Learning to Create Data-Integrating Queries

Partha Pratim Talukdar Marie Jacob Muhammad Salman Mehmood
Koby Crammer Zachary G. Ives Fernando Pereira* Sudipto Guha
University of Pennsylvania, Philadelphia, PA 19104, USA
{partha,majacob,salmanm,crammer,zives,pereira,sudipto}@cis.upenn.edu

ABSTRACT

The number of potentially-related data resources available for querying — databases, data warehouses, virtual integrated schemas — continues to grow rapidly. Perhaps no area has seen this problem as acutely as the life sciences, where hundreds of large, complex, interlinked data resources are available on fields like proteomics, genomics, disease studies, and pharmacology. The schemas of individual databases are often large on their own, but users also need to pose queries across multiple sources, exploiting foreign keys and schema mappings. Since the users are not experts, they typically rely on the existence of pre-defined Web forms and associated query templates, developed by programmers to meet the particular scientists' needs. Unfortunately, such forms are scarce commodities, often limited to a single database, and mismatched with biologists' information needs that are often context-sensitive and span multiple databases.

We present a system with which a non-expert user can author new query templates and Web forms, to be reused by anyone with related information needs. The user poses keyword queries that are matched against source relations and their attributes; the system uses sequences of associations (e.g., foreign keys, links, schema mappings, synonyms, and taxonomies) to create multiple ranked queries linking the matches to keywords; the set of queries is attached to a Web query form. Now the user and his or her associates may pose specific queries by filling in parameters in the form. Importantly, the answers to this query are ranked and annotated with data provenance, and the user provides *feedback* on the utility of the answers, from which the system ultimately learns to assign costs to sources and associations according to the user's specific information need, as a result changing the ranking of the queries used to generate results. We evaluate the effectiveness of our method against "gold standard" costs from domain experts and demonstrate the method's scalability.

1. INTRODUCTION

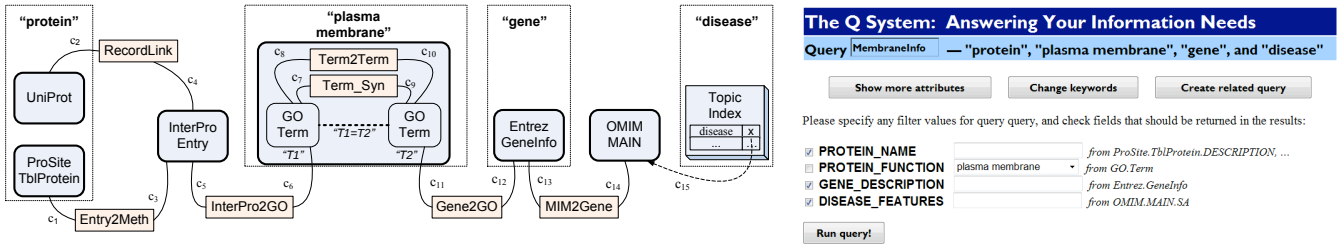
The variety and complexity of potentially-related data resources available for querying — databases, data warehouses, virtual integrated schemas — grows ever more rapidly. This is possibly most marked in the life sciences, where hundreds of large, complex, interlinked, and overlapping data resources have become available on fields like proteomics, genomics, disease studies, and pharmacology. *Within* any of these databases, schemas are often massive: for example, the Genomics Unified Schema [28], used by a variety of parasite databases such as CryptoDB and PlasmoDB, has 358 relations, many of which have dozens of attributes each. Additionally, biomedical researchers need to pose integrative queries *across* multiple sources, exploiting foreign keys, similar terms, and other means of interlinking data: using a plethora of sources can reveal previously undiscovered relationships among biological entities [6, 33], and some of these may lead to scientific breakthroughs. Currently, biologists have been restricted to queries embodied in Web forms, typically limited to a single database, that were created by full-time database maintainers; or they have to become experts in SQL, Web services, the different schemas, and scripting languages to create their own queries that span multiple schemas.

Neither of these solutions is suitable for a biomedical researcher who is not a database expert but needs to explore rich data sources to create testable scientific hypotheses. At best, the researcher will be able to obtain useful results by manually combining the results of predefined queries on different databases. This cut-and-paste process is error-prone and likely to miss data sources and less evident linkages. There is thus a great potential value for scientific discovery in providing tools that support *ad hoc* creation of new queries (or, more typically, query templates that may be parameterized with different filter conditions and reused multiple times) by end users. In this paper, we focus on supporting this need.

Of course, the problem of querying across large schemas is not unique to the life sciences: there are many enterprise, business-to-business, and Web shopping scenarios in which it would be desirable to query across large sets of complex schemas. Many have attempted to address these needs through data integration (also called EII [8]), warehousing, or middleware solutions, which present a uniform interface (queriable schema and/or APIs) over all of the data sources. Unfortunately, for large domains it is very difficult to define that uniform interface and the mappings to it (the *schema mapping* problem [35]). Beyond this problem, the uniform interface must *itself* be large and complex in order to capture all relevant concepts. Existing integration tools provide some level of mapping and mediation among sources, but (1) there is still a need to author queries that join and union together multiple relations in the mediated schema; (2) the mappings are probably incomplete with respect to any one EII environment, but there may be ways of relating data with other, external databases or EII environments; (3) some mapped sources may not be considered appropriate for a given query (e.g., due to a lack of authoritativeness or accuracy).

*Currently at Google, Inc., Mountain View, CA 94043, USA.

VLDB '08 Auckland, New Zealand



(a) Schema elements, associations, and matchings to keywords

(b) Resulting Web form

Figure 1: A sample query. Part (a) shows schema elements (nodes) matching a query about proteins, diseases, and genes related to "plasma membranes." The relations come from different bioinformatics sources, plus site-provided correspondence tables (e.g., InterPro2GO), the results of a record linking tool (RecordLink), a Term that may be used directly or combined with its superclasses or synonyms through ontologies (GO Term2Term, Term_Syn), and instance-level keyword matching (topic index). Rounded rectangles represent conceptual entities and squared rectangles represent tables relating these entities; Q considers there to be a weighted *association* edge based on the foreign key or link dereference. Part (b) shows the Web form representing a view, which allows users to fill in selection conditions and execute a query.

