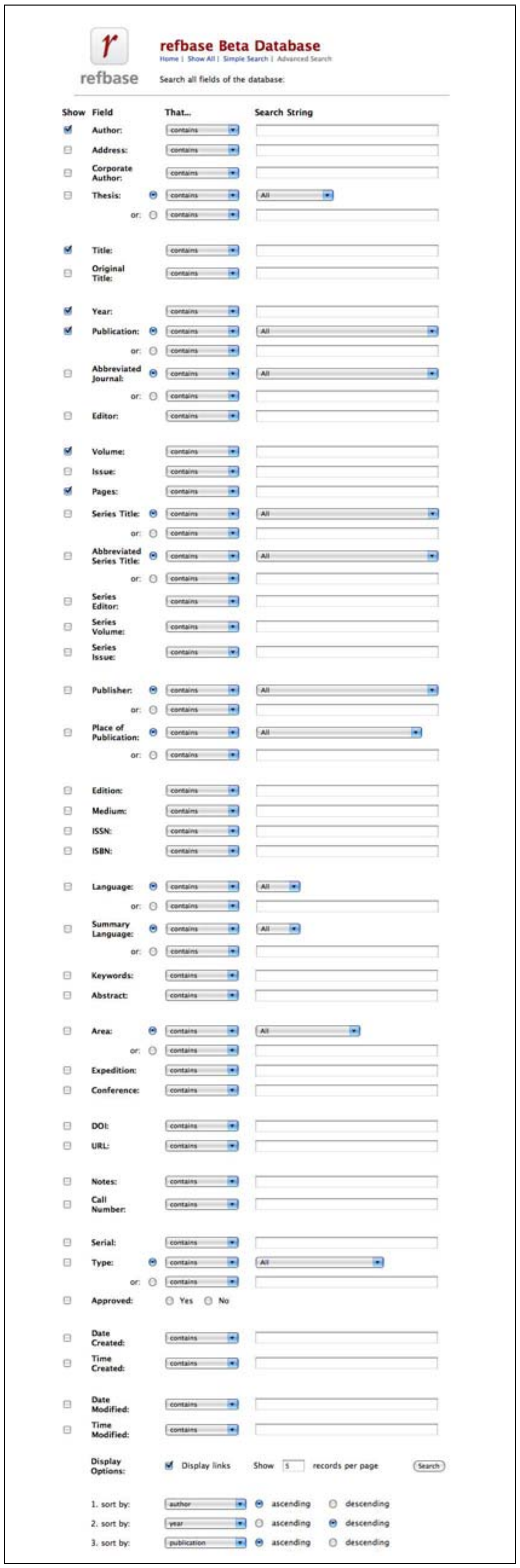
CTRY	=	
ID	=	
LINE1	=	
LINE2	=	

ZIP CODE

CODE	=	
DIV	=	
TOWN	=	

Submit | Clear

Figure 7.1: A form created for users to query a brokerage database (TPC-E benchmark)



refbase

refbase Beta Database

Home | Show All | Simple Search | Advanced Search

Search all fields of the database:

Show	Field		That...	Search String
✓	Author:		contains	
☐	Address:		contains	
☐	Corporate Author:		contains	
☐	Thesis:	●	contains	All
	or:	○	contains	
✓	Title:		contains	
☐	Original Title:		contains	
✓	Year:		contains	
✓	Publication:	●	contains	All
	or:	○	contains	
☐	Abbreviated Journal:	●	contains	All
	or:	○	contains	
☐	Editor:		contains	
✓	Volume:		contains	
☐	Issue:		contains	
✓	Pages:		contains	
☐	Series Title:	●	contains	All
	or:	○	contains	
☐	Abbreviated Series Title:	●	contains	All
	or:	○	contains	
☐	Series Editor:		contains	
☐	Series Volume:		contains	
☐	Series Issue:		contains	
☐	Publisher:	●	contains	All
	or:	○	contains	
☐	Place of Publication:	●	contains	All
	or:	○	contains	
☐	Edition:		contains	
☐	Medium:		contains	
☐	ISSN:		contains	
☐	ISBN:		contains	
☐	Language:	●	contains	All
	or:	○	contains	
☐	Summary Language:	●	contains	All
	or:	○	contains	
☐	Keywords:		contains	
☐	Abstract:		contains	
☐	Area:	●	contains	All
	or:	○	contains	
☐	Expedition:		contains	
☐	Conference:		contains	
☐	DOI:		contains	
☐	URL:		contains	
☐	Notes:		contains	
☐	Call Number:		contains	
☐	Serial:		contains	
☐	Type:	●	contains	All
	or:	○	contains	
☐	Approved:		○ Yes ○ No	
☐	Date Created:		contains	
☐	Time Created:		contains	
☐	Date Modified:		contains	
☐	Time Modified:		contains	

Display Options: ✓ Display links Show 5 records per page Search

1. sort by:	author	[✓ ascending]	descending
2. sort by:	year	ascending	[✓ descending]
3. sort by:	publication	[✓ ascending]	descending

Figure 7.2: A form provided for users to query an online bibliography management service (Refbase)

7.2 Estimating Form Usability

The usability of a form is the inverse of the "cost" of using the form. This cost may be in terms of cognitive burden on the user, time taken to specify a query, or other such measures. These are extrinsic measures, in that they are determined by user interaction with the system. To be able to improve form usability, we need a means to estimate this cost intrinsically, without having to perform a user study.

In the next several subsections we develop a model to estimate the cost of using a form. This model not only provides us with an estimate of the expected cost of using the form, but more importantly, provides guidance regarding possible usability improvements.

7.2.1 Query Cost Estimation

We call each atomic component of a form to be a *form-element*. Typical form-elements are radio buttons, text boxes, and so on. Each form field is a form element. Given a form and a query to be expressed using it, we compute the cost of expressing that query as a sum of costs of each of the form elements "involved" in the query. Not all form-elements on the form need be involved in any one query – typically only a subset are. The cost incurred by a user in specifying a piece of the query via a form-element can further be decomposed as the sum of costs of two actions:

1. Locating the form-element within the form.
2. Expressing the desired query component using that form-element (for example, filling in a textbox, checking a checkbox, selecting a value from a drop-down list, etc.).

In [35], Gajos and Weld compute the cost of changing the state of a device in a similar way—computing first a navigation cost which is incurred while searching for the desired interface element, and then the manipulation cost incurred in changing the setting controlled by that element to the desired value, and finally adding the two together.

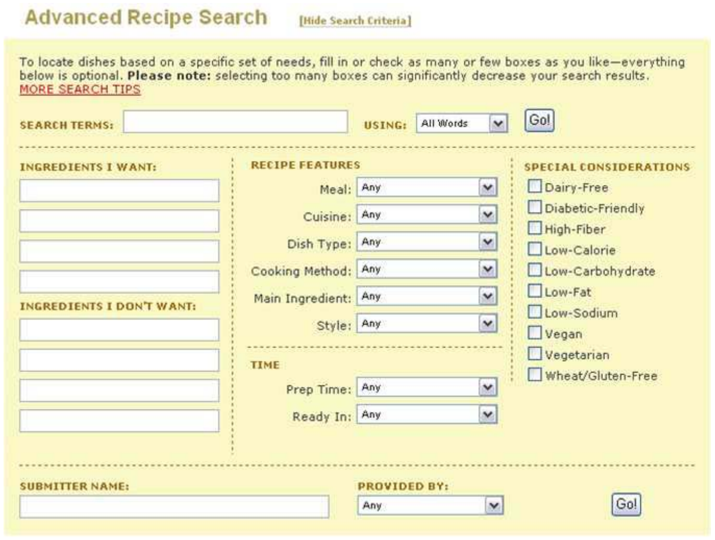
In our model, locating a form-element involves eye-balling the form and examining potentially relevant form-elements (possibly in sequence). Each such form-element incurs a *decision cost*, i.e., a user must decide whether or not it is relevant to his or her query. Finding a desired (relevant) form-element incurs a *search cost* which adds up the decision costs of all form-elements that precede it. Once a user has found a relevant form-element, the next step is to actually use it to express the desired query condition. We denote the cost of this action as the *usage cost* of that form-element. Let us take a closer look at each of these costs and the factors that influence them.

Decision Cost

Each form-element has a purpose—enabling a user to express a piece of the query. Identifying this purpose incurs a cognitive cost which can be different for different users. Each form-element has a label and a fillable form-control (e.g. textbox, checkbox, radio button, etc.). Additionally, the panel containing the form-element might also have a label that affects the decision of whether the element is relevant. These labels ultimately determine the decision cost associated with a form-element.

Search Cost

The task of finding a desired form-element within a form incurs a cost that depends not only on the form-element itself, but also on other parts of the form that are examined en route to finding it. If the form is flat, i.e., it is just a list of form-fields one below (or beside) the other, then the search cost is simply the sum of decision costs of all the form-elements the user had to examine before reaching the desired one (including itself). Sometimes, forms are organized into panels, each containing a subset of the form's elements (such as the form in Fig. 7.3 or Fig. 7.1). Such panels, which may or may not be labeled, we call *form-groups*. If a form-group is labeled, it simplifies the task of finding a desired form-element, which effectively reduces the query cost. The user just needs to examine the labels of every form-group to identify the one that most likely contains the desired form-element. For example, if a user wants to find recipes that take less than 30 minutes to prepare (using the form shown in Fig. 7.3), he or she would most likely find it under the group labeled “Time”. There might be more than one level of grouping on a form, i.e., form-groups might contain other form-groups effectively making the form a tree-like structure. If the number of form-elements is large, this type of organization can make it easier to find a desire element thereby reducing the time needed to express a desired query. Examining form-group labels incurs a decision cost much like form-element labels and this adds to the search cost for any element located within a group.


