Expressive Query Specification through Form Customization [*]

Magesh Jayapandian
Department of EECS
University of Michigan
jmagesh@umich.edu

H. V. Jagadish
Department of EECS
University of Michigan
jag@umich.edu

ABSTRACT

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. However, a typical form is static and can express only a very limited set of queries. Without room for change, query specification is limited by the expertise and vision of the interface developer at the time the form was created. If an available form cannot express a desired query, the user is stuck.

In this paper, we propose a mechanism to let a user modify an existing form to express the desired query. These modifications are themselves specified through filling forms to create an expression in an underlying form manipulation expression language we define. The technical sophistication required to modify forms is not much greater than form filling.

We have developed a form editor that implements this form manipulation language. We have also developed a query generator that modifies the form's original query based on a user's changes. We show, by means of a controlled user study, that this tool provides an effective means for specifying complex queries.

1. INTRODUCTION

Filling forms is easy. A user needs little training to learn how to fill a form correctly. In contrast, traditional database querying requires users to be able to write code in SQL (or XQuery) and to also know the database schema. It is no wonder that form-based query interfaces are so widely used by databases today.

The main drawback of a form-based query interface is that it is restrictive. A typical form is designed to do one thing, and it does not permit the user to express queries that differ from this one thing. As rich as a data collection might be, it cannot be fully utilized if its query interface is limiting. On the other hand, 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 is for users to comprehend and use it. Complexity and expressivity are conflicting goals for any forms-based interface and a trade-off is usually made.

A second drawback of current forms is that few if any of them provide users the option to specify the format and content of the query's result. Most forms, in our experience, only allow a user to specify query conditions that the results must satisfy. For example when the form in Fig. 1(a) is submitted, it produces a brief listing shown in Fig. 1(b). Typically result display is handled at extremes: either by displaying a result overview that contains multiple results per page but limited information about each, or by displaying one result per page (showing all the available information about that result) with links to the other results. The former approach, while concise, can be insufficient for users who then have to click on a number of individuals before deciding which one is closest to what they had in mind. On the other hand, the latter, which displays an exhaustive dump of all the attributes of a result object (one at a time), can be overwhelming. For example, if a person just wants to know the price, duration, fare class and seating capacity of flights from Memphis to Chicago within a specific date range, neither format is ideal. The overview display does not show all the required information, while the exhaustive display shows too much information about each result and does not show all the results together.

Often, an interface will provide an "advanced" form that provides a user with more choices than the "basic" form. However, the above drawbacks still remain in spite of the additional form complexity. To illustrate, consider the first drawback, that of limited expressivity, in the context of a database of flight listings at the website of a popular airline, Northwest Airlines. Despite the additional complexity, the advanced search form (Fig. 2) still only supports conjunctive selection queries which might be insufficient for some users. Consider the following queries to this database:

Q1. Display flights from Memphis to Chicago ordered by travel time, shortest to longest.
Q2. Show me flights from Memphis to an airport in or around Austin (within 30 miles) that depart in the next 3 days.
Q3. List all Memphis-Chicago flights today that cost less than $400.

None of these queries can be expressed using the form directly. The form 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. Without the ability to change a form, the only option a user has is to specify a query supported by the available form that returns a superset of the desired results, and examine the results manually to satisfy the unsupported query criteria. For example, in the case of $Q1$, while the advanced form does not sort the results by departure time, it can find all flights from Memphis to Chicago. A user can view the results of this partial query (which does contain the duration for each flight), and sort them mentally to find

[*]Supported in part by NSF 0438909 and NIH 1-U54-DA021519.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

Figure 1: (a) Simple Search Form (b) Results (NWA.com)

the desired flight. This is a tedious process and can be error-prone, especially if the results span multiple pages. We propose form customization as an approach to allow users to modify the forms they use to obtain precisely the results they seek. Of course, form customization is not necessary if a form is available that supports the desired query in its entirety. Moreover, if an advanced form is available, and is suited to the user's needs, the cost of switching to it will be less than that of modifying the basic form. The techniques developed in this paper are of value when the predefined (basic or advanced) forms do not meet the user's needs completely.

To draw an analogy, a user views forms in an interface as a customer would view pizzas at a restaurant. Just as the restaurant may offer several piazzas 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.

In this paper, we introduce the concept of form customization. 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. 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.

Tools that assist users in form creation have been around for over a decade. QURSED [26] is a system that eases the burden of form creators by effectively integrating the schema with the form editor visually, and by automatically constructing the equivalent declarative query template (incrementally) as the designer builds the form. The first difference from our work is that it is intended to be used by form developers, not users. Secondly, unlike this editor which generates forms from scratch, the problem we aim to solve is that of form customization, i.e., using existing "similar" forms as an effective starting point in query expression. Having chosen a form to customize, 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. Thus we are leveraging pre-existing complex queries, to support new complex queries with significantly less cost and user-expertise (which is within the capabilites of casual users as we will show in this paper). For instance, it is easy to imagine a very simple custom form for query $Q2$ above even though building it from scratch involves nesting, aggregation and joins which are non-trivial. FoxQ [10] and EquiX [20], like our system, are intended to be used by end-users. However, these tools also require users to build forms incrementally from a null starting point. Visual query builders that assist in query generation have been studied extensively [11, 13] and implemented in commercial products [4, 5, 6, 31]. These also require users to start from scratch making the burden much higher than that of form modification.

Figure 2: Advanced Search Form (NWA.com)

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.