CQ1: $q(prot, gene, typ, dis)$:- $TblProtein(id, prot, \ldots), Entry2Meth(ent, id, \ldots), InterPro2Go(ent, gid),$
$Term(gid, typ), Gene2GO(gid, giId), GeneInfo(giId, gene, \ldots), MIM2Gene(giId, mId), MAIN(mId, dis, \ldots),$
$typ =$'plasma membrane'

CQ2: $q(prot, gene, typ, dis)$:- $TblProtein(id, prot, \ldots), Entry2Meth(ent, id, \ldots), InterPro2Go(ent, gid1),$
$\mathbf{Term\_Syn}(gid1, gid2), Term(gid2, typ), Gene2GO(gid2, giId), GeneInfo(giId, gene, \ldots), MIM2Gene(giId, mId),$
$MAIN(mId, dis, \ldots), typ =$ 'plasma membrane'

CQ3: $q(prot, gene, typ, dis)$:- $TblProtein(id, prot, \ldots), Entry2Meth(ent, id, \ldots), InterPro2Go(ent, gid1),$
$\mathbf{Term2Term}(\_,$'part_of', $gid1, gid2), Term(gid2, typ), Gene2GO(gid2, giId), GeneInfo(giId, gene, \ldots),$
$MIM2Gene(giId, mId), MAIN(mId, dis, \ldots), typ =$ 'plasma membrane'

CQ4: $q(protnam, gene, typ, dis)$:- $UniProt(ac, nam, \ldots), \mathbf{RecordLink}(ac, nam, ent, protnam), Entry(ent, protnam),$
$InterPro2Go(ent, gid1), Term2Term(\_,$'part_of', $gid1, gid2), Term(gid2, typ), Gene2GO(gid2, giId),$
$GeneInfo(giId, gene, \ldots), MIM2Gene(giId, mId), MAIN(mId, dis, \ldots), typ =$ 'plasma membrane'

Table 1: Excerpts of some potential queries from the graph of Figure 1, with important differences highlighted in boldface.

An open challenge is how to provide a solution that:

- Enables *community-driven* creation of information resources tailored to different needs.
- Allows non-expert users to author new families of queries that may span multiple interlinked databases, even if these are not under a single mediated schema.
- Provides a means for the user to distinguish or select among the different data sources depending on the user's *information need*. Not all sources of data, or means of finding associations among tables, are of interest to the user.

A potentially promising approach to authoring queries without using SQL — primarily used within a *single* database — has been to start with *keyword queries* and to match terms in tuples within the data instance's tables. If multiple keywords are provided and match on different tables, foreign keys within the database are used to discover "join paths" through tables, and query results consist of different ways of combining the matched tuples [4, 23]. Path lengths are used as costs for the joined results. In general, results will be highly heterogeneous and specific to the keyword matches in the tuples. It is also possible to support approximate, similarity-based joins [11] or approximate keyword matches.

Unfortunately, the above work may be insufficient for scientific users, because such work assumes query-insensitive costs for path length, attribute similarity, and other matching steps, when scientists may need costs *specific to the context of the query* (i.e., the setting under which the query is being posed). Preferences for sources may depend on whether users are posing "what-if" types of exploratory queries or refining previous answers; they may depend on the specific query domain; or they may depend on the (perceived or actual) quality of the individual data sources. Recent bioinformatics work [6, 33] shows that there are often *many* combinations of data sources that can reveal relationships between biological entities, and biologists need ways of discovering, understanding, and assessing the quality of each of the possible ways of linking data.

Rather than simply answering queries by matching keywords to tuples and finding associations, we propose a method for defining and *interactively refining* families of queries, based on keywords. This family of queries is associated with a parameterized Web form in a way that is familiar with biologists. (For simplicity of exposition, in the remainder of this paper, we will term this parameterizable family of queries a *view* — note, however, that it really represents a view *template* to which different values may be bound.)

Learning the Right View for a Web Form

We sketch a sample interaction with our system, which we call the Q System. Figure 1 shows a running example in this paper that we will revisit in more detail in Section 3: a biologist wants to author a new Web form for the members of her lab to use. We assume Q has pre-encoded knowledge of the different schema elements (tables, attributes, XML nodes, objects, etc.) and, in special cases, tuples, in the bioinformatics domain. It also has information about *associations* between these elements: record linking tables, foreign keys, references, ontologies, etc. Such associations come from the schemas (e.g., they were specified as foreign keys, URLs, or ontology subclasses), or they may also be produced offline by human experts, schema mappers [35], or record linking [16] tools. Schema elements are modeled as nodes in a graph (Figure 1a); associations can be thought of as edges, each with a cost (the c's in the figure) that quantifies the system's bias against using that asso-

ciation to create a query template. The cost may for instance reflect the fact that an association comes from automatic schema matching tools, and is thus error prone; that an association contains suspect data; or that an association is based on approximate matches. Costs are typically pre-initialized to a default "neutral" value, and they will be *adjusted by the system through learning* to reflect user preferences conveyed by user feedback.

Suppose our biologist has a particular *information need* that depends on her domain of expertise, the precision of queries to be posed (e.g., exploratory vs. answer refinement queries), and preference for certain data sources.

1. The biologist specifies a set of keywords. In this case, these are to create a (parameterizable) view for **protein**s, **disease**s, and **gene**s related to the **plasma membrane**s of cells. The biologist accepts the defaults, which are to create the view using the 4 most promising data integration queries the Q System can find.

2. By matching the keywords against the schema graph, finding trees in the graph that contain a match to each keyword, and summing the costs of the edges within each tree (which represents a conjunctive query), Q finds the 4 *best-ranked* conjunctive queries (CQs), shown in Datalog notation in Table 1. The top-ranked query follows associations and intermediate relations to connect *TblProtein*, a single GO *Term*, an entry in *GeneInfo*, and an entry in *OMIM MAIN* (identified as a disease database). The next two queries consider alternative means of using **two different GO Terms** T1 and T2 (using *Term_Syn*onyms or *Term2Term* relationships such as parent concept-subconcept), to relate *TblProtein* and associated InterPro *Entry* tuples with a *GeneInfo* tuple. The final query uses the *UniProt* table instead of *TblProtein*, and here needs a cross reference from a record linking tool in order to find the corresponding Interpro *Entry*.

3. The biologist takes the output of Q — a suggested union over CQ1-CQ4 — and makes final refinements (projecting out attributes that are uninteresting). (See the Appendix for a screen shot.) The result is a Web form (Figure 1b), which is saved to a permanent URL.

4. The biologist, and others in her lab, make use of this Web form, parameterizing it with specific disease features or protein names. When they get query results, some of these are more meaningful than others. Each result tuple will be given the rank of the (highest-ranked) query that produced it. Suppose it is more useful to exploit parent concept-subconcept relationships among *GO Term*s than to exploit *Term* synonyms. The users provide *feedback* on answers from CQ3, specifying that these should be ranked above answers returned by CQ2. (The Appendix includes a screen capture.)

5. The system will accept this feedback on the answers, determine the queries that produced those answers (via *data provenance* [7, 13, 20]), and adjust the ranking of the queries. (Note that the feedback on single tuples is *generalized* to other tuples from the same query. In turn, adjustments to one query may generalize to other queries, e.g., those that use synonyms and parent concept-subconcept relationships). Q will then recompute the **new** top-4 queries, which may include queries that previously had lower ranking.

