

Automated Creation of a Forms-based Database Query Interface*

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

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

ABSTRACT

Forms-based query interfaces are widely used to access databases today. The design of a forms-based interface is often a key step in the deployment of a database. Each form in such an interface is capable of expressing only a very limited range of queries. Ideally, the set of forms as a whole must be able to express all possible queries that any user may have. Creating an interface that approaches this ideal is surprisingly hard. In this paper, we seek to maximize the ability of a forms-based interface to support queries a user may ask, while bounding both the number of forms and the complexity of any one form. Given a database schema and content we present an automated technique to generate a good set of forms that meet the above desiderata. 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. Our experimental analysis shows that our techniques can create a reasonable set of forms, one that can express 60–90% of user queries, without any input from the database administrator. Human experts, without support from software such as ours, are often unable to support as high a fraction of user queries.

1. INTRODUCTION

A form is a simple and intuitive query interface frequently used to provide easy database access. It requires no knowledge, on the part of the user, of how the data is organized in storage and no expertise in query languages. For these reasons, forms are a popular choice for most of today's databases. However, while easy to use, forms provide the user a constrained view of the underlying data. If a user requires some information that is present in the database but inaccessible via the available forms, he or she is helpless without a querying alternative. While in some cases certain query types are intentionally disallowed for security, performance or other reasons, it is often the case that a query isn't supported simply because the demand for it wasn't anticipated by the interface developer. On the other hand, it is not practical to support all possible queries, particularly if the schema of the database is complex: the interface would

need far too many forms and each form would be too complex, overwhelming users and negating the benefits of a forms-based interface. Hence a trade-off needs to be made between expressivity and complexity while designing forms. This trade-off is critical to interface usability and is non-trivial due to the potentially wide range of querying needs of intended users.

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. Our goal in this paper is to automate the task of form generation in an attempt to significantly reduce, if not eliminate, the developer's role in the process. We do so by developing an automated procedure, based on a set of heuristics, to analyze the database – its schema as well as its content – to identify zones of potential interest. We then generate a set of forms that highlight those parts of the data and support as many and as diverse queries as possible to those areas. While a real user query log can help produce an even better set of forms, such queries are not typically easy to acquire before the database is even publicly accessible. Hence, being able to generate a reasonably good starting set of forms without the benefit of user queries is important. 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. Our problem statement thus becomes: *Design a form interface that maximizes expressivity while respecting specified upper-bounds on interface complexity.* We define expressivity of an interface as the range of queries that can be expressed using it. Interface complexity, as defined above, is limited through thresholds on size and individual form complexity.

1.1 Motivation

The effectiveness of a manually designed forms-based interface largely depends on the developer's understanding and estimation of its users' needs. This is evident from observable differences between two or more interfaces designed to serve the same purpose but by different UI designers. For example, consider the task of buying a used car. There are several database-backed websites that help users buy used vehicles and several of them provide forms-based interfaces to help a user find exactly the type of car he or she is looking for. While the task is common to all these websites, the interfaces they provide are quite different. Specifically, the set of queries that they allow users to ask about the desired car are not the same. This can make some more desirable for a specific information need even if the data is the same in all of them. We analyzed the interfaces provided by five such websites: Car.com, Cars.com, AutoTrader.com, CarsDirect and eBay Motors. Only sections concerning used cars were considered. Table 1 provides a summary.

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

Permission to make digital or hard copies of portions 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. Copyright for components of this work owned by others than VLDB Endowment must be honored.

Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

| | Website | # Forms | # Fields in each form |
|-------|----------------|---------|-----------------------|
| W_1 | AutoTrader.com | 1 | 30 |
| W_2 | Car.com | 3 | 5, 7, 8 |
| W_3 | Cars.com | 5 | 2, 5, 11, 11, 12 |
| W_4 | CarsDirect | 4 | 3, 3, 5, 13 |
| W_5 | eBay Motors | 2 | 3, 16 |

Table 1: Comparison of Websites selling Used Cars

While all of these websites serve the same purpose (helping a user find and buy a used car) and have the same underlying data (used car listings) with more or less the same set of attributes for each listing, the ways in which their query forms are structured and presented to users are quite different. These differences are not only in the number of forms and how they are positioned relative to each other, but also in the content and layout of each form. The content of a form directly impacts the range of queries that a user can issue using it. Consider the following queries:

- Q1.** Find a certified pre-owned Acura that comes with side airbags.
- Q2.** Show all 2005 Chrysler Sebrings listed in the past two weeks.
- Q3.** List all Mazda Proteges with fewer than 50,000 miles on them that are selling for under \$20,000.
- Q4.** Show minivans available for a monthly payment below \$200.
- Q5.** Find a car that has a 600hp V12 engine.