To enable form customization, one needs to understand how forms are structured and given this understanding, how this structure can be altered ultimately by an end-user. Our first contribution is a careful characterization of form components, and separation of form structure and function in Sec. 2. We introduce a role-based separation of form fields that makes forms both more readable and easier to modify by end-users. We introduce a form expression language in Sec. 3, which describes the operations that can be performed to modify forms. Based on this, we present, in Sec. 4, the technique for dynamic form generation including how to design form layout. We discuss briefly, in Sec. 5, the provision for advanced users to modify forms to a greater extent than typical form users to demonstrate the expressivity of the interface. Given a filled form, we can generate an equivalent declarative query expression (since one cannot be hardcoded if the form is not static) which we present in Sec. 6. The experimental section that follows (Sec. 7), presents the results of a controlled user study we conducted to evaluate the system's effectiveness. While the current implementation is XML-based, we believe this 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.

2. FORM DEFINITION

Data providers have largely viewed forms just as user-friendly wrappers over declarative queries. This is evident from the absence of any formal language or generic model capable of representing forms. Given a query, the corresponding form has various components, each a collection of form controls, that correspond to different parts of the query expression. The composition of a form component depends on the query fragment it represents. For a given type of query fragment, such as a selection predicate or a result ordering attribute, different interface designers may design the corresponding form component differently. Consider a selection predicate on a date field. In a form, this query condition can be represented as a labeled text-box, a set of three drop-down lists (for day, month and year) or even a calendar widget that allows users to click on a date. Apart from form components themselves, the form's layout (the positioning of its components) is also arbitrary. While this heterogeneity among forms may not be of much interest to interface designers (who are only concerned with the usability of their own forms), in the interest of making forms easier to create and manipulate, a well-defined, systematic approach to form design is needed. Not only will such an infrastructure make forms easier to evolve, but it will also make visual interfaces easier to create, manage and troubleshoot. For such an approach to be feasible, we must define a canonical form representation as well as a set of form manipulation operators that allow the contents of a form to be altered. In this section we define a logical form representation distinct from its appearance. Having such an abstraction allows each form to have both an internal representation (which is machine-readable thereby making it possible to develop form manipulation tools) and an external representation (which is human-readable so people can use it). Under this logical representation, a form is defined as a collection of *form-elements* laid out according to their purpose and relationships with one other.

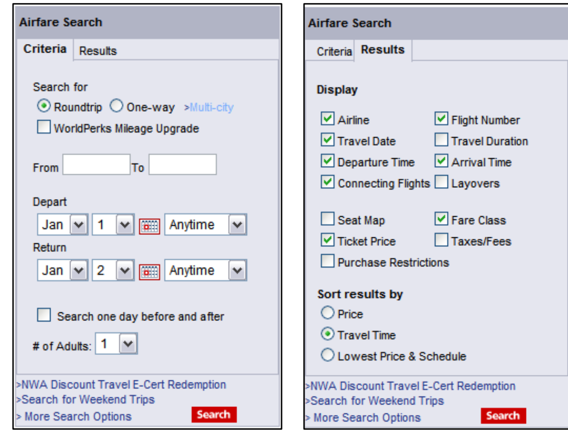

Figure 3: Two-paned version of Search Form in Fig. 1(a).

2.1 Form Elements

A form-element is an object designed to translate a user's input into a query fragment. A simple example is a labeled text-box for a search query that accepts a keyword from a user and creates a selection predicate for the corresponding table-column in the database. For completeness, we have identified the most common query operations in a declarative query language and designed form-elements for each of them. But this is not the complete set of form-element types. It is extensible within the framework of the form manipulation language (introduced in Sec. 3) to support querying needs.

Constraint Specification Element: A form-element that allows a user to specify a value for a data attribute. This is mapped to a selection predicate in the underlying query. For example, a text-box labeled Car Model in a reservation form allowing a user to specify the desired model is a constraint specification element.

Result Display Element: This form-element type enables a user to choose which fields to display in the result of the query. For example, a check-box labeled Show/Hide Pictures that can display or hide photographs of listings in a real-estate search form is a result display element. It allows users to customize the query's result.

Result Ordering Element: A result-ordering form-element lets users specify how the result of the form should be sorted. Flight reservation forms provide a set of radio-buttons marked Sort by price, Sort by schedule, etc. to specify the order in which matching flights are displayed. These are result ordering elements.

Aggregate Computation Element: This form-element denotes an aggregation operation, either a selection or a projection, in the underlying query. Some sports statistics query forms can be used to find the Highest Score by any team in any game at any venue. The form-element that allows users to find this maximum score is, in effect, an aggregate computation element.

Disjunction Element: A disjunction element is a set of constraint specification elements at least one of which must be satisfied. It is mapped to a set of selection predicates separated by *OR*'s in the underlying query. One use is to allow a user to specify a list of trusted sellers while searching for an item in an auction database.

Join Specification Element: While most forms hide inter-relationships between fields, the flexibility of a form can be increased by the presence of form-elements that allow users to specify these relationships. Consider a biological database query form allow users to find a protein that either interacts with a given protein (specified using constraint elements in the same form) or is homologous to it. A join specification element can give the user a choice between relating the two proteins by interaction or by homology.

2.2 Form Element Organization

A form with multiple form-elements can display them in many different ways, not all of which are meaningful. The arrangement of elements in the form (along with individual labels) indicate to a user what each element denotes and how it relates to other elements. The layout of these elements involves organizing them into collections spatially within the form, and intuitively labeling them to tell users what they represent. We term these collections as *form-groups*. In Fig. 2, for example, *'Where and When do you want to travel?'*, *'Who is going on this trip?'* and *'Do you have any preferences?'* are form-groups, From, To, etc. are form-elements, and the form controls shown include check-boxes (e.g. Nonstop flights only), text-boxes (e.g. From) and drop-down menus (e.g. Search for Cabin). Thus a form-element is just a set of form-controls, and a form-group is simply a set of form-elements or in some cases, other form-groups.