Advanced Recipe Search [Hide Search Criteria]

To locate dishes based on a specific set of needs, fill in or check as many or few boxes as you like—everything below is optional. **Please note:** selecting too many boxes can significantly decrease your search results.
MORE SEARCH TIPS

SEARCH TERMS: USING: All Words Go!

INGREDIENTS I WANT:

INGREDIENTS I DON'T WANT:

RECIPE FEATURES

Meal:	Any
Cuisine:	Any
Dish Type:	Any
Cooking Method:	Any
Main Ingredient:	Any
Style:	Any

TIME

Prep Time:	Any
Ready In:	Any

SPECIAL CONSIDERATIONS

- ☐ Dairy-Free
- ☐ Diabetic-Friendly
- ☐ High-Fiber
- ☐ Low-Calorie
- ☐ Low-Carbohydrate
- ☐ Low-Fat
- ☐ Low-Sodium
- ☐ Vegan
- ☐ Vegetarian
- ☐ Wheat/Gluten-Free

SUBMITTER NAME: PROVIDED BY: Any Go!

Figure 7.3: Example Form (AllRecipes.com)

We can put the ideas above into a forumla, but to do so we need some notation. Let $PG(e)$ be the *parent group of* e, in other words, the form-group containing form-element (or form-group) e. This is the form itself if there is no intervening form-group. Let $AG(e)$ be the set of Ancestor-Groups of e, i.e., all form-groups in the ancestry of e, which include the innermost group $g_1 = PG(e)$ that contains e, followed by group $g_2 = PG(g_1)$ and so on until the outermost group in the form $($which we call the root group$)$. Finally, we write $e' \prec e$ if e' is a form-element or form-group that precedes e in a user’s scan of the form (top to bottom and left to right in most of the Western world).

Formula 12. *(SEARCH COST)* The search cost of a form-element e can be expressed in terms of decision costs of form-elements and form-groups that precede it on the form.

$$C_S(e) = \sum_{g \in AG(e)} \sum_{\substack{g' \preceq g \\ PG(g')=PG(g)}} C_D(g') \quad + \sum_{\substack{e' \preceq e \\ PG(e')=PG(e)}} C_D(e')$$

Here C_D denotes decision cost and is computed for all form-elements e' preceding e within the same form-group $(PG(e)$ and $PG(e')$ are one and the same$)$.

If more than one form-element is needed to express a desired query, the query cost is the sum of the search costs of each of them individually. However, if two or more form-elements are located within the same form-group or have some shared hierarchy, this may not be true. In such cases, the decision cost of the form-group in common may only be incurred once. However, to be consistent in our measurement, we only consider the *worst case*, i.e., that a user does not know or realize the common hierarchy of multiple form-elements. Thus the search cost for multiple form-elements is computed simply as the sum of their individual search costs (in isolation).

Usage Cost

Once a desired form-element is located the user needs to manipulate it to specify the query predicate he or she has in mind. We call the cost associated with this action the *usage cost* of the form-element. It depends not only the condition itself but also on the type of input field provided. A simple textbox, for instance, requires that the user fully enter the value of the chosen attribute which can be expensive (in terms of keystrokes and/or mouse-clicks). Checkboxes, radio buttons, and drop-down lists (which can be used if the data domain is small) are typically cheaper as they only require mouse-clicks. Although there is the cognitive overhead of examining other values, the user does not have to worry about typographical errors or terminological inconsistencies.

Query Cost

We can use the above measures (decision cost and usage cost) to estimate the cost incurred by a user in expressing a desired query.

Formula 13. *(QUERY COST)* The cost of a query q to be expressed using a form f is computed as follows[2]

$$C(q, f) = \sum_{e \in (E_f \sqcap q)} C_S(e) \quad + \sum_{e \in (E_f \sqcap q)} C_U(e)$$

Here $C_S(e)$ denotes the search cost of form-element e, $C_U(e)$ refers to the usage cost of e, E_f is the set of all form-elements in form f and $(E_f \sqcap q)$ is the subset of form-elements in f relevant to query q.

It may not be obvious to a user whether or not a form-element is relevant to the desired query. This may be due to one or more form-elements having labels that are similar (textually or semantically) to the desired form-element or just misleading. If the user erroneously uses irrelevant form-elements or searches the entire form unsuccessfully for a particular form-element, the query cost could be higher than the above estimate. In fact, the query may even have to be expressed multiple times if the error is not realized until after the form is submitted (if the user knows or feels that the results are incorrect). Since the increase in cost due to form's mismatch with a user's expectation cannot be generalized, or even quantified probabilistically, we ignore such scenarios while computing query cost. In other words, we assume that the query is correctly specified in a single attempt, i.e., all desired form-elements are correctly filled and no irrelevant form-elements are examined or used.

[2]Note that this formula does not account for user expertise or familiarity with the system or any other external factor that could have an impact on the time actually taken to express a query. It is simply an estimate that is easy to compute and suitable as a basis for comparison of different designs of a query form.

7.2.2 Computing Usability

The usability of any object depends at least in part on the purpose for which it is to be used. A single form may permit the specification of multiple queries—for instance through optional and conditional form elements. If two users attempt to pose two different queries using a particular form, each user might have a different opinion of the form's usability depending on how easy it is to pose their query using the form. Hence, we consider all queries expressed using a form, compute the cost incurred to express each of them using the form, take the average and call this average the usability of the form.

Formula 14. *(FORM USABILITY)* The usability of a form is computed as the inverse of the average query cost of all queries posed using the form.

$$U(f, Q_f) = C(f, Q_f)^{-1} = \left(\frac{\sum_{q \in Q_f} C(q, f)}{|Q_f|} \right)^{-1}$$

Here, $U(f, Q_f)$ and $C(f, Q_f)$ denote the usability and the cost of the form f respectively for a given set of queries Q_f of cardinality $|Q_f|$. $C(q, f)$ refers to the query cost of a query q if expressed using form f.

Ideally, the average should be weighted based on how frequently each query occurs. However, the actual set of queries for which a form is used cannot usually be known before the form is actually deployed. So Formula 14 presents an equi-weighted average. If weights happen to be known, or guessed, it is easy to accommodate these in the formula.

The formulae in the model above appear to be complex at first glance. However, careful examination shows that the only parameters required are C_D and C_U for each form element (and form group), and a (weighted or unweighted) query workload Q_f. Furthermore, the specific values for the cost parameters are immaterial – it is only their relative magnitude that matters. For simplicity, we can set these cost parameters to 1, except where a different value is explicitly warranted. That is, we will use $C_D = C_U = 1$ for all form elements and form groups by default in this paper. Where some user decision or action is particularly simpler (or harder), we will use an appropriate fraction (or multiple) for the corresponding cost.

7.3 Form Enhancement

Both automatically-generated forms (e.g. Fig. 7.1) and manually-generated forms (e.g. Fig. 7.2) can sometimes be difficult to use if their respective schemas are very complex. In this section, we show some of the ways in which these and other such forms can be made more usable (automatically). Consider the schema fragment shown in Fig. 7.4. This sub-schema has been taken from the brokerage dataset used by the TPC-E Benchmark [12], which simulates the OLTP workload of a brokerage firm. The main entity in this portion of the schema is CUSTOMER with attributes corresponding to a customer's name (C_F_NAME,C_L_NAME and C_M_NAME) , gender (C_GNDR), address (ADDRESS), etc. Automatic form-generation uses the entire schema to create an initial set of forms that can be used to query this and other entities in the database. Howewver, this often results in poor forms because of:

1. Schema complexity
2. Cryptic table (entity) and column (attribute) names
3. High degree of normalization

For this TPC-E fragment specifically, one of the forms created is shown in Fig. 7.1. In this section we present three approaches to improving the usability of forms such as this.

7.3.1 Textual Enhancement

The readability (and hence usability) of a form depends to a large extent on its textual content. The labels associated with form-elements and form-groups have a direct impact on the cost (to the user) of expressing queries using the form (specifically search cost, $C_S(e)$, in Formula 12). We examine how these labels are generated and how they can be modified and/or supplemented to improve the form's usability.