We found that even though results matching all five queries were available in all five websites, not all of these queries could be issued using the forms provided. Query Q_1 could only be asked in two of the five websites (W_1 and W_5). Q_2 was supported only by the forms provided in W_3 while Q_4 was only issuable using W_2 . All sites except W_2 allowed Q_3 to be asked. Query Q_5 could not be expressed using any of the five query interfaces. It bears repeating that, for the unsatisfactory websites, the data attributes used in these queries were *available* in the databases but not *queriable* using the interfaces. This heterogeneity shows that finding a universally satisfying solution to any form design task is hard, and that even if done manually by human experts, they can have limitations that lead to unsatisfied users¹.

Just in case used car sales were an anomaly, we analyzed the forms-based query interfaces of 12 additional web databases. This set included 3 scientific databases (including NCBI BLAST Search and the Human Protein Reference Database), 3 commercial (including Northwest Airlines and Realtor.com), 3 statistical (such as CricInfo’s Stats Guru) and 3 informational databases (the Internet Movie Database, Yahoo! Movies and AllRecipes.com). A summary of our analysis is shown in Table 2. We used the forms and the query results to infer the schema used by each database. Since the schemas were inferred, and not known explicitly, several rows have been marked “(estimate)”. On average (median), each database schema had about 5 entities and each entity had about 5 at-

¹It is conceivable that some of these websites intentionally decided not to support some queries for business or performance reasons. However, we were not able to imagine a plausible explanation to cover the cases we observed. Moreover, for many of these queries, it is possible to answer them by issuing a more general query first and then visually inspecting its results and sifting through them manually to find the relevant results while ignoring irrelevant results. For example, for Q_2 on W_1 , a user can ask for all 2005 Chrysler Sebrings and then manually look through these to find the ones listed recently, in the past two weeks. Surely, there is no performance advantage in forcing the user to perform this query in such an awkward way.

| | Min. | Max. | Avg. | Med. |
|------------------------------------|------|------|-------|------|
| # Entities per Database (estimate) | 1 | 36 | 7.08 | 5 |
| # Attributes per Entity (estimate) | 1 | 101 | 12.21 | 5 |

| | | | | |
|--|---|-----|-------|----|
| # Forms per Database | 2 | 102 | 20.42 | 10 |
| # Fields per Form | 1 | 64 | 6.23 | 5 |
| # Entities per Form (estimate) | 1 | 6 | 1.01 | 2 |
| # Entity-Relationships per Form (estimate) | 0 | 5 | 0.32 | 0 |

| | | | | |
|--|---|----|------|---|
| # Queriable Attributes per Entity (est.) | 1 | 64 | 6.94 | 3 |
| # Non-queriable Attributes per Entity (est.) | 0 | 37 | 5.26 | 2 |

Table 2: Statistics of 12 Form Interfaces on the Web

tributes. About 10 forms were created for each database for users to issue their queries. Forms are typically not complex, each with only 5 form fields on average (these may correspond to multiple entities). The limitations of forms that we showed above through anecdotal evidence can be seen in the observation that of the 5 attributes per entity, only 3 are queriable via a form. The remaining 2 attributes (which can be viewed on the result page) cannot have conditions specified on them at form-submission time. An example of such an attribute in the used car example is *listing date*, which is shown to users in the result but cannot be used while querying in 4 of the 5 car databases mentioned in Table 1.

To recap, many databases use forms-based interfaces but designing a good forms-based interface is hard. We see a wide range of limitations on the expressive power of forms even in popular deployed databases. Our goal in this paper is to enable the design of an expressive forms-based interface while keeping the interface simple. Keeping it simple means limiting the number of forms and also the number of elements on a form – we set thresholds on both.