In the logical representation of a form, elements are grouped hierarchically with the placement of elements governed by how they are grouped. Grouping helps resolve ambiguities between like-named elements that may co-occur in a query. In the form, related form-elements, i.e., those belonging to a single group, are juxtaposed, to the maximum extent possible, to help users infer what they mean in the context of the query. Form-elements in a group can also be related to one another in terms of the query as we will discuss in Sec. 5. Multiple form-elements that contain attributes of the same entity (based on the database schema) are placed in a single group which represents the entity. In Fig. 2, form-groups are separated by panels and the label for each group is prominently indicated at the top left of the group. Our system uses labeled rectangular boxes to denote form-groups. Grouping is recursive—form-groups may be comprised of form-elements or other form-groups.

2.3 Canonical Structure

It is useful to think of a form as having three main components namely inputs, outputs and relationships. The fields whose values are filled in by a user can be thought of as the form's *inputs*. The fields whose values are returned to the user in the results of the query can be considered the *outputs* of the form (from a user's perspective). Finally, the associations between the schema entities that are queried are the entity *relationships* underlying the form. We find it convenient to separate these three components in the logical representation. Each of the three components can be represented as a tree owing to the hierarchical nature of grouping. We name these components as the *input tree*, *output tree* and *relationship tree*. These components are logically distinct and together form a canonical representation that can be used to represent any form.

DEFINITION 1. (CANONICAL FORM) *A form can be canonically represented as a 3-tuple, $F = \langle IT, OT, RT\rangle$, where: IT is its* input tree*, OT is its* output tree *and RT is its* relationship tree*.*

Each field in a form has a specific role with respect to the query that the form evaluates. Typically a field is used to specify a search criterion, making it an *input* to the underlying query. If a field pertains to the content or sort-order of the query's result, it is manipulating the *output* of the query. Consider the form in Fig. 1. It allows users to specify the origin, destination as well as departure and arrival dates for a desired flight, but it does not let the user choose what information about each flight he or she would like to see. A modification of this form that expands its functionality by allowing users to select the fields in the result display is shown in Fig. 3. This form has two panes—one for search criteria and a second to specify the content and order of search results. It is more flexible and intuitive, highlighting the benefits of logical separation. In this section, we introduce the *input tree* and *output tree*. These are the parts of the form most used by typical users. We introduce the *relationship tree* in Sec. 5 in which we describe more complex query specification intended only for advanced and database-savvy users. The primary purpose of the relationship tree is to capture inter-entity relationships and allow them to be modified if needed.

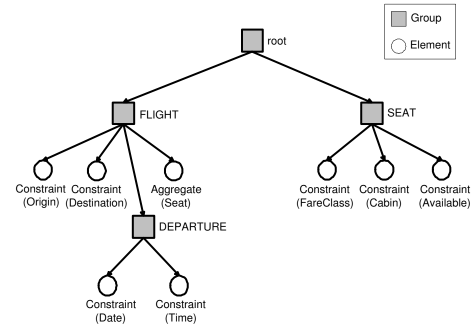

Figure 4: Logical Representation of a Form (Input Tree)

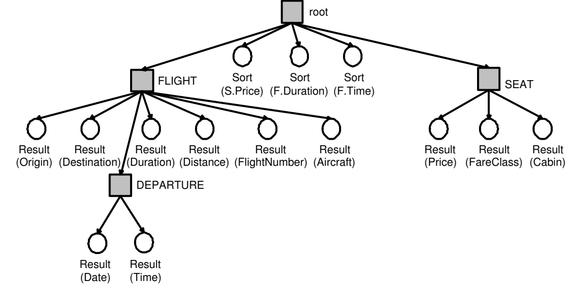

Figure 5: Logical Representation of a Form (Output Tree)

DEFINITION 2. (INPUT TREE) *The* input tree *of a form is a tree, $IT = \langle \xi, \Gamma, \varepsilon\rangle$,where:*
– ξ is a finite set of form-elements that will serve as inputs to the query of interest;
– Γ is a finite set of form-groups;
– ε, a subset of $(\xi \bigcup \Gamma) \times \Gamma$, is a group membership relation between elements/groups and the groups to which they belong.

In this tree, form-elements are the leaves and groups make up the intermediate nodes, including the root. 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. 4 where squares represent form-groups and circles correspond to form-elements. This tree has several constraint-specification elements grouped by their associated schema-entities such as Flight and Seat.

DEFINITION 3. (OUTPUT TREE) *The* output tree *of a form is a tree, $OT = \langle \xi, \Gamma, \varepsilon\rangle$,where:*
– ξ 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 is a finite set of form-groups;
– ε, a subset of $(\xi \bigcup \Gamma) \times \Gamma$, is a group membership relation between elements/groups and the groups to which they belong.

In this tree as well, leaf nodes correspond to form-elements, while internal nodes represent form-groups. Result specification operators can be *result-display* or *result-ordering* elements (described in Sec. 6.1). Fig. 5 shows an output tree. The trees in Fig. 4 & 5 are manifested in the form shown in Figs. 6 & 7. In [24], we show that this type of form can be generated efficiently given a query workload. In related work, the TQL language (used in QURSED [26,

Figure 6: Visual Representation of a Form (Criteria Pane)

27]) also logically separates each form into two sections – the *condition tree*) and the *result tree*. Some visual query builders [20, 22] also divide query specification this way, i.e., into a selection specification and a projection specification step.