Challenges. This mode of interaction creates a number of fundamental challenges. First, we must have a unified, end-to-end model that supports computation of ranked queries, which produce correspondingly ranked results, and it must be possible to *learn new query rankings from feedback* over the results, ultimately converging to rankings consistent with user expectations. In support of this, we must be able to find the top-k queries, begin producing answers, and learn from feedback at interactive-level speeds. We must always be able to determine results' provenance, as a means of connecting feedback to the originating query or queries. Additionally, it is essential that we be able to *generalize* feedback to results other than the one to which the user directly reacted. In part this is due to the "curse of dimensionality": we must generalize if we are to learn from small numbers of examples.

Contributions. In our approach, edge weights encode shared learned knowledge about the usefulness of particular schema elements, across multiple queries and users with similar preferences and goals. Multiple users in the same lab or the same subfield may wish to share the same view and continuously refine it. They may also wish to pose other related queries, and have feedback affect the entire set of queries together. On the other hand, groups of users with very different goals (say, highly speculative exploration of associations, versus refinement of results to find the highest-reliability links) can have their own set of views with a different set of weight assignments. In essence, each sub-community is defining its own integrated schema for viewing the data in the system — which includes not only a set of schema mappings (associations) but also a set of weights on the mappings and source relations. This represents a *bottom-up*, *community-driven* scheme for integrating data. Our paper makes the following contributions:

- We bring together ideas from data integration, query-by-example, data provenance, and machine learning into a novel unified framework for authoring *through feedback* families of queries corresponding to a bioinformatics Web form, but potentially spanning many databases. We exploit the output from record linking and schema mapping tools.
- We develop efficient search strategies for exploring and ranking associations among schema elements, links, synonyms, hypernyms, and mapping tables — producing top-ranked queries at interactive speeds. This depends on a new approximation scheme for the K-best Steiner tree problem (explained in Section 5), which scales to much larger graphs than previous methods.
- We develop efficient techniques for integrating data provenance and online learning techniques — learning at interactive speeds.
- We experimentally validate the efficacy of our solutions, using real schemas and associations.

Roadmap. Section 2 briefly reviews the state of the art in related work. Section 3 presents our basic architecture and operation. Section 4 describes how queries are answered in our system, and Section 5 describes how feedback is given and processed. We present experimental results showing scalability and rapid learning in Section 6, and we conclude and discuss future work in Section 7.

2. RELATED WORK

The problem of providing ranked, keyword-based answers to queries has been the subject of many studies. Most focus on providing *answers* based on the keywords, rather than on constructing persistent *views* that can be used for multiple keywords. We briefly review this work and explain why our problem setting is different.

Information retrieval [1] focuses on providing ranked documents as query answers, given keyword queries. It does not generate or learn structured queries that combine data from multiple databases. Similarly, while natural-language query interfaces have been studied that create structured queries over databases, e.g., [34], our goal is to take keyword queries, not natural language, and a large and

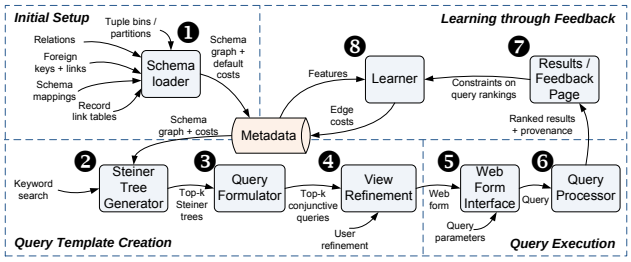


Figure 2: Architectural stages of the Q System.

diverse set of data sources and associations, and to learn the appropriate structural queries.

At a high level, our work seems most similar to keyword search over databases [4, 5, 23, 25]. There, keyword matches involve both data items and metadata. Results are typically scored based on a combination of match scores between keywords and data values, length of the join path between the matched data items, and possibly node authority scores [2, 21]. The NAGA system [26] additionally considers an ontology-like knowledge graph and a generative model for ranking query answers. In many of these systems, ranking is based on costs that are computed in additive fashion (much like the cost model we adopt). In contrast to all of these systems, we seek to *learn* how to score results based on user preferences, since associations between scientific data sources do not necessarily reflect authority, and in any case perceived authority may vary with user characteristics and needs. This naturally complements a number of existing bioinformatics query authoring systems that rely on expert-provided scores [6, 33].

Existing "top-k query answering" [11, 18, 30, 31] provides the highest-scoring answers in answering ranked queries. Our goal is to identify possible queries to provide answers by connecting terms, to separately rank each combination, to output the results using this rank, and finally to enable feedback on the answers. The step in which results are output could be performed using top-k query processing algorithms.

Our work uses machine learning in a way that is complementary to other learning-based tools and techniques for data integration. Schema matching tools [35] provide correspondences and possibly complete schema mappings between different pre-existing schemas: we use mappings as input, in the form of associations that help our system to create cross-site queries. We can also learn the quality of the mappings based on feedback. Finally, recent work focuses on learning to construct mashups [36], in a way that generalizes the information extraction problem and suggests new columns to be integrated: this is also complementary to our work, which focuses on determining how to decide which queries and answers should be used to populate the results.

3. ARCHITECTURE AND OPERATION

We divide system operation into four major phases: initial setup, query template creation, query execution, and learning through feedback. We discuss each of these phases in order, focusing on the modules and dataflow.

3.1 Initial Setup

Refer to Figure 2 to see the components of the Q System. During initial setup, Q's **Schema Loader** (the box highlighted with the numeral 1 in the figure) is initially given a set of data sources, each with its own schema. Data items in each schema might optionally contain links (via URLs, foreign keys, or coded accession numbers) to other data sources. Additionally, we may be given certain known correspondences or transformations as the result of human input or data integration tools: for instance, we may have schema mappings between certain elements, created for data import, export, or peer-to-peer integration [22]; some data items may be known to reference an externally defined taxonomy or ontology such as GeneOntology (GO); and tools may be able to discover (possibly approximate) associations between schema elements. All such information will be encoded in the *schema graph*, which is output by the Schema Loader and saved in a metadata repository.

Figure 1 features two classes of relations as nodes: blue rounded rectangles represent entities, and orange elongated rectangles represent cross-references, links, or correspondence tables. Edges represent associations between nodes (generally indicating potential joins based on equality of attributes). The schema graph in the example illustrates a common feature of many bioinformatics databases, which is that they frequently contain cross-referencing tables: *Entry2Meth*, *InterPro2GO*, etc., represent the database maintainers' current (incomplete, possibly incorrect) information about inter-database references. Additionally, our example includes a correspondence table, *RecordLink*, that was created by a schema mapping/record linking tool, which matches *UniProt* and *InterPro* tuples. As previously described, any of the associations encoded as edges may have a *cost* that captures its likely utility to the user: this may be based on reliability, trustworthiness, etc., and the system will attempt to learn that cost based on user feedback. These costs are normally initialized to the same default value.