1.2 Our Approach

The ideal metric for the expressive power of an interface is the fraction of user queries satisfied in a representative user query log. But obtaining such a log is non-trivial. In fact, it is impossible for a new database deployed for the first time. Thus our challenge is to design a good set of forms without having an actual query log at hand. Based solely on the schema and data content of a database, we must advise a designer what forms to create. Given the large variety of queries that could potentially be posed against any database, this barrier appears to be insurmountable – how can we select a small number of queries to support using a few simple forms from this large set of possibilities?

To gain an intuition for how schema and content of the database can provide a reasonable starting point to estimate user query needs, consider a movie database. It is likely to have movies as the most important and highest populated entity. 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. A slightly more complex schema that many database researchers are familiar with is the XMark benchmark [5, 24]. In its schema, the primary entities are *open_auction*, *closed_auction*, *item* and *person*. Intuitively these are the entities of most interest to users whose queries may involve one or more of these entities. Most current databases, however, have much more complex schemas with numerous entities, attributes and relationships. Many of these databases 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 with more than one entity of interest. In this paper, we describe our approach to determine exactly that. We can break the forms interface design problem down into three challenges discussed below.

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. 2, 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. 3.

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.

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. 5. 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. $> 80\%$).

2. 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.

2.1 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 dis-

tinguish 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 1. (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 [28] 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.

2.2 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) [28]. 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 .

2.3 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).

2.3.1 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 entities². Hence we use a recursive formula to compute queriability making use of postulates P1 and P2 (like how schema element importance was computed in [28]).

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 convergence³ is reached.*

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

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

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.

2.3.2 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

²This 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.

³Proof of convergence can be seen in an analogous problem in [19].

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 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 individual queriabilities of 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.*

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

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

Here $R(e_i \rightarrow e_j)$ denotes the **participation ratio** of e_i in the relationship between e_i and e_j . $N(e_i \rightarrow e_j)$ 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 data⁴. 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. 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 seller), the queriability of the related entity pair is simply the product of queriabilities of the two entities.

$$Q(\text{closed_auction} \wedge \text{person}) = Q(\text{closed_auction})Q(\text{person})$$

$$= 0.0128 * 0.0372 = 0.0005$$

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 $N_\pi(e_1, \dots, e_m) = m!$ 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

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

⁴Note 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.

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:

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

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

2.3.3 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. In this paper, we only consider attributes of entities. Attributes of relationship can be incorporated as a simple extension but are beyond the scope of this paper.

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.*

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

Here $C(e)$ is the **absolute cardinality** of the entity e , i.e., the number of times it occurs in the database, and $C(e \rightarrow a)$ 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's *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.

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

3. FORM COMPOSITION

Having computed the queribilities 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 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.

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 queribilities 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. 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. 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 computa-