Symbolic Representation: A form, given its logical representation, can be represented textually by a symbolic expression. Each form-element type has a representative symbol and form-groups are expressed using curly braces. A form-group is an unordered comma-separated list of its component elements and sub-groups placed within a pair of 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 denoted with three pairs of curly braces representing the root-level form-group in each of the form-trees. An empty form is represented as $F = \{\}_{\mathbf{I}}\{\}_{\mathbf{O}}\{\}_{\mathbf{R}}$, where $\{\}_{\mathbf{I}}$, $\{\}_{\mathbf{O}}$ and $\{\}_{\mathbf{R}}$ denote the input, output and relationship trees respectively. Brackets also mark the extents of a form-group, but these have a subscript which uniquely identifies them. In this scheme, the example form is expressed as follows (only the input tree is shown in full).

$$
\begin{aligned}
&\{\{c(\text{Flight:Origin}), c(\text{Flight:Destination}),\\
&\quad a(\text{Flight:Seat,Northwest:Flight}),\\
&\quad \{c(\text{Departure:Date}), c(\text{Departure:Time})\}_2\}_1,\\
&\ \{c(\text{Seat:Cabin}), c(\text{Seat:FareClass}),\\
&\quad c(\text{Seat:Available})\}_3\}_{\mathbf{I}},\\
&\{\{r(\text{Flight:Origin}), r(\text{Flight:Destination}), ...\}_1, ...\}_{\mathbf{O}},\\
&\{\{j(\text{Flight:FlightNumber, Seat:FlightNumber}), ...\}_1, ...\}_{\mathbf{R}}
\end{aligned}
$$

In the above expression, the functions $c()$, $r()$, $a()$, $s()$ and $j()$ denote constraint-specification, result-display, aggregate, result-ordering and join-specification form-elements respectively (see Sec. 2 and Sec. 6.1 for brief descriptions of these element types). We defer discussion of the relationship tree to Sec. 5.

3. FORM MANIPULATION

In this section, we present the set of available form manipulation operators that users can choose from to customize a form. The result of any operation is a new form. Hence 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. 4. We use the notation F^t to denote the t tree of the form F, where $t \in \{\mathbf{I}, \mathbf{O}, \mathbf{R}\}$.

Figure 7: Visual Representation of a Form (Results Pane)

Initially, the input and output trees of the running example shown in Figs. 4 and 5 are:

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

We have used initials to identify elements in the above expressions. For example, looking at Fig. 4, we can recognize that F.o is Flight:Origin, S.fc is Seat:FareClass, etc.

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 $\lambda_{(e,g)}(F^t)$.

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

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 $\phi_e(F^t)$.

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

Form-element Move (ψ): This operator moves an existing form-element from one group to another. The element and the target group must be specified and the operation is written as $\psi_{e,g}(F)$.

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

Form-group Insertion (Λ): This operator inserts a form-group into a given form. It requires the group and optionally its parent group which must be present in 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

Figure 8: Criteria Pane of the Modified Form

is not empty. We choose to allow deletion of non-empty groups in the language and let the implementation decide whether a group-delete command automatically triggers element-delete commands for each form-element in it or simply relocates them.

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

Form-group Move (Ψ): A form-group can be moved from one group to another 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. 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 existing forms to be combined. This enhances reusability and minimizes duplication of effort.

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 the operations we just performed is shown in Figs. 8 & 9.

4. FORM GENERATION

With any form as a starting point, a user can edit it using the form editor in multiple iterations until the desired form is obtained. The editor provides a canvas that holds the current form, and button-activated operations to modify it. We discussed the supported operations in Sec. 3. Here we describe how each operator is realized.

Form-Element Insertion: Inserting a form-element involves selecting a form-pane, and one or two fields (element-dependent).
Pane Selection: The first choice is the type of query operation it involves. As shown in Fig. 6, the criteria pane contains the simple constraint elements and aggregate constraint elements. Fields to be

Figure 9: Results Pane of the Modified Form

returned in the result as well as the ordering of the result are dictated by the contents of the results pane (Fig. 7). 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 associated with the form-element to be inserted. If the form has been partially built (typically the case in form customization), 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 in 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 in 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. In such cases, the user is prompted to re-visit the schema or a fragment of it, and select the desired field.

Form-Element Deletion: If the form contains elements that are irrelevant to a user's query or if a user inserted one or more elements in error, these may be removed from the form. An element can be deleted by clicking on its remove button (marked 'X').

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 in the input or output trees. Secondly, if two elements reference attributes of two different entities that are joined in the query, they will be placed in a single group 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 (this is the case if the threshold on minimum size of a group is two; in reality, it is tunable).

Form-Group Deletion: After the deletion of one or more form-elements, the associated form-groups might also need to be removed. If the number of elements drops below the threshold, the presence of the group is no longer necessary. Similarly, if a join condition is removed, its group must also be deleted. This reorganization 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 [26, 27] 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 [1] 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.

5. ADVANCED MANIPULATION

Casual form users are not database experts. By keeping the task of form-alteration as simple as possible, an interface can be made more flexible without decreasing its usability significantly. However, a form can be made even more expressive if more complex modifications are allowed. These are beyond the scope of casual users and are intended only for experts. In this section, we describe how experienced users can take advantage of form customization tools to make complex modifications. These are presented purely to illustrate the our system's expressive power. We describe both the operations themselves and how they can be used. But first, we define the third component of the form model, the *relationship* tree.