3.2 Query Template Creation

The user defining a query template poses a **keyword query**

```
protein "plasma membrane" gene disease
```

which is matched against the schema graph by the **Steiner Tree Generator** (box #2 in Figure 2). A pre-processing step consists of matching keywords against graph elements: We can see from the figure that the first term matches against *UniProt* and *TblProtein* (based on substring matching against both relation and attribute names). The term "plasma membrane" does not match against any table names or attributes — but rather against a term in the GO ontology, which includes (as encoded data) standardized terms. Terms in the ontology have both subclasses (*Term2Term*) and synonyms (*Term_Syn*), and hence the system must consider these in the query answers as well. The keyword "gene" matches as a substring against *GeneInfo*, and finally, "disease" matches against an entry in an index from topics to databases. Implicitly, as part of the keyword matching process, the Q System adds a node to the schema graph for each keyword, and an edge to each matching node. (For visual differentiation in the figure, we indicate these edges by drawing a dashed rectangle around each keyword and its matching nodes.)

Now, given keyword matches against nodes, the Steiner Tree Generator can determine the *best* (top-k) queries matching the keywords. Its goal is to find the k trees of minimal cost contained in the schema graph, each of which includes all of the desired (keyword) nodes, *plus* any additional nodes and edges necessary to connect them. This is technically a Steiner tree; the cost of each Steiner tree is the sum of edge costs. (We discuss below how edge costs are obtained.) Note the subtlety that this module does not generate queries to compute the top-k *answers*; rather, it produces the top-k-scoring *queries* according to our schema graph. These may return more or fewer than k answers; but commonly each query will return more than one answer.

The **Query Formulator** (box 3) takes each of the top-k Steiner trees and converts it into a conjunctive query (nodes become relations, edges become joins, and the cost of the query is the sum of the costs of the edges in the Steiner tree). Consider, e.g., two queries from Table 1:

- One query, CQ1, might represent the tree connecting *TblProtein* through *Entry2Meth* to *Entry*, followed by joining *InterPro2GO* to get to *Term*, then joining with *Gene2GO* to get a *GeneInfo*, and finally joining with *MIM2Gene* and *OMIM MAIN* to get a complete result. This query is shown in Datalog form as the first row in Table 1, and its cost would be computed as $c_1 + c_3 + c_5 + c_6 + c_{11} + c_{12} + c_{13} + c_{14} + c_{15}$.

- A different query, CQ4, uses a correspondence table from a record linking (entity matching) tool that produces potential matches between *UniProt* proteins and InterPro *Entry* tuples. In general, the quality of those matches might vary with the matched tuple, and the corresponding cost contribution would then depend on the tuple. However, as we discuss below in more detail, we simplify the problem here to pay a single cost for any use of record linking, obtaining the tree cost $c_2 + c_4 + c_5 + c_6 + c_{11} + c_{12} + c_{13} + c_{14} + c_{15}$.

In principle, queries like CQ4 may assign different scores to different tuples, because record linking is not equally reliable for all tuples. However, we believe there are several disadvantages in fully adopting a per-tuple ranking model. First, in general the overall score of a tuple is based on features it likely shares with other tuples. We would like knowledge about scores to transfer to other tuples with similar features — to take what we learn on a per-tuple basis, discover common classes among the tuples, and then apply what we learned to our classes. Unfortunately, the finer-grained our ranking model, the more examples we would need to learn anything useful; this mirrors the so-called "curse of dimensionality" that is often discussed in the data mining literature. Second, simply *processing* queries in such a model creates a very large top-k-results query processing problem: for a large schema graph we may have thousands of potential queries to merge together, and current top-k query processing algorithms have not tackled this scale.

Our approach is to use a "binning" strategy whereby tuples within a relation are grouped or *binned* according to administrator-defined partitioning functions — typically common features or score ranges. For example, in an ontology we may wish to put all direct superclasses into one "bin"; all grandparent classes into a different bin; all sibling or synonymous classes into yet another bin; etc. (This binning is indicated, for instance, in Figure 1 for the *Term* relation.) For a record linking table, we may wish to group together all associations above a certain threshold into one "high confidence" bin; another range of values into "medium confidence"; etc. Now we treat each bin as a separate node in the schema graph, and we separately handle each bin with respect to top-k query generation and learning. This approach allows us to "share learning" across a group of tuples, and also to efficiently prune the space of possible top-k queries at query generation time, rather than choosing among tuples at runtime.

At the **View Refinement** stage (box 4), the top-scoring queries are combined into a disjoint union (i.e., aligning like columns and padding elsewhere with nulls, as described in Section 5), forming what we term a *union view*. Next, the query author may *refine* the query, adding projections, renaming or aligning columns, and so on. At this stage, the view is optionally given a name and made persistent for reuse. An associated Web form is automatically generated, as in Figure 1b. (Recall that our "view" actually represents a template for a family of queries with similar information needs, which are to be parameterized by the user to actually pose a query.)

3.3 Query Execution

Any user with permissions (not only the author) may access the **Web Form Interface** (box 5), parameterize the fields of the query through the Web form, and execute it. This invokes the **Query Processor** (box 6), which is a distributed relational query engine extended to annotate all tuples with their *provenance* or lineage [7, 13, 20], which is essential for later allowing the system to take feedback on tuples and convert it into feedback on queries. Of course, the query processor must also return these annotated results in increasing order of cost, where they receive the cost of the query that produces them. (If a tuple is returned by more than one query, it is annotated with the provenance of all of its producer queries, and given the cost of the lowest-cost query.)

3.4 Learning through Feedback

Once the user has posed a query, he or she may look over the results in the **Results/Feedback Page** (box 7) and provide *feedback* to the system, with respect to the relative ordering and set of answers. The system will generalize this feedback to the queries producing the answers. Then the **Learner** (box 8) will adjust costs on the schema graph, thus potentially changing the set of queries associated with the Web form, and altering the set of answers to the query. The new results are computed and returned at interactive speeds, and the user may provide feedback many times. Our goal is to learn the costs corresponding to the user's mental model of the values of the respective sources.

In the subsequent two sections, we discuss the implementation of the main modules in detail. We omit further discussion of Module 1, the Schema Loader, as it is straightforward to implement. Our discussion begins with the query creation and answering stages (boxes 2-6), and then we move on to discuss the feedback and learning stages (boxes 7 and 8).