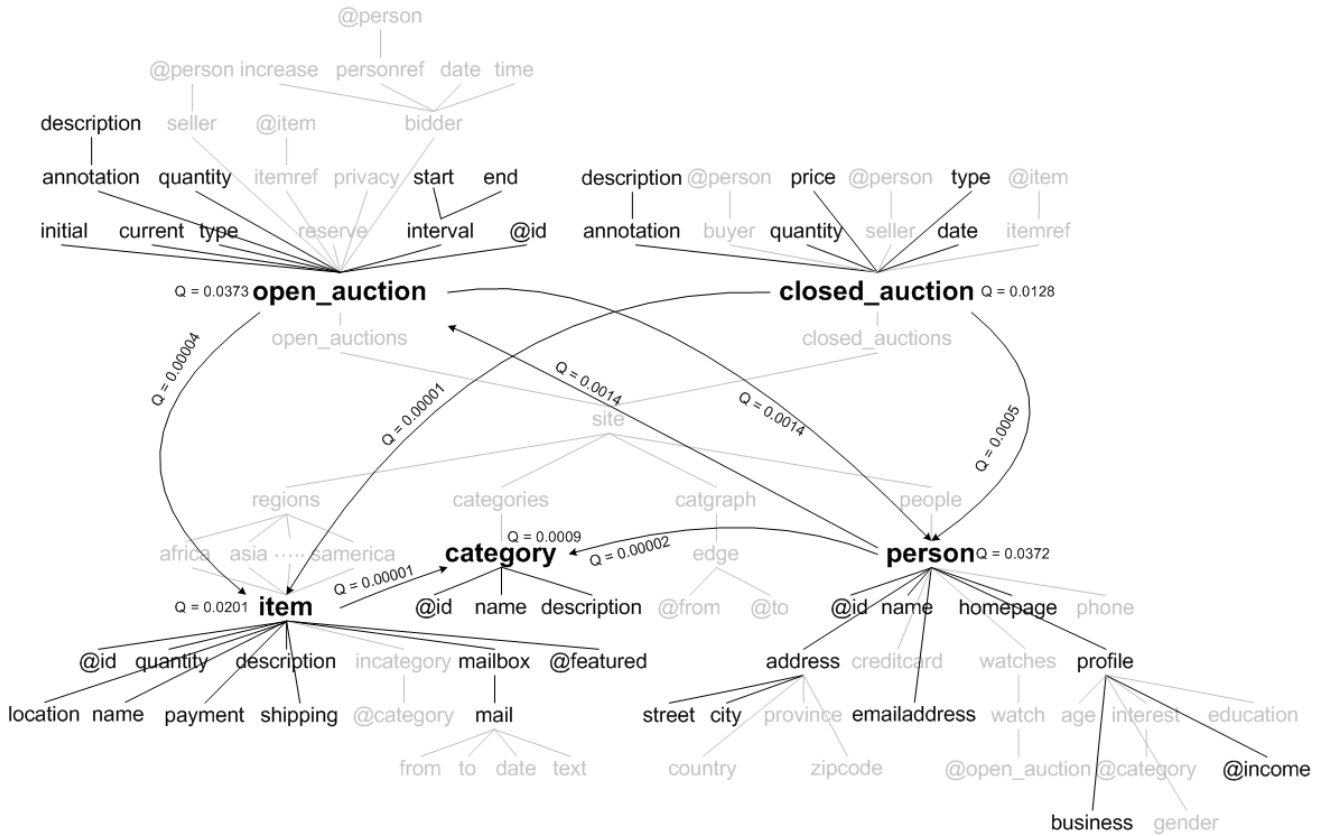


Figure 1: XML Schema Definition of the XMark Dataset showing the most querable entities and attributes highlighted (entities and attributes are inferred from the XSD during schema analysis). The most querable related entity pairs are shown with dark arrows.

tion costs, the system only allows one join or a single nested subquery. 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.

4.1 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.*

$$Q_\sigma(a) = w_\sigma(a) * N(a) \quad w_\sigma(a) = \frac{r_a}{C(a)}$$

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.*

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

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 .

Sort: “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.*

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

Algorithm 1: Algorithm GenerateForms

Input: A Database D with a schema S

Input: Complexity thresholds: k_e (for entities), k_a (for attributes), k_{σ} , k_{π} , k_{ψ} , k_{γ} (for operator-specific attributes) and k_r (for related entity collections)

Output: A set of forms F

$G = \text{AnalyzeSchema}(D, S);$

$F = \text{AssignSchemaComptsToForms}(G, k_e, k_a, k_r);$

$F = \text{CreateFormComponents}(F, k_{\sigma}, k_{\pi}, k_{\psi}, k_{\gamma});$

4.2 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 1 through 4. A sample form (created for the Geoquery schema [1]) can be seen at <http://www.eecs.umich.edu/~mjayapan/vldb2008/GeoqueryForm.html>.

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 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.

5. EVALUATION

We implemented our system on top of the TIMBER [17] 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

Algorithm 2: 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);

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.

5.1 Experimental Methodology

In this paper, 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

Algorithm 3: Algorithm AssignSchemaComptsToForms

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 ;

the threshold variation experiments serve to help data providers develop an intuition for good threshold values for their own databases.

5.2 Data Sets and Query Workloads

The system was evaluated using three datasets: a real-world database, MiMI [3], and two natural language querying benchmarks, Geoquery [1, 26] and Jobsquery [2, 9, 26]. 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. A description of each dataset and its accompanying query set can be found in Appendix A. In this section, we show how our system performs in each environment.

5.3 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 (see Appendix A.2 for details). Unfortunately, our system only considers SPJ queries with sorting and aggregation containing one join or nested sub-query⁵. 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 present in this paper. 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

⁵Support 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.