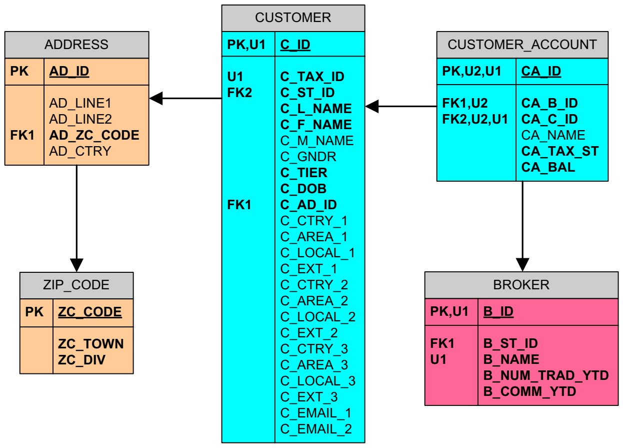


Figure 7.4: Fragment of the TPC-E schema

Label Supplementation

Entity and attribute naming conventions vary from database to database and are created with the goal of efficient data organization. This convention and the names themselves may or may not be ideal for use in a user interface to that database. While attribute names used in the logical schema can sometimes be cryptic, there is often accompanying documentation that clarifies the purpose and usage of all tables and columns in the dataset. These entity-level and attribute-level descriptions can be appended to their corresponding form-elements to guide users on how to use them while specifying their queries. Fig. 7.5 shows how these clues can be placed next to their corresponding form-elements and form-groups to aid querying users. In this enhanced form, a user can mouse-over an icon next to the field label and view the documentation corresponding to that field. (See the small pop up box below th “COMM TOTAL” field under the “BROKER” group in Fig. 7.5). The group description is always visible at the top of the panel demarcating it.

Sample Values

In addition to the field description, each field can also display one or more sample values that best represent what the attribute denotes. One of the best ways to explain what a form-field is meant for and how to use it is to show the user an example of a valid entry for that field. If values of the attribute are of a specific format, an example can convey this information well. We simply select a frequently occurring value from the database, or if a query workload is available, a frequently occurring value that other users enter in that field. Some query forms on the web provide sample values to assist users. Typically these values are chosen by the interface designer or domain expert. In Formula 13, we account for the effect this enhancement has on usability in the term $C_U(e)$ which denotes the usage cost of a form-element e.

Interactional Enhancement

The cost of specifying the desired query criterion depends to an extent on the mode of user input. A textbox requires textual input, i.e., the user has to fill in one or more words that specify the condition. A drop-down list or a collection of checkboxes or radio buttons can also be used and are often easier to use. This is because of two reasons: the user does not have to type out each individual character of the desired value. Secondly, this desired value is already present on the form and just needs to be selected. But this is not always the better alternative. If the domain of the attribute is large, just scanning through all possible values in the drop-down list can be more difficult than typing in the desired value. Our approach to enhancing user interaction involved in form filling is to analyze the data domain of each attribute and use an appropriate form control based on the attribute's data type and the cardinality of the data domain. We summarize these decisions in Table 7.1. Our example form (shown in Fig. 7.5), for instance, uses a calendar widget for CUSTOMER.DOB and radio buttons for CUSTOMER.GNDR and ADDRESS.CTRY.

7.3.2 Layout Enhancement

A second way to improve the usability of a form is to change the layout of form-elements and form-groups on the form. A form that satisfies a user's querying need still may not be usable if it is too complex and the user finds it difficult to locate the relevant form-elements. If this is the case, the cost of expressing the desired query is too high. For

Data Type	Domain Cardinality (# unique values)	Form-control/Widget
String	1–5	Radio Buttons
String	6–10	Drop-down List
String	10–1000	Auto-completion
String	$>$ 1000 or 0	Text Box
Date	any	Calendar
Integer	2–100	Slider
Integer	0–1 or $>$ 100	Text Box
Other	any	Text Box

Table 7.1: Selection criteria for input form field controls

instance, consider the following query:

Q: *Which tier-1 customers have non-taxable accounts with inactive brokers?*

Let us first consider the form in Fig. 7.1. This query can be expressed by filling in ‘1’ in the field labeled CUSTOMER.TIER, ‘0’ in the field CUSTOMER ACCOUNT.TAX ST and “inactive” in the BROKER.ST ID. Using Formula 12, we can compute the cost of this query as follows.

$$\begin{aligned}
C(Q,f) &= \sum_{e\in(E_f\sqcap q)} C_S(e) \quad + \sum_{e\in(E_f\sqcap q)} C_U(e) \\
&= [C_S(\text{CA.TAX ST}) + C_U(\text{CA.TAX ST})] \\
&+ [C_S(\text{B.ST ID}) + C_U(\text{B.ST ID})] \\
&+ [C_S(\text{C.TIER}) + C_U(\text{C.TIER})] \\
&= [(1*1+4*1)+1*1] \\
&+ [(2*1+5*1)+1*1] \\
&+ [(3*1+21*1)+1*1] \\
&= [6]+[8]+[25] \\
&= 39
\end{aligned}$$

We can simplify this form by carefully choosing portions of it to hide temporarily so that the time taken to scan the form is low. The contents of certain panels (form-elements) or even entire panels (form-groups) can be hidden from users initially if their absence could potentially reduce the time taken to express a query using the form. The idea here is that more frequently used parts of the form remain visible while the rest of the form is hidden but viewable if desired by users. By reducing the amount of “clutter” on the form, average search cost can be reduced. However, care must be taken while determining which elements or groups to hide. Even though hidden components can still be viewed by expanding the containing form-group, the benefits of layout enhancement are lost if group expansion is required for most user queries.

In our scenario, the “hidden” schema elements correspond to form-elements that are temporarily invisible to the user. Hidden panels are replaced by anchors containing just the name of the suppressed form-group. This can be clicked on (by a user) to reveal (unhide) the hidden panel. Then one or more of the newly displayed form-elements can be used to specify a query. If the user wishes, the reverse is also possible, i.e., form-elements and form-groups can be re-hidden once they have been made visible.

Query-based Hiding

If a collection of real queries to the database is available, we can use each individual query to estimate the relative importance of each form-element and form-group. The best possible scenario is if a previous user of the form asked exactly the same query. If our example query Q was issued by a user using our example form (Fig. 7.1), and this was the only observed query so far, the form can be adjusted to only show the form-elements and form-groups used for this query initially (hiding the rest). The form would now look like the form in Fig. 7.6. Initially the cost of this query as computed earlier in this section would be 39 units. But after the new query-based enhancement, if another user were to ask the same query the cost is now:

$$\begin{aligned}
C(Q,f) &= [C_S(\textsf{CA.TAX ST}) + C_U(\textsf{CA.TAX ST})] \\
&+ [C_S(\textsf{B.ST ID}) + C_U(\textsf{B.ST ID})] \\
&+ [C_S(\textsf{C.TIER}) + C_U(\textsf{C.TIER})] \\
&= [(1*1+1*1)+1*1] \\
&+ [(2*1+1*1)+1*1] \\
&+ [(3*1+1*1)+1*1] \\
&= [3]+[4]+[5] = 12
\end{aligned}$$

Of course, this is an extreme best case scenario where the form is custom-fit for the single query of interest. More typically, there will be a collection of expected queries, and the visibility of the form should be adjusted to minimize the future cost of any query similar to any query that has been expressed before. One approach is to show all form-elements and form-groups that were previously used while hiding all others. This approach is outlined in Algorithm 8. Alternatively, one could set a threshold parameter to set the minimum number of times a form-element or form-group had been used to render it visible to future users. So if the threshold is set at 5, then any form-element or form-group that had been used fewer than five times would be hidden from view initially. The less popular fields are hidden, not deleted, so they can be reinstated on demand, but require additional user effort.

Queriability-based Hiding

If we do not have access to a query workload or even a set of queries that actual users of the system would like to ask, we can still make use of the database itself to decide which parts of a form are more likely to be used to query the data. Using the notion of queriability presented in [48], we can choose which form components to hide initially (the form-elements and form-groups that correspond to attributes and entities in the schema

Algorithm 8: Algorithm `EnhanceLayout`

```
Input: Original form $F$
Input: Database $D$
Input: Schema $S$ (of $D$)
Input: Complexity thresholds: $k_t$ (table), $k_c$ (column)
Input: Query Workload $Q$ (optional)
Output: Layout Enhanced form $F_L$
foreach form-group $g \in F$ do
    Set $visibility(e) \leftarrow$ false;
foreach form-element $e \in F$ do
    Set $visibility(g) \leftarrow$ false;
Compute the queriability of every table and column in $S$ using $S$ and $D$ (using formulae
defined in [48]);
Rank all tables and put the top-$k_t$ tables in a new set $QT$;
Rank all columns within each table and put the top-$k_c$ columns in a new set $QC(t)$;
if $Q \neq \{\phi\}$ then
    foreach query $q \in Q$ do
        foreach table $t$ referenced by $q$ do
            Let $g$ be the form-group associated with table $t$ in the schema;
            Set $visibility(g) \leftarrow$ true;
        foreach column $c$ referenced by $q$ do
            Let $e$ be the form-element associated with column $c$ in the schema;
            Set $visibility(e) \leftarrow$ true;
foreach form-group $g \in F$ do
    Let $t$ be the table associated with form-group $g$ in the form;
    if $t \in QT$ then
        Set $visibility(g) \leftarrow$ true;
        foreach form-element $e \in g$ do
            Let $c$ be the column associated with form-element $e$ in the form;
            if $c \in QC(t)$ then Set $visibility(e) \leftarrow$ true;
```

with a low queriability score). The formula used to compute queriability was based on the summary importance formula proposed in [79] which is used to summarize complex schemas so that they can be browsed and queried more easily. To summarize a schema, [79] compute an “importance” score for each schema element and along with schema “coverage” computations, determine which parts of the schema to make visible to users, and which parts to hide from sight initially. This approach is outlined in Algorithm 8. One can see the parallel with our purpose to use such a metric.