4. QUERIES AND QUERY ANSWERS

In this section, we begin by discussing the details of the schema graph (Section 4.1) and cost model (Section 4.2), which form the basis of all query generation. Section 4.3 then considers how keywords are matched against the graph, and Section 4.4 addresses the key problem of finding the best queries through Steiner tree generation. Finally, we discuss how Steiner trees are converted into query templates (Section 4.5), and how these templates are parameterized and executed (Section 4.6).

4.1 Foundation: the Schema Graph

As its name connotes, the *schema graph* is primarily at the schema and relationship level: nodes represent source relations and their attributes and edges represent associations between the elements. Our query system additionally supports matches at the tuple level — which is especially useful when searching topic indices and ontologies (as in Figure 1) — but our emphasis is primarily on the metadata level, as explained in the previous section.

Nodes. Nodes represent source relations containing data that may be of interest. The query answers should consist of attributes from a set of source nodes.

Edges. Within a given database, the most common associations are **references**: a foreign key pointing to another relation, a hyperlink pointing to content in another database, etc. However, a variety of additional associations may relate nodes, particularly across different sources: **subclass** ("is-a") is very common in ontologies or class hierarchies; **maps-to** occurs when there exists a view, schema mapping, synonym, or correspondence table specifying a relationship between two different tables; **similar-to** describes an association that requires a similarity join. All edges have cost expressions associated with them.

4.2 Cost Model

The *costs* associated with edges in the schema graph are simple weighted linear combinations of edge *features*. Features are domain-specific functions on edges that encode the aspects of those edges that are relevant to user-ranking of queries: in essence, they capture distinctions that may be relevant to a user's preference for an edge as part of the query. The identities of edge end-nodes are the simplest and most obvious features to use: the cost will be a function of the nodes being associated by the edge. However, more general features, for instance the type of association (subclass, maps-to, similar-to) used to create an edge, are also potentially useful. Each feature has a corresponding *weight*, representing the relative contribution of that feature to the overall cost of the query: this is set to a default value and then *learned*. Crucially, the use of common features in computing costs allows the Q System to share information about relevance across different queries and edges, and thus learn effectively from a small number of user interactions.

We discuss features and how they are learned in Section 5. For purposes of explaining query answering in this section, we note that the cost of a tree is a weighted linear combination of the features of the edges in the tree. This model was carefully chosen: it allows simple and effective learning of costs for the features from user feedback [12].

Intuitions behind the cost model. An edge cost in our model can be thought of as the logarithm of the odds (in the sense of betting) that using that edge in a query leads to *worse* answers (from the user's point of view) than including the average alternative edge. Conversely, lower costs correspond to better odds that using the edge will lead to better answers. Since the costs are parameterized by a shared weight vector $\mathbf{w}$, feedback from a few queries will typically affect edges involved in many different queries. Selecting query trees according to the updated weights will increase the odds that user-favored answers are shown first.

We observe that our cost model somewhat resembles that of other keyword query systems (e.g. [25]), which do not use features or learning, but often use an additive model based on edge costs. Our notion of cost and its use in query construction is different from the probabilities in probabilistic databases [14], which represent uncertainty about whether specific relationships hold. A low-cost answer tuple in our model is not necessarily very probable, but simply one that was derived by a query that involves associations favored by the user. Our costs are also different from edge capacities in authority flow models [2, 37], which encode the relative strength of edges as carriers of authority between nodes. A low-cost edge in our model is not necessarily one that passes more authority from its source to its target, but simply one that supports a join that has proven useful.

4.3 Matching Keyword Queries

Given a user's keyword query, the Q System begins by matching it against nodes in the schema graph. A keyword query consists of a set of terms $Q = \{q_1, \ldots, q_n\}$. Let N_q be the set of nodes in the schema graph that match $q \in Q$, and let $N = \bigcup_{q \in Q} N_q$. A node matches a term if its label (consisting of the relation and attribute names) contains the term as a substring, or, in special cases (e.g., for taxonomies and synonym tables), the *instance* of the relation represented by the node contains the term.

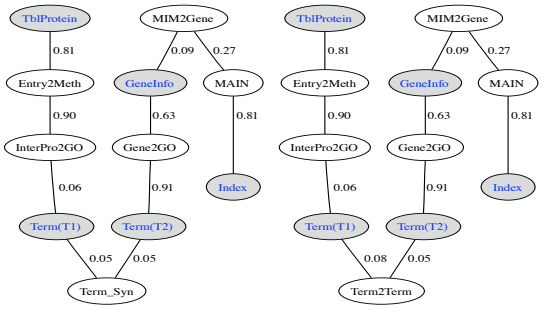
For each $q \in Q$, we add a special *keyword node* q to the graph, and also edges (q, n) for all $n \in N_q$. These new edges can be assigned costs according to an appropriate scoring function, for instance TF/IDF. The system now attempts to find the k lowest-cost *Steiner trees* that contain all of the keyword nodes.

Each such tree T also includes non-keyword nodes that are needed to complete a (connected) tree. As discussed previously, the cost of T is the sum of costs of its edges, and those costs are weighted combinations of edge features. Formally, the feature weights form a *weight vector* $\mathbf{w}$, and the cost $C(T, \mathbf{w})$ of T is the sum of $\mathbf{w}$-dependent edge costs:

$$C(T, \mathbf{w}) = \sum_{e \in E(T)} C(e, \mathbf{w}) \quad (1)$$

where $E(T)$ is the set of edges of T.

We next discuss the process of finding Steiner trees. The goal here is to quickly (at interactive rates) produce an ordered list of subtrees of the schema graph that purport to satisfy the information need specified by a set of keywords. That ordered list is determined by the current feature weight vector $\mathbf{w}$. Later, the learning process will adjust this weight vector so that the order of the returned query trees corresponds better to user preferences about the order of the corresponding answers.



(a) Cost=4.58 (b) Cost=4.61

Figure 3: Steiner trees for queries CQ2 and CQ3 in Table 1. Nodes matching query keywords are shaded, with blue text.

4.4 Steiner Tree Generation

The task of our Steiner Tree Generator is not merely to find a single Steiner tree in the graph, as is customary in the literature — but to find the *top k Steiner trees* in order to find the k best queries. Here, we are faced with the question of whether to find the actual k lowest-cost Steiner trees, or to settle for an approximation. For small graphs we use an exact algorithm for finding the k lowest-cost Steiner trees, and for larger graphs we develop a heuristic. This allows us to find the optimal solution for small schema graphs, and yet to scale gracefully to larger schemas.