Algorithm 4: 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;

```

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 queryable 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 ⁶

In Sec. 1.1 we used a real example to show the limitations of manually designed forms. In this section we attempt to measure their performance and compare them with automatically generated forms 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

⁶More information about the results of this experiment can be found in Appendix B.

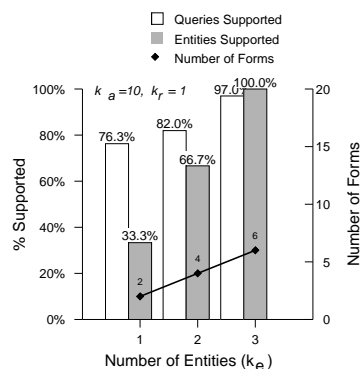


Figure 2: Effect of Entity Threshold in MiMI (k_e)

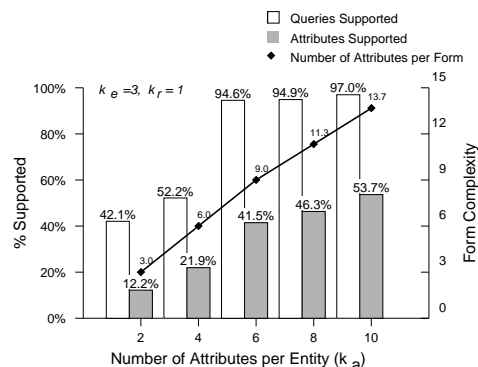


Figure 3: Effect of Attribute Threshold in MiMI (k_a)

supports 80% of the query set. Even if we restrict it to 12 fields (not a good choice as evident from Fig. 9), 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.

5.4 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).

5.4.1 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 queryable for each selected entity, k_a was kept constant at 10. Our observations are shown in Fig. 2. We can see a clear bias in the queries towards the most queryable 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 queryable 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. 3. 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 generated automatically. We also observe that for k_a

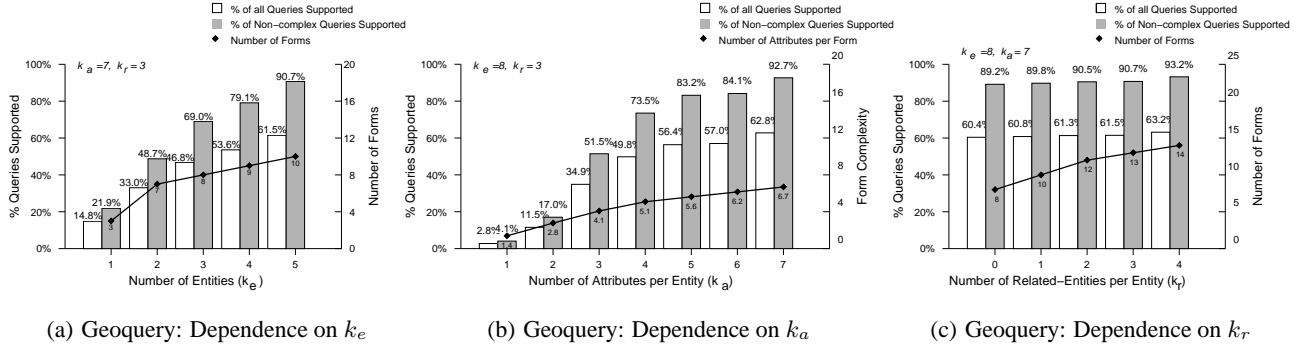


Figure 4: Effect of Thresholds on Query Support in Geoquery

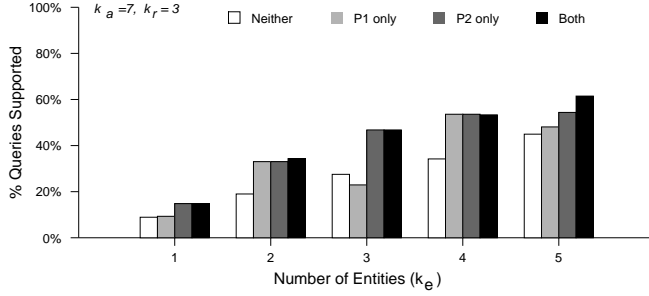


Figure 5: Effect of Postulates P1 and P2

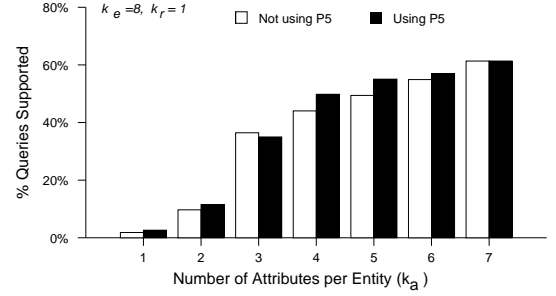


Figure 7: Effect of Postulate P5

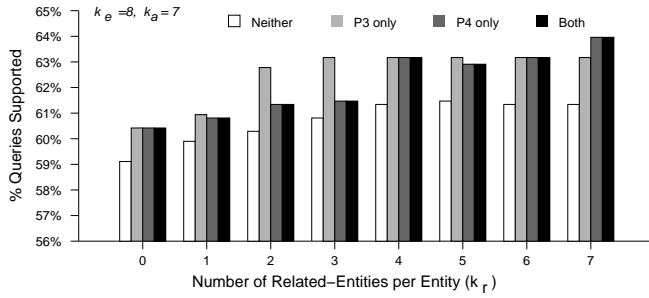


Figure 6: Effect of Postulates P3 and P4

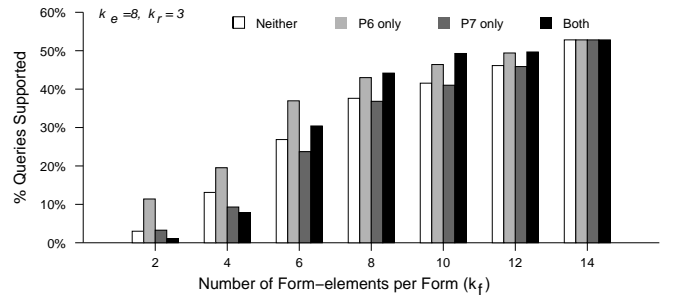


Figure 8: Effect of Postulates P6 and P7

> 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.

5.4.2 Geoquery

We altered the system thresholds k_e , k_a and k_r one at a 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. 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(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(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. 5) 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. 6. Finally, we evaluated the effectiveness of postulates P5 through P7 which are captured in Figures 7 and 8. 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.

5.4.3 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.

5.5 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 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. As we showed in Table 2, many current database-backed websites employ over 10 different forms, some as many as 102 [4]. 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

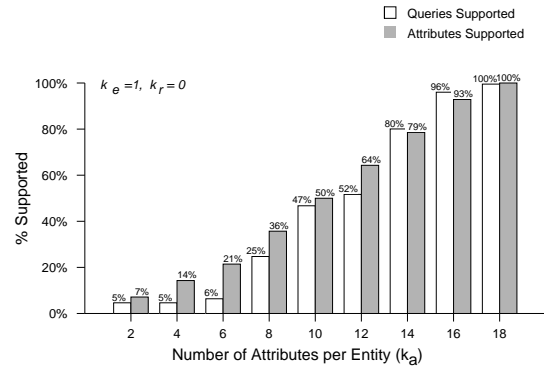


Figure 9: Effect of k_a on Query/Attribute Support (Jobsquery)

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.

6. RELATED WORK

While we have seen little work that automatically determines the