For example, the form in our example has form-groups corresponding to CUSTOMER ACCOUNT, BROKER, CUSTOMER, ADDRESS and ZIP CODE. Based on our computations, we find CUSTOMER and BROKER to have the highest queriability scores. Hence, as Fig. 7.7 shows, the form-groups corresponding to these entities are visible while the rest of the form is hidden. Within the CUSTOMER group, notice that some form-elements are also hidden. These elements can be made visible simply by clicking on the link "Click here to see more fields". The abbreviated form can be easier to use if the query desired does not involve the hidden groups or elements.

Our example query Q, for instance, requires the use of two form-elements that are visible and one that is hidden. If we compute the new cost of expressing this query using the layout-enhanced form, we can see the effect of this enhancement on the usability of this form for this particular query. Consider the cost of fully expanding a form-group (one or at most two mouse clicks) to be 1 unit per click.

$$\begin{aligned}
C(Q,f) &= [C_S(\text{CA.TAX ST}) + C_U(\text{CA.TAX ST})] \\
&+ [C_S(\text{B.ST ID}) + C_U(\text{B.ST ID})] \\
&+ [C_S(\text{C.TIER}) + C_U(\text{C.TIER})] \\
&= [(1*1+1+4*1)+1*1] \\
&+ [(2*1+5*1)+1*1] \\
&+ [(3*1+10*1)+1*1] \\
&= [7]+[8]+[14] \\
&= 29
\end{aligned}$$

The query cost is less than that using the original form in Fig. 7.1 due to the reduced search cost of the field CUSTOMER.TIER within the group CUSTOMER. Although an extra click is needed to find CUSTOMER ACCOUNT.TAX ST, the overall cost is lower.

Algorithm 9: Algorithm `EnhanceStructure`

```
Input: Original form $F$
Output: Structure Enhanced form $F_S$
Let $T$ be the form-tree associated with the form $F$;
Let $root$ be the root group of $T$;
Let $G$ be the set of all form-groups in $F$;
Initialize a queue $Q$ to be empty;
Add $root$ to $Q$;
while $Q$ is not empty do
    Remove group $g$ from $Q$;
    Let $t$ be the associated table of $g$ in the schema;
    foreach form-group $g' \in F$ do
        Let $t'$ be the associated table of $g'$ in the schema;
        if $t$ references $t'$ in the schema
        and $t$ is not a relationship table then
            Add $g'$ to $Q$;
            Let $p'$ be the parent group of $g'$ in $T$;
            if $p' \neq root$ then
                Create a deep copy of $g'$ called $g'_1$;
                Add $g'_1$ as a child of $g$ in $T$ (and thus in $F$);
            else
                Remove $g'$ from $T$ (and thus from $F$);
                Add $g'$ as a child of $g$ in $T$ (and thus in $F$);
```

7.3.3 Structural Enhancement

We now consider the skeletal structure of a form, i.e., the initial positioning of form-elements and form-groups relative to each other. There is no single optimal positioning, as one can easily determine by looking at the (presumably well-designed) forms created for competing web merchants, such as Travelocity, Expedia, and Orbitz. However, we can subjectively criticize at least some structures. For example, Consider the form shown in Fig. 7.1. We see five form-groups two of which are: ADDRESS and ZIP CODE. Just by looking at the form, it might be unclear to a user whether ADDRESS denotes the address of the customer or of the broker. The group labels themselves do not make the form unambiguous.

The root cause of poor designs as in Fig. 7.1 is that the structure of the form directly reflects the schema of the database. Novice human designers may do this since they do not know any better. Even more experienced human designers may sometimes do this because it is the path of least effort. This direct reflection of database structure is endemic to automatic and semi-automatic form generation tools [47, 57, 58, 66].

The problem with this approach to form design is that database schemas are designed for efficiency in storage, efficiency in query evaluation and minimization of redundancy among several other performance goals. Forms, on the other hand, should be designed for usability and expressive power (usually a trade-off is made between these two often conflicting goals). Hence it is not wise to emulate schema structure when structuring query forms. One of the main problems is normalization—entities in a database are often fragmented considerably to reduce duplication of data due to functional dependency between columns. Hence one way to improve form structure is to de-normalize form-groups that correspond to individual tables in the database. Displaying the explicit join relationships between the tables (in the form) is one approach, but form designers traditionally shy away from this to make forms simpler.

Returning to our example in Fig. 7.1, we recognize the relationship of ZIP CODE and ADDRESS with the entity CUSTOMER at form-generation time, and use it to re-structure the form, making it more hierarchical. By nesting form-group ZIP CODE within ADDRESS and form-group ADDRESS within CUSTOMER, the relationship between them, which is clear in the schema (due to primary key–foreign key relationships), is now also clear to a user. By analyzing primary key–foreign key relationships in the schema and ignoring relationship tables (these can be identified using heuristics), the right way to nest groups can be determined. As outlined in Algorithm 9, the system can be made to automatically determine which groups to nest which other groups. If similar and related form-groups are placed together, once one of them is found, the others become easier to locate. Consider the query:

Q': *Which tier-1 customers live in Vancouver, Canada?*

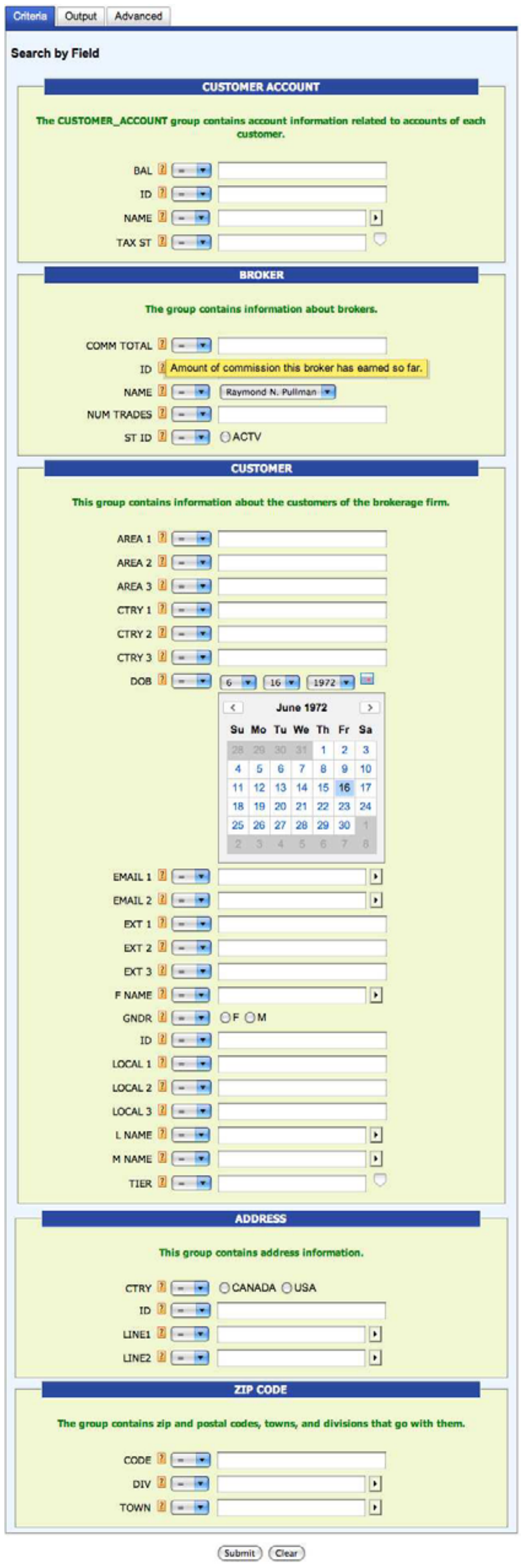
While this form in Fig. 7.1 can be used to express this query, its usability can be improved. The query cost defined by Formula 13 can be computed as follows.

$$\begin{aligned}
C(Q', f) &= [C_S(\textsf{C.TIER}) + C_U(\textsf{C.TIER})] \\
&+ [C_S(\textsf{A.CTRY}) + C_U(\textsf{A.CTRY})] \\
&+ [C_S(\textsf{Z.TOWN}) + C_U(\textsf{Z.TOWN})] \\
&= [(3 * 1 + 21 * 1) + 1 * 1] \\
&+ [(4 * 1 + 1 * 1) + 1 * 1] \\
&+ [(5 * 1 + 3 * 1) + 1 * 1] \\
&= [25] + [6] + [9] \\
&= 40
\end{aligned}$$

While enhancing the form's structure, two form-groups, ADDRESS and ZIP CODE are moved into the CUSTOMER group. Even a query might not involve form-elements from these groups, the overall cost of the query may be reduced simply by making the other form-elements easier to find. Using the enhanced form (Fig. 7.8), the cost of query Q' now becomes

$$
\begin{aligned}
C(Q', f) &= [C_S(\text{C.TIER}) + C_U(\text{C.TIER})] \\
&+ [C_S(\text{A.CTRY}) + C_U(\text{A.CTRY})] \\
&+ [C_S(\text{Z.TOWN}) + C_U(\text{Z.TOWN})] \\
&= [(1 * 1 + 21 * 1) + 1 * 1] \\
&+ [(1 * 1 + 1 * 1) + 1 * 1] \\
&+ [(1 * 1 + 3 * 1) + 1 * 1] \\
&= [24] + [3] + [5] \\
&= 32
\end{aligned}
$$

The difference in the total cost arises from having to scan fewer form-groups to locate the form-elements ADDRESS.CTRY and ZIP CODE.TOWN. Since ADDRESS is located within CUSTOMER, once the condition on CUSTOMER.TIER has been specified, the user need not examine other top-level form-groups to find ADDRESS.CTRY. Hence the cost of specifying this condition has been reduced. Similarly rather than scan the entire form (all five form-groups) to find ZIP CODE, having already scanned CUSTOMER the user now knows exactly where to find ADDRESS (to specify the condition ADDRESS.CTRY = “CANADA” and then ZIP CODE to specify that ZIP CODE.TOWN = “Vancouver”. The savings in cost is reflected below. Notice the reduced search cost for ADDRESS.CTRY and ZIP CODE.TOWN.