4.4.1 Steiner Trees via Integer Programming

We first formalize the Steiner tree problem. Let G be a directed graph with nodes and edges given by $V(G)$ and $E(G)$, respectively. Each edge $e = (i, j) \in E(G)$ has a cost $C(e)$. We also have a set of nodes $S \subseteq V(G)$. A directed subtree T in G connecting the nodes in S is known as a Steiner tree for S. The nodes in $V(T) \setminus S$ are called Steiner nodes. The cost of T is $C(T) = \sum_{e \in E(T)} C(e)$. Finding the minimum cost Steiner tree on a directed graph (STDG) is a well-known NP-hard problem [41, 17].

Finding a minimum-cost Steiner tree on a directed graph [41] can be expressed as a mixed integer program (MIP) [40] in a standard way (Figure 4) [41]. This encoding requires one of the nodes r in $V(G)$ to be chosen as root of the Steiner tree. Hence, the minimum cost Steiner tree can be obtained by running the appropriate MIP with each node in $V(G)$ taken as root separately and then selecting the lowest-cost tree from at most $|V(G)|$ candidates. This can be time consuming especially for large schema graphs. For the experiments reported in this paper, we convert every schema graph edge (which, despite describing a foreign key, is really a

STEINER (G, S, C) :

$$\min_{\substack{\mathbf{x},\mathbf{y} \\ r \in V(G)}} \sum_{(i,j) \in E(G)} C(i,j) \times y_{ij}$$

s.t. $S' = S - \{r\}$

$$\sum_{h \in V(G)} x^k_{rh} - \sum_{j \in V(G)} x^k_{jr} = 1 \ \ \forall k \in S' \quad \text{(C1)}$$

$$\sum_{h \in V(G)} x^k_{kh} - \sum_{j \in V(G)} x^k_{jk} = -1 \ \ \forall k \in S' \quad \text{(C2)}$$

$$\sum_{h \in V(G)} x^k_{ih} - \sum_{j \in V(G)} x^k_{ji} = 0 \ \ \forall i \in V(G) \setminus S \quad \text{(C3)}$$

$$x^k_{ij} \leq y_{ij} \ \forall (i,j) \in E(G), k \in S' \quad \text{(C4)}$$

$$x^k_{ij} \geq 0 \ \forall (i,j) \in E(G), k \in S' \quad \text{(C5)}$$

$$y_{ij} \in \{0,1\} \quad \text{(C6)}$$

Figure 4: Mixed integer program for min-cost Steiner trees.

STEINERIE(G, S, I, X, C) :

$$\min_{\substack{\mathbf{x},\mathbf{y} \\ r \in S}} \sum_{(i,j) \in E(G)} c(i,j) \times y_{ij}$$

$$S^+ = S \cup \{i : (i,j) \in I\} \quad \text{(T1)}$$

s.t.

Constraints C1-C6 from STEINER(G, S^+, C)

$$\sum_{h \in V(G)} y_{hr} = 0 \quad \text{(C7)}$$

$$\sum_{h \in V(G)} y_{hi} \leq 1 \ \ \forall i \in V(G) \setminus \{r\} \quad \text{(C8)}$$

$$\sum_{k \in S'} x^k_{ij} \geq 1 \ \ \forall (i,j) \in I \quad \text{(C9)}$$

$$y_{ij} = 0 \ \ \forall (i,j) \in X \quad \text{(C10)}$$

Figure 5: MIP for Steiner tree with inclusions and exclusions.

bidirectional association) to a pair of directed edges. With such bi-directional edges, one can find the minimum cost Steiner tree by solving STEINER (G, S, C) with any of the nodes in S fixed as root, avoiding the need to iterate over all the vertices in the graph. Unless otherwise stated, we assume the graph to be bi-directional in what follows. In STEINER (G, S, C), an edge $(i, j) \in E(G)$ is included in the solution iff $y_{ij} = 1$. The MIP STEINER (G, S, C) that finds the lowest-cost (according to cost function C) Steiner tree in G and containing nodes S can be viewed as a network flow problem where x^k_{ij} specifies the amount of flow of commodity k flowing on edge (i, j). Flow on an edge (i, j) is allowed only if that edge is included in the solution by setting $y_{ij} = 1$. This is enforced by constraint C4. All flows are nonnegative (constraint C5). Flow of commodity k originates at the root r (constraint C1) and terminates at node k (constraint C2). Conservation of flow at Steiner nodes is enforced by constraint C3.

However, we need more than just the minimum-cost tree: we need the k lowest-cost trees. To achieve this, we modify the MIP of Figure 4 so that it can be called multiple times with constraints on the sets of edges that the solution can include. The modified program is shown on Figure 5.

The MIP STEINERIE(G, S, I, X, C) finds the lowest cost (according to cost function C) Steiner subtree of G rooted at r that contains the nodes in S, which must contain the edges in I and cannot contain any edge in X. C9 guarantees that there is flow of at least one commodity on all edges in I. T1 with C7-C9 enforce the inclusion constraints, while the exclusion constraints are enforced by C10. We must also ensure the result will be a tree by requiring flow to pass through the source nodes of the edges in I. Step T1 expands S by including source nodes of the edges in I. This ensures there is a directed path from root r to the source nodes of the edges that must be included. C7 ensures that there is no incoming active edge into the root. C8 ensures that all nodes have at most one incoming active edge.

4.4.2 K-Best Steiner Trees

Algorithm 1 KBESTSTEINER(G, S, C, k). **Input**: Schema graph G, keyword nodes S, edge cost function C, number of returned trees k. **Output**: List of at most k trees sorted by increasing cost.

```
1: Q ← empty priority queue
   {Q contains triples (I, X, T) sorted by T's cost.}
2: T = STEINERIE(G, S, ∅, ∅, C)
3: if T ≠ null then
4:    Q.INSERT((∅, ∅, T))
5: end if
6: A ← empty list
7: while Q ≠ ∅ ∧ k > 0 do
8:    k = k − 1
9:    (I, X, T) ← Q.DEQUEUE()
10:   A.APPEND(T)
11:   Let {e_1, ..., e_m} = E(T) \ I
12:   for i = 1 to m do
13:      I_i ← I ∪ {e_1, ..., e_{i−1}}
14:      X_i ← X ∪ {e_i}
15:      T_i ← STEINERIE(G, S, I_i, X_i, C)
16:      if T_i is a valid tree then
17:         Q.INSERT((I_i, X_i, T_i))
18:      end if
19:   end for
20: end while
21: return A
```

To obtain the k lowest-cost Steiner trees, where k is a predetermined constant, we use KBESTSTEINER (Algorithm 1), which uses the MIP STEINERIE as a subroutine. KBESTSTEINER is a simple variant of a previous top k answers algorithm [27, algorithm *DQFSearch*], which in turn generalizes a previous k-best answers algorithm for discrete optimization problems [29].