correct set of forms for a database, there has been a great deal of work in simplifying the form specification task for interface developers. Even with standard form building tools like Cold Fusion [12], Visual Basic [14], CGI [13], ASP, JSP, business tools like InfoPath, XAML, Ariba, SAP, SAS, Access, Visual Web Developer and user-friendly solutions like WUFOO, WyaWorks, ZOHO Creator and Coghead having come a long way, the task of form creation is still largely manual. An exception is [16] in which forms are generated dynamically based on metadata embedded in relational tables. This system however, only creates survey forms whose purpose is data-entry rather than data-retrieval. They essentially support only a single insertion query and are orthogonal to our work, which supports a wide range of retrieval queries. Other systems include powerful tools like the QURSED Form and Report Editor [22, 23] which enables developers to create form-based interfaces in fewer simpler steps. This editor uses the schema of the database as a starting point (like our system does) but it is up to the developer to select the desired schema elements and attributes to use in forms. It allows 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. Earlier work in UI design includes DRIVE [20], a runtime user-interface development environment for object-oriented databases. This system used the NOODL data model enabling context-sensitive interface editing. [27] proposed 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. FoXQ [6] and EquiX [11] are examples of work done to hide from users the intricacies of a declarative language (XQuery). Both are systems where users build queries incrementally by navigating through layers of forms. Visual querying began as early as in 1975 with QBE [29]. Since then, there have been several graphical query languages designed to make declarative querying more intuitive. Some examples of these are XQBE [8], VISIONARY [7], Kaleidoquery [21], QBT (Query By Templates) [25], Marmotta [10] and GRIP [15]. IBM distributes a query builder called Visual XQuery Builder with its DB2 Developer Workbench. These help users create declarative language queries instead of using forms with pre-embedded declarative queries.

7. CONCLUSION

A forms-based interface is the gateway to many a database and it ultimately determines the availability and usefulness of the data. Designing a good set of forms is hard. Insufficiently expressive forms can frustrate a user. In this paper, we have developed a mechanism to generate a forms-based interface with nothing more than the database itself. In the absence of real user queries to guide interface design prior to database deployment, this is a challenging problem, but one of practical importance. 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 presented our approach to form building using an XML implementation. However, in principle, our concepts are equally applicable to relational databases. 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. In comparison, our analysis of deployed databases shows that this level of coverage is difficult to achieve today, even with expert design.

8. REFERENCES

- [1] Geoquery Database:
<http://www.cs.utexas.edu/users/ml/geo.html>.
- [2] Jobsquery Database:
<http://www.cs.utexas.edu/users/ml/nldata/jobquery.html>.
- [3] MiMI – Michigan Molecular Interaction Database:
<http://mimi.ctaalliance.org>.
- [4] NCBI BLAST:
<http://www.ncbi.nlm.nih.gov/blast/Blast.cgi>.
- [5] XMark: An XML Benchmark Project:
<http://www.xml-benchmark.org/>.
- [6] Robin Abraham. FoxQ–XQuery by Forms. In *HCC*, 2003.
- [7] Francesca Benzi, Dario Maio, and Stefano Rizzi. VISIONARY: a Viewpoint-based Visual Language for Querying Relational Databases. *Journal of Visual Languages and Computing*, 10(2), 1999.