Search by Field

CUSTOMER ACCOUNT

The CUSTOMER_ACCOUNT group contains account information related to accounts of each customer.

BAL
ID
NAME
TAX ST

BROKER

The group contains information about brokers.

COMM TOTAL
ID — Amount of commission this broker has earned so far.
NAME — Raymond N. Pullman
NUM TRADES
ST ID — ACTV

CUSTOMER

This group contains information about the customers of the brokerage firm.

AREA 1
AREA 2
AREA 3
CTRY 1
CTRY 2
CTRY 3
DOB — 6 / 16 / 1972

June 1972

Su	Mo	Tu	We	Th	Fr	Sa
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1
2	3	4	5	6	7	8

EMAIL 1
EMAIL 2
EXT 1
EXT 2
EXT 3
F NAME
GNDR — F M
ID
LOCAL 1
LOCAL 2
LOCAL 3
L NAME
M NAME
TIER

ADDRESS

This group contains address information.

CTRY — CANADA USA
ID
LINE1
LINE2

ZIP CODE

The group contains zip and postal codes, towns, and divisions that go with them.

CODE
DIV
TOWN

Figure 7.5: A form enhanced by supplementing it with textual descriptions and hints

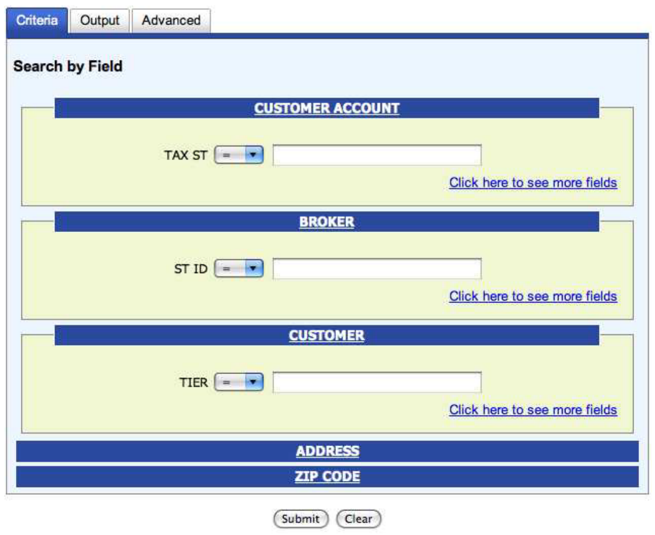


Figure 7.6: A form enhanced by hiding less important form-elements and form-groups based on a query workload

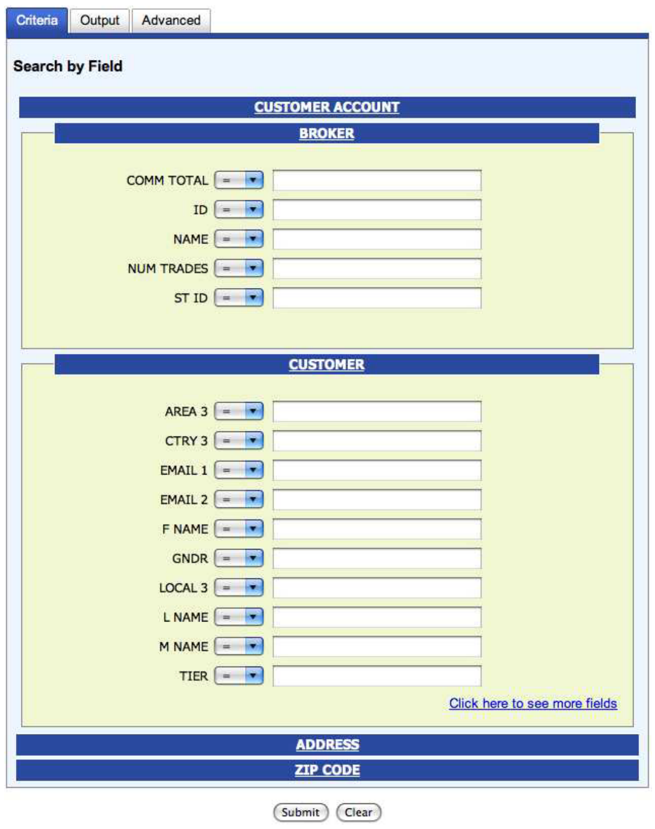
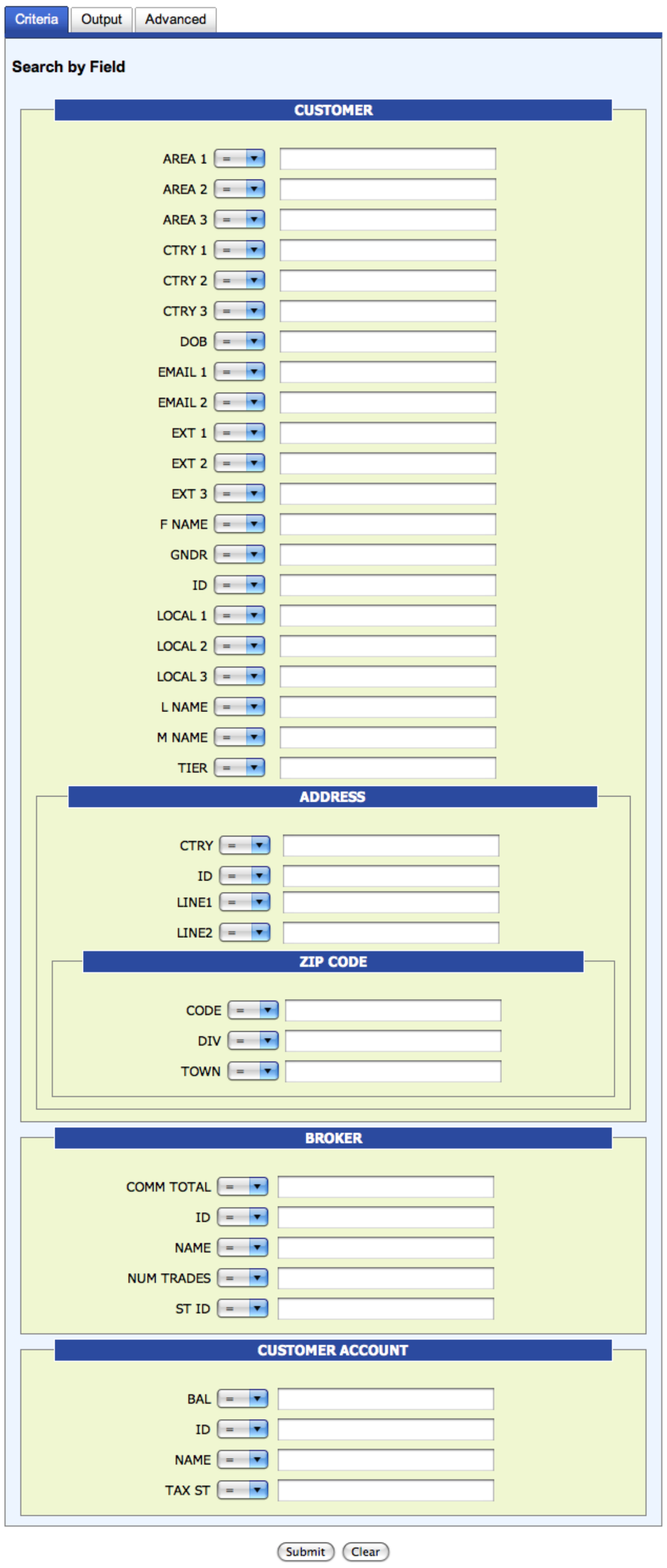


Figure 7.7: A form enhanced by hiding less important form-elements and form-groups based on schema properties



Search by Field

CUSTOMER

AREA 1	=	
AREA 2	=	
AREA 3	=	
CTRY 1	=	
CTRY 2	=	
CTRY 3	=	
DOB	=	
EMAIL 1	=	
EMAIL 2	=	
EXT 1	=	
EXT 2	=	
EXT 3	=	
F NAME	=	
GNDR	=	
ID	=	
LOCAL 1	=	
LOCAL 2	=	
LOCAL 3	=	
L NAME	=	
M NAME	=	
TIER	=	

ADDRESS

CTRY	=	
ID	=	
LINE1	=	
LINE2	=	

ZIP CODE

CODE	=	
DIV	=	
TOWN	=	

BROKER

COMM TOTAL	=	
ID	=	
NAME	=	
NUM TRADES	=	
ST ID	=	

CUSTOMER ACCOUNT

BAL	=	
ID	=	
NAME	=	
TAX ST	=	

Figure 7.8: A form enhanced by altering its structure

7.4 Evaluation

We evaluated our proposed form enhancements in two ways: analytically, using our cost formula on a real dataset with real user queries, and empirically, by measuring the time taken by users to express queries on both original and enhanced forms.