We are not the first to use lowest-cost Steiner trees to rank keyword query results, but we are the first to use the resulting rankings for learning. In addition, in the previous work [27], the graph represents actual data items and their associations, and the Steiner trees are possible answers containing given keywords. Since data graphs can be very large, the method is primarily of theoretical rather than practical interest. In our application, however, we work on much smaller schema graphs, and each tree corresponds to a whole query that may yield many answers, not a single answer.

4.4.3 K-Best Steiner Tree Approximation

As we show in Section 6, our Steiner formulation works for medium-scale schema graphs (around 100 nodes). To scale k-best inference to much larger schema graphs, we developed the following novel pruning heuristic.

Shortest Paths Complete Subgraph Heuristic (SPCSH) We explore using reductions [15, 39] to prune the schema graph to scale up KBESTSTEINER to larger schema graphs. SPCSH keeps only

the subgraph induced by the m shortest paths between each pair of nodes in S. The intuition for this is that there should be significant edge overlap between the k-best Steiner trees and the subgraph induced by the m-shortest paths, thereby providing good approximation to the original problem while reducing problem size significantly. SPCSH then computes the k-best Steiner trees by invoking KBESTSTEINER on the reduced subgraph.

Heuristic 2 SPCSH(G, S, C, k, m). **Input**: Schema graph G, keyword nodes S, edge cost function C, number of returned trees k, number of shortest paths to be used m. **Output**: List of at most k trees sorted by increasing cost.

```
L ← empty list
for all (u, v) ∈ S × S do
    P ← G.SHORTESTPATHS(u, v, C, m)
    L.APPEND(P)
end for
G_(S,C,m) ← G.SUBGRAPH(L)
return KBESTSTEINER(G_(S,C,m), S, C, k)
```

In SPCSH, G.SHORTESTPATHS(u, v, C, m) returns at most m shortest (least costly) paths between nodes u and v of G using C as the cost function. Efficient algorithms to solve this problem are known [42]. SPCSH is similar to the distance network heuristic (DNH) for Steiner tree problems on undirected graphs [38, 39], but there are crucial differences. First, DNH works on the set S-induced complete distance network in G while SPCSH uses a subgraph of G directly. Second, DNH uses a minimum spanning tree (MST) approximation while we use exact inference, implemented by KBESTSTEINER, on the reduced subgraph. Third, DNH considers only the shortest path for each vertex pair in $S \times S$, while SPCSH considers m shortest paths for each such vertex pair.

4.5 From Trees to Query Templates

The next task is to go from top k Steiner trees to a set of conjunctive queries, all outputting results in a common schema and returning *only* attributes in which the query author is interested. This is accomplished by first converting the top Steiner trees into conjunctive queries; then combining the set of conjunctive queries into a union view that produces a unified output relation; next, supporting user refinement of the view, e.g., to add projections; finally, naming and saving the view persistently with a Web form.

Converting Steiner trees to conjunctive queries. The task of generating conjunctive queries from Steiner trees is fairly straightforward. Each node in the Steiner tree typically represents a relation; traversing an edge requires a join. (In a few cases, a keyword may match on a tuple in, e.g., a topic index or ontology, and now the match represents a selection predicate.) In our implementation, the edges in our schema graph are annotated with the appropriate dereferencing information, typically foreign keys and keys. Here the query is formed by adding relations plus predicates relating keys with foreign keys. We require a query engine that supports queries over remote sources (such as the ORCHESTRA engine we use, described in Section 4.6), and we assume the existence of "wrappers" to abstract non-relational data into relational views.

Union view. The union of the queries derived from the top k Steiner trees form a single *union view*. Since each tree query may consist of source data from relations with different schemas, an important question is how to represent the schema for the union view. To create an initial version of the union view, we adopt a variation of the *outer union* (disjoint union) operator commonly used in relational-XML query processing [9]. Essentially, we "align" keys and attributes that have the same name, and pad each result with nulls where it does not have attributes.

View refinement. Next, allow the user to refine the view definition by adding projections, or aligning additional attributes from different source relations. This is done through an AJAX-based Web interface, which provides rapid feedback on how user selections affect the output. Projection and attribute alignment are achieved as follows. In a scrollable pane, we create a column for each keyword k_i. Then, for each conjunctive query in the view, we output a row in this pane, in which we populate each column i with the schema of the relation r_i that matches k_i. Each attribute in the schema is associated with a check box — unchecking the check box will project the attribute out from the view. Additionally, there is a text field through which the attribute can be renamed as it is output in the view. If two source attributes are renamed to the same name, then their output will be automatically aligned in the same output column.

Web form. The result of refinement is an intuitive Web-based form created from (and backed by) the view, as previously shown in Figure 1b. To reiterate, this form represents not one query but a *family* of queries, as it may be parameterized the the user. The query author will name and save the view and Web form, making it available for parameterization and execution.

4.6 Executing a Query

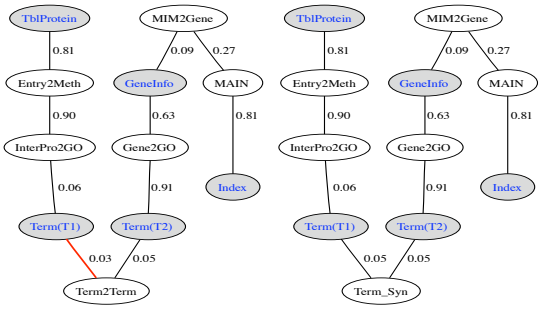
The user of a Web form (who may or may not be its creator) may retrieve the form via a bookmark, or by looking it up by its name and/or description. Each field in the Web form has a check box, which can be deselected to further project out information. The biologist may add selection predicates by filling in values in text boxes, or, for attributes with only a few values, by selecting from a drop-down list. Finally, alongside each item, there is a description of one or more sources from which the attribute is obtained — depending on space constraints — to help the biologist understand what the attribute actually means.

Query execution with provenance. Once the query is parameterized, the user will request its execution. Based on the query or queries that produced it, each tuple output by the query processor receives a *score*, which is the cost of the query that generated it. If a tuple is derived from multiple queries, it receives the lowest (*minimum-cost*) score. Rather than build our own query engine specifically for the Q System, we adopt the query processor used in the ORCHESTRA system [19].

When computing query results, ORCHESTRA also records their provenance in the form of a derivation graph, which can be traversed and retrieved. The same tuple may be derived from more than one query: hence in queries produced by the Q System, the provenance of a tuple is a tree-structured representation specifying which queries were applied to which source tuples, in order to derive the result tuple.