- [8] Daniele Braga, Alessandro Campi, and Stefano Ceri. XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. *ACM TODS*, 30(2), 2005.
- [9] Mary Elaine Califf and Raymond J. Mooney. Relational Learning of Pattern-Match Rules for Information Extraction. In *AAAI*, 1999.
- [10] Fabrizio Capobianco, Mauro Mosconi, and Lorenzo Pagnin. Looking for convenient alternatives to forms for querying remote databases on the Web: a new iconic interface for progressive queries. In *AVI*, 1996.
- [11] Sara Cohen et al. EquiX–A Search and Query Language for XML. *JASIST*, 53(6), 2002.
- [12] Ben Forta, Raymond Camden, Leon Chalnack, and Angela C. Buraglia. *Macromedia ColdFusion MX 7 Web Application Construction Kit*. Macromedia Press, 2005.
- [13] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly, 1996.
- [14] Michael Halvorson. *Microsoft Visual Basic 2008 Step by Step*. Microsoft Press, 2008.
- [15] Harald Schöning. A Graphical Interface to a Complex-Object Database Management System. In *IDS*, 1992.
- [16] D. J. Helm and B. W. Thompson. An Approach for Totally Dynamic Forms Processing in Web-Based Applications. In *ICEIS (2)*, 2001.
- [17] H. V. Jagadish et al. TIMBER: A Native-XML Database. *VLDB Journal*, 11(4), 2002.
- [18] Magesh Jayapandian, Adriane Chapman, et al. Michigan Molecular Interactions (MiMI): Putting the Jigsaw Puzzle Together. *Nucleic Acids Research (Database Issue)*, 35, 2007.
- [19] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*. Chapter 7. PhD thesis, University of Leipzig, 2004.
- [20] Kenneth J. Mitchell and Jessie B. Kennedy. DRIVE: An Environment for the Organized Construction of User-Interfaces to Databases. In *Interfaces to Databases (IDS-3)*, 1996.
- [21] Norman Murray, Norman Paton, and Carole Goble. Kaleidoquery: A Visual Query Language for Object Databases. In *AVI*, 1998.
- [22] Yannis Papakonstantinou, Michalis Petropoulos, and Vasilis Vassalos. QURSED: Querying and Reporting Semistructured Data. In *SIGMOD*, 2002.
- [23] Michalis Petropoulos, Yannis Papakonstantinou, and Vasilis Vassalos. Graphical Query Interfaces for Semistructured Data: The QURSED System. *ACM TOIT*, 5(2), 2005.
- [24] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI.
- [25] Arijit Sengupta and Andrew Dillon. Query by Templates: A Generalized Approach for Visual Query Formulation for Text Dominated Databases. In *ADL*, 1997.
- [26] Lappoon R. Tang and Raymond J. Mooney. Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing. In *ECML*, 2001.
- [27] Anders Tornqvist, Chris Nelson, and Mats Johnsson. XML and Objects-The Future for E-Forms on the Web. In *WETICE*. IEEE Computer Society, 1999.
- [28] Cong Yu and H. V. Jagadish. Schema Summarization. In *VLDB*, 2006.
- [29] Moshé M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. In *VLDB*, 1975.

APPENDIX

A. DATASETS AND QUERY SETS

We provide a detailed description of the datasets and query sets used in our experiments (Sec. 5) in this section.

A.1 MiMI

MiMI (Michigan Molecular Interactions) [3] 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 [18].

A.1.1 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.

A.2 Geoquery

The Geoquery database [1] contains geographical information about the United States including states, cities, roads, rivers, etc. (schema shown in Fig. 10). 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.