Our primary hypothesis is that forms are more usable after the techniques developed in this paper have been applied. A more usable form should incur less cost on part of the user to express desired queries. This cost may include many aspects, such as cognitive effort, confidence in correctness of query expression, and so on. But such aspects are hard to measure in isolation, and are generally believed to have an impact on the time to express a query. The time taken by a user, of course, is an important cost in itself, and it is easy to measure. Therefore, we will use time to express query as our surrogate for (inverse of) usability.

Our secondary hypothesis is that the cost formula we developed accurately reflects actual user cost. By correlating actual user time against cost predicted by the formula in the experiments mentioned above, we can test this hypothesis as well without requiring additional experiments.

7.4.1 Experimental Methodology

We have proposed 4 enhancement techniques, each of which can be applied independently. These are, respectively: Textual Enhancement, Interactional Enhancement, Layout Enhancement (via selective hiding), and Structural Layout Enhancement. Layout Enhancement via selective hiding comes in two flavors: based on query workload and based on queriability analysis. Since each of the 4 enhancements is independent, each can be either applied or not, with two ways to apply Layout Enhancement. We apply each enhancement in isolation, and also generate two forms with all enhancements—since Layout Enhancement can be done in two mutually exclusive ways, we generate one with each of them to go along with all of the other enhancements. Thus we have a total of 8 different choices of enhancement combinations (listed in Table 7.2) if we include one with no enhancement. We measure all 8.

Form	Enhancement(s)
1	None
2	Textual
3	Structural
4	Layout 1 (Queriability-based)
5	Layout 2 (Query-based)
6	Interactional
7	All (with Layout 1)
8	All (with Layout 2)

Table 7.2: Combinations of Form Enhancements

The starting point, for all of these enhancements, is an original un-enhanced query form. We use an automatic form-generator [48] to generate forms based on the selected application.

7.4.2 Dataset

We chose the TPC-E benchmark [12] dataset for our experiments, because of its wide availability. This is a moderately complex dataset that simulates the database of a stock brokerage firm. It has 33 tables in three main categories: Customer, Broker and Market. Each table has between 2 and 24 columns. By adjusting the benchmark parameters for data generation, we created a dataset that was 7.1 GB in size (when stored as tab-delimited flat files). Unlike earlier TPC benchmarks, the data values that constitute the content of the database are, for the most part, meaningful[3].

7.4.3 Query Workload

While the TPC-E data is an attractive base for our experiments, the accompanying queries are not. Most of the queries are rather complex, involving PL/SQL, and several are

[3]This is an important feature because it allows us to make use of many of our form enhancement techniques that are based on attribute values

updates. These queries are far too complex to be represented in a reasonable form. So we decided to create our own set of queries against the TPC-E data, by polling the general public for queries they find interesting to ask of this financial data store.

Specifically, we manually created 4 sub-schemas of TPC-E, each roughly equal in size, and with limited overlap between sub-schemas. We made use of Amazon's Mechanical Turk [1], an Amazon web service, to present the TPC-E database to a diverse audience and obtain from them a list of queries whose results would be of some interest to them. Using this approach, we obtained 400 queries from 20 respondents to the four sub-schemas of the TPC-E schema (five queries each). Some of the queries suggested were similar, but we did not eliminate duplicates since we thought that retaining the full set would be more representative of actual user queries.

We then used the queries in this set for our experiments, both user studies and model-computation. Note that all queries obtained were simple English sentences. So including a query in a user study was straightforward. For many other purposes, precise query statements were required, and these were generated by hand from the English specification.

7.4.4 Analytical Evaluation

For each of the four different forms, for each query in the workload, we compute the cost of using the form according to Formula 14. Then we assign to each form a final cost computed as the sum total of the costs of each query in the workload (that can be expressed using it) normalized by the number of such queries. This gives us the base cost for each form before enhancements.

Then, in turn, we apply each of 7 possible combinations of enhancements to each form, and compute the cost again, using the same procedure. This permits us to compare costs across 8 different versions of each of the 4 forms.

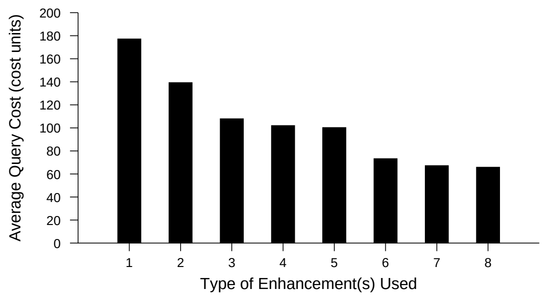


Figure 7.9: Analytical Usability of each Enhancement Type

Design and Measurements

We use four different forms each corresponding to a different subset of the database schema. For each query in the workload, we compute the cost of specifying all interface elements needed to issue the query correctly. This will depend on the number, location and relative position (relative to the previous form-element in the sequence) of each form-element. The cost we assign to a particular form is simply the sum total of the costs of each query in the workload (that can be expressed using it) normalized by the number of such queries. We apply different enhancement techniques both in isolation and in compatible combinations and record the costs they incur. This enables us to compare individual techniques and also shows us how they might influence one another.

Procedure

We start with the original un-enhanced forms. We compute the decision cost of each form-element and form-group and record them. Using these costs, we can compute the search cost of each form-element (using Formula 12) in the form. The usage cost of each form-element is then computed and added to the search cost of the form-element to obtain the total cost that the form-element adds to the cost of a query that involves it. We separate the query workload into four query subsets, one for each of the four forms created for this evaluation. Each query can only be expressed using one of the four forms and this allows us to partition the workload easily. We iterate through each query subset, examining one query at a time and determining which form-elements are needed to express it. For each relevant form-element, we retrieve its total cost (search cost plus usage cost) and add all of them together to compute the cost of the query. Next, using Formula 14, we compute the usability of each of the four forms using all the queries in the subset of the workload associated with it.

Since usability is dependent on the query workload, we do not compare the usability of different forms with each other. Instead, we apply different enhancements to each form and compare the performance of the resulting forms with the original form and with each other. We record our observations and use them to draw conclusions about their effectiveness and synergy.

Results

In Fig. 7.9, we show the results for one of the four sub-schemas. We observe that each enhancement leads to a form that is more usable than the original un-enhanced form. The biggest improvement is made by the interactional enhancement. Further, when we apply all of them together, the resulting form becomes even easier to use.

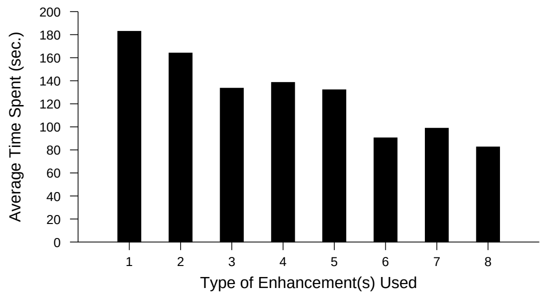


Figure 7.10: Task Times for each Enhancement Type

7.4.5 User Study

Our second evaluation method is a user study in which people, who volunteer to be test subjects, attempt a series of querying tasks each requiring a pre-specified query

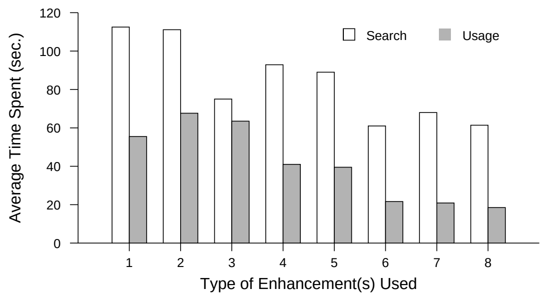


Figure 7.11: Search and Usage Costs for each Enhancement Type

(in English) to be expressed using the form provided. We measure response times of each user for each task and use them to evaluate the effectiveness of our proposed form enhancements.

Design and Measurements

As in our analytical evaluation, we start with the same four forms each allowing a limited range of queries to the TPC-E database. We select eight queries (two for each sub-schema, chosen at random from the query workload described in Sec. 7.4.3) and use them as querying tasks that test subjects are asked to express using the forms provided. Thus we have 8 such tasks—we order them such that the distance between two tasks with the same sub-schema is maximum (to limit the effects of learning the schema).

We not only measure the time taken for a query to be specified completely but also the time spent on individual form-elements (both on search and on usage) to obtain greater insight into the advantages and disadvantages of each form enhancement.