DEFINITION 4. (RELATIONSHIP TREE) *The* relationship tree *of a form is a tree,* $RT = \langle \xi, \xi_j, \Gamma, \varepsilon \rangle$*,where:*
– ξ *is a finite set of form-elements which are also present in either the* input *or* output trees;
– ξ_j *is a finite set of* join *form-elements that specify the relationships between the entities of the form;*
– Γ *is a finite set of form-groups;*
– ε*, a subset of* $(\xi \bigcup \xi_j \bigcup \Gamma) \times \Gamma$*, is a group membership relation between elements/groups and the groups to which they belong.*

The relationship tree captures the relationships between entities in a form. An example of such a tree is shown in Fig. 10 where the related entities are Flight and Seat. Relationship trees are similar to *input* and *output* trees in that they contain form-elements at their leaves and form-groups at intermediate nodes. Each form-group in this tree indicates a relationship between two entities. If the relationship is ternary, it is captured by two form-groups, one within the other. There will be two join elements in such a scenario each within one of the two form-groups. Typically every form-group contains a single join element. In cases where two entities have more than one relationship between them, additional join elements will be present within the form-group that relates the two entities.

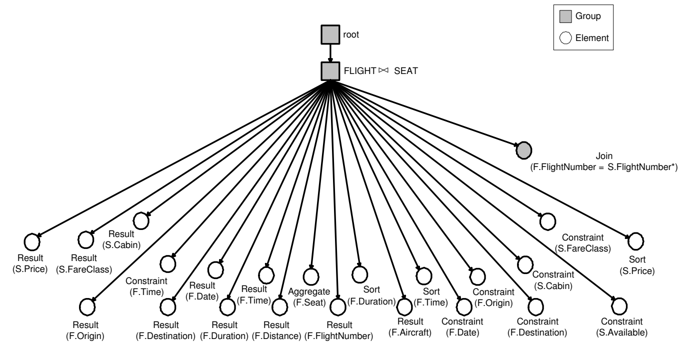

Figure 10: Logical representation of a form's Relationship Tree

A difference between the relationship tree and input or output trees is that it includes *all* the form-elements in a form, not just those of a specific type. The form-elements in a relationship tree are thus a union of the form-elements found in the other two trees and additionally contain the join elements. Secondly, the reason for grouping form-elements is different in the relationship tree than it is in the other two trees. Instead of semantic closeness between form-elements, relationship trees group elements by how they relate to one another in the query (based on value-joins).

The benefit of having a separate relationship tree for a form is two-fold. First, it makes complex query relationships easy to examine. Simply by looking at the form-groups in this tree and how they are placed in the tree hierarchy, one gets a sense of how the various queried entities relate to each other. This representation also makes relationships easier to modify if necessary. Second, this tree provides the starting point for the evaluation of the query specified by an end-user using a particular form. Once the form is filled, its relationship tree will contain all the form-elements actually used to specify the desired query along with the actual values provided by the user. In addition, with the relationships between the form's entities defined in this tree, the desired declarative query can be generated. The query generation process is described in Sec. 6.

The relationship tree is also the basis of the third pane of the form called the *Advanced* pane. It is so named to discourage naïve users from viewing or modifying the relationships between the form's entities. Although the relationship tree contains *all* the form's elements, only the join elements are displayed (since the other elements are already displayed in either of the first two panes). The form-group containing each join element is also displayed, and is manifested as a labeled box (with the join element within it).

Form Customization: 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 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 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. 11).

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. QUERY GENERATION

A traditional form is static and has a hard-coded query built into it. While the query generated can vary depending on what a user fills in, the variation is predefined and hence limited. In contrast the modifiable forms proposed in this paper permit the user to alter even the structure of a form making it impossible for the system to predetermine the query to be generated. Hence the query must be formulated dynamically, when the form is submitted. In this section, we describe the translation scheme that we use to generate queries from filled forms, based on a systematic conversion of the form representation formula into a query statement. Our implementation is based on an XML data environment, hence the declarative language we use is XQuery. We use XML Schema Definitions to define data structure and preserve element and attribute names as well as their structural relationships in the forms generated. The actual constructs in the query generated depend on the actual data manipulation operators in the form. We first present a set of primitive data manipulation operators in Sec. 6.1 and assume that we have available a translation scheme for each. In Sec. 6.2, we consider query generation for simple as well as complex queries and explain how their corresponding forms can be mapped to query language expressions and subsequently evaluated.

6.1 Form Elements

As defined in Sec. 2, a form-element represents a single data manipulation operation and it contains references to one or more entities in the schema. We now present a few common types of form-elements and the data manipulation operations they perform.

[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.

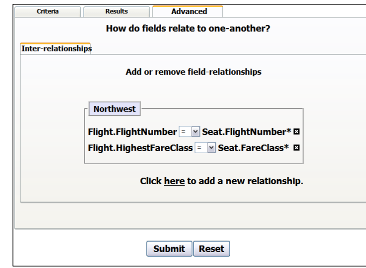

Figure 11: Advanced Pane of the Modified Form

This is the set that we implemented and is presented for concreteness. It is not an exhaustive list but it directly impacts the query expressivity of a form in our system. We briefly mentioned these element types in Sec. 2.1 and they are sufficient to express relational algebra with aggregates, but without negations, set operations or quantification. They are also enough to express XQuery core [9] excluding output structuring, set operations and quantified or negated predicates. We emphasize that the form manipulation mechanisms, which are the main contribution of this paper, do not depend on, and are not limited by, this particular set of operators. By choosing an appropriately rich set of form-elements, one can make generated forms arbitrarily expressive.