A.2.1 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.

A.3 Jobsquery

The Jobsquery database [2, 9] 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 [2] 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.

A.3.1 Query Workload

Here again, we needed to translate the data from Prolog to XML and deduce an XML schema. The queries had to be converted 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

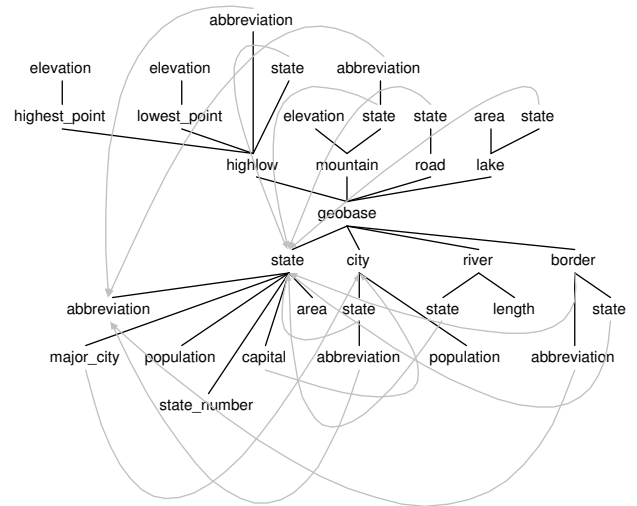


Figure 10: Geoquery Schema

the total). These queries include a wide range of selection queries including simple, conjunctive, disjunctive, quantified and negated selection conditions.

B. COMPARISON WITH HUMAN-DESIGNED FORMS

In Sec. 5.3, we presented the overall results of an experiment comparing an automatically generated form with a form designed for a popular publicly accessible website (Monster.com) in the same domain (job listings). Here we shed some light onto why the usefulness of the two forms (by our evaluation metric) differed so widely. First we show what the two forms look like: Fig. 13 shows the automatically generated form while Fig. 11 shows the human generated form.

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. 12). 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 form:

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 ?

Figure 11: Monster.com Query Form (Advanced)

| 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 | | |

Monster recommends using [Apply Now](#). [Learn more](#) [Save to job file](#)

Figure 12: Sample Monster.com Query Result

- show the jobs with the title systems analyst requiring 2 years of experience ?
- show jobs in austin that require a bscs ?
- what are the jobs in washington that require at least 5 years of experience ?
- can i find a job making more than 40000 a year without a degree ?
- i want a job that doesnt use windows ?
- are there any jobs for people in austin that want to program in lisp but do not have a degree ?
- 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).

C. COST OF FORM GENERATION

The steps taken in generating forms automatically were outlined in Algorithms 1 - 4. 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:

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

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

Figure 13: Form automatically generated for Jobsquery

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 complete⁷. 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.

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

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

⁷Database 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 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 each form contains k_f form-elements per entity. The cost of this form design step is given by:

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

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:

$$\begin{aligned} C_{fg} &\in O(n^2 + n_e(d_e + n_{e_a} d_{e_a} + n_{e_r} d_{e_r}) \\ &\quad + k_e(n_{e_a} \log n_{e_a} + n_{e_r} \log n_{e_r})) \end{aligned}$$

D. 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.

D.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 automatically 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 having more than two or three entities.

D.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, 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.