Tasks and Subjects

Each query is associated with eight tasks each of which involves a different form—these are the original form (un-enhanced), the form textually enhanced, the structurally enhanced version of the form, the form layout enhanced based on queriability, layout enhanced based on the query workload, the interactionally enhanced form and finally the two versions with all (non-mutually exclusive) enhancements combined. Each subject was assigned eight tasks: one with each of the eight enhancement combinations, one with each of the eight queries, and two for each of the four schemas. Their order was randomly selected. Thus in the final evaluation, each enhancement combination will have had an equal contribution from each of the eight users and will have been associated with a different query for each user.

Our test subjects again are drawn from the general public using [1] but this time we record their age, education level and comfort-level and expertise with database querying. Our techniques should result in forms that are usable by casual users, but we need this information to put our results in perspective. Four of our eight users did not have a college degree (but may have taken some college courses), two had bachelor's degrees and the remaining two had graduate degrees. Four of them had had some experience with databases, while the other four did not. Their ages were 20, 21, 25, 32 (two) and 35 (three).

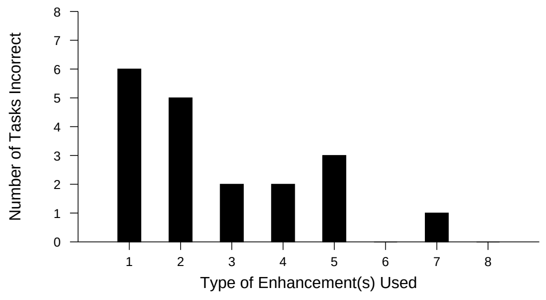


Figure 7.12: Task Errors for each Enhancement Type

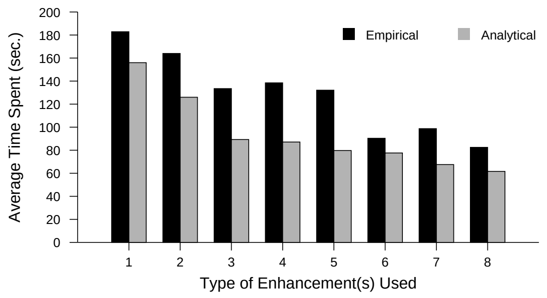


Figure 7.13: Empirical vs. Analytical Usability of each Enhancement Type

7.4.6 Results

We measured the time taken for each user to complete each task and added a two-minute penalty for tasks that were incorrectly done (since in a real-world situation, this would usually involve the user having to go back to the form and re-issue the query). We aggregated these response times by enhancement combination and our results can be seen in Fig. 7.10. We also used the event logs to isolate the search cost and usage cost for each task and aggregate these in the same way (see Fig. 7.11). Finally we counted the number of tasks that were partially or completely incorrect and show these in Fig. 7.12. We observe that enhancements lead to improvement in usability. Further, the search costs show marked improvement in each of the enhanced forms (more so when used in combinations) and the interactional enhancement greatly reduces the usage costs. We also note that the un-enhanced form has the most instances of error which reduces as enhancements are applied.

Effectiveness of Cost Model

We compare the results of the two experiments to see how they correlate with each other. Since the analytical experiment is based on our cost model and the user study gives us actual response times, we can compare these, since they involve the same dataset and querying tasks, to measure the accuracy of our cost formula. If one cost unit is considered one second, the comparison in terms of seconds per task (on average) is shown in Fig. 7.13.

Comparison with Human-generated Forms

We also compared the usability of our most enhanced form with that of a human-generated form taken from a dataset with a similar purpose. Querying tasks for this study were chosen such that they applied to both databases, especially to both forms. We recruited 10 users for this study and gave them each 10 tasks (5 involving the manually-generated form and 5 to the automatically-generated and enhanced form). Our observations are shown in Fig. 7.14. On average, each task took 49.7 seconds on the human-generated form versus

54.6 seconds on the auto-generated form. We used a 60-second penalty for incorrect responses.

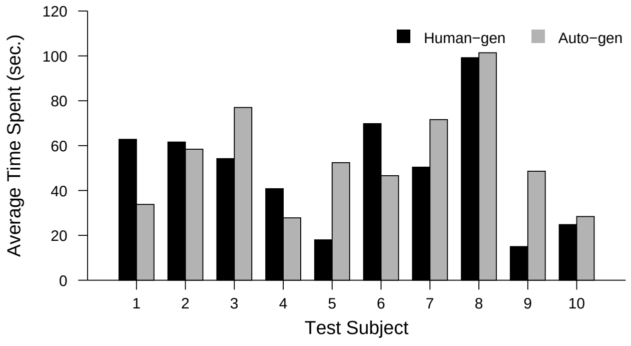


Figure 7.14: Human-generated Forms vs. Automatically-generated Forms

CHAPTER VIII

CONCLUSIONS

The query interface is an important component of a database. It is the vehicle provided to each user to navigate the data and extract desired information. As valuable as the content of the database is, it is the interface that determines the accessibility of the data and ultimately, the usefulness of the database. Among all known querying mechanisms, form-based query interfaces are widely regarded as the most intuitive and the most effective, especially to non-database-experts. Such interfaces are an important part of databases in the real world [19]. However, they are not without their own shortcomings—good forms take substantial time and manual-effort to design and generate, and even the best forms cannot guarantee the satisfaction of all their users. In this thesis, we introduced a new type of form which resembles many of today's forms (which are manually generated) in appearance, but are more rigidly structured internally (Chapter III). This structure allows us to both generate and analyze forms in an automated fashion. These forms are now machine-readable which allow them to be created, rendered and executed without human intervention. We also showed how a filled form was translated to a machine-readable query that can be executed by a standard database system.

In Chapter IV, we introduced a mechanism to generate a forms-based interface with nothing more than the database itself. We introduced metrics to estimate the usefulness of various schema components from a querying angle based on an analysis of the schema and the content of the database. We evaluated our system's performance on two public benchmark datasets and query sets. We also conducted experiments on an in-house compiled database available to the public. We observed that the interfaces we generate satisfy a large fraction (60-90%) of actual queries [48]. In comparison, our analysis of deployed databases shows that this level of coverage is difficult to achieve today, even with expert design.

In Chapter V, we presented mechanisms to refine an initial form-set (also automatically) given a real or expected query workload. We introduced notions of query-similarity and similarity-based clustering to enable our system to generate (or modify) the fewest forms necessary to support all of the observed queries. We presented a study of system performance using a real query trace, as well as queries from two standard XML benchmarks [47, 50].

In spite of the best efforts and intentions of an interface developer (or an automated system) a user still might not be able to use the interface to express a desired query. In Chapter VI, we proposed a way for users to take matters into their own hands and alter the deployed forms to serve their needs without a steep learning curve. We introduced a form manipulation language to define these alterations and implemented it in a visual editor that allows users to modify existing forms to support previously unsupported queries. Large-scale modifications can be made iteratively one simple step at a time. We conducted experiments on real users that validated our system's usefulness [49]. Using our form editor, users of databases who have little or no knowledge of sophisticated query languages and tools can still formulate queries giving them virtually unbridled access to their data of interest. While the technologies suggested in this chapter can be used to make major changes to existing forms and even to define new forms from scratch, they are of greatest value when the modifications required are small.

After addressing the expressive power side of forms for majority of this thesis, we analyzed their usability side in Chapter VII. We developed a cost model for form usage and used this to compare multiple forms on the basis of usability. Further, this understanding also led us to come up with different ways to enhance the usability of a given form. We applied these enhancements and showed the increase in form usability both analytically (using the cost model) and empirically (via a controlled user study).

In conclusion, we feel that such an automated self-managing interface-builder can significantly reduce if not eliminate the effort on the part of database owners to provide users access to their data, and also maximize user satisfaction with these data resources. While we presented our approach to form building using an XML implementation, in principle, our concepts are equally applicable to other data models including relational databases.

8.1 Future Work

A comprehensive evaluation comparing automatically-generated forms with human-generated forms side-by-side would give a clear indication of the relative usability of the forms we generate. Such an evaluation could involve hiring a human expert who can be asked to develop forms by hand for a given schema-data setup and compare them directly with forms generated automatically for the same database. A user study would best reflect this comparison since it involves people who would actually use the database in question to retrieve some information. Such a human-intensive evaluation is potentially very expensive but is certainly worth pursuing, in future, to extend this research.

Manual as well as automatic form generation techniques each have their advantages and disadvantages. It is potentially interesting to devise hybrid form interfaces, i.e., those that are functionally in between the two extremes and borrow from both approaches in a way that maximizes their strengths and minimizes their shortcomings. Designing such interfaces is one of the logical next steps in this research direction and it can yield greater insight and understanding into the dynamics of database querying.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Amazon Mechanical Turk: `http://aws.amazon.com/mturk/`.

[2] Apache Struts project: `http://struts.apache.org/`.

[3] Apple - iLife - iTunes - Smart Playlists: `http://www.apple.com/lae/itunes/smartplaylists.html`.

[4] Cold Fusion: `http://www.adobe.com/products/coldfusion/`.

[5] Django Web Framework: `http://www.djangoproject.com/`.

[6] Geoquery Database: `http://www.cs.utexas.edu/users/ml/geo.html`.

[7] Jobsquery Database: `http://www.cs.utexas.edu/users/ml/nldata/jobquery.html`.

[8] MiMI: Michigan Molecular Interaction Database – `http://mimi.ncibi.org`.

[9] NCBI BLAST: `http://www.ncbi.nlm.nih.gov/blast/Blast.cgi`.