Data Manipulation Operators:

Constraint Specification: A constraint specification operator denotes a selection predicate and is represented as $c(E : n)$, n being the qualified name of the schema attribute involved relative to E, the associated entity. Multiple such operators can be combined conjunctively or disjunctively to construct complex query predicates.

Result Display: A result display operator corresponds to a query projection and is represented as $r(E : n)$, where E and n are the same as in a constraint specification operator.

Result Ordering: A result ordering operator is expressed as $s(E : n, o)$ which in addition to the name of the attribute, specifies the sort order, o which can be either *ascending* or *descending*. The result of a query can be sorted using one or more such operators.

Aggregate Computation: An aggregate computation operator is represented as $a(E_n : n, E_b : b)$, where n denotes the name of the aggregated schema element (or attribute) and b corresponds to the qualified name of the grouping basis or scope (of the aggregation). E_n and E_b are their associated schema entities respectively. As its name suggests, the aggregate computation operator corresponds to an aggregation operation at the query processing level.

Disjunction: A disjunction operator combines two or more constraint-specification operators into a single group from which at least one needs to be satisfied. It can be expressed as $d(c_1, c_2, ..., c_n)$ where each c_i denotes a constraint specification operator.

Join Specification: A join specification operator is expressed as $j(E_1 : n_1, E_2 : n_2)$. n_1 and n_2 correspond to attributes involved in the join condition, i.e., the relationship between E_1 and E_2.

Operator Translation: Each of the above data manipulation operators can be used to construct a corresponding form-element on an actual query form using a specific set of form controls.

Constraint Specification: The constraint specification operator is manifested using the following set of form controls: a static form-label denoting the schema attribute, a drop-down list of comparison operators ($<, \leq, =, \geq, >$ and $\neq$), and a text-box for the user to fill in the attribute value. The $\neq$ comparison operator allows users to specify negative conditions in the form.

Result Display: The result display operator is shown in a form as a set containing a static label and a check-box to indicate whether the result includes or excludes the attribute.

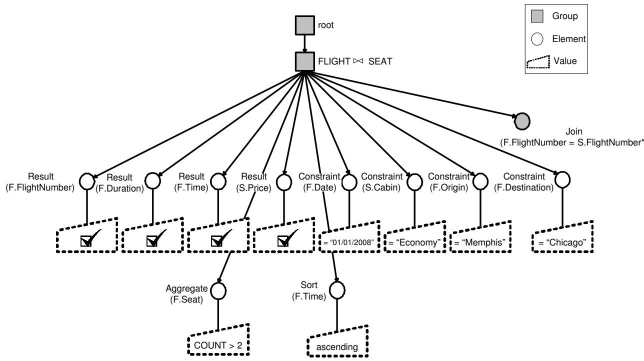

Figure 12: A Filled Form

Result Ordering: The result ordering operator is translated to a set containing a static label, a drop-down list for the sort-order (ascending or descending) and a check-box.
Aggregate Computation: In a form, aggregate computation is represented by a single label followed by a drop-down list of comparison operators, a text-box similar to the constraint specification element, and a check-box similar to the result display element. The label contains the qualified name of the entity (or attribute) whose value or occurrence is aggregated, with the grouping basis highlighted (or hidden if it matches the label of the group containing it).
Disjunction: A disjunction operator is displayed as a set of constraint specification elements, each on a separate line within the form. These elements are placed together in the form within a labeled rectangular box (similar in appearance to a form-group). If all of them denote conditions on attributes belonging to the same schema entity, the name of the entity is used as the box's label.
Join Specification: A join specification operator maps to a set of controls that includes the labels of the two elements (attributes) that constitute the join condition, and a drop-down list of comparison operators (as in constraint specification) for the join relationship.

6.2 Translation Procedure

The algorithm for operator translation is shown in Fig. 13. We discuss the generation of declarative expressions for two query types.
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 predicates in the where-clause and finally, all result-display elements are mapped to projected elements or attributes in the return-clause. If there is an aggregate-computation element, it provides both the attribute to be grouped and 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.
Complex Queries: Unlike simple queries, forms corresponding to complex queries such as join queries and queries with nested sub-queries have more than one group in their relationship trees. In each group, the join condition is examined and all descendent elements are partitioned into two groups, one corresponding to the left-hand side of the join condition and the other, 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 aggregations 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 final query. As an example of a nested query, consider the filled form in Fig. 12 which generates the following XQuery:

Figure 13: Algorithm `GenerateQuery`

Input: A filled form F
Output: An XQuery expression X

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;
  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;

foreach *form-group* $g \in T$ **do**
  Partition the form-elements in g between the two binding variables by schematic similarity;

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;
  Create a predicate for each constraint-specification or join-specification element and add it to the **where clause**;
  Create a projection for each result element and add it to the **return clause**;
  Create an orderby attribute for each sort 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;

```
for $f in doc("northwest.xml")//Flight
let $s := for $st in doc("northwest.xml")//Seat
          where $st/FlightNumber = $f/FlightNumber
            and $st/Cabin = "Economy"
          return $st
where $f/Origin = "Memphis" and $f/Destination = "Chicago"
  and $f/Departure/Date = "01/01/2008"
  and COUNT($s) > 2
return {$f/FlightNumber} {$f/Departure/Time} {$f/Duration}
       {$s/Price}
order by $f/Departure/Time
```