[10] Refbase users: `http://wiki.refbase.net/index.php/Refbase_users`.

[11] Ruby on Rails: `http://www.rubyonrails.org/`.

[12] TPC Benchmark E: `http://www.tpc.org/tpce/tpc-e.asp`.

[13] TurboGears: `http://www.turbogears.org/`.

[14] WebObjects: `http://www.apple.com/webobjects/`.

[15] XMark: An XML Benchmark Project: `http://www.xml-benchmark.org/`.

[16] R. Abraham. FoxQ–XQuery by Forms. In *HCC*, 2003.

[17] E. Augurusa, D. Braga, et al. Design and Implementation of a Graphical Interface to XQuery. In *SAC*, 2003.

[18] F. Benzi, D. Maio, and S. Rizzi. Visionary: a Viewpoint-based Visual Language for Querying Relational Databases. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, 10(2), 1999.

[19] P. Bernstein et al. The Asilomar Report, 1998.

[20] D. Braga, A. Campi, and S. Ceri. *XQBE* (*XQ*uery *B*y *E*xample): A visual interface to the standard XML query language. *ACM Trans. Database Syst.*, 30(2), 2005.

[21] M. E. Califf and R. J. Mooney. Relational Learning of Pattern-Match Rules for Information Extraction. In *AAAI*, 1999.

[22] F. Capobianco, M. Mosconi, and L. Pagnin. Looking for convenient alternatives to forms for querying remote databases on the Web: a new iconic interface for progressive queries. In *Advanced Visual Interfaces*, 1996.

[23] T. Catarci et al. An Ontology Based Visual Tool for Query Formulation Support. In *ECAI*, 2004.

[24] T. Catarci and G. Santucci. Are Visual Query Languages Easier to Use than Traditional Ones? An Experimental Proof. In *HCI '95: Proceedings of the HCI'95 Conference on People and Computers X*, 1995.

[25] S. Ceri et al. XML-GL: A Graphical Language for Querying and Re-structuring XML Documents. *Computer Networks*, 31(11-16), 1999.

[26] K. C.-C. Chang, B. He, and Z. Zhang. Toward Large Scale Integration: Building a MetaQuerier over Databases on the Web. In *CIDR*, 2005.

[27] J. Choobineh. Formflex: A User Interface Tool for Forms Definition and Management. *Human Factors in Management Information Systems*, 1988.

[28] J. Choobineh, M. V. Mannino, and V. P. Tseng. A Form-Based Approach for Database Analysis and Design. *Communications of the ACM*, 35(2), 1992.

[29] S. Cohen, Y. Kanza, Y. Kogan, Y. Sagiv, W. Nutt, and A. Serebrenik. EquiX–A search and query language for XML. *JASIST*, 53(6), 2002.

[30] A. Cornuéjols. Getting Order Independence in Incremental Learning. In *European Conference on Machine Learning*, 1993.

[31] D. W. Embley. NFQL: The Natural Forms Query Language. *ACM Transactions on Database Systems*, 1989.

[32] M. Erwig. A Visual Language for XML. In *IEEE Symposium on Visual Languages*, 2000.

[33] D. Esposito. *Programming Microsoft ASP.NET 2.0 Core Reference*. Microsoft Press, 2005.

[34] B. Forta, R. Camden, L. Chalnick, and A. C. Buraglia. *Macromedia ColdFusion MX 7 Web Application Construction Kit*. Macromedia Press, 2005.

[35] K. Gajos and D. S. Weld. SUPPLE: Automatically Generating User Interfaces. In *IUI*, 2004.

[36] N. C. Goodwin. Functionality and Usability. *Communications of the ACM*, 30(3), 1987.

[37] D. P. Groth. Visual Representation of Database Queries using Structural Similarity. In *IV*, 2003.

[38] S. Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly, 1996.

[39] M. Halvorson. *Microsoft Visual Basic 2005 Step by Step*. Microsoft Press, 2005.

[40] P. Hanna. *JSP 2.0: The Complete Reference*. McGraw-Hill Osborne Media, second edition edition, 2002.

[41] Harald Schöning. A Graphical Interface to a Complex-Object Database Management System. In *Interfaces to Databases (IDS)*, 1992.

[42] D. J. Helm and B. W. Thompson. An Approach for Totally Dynamic Forms Processing in Web-Based Applications. In *ICEIS (2)*, 2001.

[43] M. Y. Ivory and M. A. Hearst. The state-of-the-art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4), 2001.

[44] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Socit Vaudoise des Sciences Naturelles*, 37, 1901.

[45] H. V. Jagadish et al. TIMBER: A Native-XML Database. *VLDB Journal*, 11(4), 2002.

[46] M. Jayapandian, A. Chapman, et al. Michigan Molecular Interactions (MiMI): Putting the Jigsaw Puzzle Together. *Nucleic Acids Research (Database Issue)*, 35, 2007.

[47] M. Jayapandian and H. V. Jagadish. Automating the Design and Construction of Query Forms. In *ICDE*, 2006.

[48] M. Jayapandian and H. V. Jagadish. Automated Creation of a Forms-based Database Query Interface. In *VLDB*, 2008.

[49] M. Jayapandian and H. V. Jagadish. Expressive Query Specification through Form Customization. In *EDBT*, 2008.

[50] M. Jayapandian and H. V. Jagadish. Automating the Design and Construction of Query Forms. *TKDE*, 2009.

[51] D. R. O. Jr., S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal Interaction using Xweb. In *UIST*, 2000.

[52] M. Levandowsky and D. Winter. Distance between Sets. *Nature*, 234(5323), 1971.

[53] S. Melnik. *Generic Model Management: Concepts and Algorithms.* Chapter 7. PhD thesis, University of Leipzig, 2004.

[54] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *PODS*, 2002.

[55] K. Mitchell and J. Kennedy. DRIVE: An Environment for the Organized Construction of User-Interfaces to Databases. In *Interfaces to Databases (IDS-3)*, 1996.

[56] P. Mukhopadhyay and Y. Papakonstantinou. Mixing Querying and Navigation in MIX. In *ICDE*, 2002.

[57] K. D. Munroe and Y. Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *VDB*, 2000.

[58] N. Murray, N. Paton, and C. Goble. Kaleidoquery: A Visual Query Language for Object Databases. In *Advanced Visual Interfaces*, 1998.

[59] J. Nichols, D. H. Chau, and B. A. Myers. Demonstrating the Viability of Automatically Generated User Interfaces. In *CHI*, 2007.

[60] J. Nichols, B. A. Myers, and K. Litwack. Improving Automatic Interface Generation with Smart Templates. In *IUI*, 2004.

[61] J. Nichols, B. A. Myers, and B. Rothrock. UNIFORM: Automatically Generating Consistent Remote Control User Interfaces. In *CHI*, 2006.

[62] J. Nichols, B. Rothrock, D. H. Chau, and B. A. Myers. Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances. In *UIST*, 2006.

[63] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: Querying and Reporting Semistructured Data. In *SIGMOD*, 2002.

[64] A. Parush, R. Nadir, and A. Shtub. Evaluating the Layout of Graphical User Interface Screens: Validation of a Numerical Computerized Model. *International Journal of Human-Computer Interaction*, 10(4), 1998.

[65] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. Query Set Specification Language. In *WebDB*, 2003.

[66] M. Petropoulos, Y. Papakonstantinou, and V. Vassalos. Graphical query interfaces for semistructured data: the QURSED system. *ACM Transactions on Internet Technologies*, 5(2), 2005.

[67] M. Petropoulos, V. Vassalos, and Y. Papakonstantinou. XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces. In *WWW*, 2001.

[68] A. Puerta and J. Eisenstein. Towards a General Computational Framework for Model-based Interface Development Systems. In *IUI*, 1999.

[69] A. Puerta and P. Szkeley. Model-based Interface Development. In *CHI*, 1994.

[70] R. E. Sabin and T. K. Yap. Integrating Information Retrieval Techniques with Traditional DB Methods in a Web-Based Database Browser. In *SAC*, 1998.

[71] A. R. Schmidt et al. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[72] A. Sengupta and A. Dillon. Query by Templates: A Generalized Approach for Visual Query Formulation for Text Dominated Databases. In *ADL*, 1997.

[73] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, first edition, 1949.

[74] D. Simovici, N. Singla, and M. Kuperberg. Metric Incremental Clustering of Nominal Data. In *ICDM*, 2004.

[75] S. Sinha et al. Accessing a Medical Database using WWW-Based User Interfaces. Technical report, 1998.

[76] L. R. Tang and R. J. Mooney. Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing. In *ECML*, 2001.

[77] A. Tornqvist, C. Nelson, and M. Johnsson. XML and Objects-The Future for E-Forms on the Web. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*. IEEE Computer Society, 1999.

[78] V. Venkatesh and F. D. Davis. A Model of the Antecedents of Perceived Ease of Use: Development and Test. *Decision Sciences*, 27(3), 1996.

[79] C. Yu and H. V. Jagadish. Schema Summarization. In *VLDB*, 2006.

[80] Z. Zhang, B. He, and K. C.-C. Chang. Light-weight Domain-based Form Assistant: Querying Web Databases On the Fly. In *VLDB*, 2005.

[81] M. M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. In *VLDB*, 1975.