In related work, TQL (basis of the QURSED Editor [26, 27]) is a query language designed specifically for form design. It automatically generates query fragments corresponding to form-elements chosen by the form creator based on their type. While TQL covers a majority of query types, it is not as expressive as traditional declarative query languages like SQL or XQuery (although TQL queries can be translated to XQuery for evaluation). In comparison, the system we propose can be made as expressive as needed. Form generation tools like Ariba, Caspio Bridge, Microsoft Visual Web

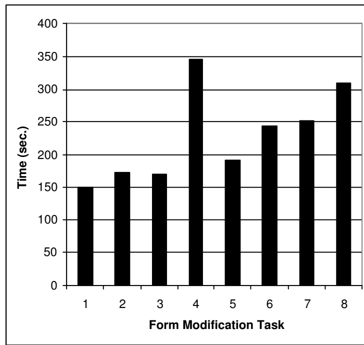

Figure 14: Time spent on each query task in lay user study

Developer, WUFOO [7] and Microsoft InfoPath also automatically generate query templates corresponding to forms created, but these only create data-entry forms for which query generation is trivial.

7. EVALUATION

We implemented the form construction and customization ideas described above as a front end to TIMBER [23], an XML database system. Since the ultimate measure of an interface is its usability, we conducted usability studies to evaluate the effectiveness and usefulness of form modification. The purpose of a customizable form is to enable a user to change a form from one that does not support a desired query to one that does. A traditional static form is of little use if the desired query is unsupported. This is why a one-on-one comparison between static and customizable forms for unsupported queries is of little use. But what is the cost of form customization to a user? Is it realistic to expect casual users to customize forms? These are the questions we aimed to answer with our system evaluation. First, we created a set of forms and a set of queries not fully supported by them. We measured how long and how correctly casual users were able to perform the required modifications. Secondly, we evaluated the system's usability for expert users as well. Since the experts were, by definition, familiar with declarative query languages, this allowed us to perform a direct comparison between form modification and query re-writing to satisfy a set of information needs that the original forms and queries could not. We present our results and our inferences 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.

7.1 Casual User Study

To evaluate the general case, we ran an experiment with 10 technically unsophisticated users, knowledgeable neither in formal querying techniques nor in the data domain. After a short tutorial to gain familiarity with the tools at their disposal, we presented 8 querying tasks on two different schemas, and measured their response times for the customizations 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 included both simple and complex queries, where complexity was defined in multiple ways (Table 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 1: Task complexities in lay user study

Independent Variables: There are three complexity factors affecting the performance measures of the subjects:
Schema Complexity: This denotes the schema of 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 schema complexity as 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 is 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 requires less than 10 elements (lowest was 3, highest was 8), while a complex query needs 10 or more elements to be specified for it to be generated (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 either multiple fields in the schema to be specified or 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, to do one sample task whose solution was provided. 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, if the subject needed to. For each task, subjects were asked to submit their solutions (by clicking a button) and proceed to the next.
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. We find these to be reasonable compared given that the alternative is to use (non-customizable) forms that force the user to pose an incomplete query and then spend time manually analyzing the results which is both tedious and error prone. 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. 15). Response times for tasks in-

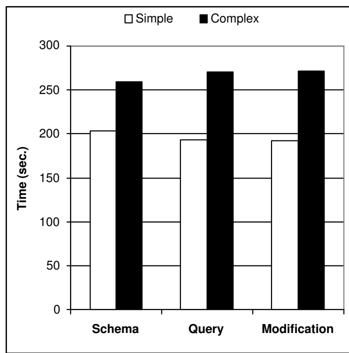

Figure 15: Factors affecting response time in lay user study

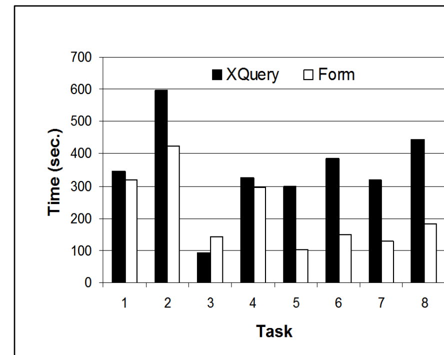

Figure 16: Time spent on each query task expert-user study

volving 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 modification itself was complex than for forms needing minor changes. These relatively moderate increases in response time verify the system's robustness to increased complexity of schema, query or modification.
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).

7.2 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 databases. In this study we compared the ease of modification of declarative query statements versus forms. We chose to use datasets that were different from those used in the lay user study, but were from a domain that these expert users were intimately familiar with. In any case, the schemas were provided for reference. The study had 10 tasks each of which started with a query and a form, both of which needed to be modified to pose the same new query. In other words, the user neither had to design a form nor write an XQuery from scratch — only modify an existing form and query. The first two tasks were purely instructional and had the correct answers revealed (these were to a separate database so no schema familiarity was gained).
Independent Variables: There is only one factor affecting how well a subject performed each task:
Interface: Each user had to attempt each querying task using two different querying mechanisms (XQuery and a form) one after the other, in random order, one task at a time.
Dependent Variables: There are two measures of interest:
Efficiency: This refers to 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 proceed to the next task. 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 [8, 29] dataset and a biological database, MiMI [3], and were equally distributed (4 each). The required modifications ranged from simple element insertions, to introducing new entities and relationships via joins.
Results & Discussion:*Efficiency:* We recorded the time taken for each task by each subject and charted the average time taken per task for each querying mechanism (see Fig. 16). 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 form modification time from one task to the next. We attribute this to increased familiarity with the interface. In contrast, XQuery specification shows no such trend.
Impact: We compared the performance of subjects familiar with one querying mechanism (XQuery) and unfamiliar with the other (customizable forms). Still, it is the latter that shows better usability and fewer errors. The results of the two user studies suggest that customizable forms are effective regardless of querying expertise.

7.3 Query Builder Comparison

Visual query builders are tools that allow a user to visually construct a declarative query iteratively. One such tool is the DB2 Visual XQuery Builder packaged with the IBM DB2 Developer Workbench [2]. Since forms and query builders are both approaches to visual query specification, we conducted an experiment to compare our system with the DB2 VXB. 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 a null 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 that time. 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 query building is not as easy as form customization.

8. RELATED WORK

Regarded as the simplest mode of machine-readable query input, forms see very wide use in databases today. Early work on form-based querying mechanisms include [18, 21] which provided users visual tools to frame their queries as well as to perform other operations such as database design [19] and view definition. The GRIDS

system [28] generated forms that allowed users to pose queries in a semi-IR, semi-declarative fashion. In the relational world, visual querying began with QBE [33] as early as 1975. A survey of early visual languages, both iconic and form-based can be found in [14]. Query-By-Diagram (QBD*) [16] was a visual querying language based on the ER model that allowed users to manipulate diagrams instead of having to write SQL statements. The SEWASIE project includes an intelligent query interface [15] that improves the usability of a visual query language by using ontology to overcome vocabulary ignorance of users. Standard form design tools such as Microsoft Visual Basic, Visual Studio and Java Swing are used by interface developers to create forms from scratch. More recently, user interface languages such as XAML, JavaServer Faces, XForms and XUL were developed specifically for UI design and these drastically reduce the amount of code needed to create forms. If only data-entry forms are required (these only allow insertion of data, not querying), there are more user-friendly tools like Ariba, WUFOO [7] and Microsoft InfoPath that automate lower-level tasks and greatly simplify the form design process. Caspio Bridge is a simple search from creation tool that helps users design data-entry as well as very simple search forms. In the area of business intelligence, sophisticated solutions such as SAP eRFx, SAP BW, SAS and Microstrategy are available that that help database experts generate forms and reports especially for OLAP-style queries.

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 [10] is a system that helps users build queries incrementally by navigating through layers of forms, and helps them view results in the same way. EquiX [20] is another such language that helps users build queries in steps using a GUI. Both these approaches require users to start from scratch making the cost of querying much higher than simply customizing existing forms. In the area of data integration, systems such as MetaQuerier [17, 32] build integrated query interfaces from multiple similar source interfaces to provide a single query point for users. There are also form editing tools such as the QURSED editor [26, 27] that simplify the task of form construction (and report building) for interface developers by automatically generating form-elements for a schema entities based on their types. Other visual languages such as XQBE [13], Xing [22], MIX [25] and QBT (Query By Templates [30]) adopt different approaches to visual query specification, though these are not form-based. Among commercial tools, IBM DB2 provides a Visual XQuery Builder with its Developer Workbench [2]. This tool allows developers to specify queries visually which are then translated into XQuery.

9. CONCLUSION

Query interfaces play a vital role in determining the usefulness of a database. A form-based interface is widely regarded as the most user-friendly querying method. Form-based interfaces are an important part of databases in the real world [12]. In this paper, we presented mechanisms to overcome the challenges that limit the usefulness of forms: their restrictive nature and the tedious manual effort required to construct them well. We introduced a form manipulation language that we implemented in a visual editor which allows users to customize an existing form to support a previously unsupported query. Large-scale modifications can be made iteratively one simple step at a time. Finally, we conducted experiments on real users that validated the system's usefulness. Using the 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. We presented our approach to form building in the context of XML databases. However, the bulk of the ideas described in this paper are equally applicable to SQL-based databases. While the technologies suggested in this paper 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.

10. REFERENCES

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

[2] IBM DB2 Developer Workbench: http://www-306.ibm.com/software/data/db2/ad/dwb.html.

[3] MiMI: Michigan Molecular Interactions Database – http://mimi.ncibi.org.

[4] Rapid SQL: http://www.embarcadero.com/products/rapidsql/.

[5] SQL Manager Advanced Query Builder: http://www.sqlmanager.net/products/tools/querybuilder.

[6] Stylus Studio XQuery Editor: http://www.stylusstudio.com/xquery_editor.html.

[7] WUFOO: http://www.wufoo.com/.

[8] XMark: http://www.xml-benchmark.org/.

[9] XQuery 1.0: http://www.w3.org/tr/xquery/.

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

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

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

[13] D. Braga, A. Campi, and S. Ceri. *XQBE*: A visual interface to the standard XML query language. *TODS*, 30(2), 2005.

[14] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 8(2), 1997.

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

[16] T. Catarci and G. Santucci. Query by Diagram: A Graphical Environment for Querying Databases. In *SIGMOD*, 1994.

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

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

[19] J. Choobineh et al. A Form-Based Approach for Database Analysis and Design. *CACM*, 35(2), 1992.

[20] S. Cohen et al. EquiX–A search and query language for XML. *JASIST*, 53(6), 2002.

[21] D. W. Embley. The Natural Forms Query Language. *TODS*, 1989.

[22] M. Erwig. A Visual Language for XML. In *VL*, 2000.

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

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

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

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

[27] M. Petropoulos, Y. Papakonstantinou, and V. Vassalos. Graphical query interfaces for semistructured data: the QURSED system. *TOIT*, 5(2), 2005.

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

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

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

[31] J. L. Viescas. *Microsoft Office Access 2003 Inside Out*. 2004.

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

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