The existing ORCHESTRA system encodes provenance as a graph represented in relations, since it must support *recursive* queries whose provenance may be cyclic. Since all queries from the Q System are *tree-structured* and thus acyclic, we modified the query answering system to compute the provenance *in-line* with the query results: each tuple is annotated with a string-typed attribute containing the provenance tree expression, including the keys and names of the specific source tuples, and any special predicates applied (e.g., tests for similarity). This annotation adds only the overhead of casting attributes to strings and concatenating them to query processing — rather than materializing extra relations.

We note that, for situations in which all of the top k queries' cost



(a) Cost=4.56 (b) Cost=4.58

Figure 6: Re-ranked Steiner trees with costs updated as discussed in the text. The updated edge is thicker and red.

expressions are independent of tuple data, we can simplify even further, and simply tag each tuple with the ID of the query. However, for regularity across all answers, we use the previous scheme that encodes full details about the source tuples.

In our experience and that of our collaborators, the majority of bioinformatics queries have selective conditions, so we work under the assumption that any given query typically returns few answers. This has an important benefit in our context: it means that we can compute the entire set of answers satisfying the top queries — and as a result, compute the complete provenance for each tuple in terms of the queries. We need this complete information in order to provide proper feedback to the learning stages of the system, which we describe next.

5. LEARNING FROM FEEDBACK

Interaction with the Q System does not stop once query answers have been returned. Instead, the user is expected to provide feedback that helps the system learn which answers — thus, which queries and ultimately which *features* in the schema graph — are of greater relevance to the user.

The user provides feedback through the Results/Feedback Page, which shows query results in a table. When the user "mouses over" a tuple, Q provides a pop-up balloon showing the provenance of the tuple, in terms of the Steiner tree(s) that produced it; in many situations this is useful in helping the user understand how the tuple was created. The user may click on a button to tell our Q System that a given tuple should be *removed* from the answer set, another button instructing Q to move the tuple to the *top* of the results, or may input a number to indicate a new position this tuple should have in the output. In the cases we consider here, the cost (and thus rank) of a tuple is dependent solely on the query, and therefore the feedback applies to all tuples from the same query. (Recall that our *binning* process can divide tuples from the same relation into closely related groups, each of which corresponds to a different query, hence per-query feedback need not be coarse-grained.)

5.1 Basis of Edge Costs: Features

As we discussed previously, edge costs are based on features that allow the Q System to share throughout the graph what it learned from user feedback on a small number of queries. Such features may include the identity of nodes or edge end-nodes, or the overall quality of the match for an edge representing an approximate join. We now define features and their role in costs more precisely. Let the set of predefined features be $F = \{f_1, \ldots, f_M\}$. A feature maps edges to scalar values. In this paper, all feature values are binary, but in general they could be real numbers measuring some property of the edge. For each edge (i, j), we denote by $\mathbf{f}(i, j)$ the *feature vector* that specifies the values of all the features of the edge. Each feature f_m has a corresponding weight w_m. Informally, lower feature weights indicate stronger preference for the edges that have those features. Edge costs are then defined as follows:

$$C((i,j),\mathbf{w}) = \sum_m w_m \times f_m(i,j) = \mathbf{w} \cdot \mathbf{f}(i,j) \qquad (2)$$

where m ranges over the feature indices.

To understand features, weights, and the learning process, consider an example with the two Steiner trees in Figure 3, which correspond to queries CQ2 and CQ3 in Table 1. Their costs are derived from features such as the following, which test the identity of edge end-nodes:

$$f_8(i,j) = \begin{cases} 1 & \text{if } i = \textit{Term(T1)} \ \&\ j = \textit{Term2Term} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{25}(i,j) = \begin{cases} 1 & \text{if } i = \textit{Term(T1)} \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $w_8 = 0.06, w_{25} = 0.02$. Then the score of the edge (i = *Term(T1)*, j = *Term2Term*) in Figure 3(b)[1] would be $C(i,j) = w_8 \times f_8(i,j) + w_{25} \times f_{25}(i,j) = 0.08$.

Now suppose that, as mentioned in the previous section, the user is presented with tuples generated by the tree queries of Figures 3(a) and (b), annotated with provenance information. Since CQ2's tree has a lower cost than CQ3's tree, tuples generated by executing CQ2 are ranked higher. The difference between CQ2 and CQ3 is that while CQ2 uses the synonym relation (*Term_Syn*), CQ3 uses the ontology relation (*Term2Term*). Suppose that the user prefers tuples produced by CQ3 to those produced by CQ2. To make that happen, the learning algorithm would update weights to make the cost of the second tree lower than the cost of the first tree so that in a subsequent execution, tuples from the second tree are ranked higher. Setting $w_8 = 0.01, w_{25} = 0.02$ would achieve this, causing the two tree costs be as shown in Figure 6. Of course, the key questions are *which* weights to update, and by *how much*. We now discuss the actual learning algorithm.

5.2 Learning Algorithm

We use an *online* learning algorithm, that is, an algorithm that updates its weights after receiving each training example. Algorithm 3 is based on the Margin Infused Ranking Algorithm (MIRA) [12]. MIRA has been successfully applied to a number of learning problems on sequences, trees, and graphs, including dependency parsing in natural-language processing [32] and gene prediction in bioinformatics [3].

The weights are all zero as Algorithm 3 starts. After receiving feedback from the user on the r^{th} query S_r about a top answer, the algorithm computes the list B of the k lowest-cost Steiner trees using the current weights. The user feedback for interaction r is represented by the keyword nodes S_r and the *target tree* T_r that yielded the query answers most favored by the user. The algorithm then updates the weights so that the cost of each tree $T \in B$ is worse than the target tree T_r by a margin equal to the mismatch or *loss* $L(T_r, T)$ between the trees. If $T_r \in B$, because $L(T_r, T_r) = 0$, the corresponding constraint in the weight update is trivially satisfied. The update also requires that the cost of each edge be positive, since non-positive edge costs are not allowed in the Steiner MIP. An example loss function, which is used in the experiments reported in this paper, is the *symmetric loss*:

$$L(T,T') = |E(T) \setminus E(T')| + |E(T') \setminus E(T)| \qquad (3)$$

[1]For the sake of simplicity, we consider only simple paths here. However, the Q System is capable of handling arbitrary tree structures. This is an improvement over previous systems [6] that can handle path queries only.

Algorithm 3 ONLINELEARNER(G, U, k). **Input**: Schema graph G, user feedback stream U, required number of query trees k. **Output**: Updated costs of edges in G.

$\mathbf{w}^{(0)} \leftarrow \mathbf{0}$
$r = 0$
while U is not exhausted **do**
  $r = r + 1$
  $(S_r, T_r) = U.\text{NEXT}()$
  $C_{r-1}(i,j) = \mathbf{w}^{(r-1)} \cdot \mathbf{f}_{ij} \;\; \forall (i,j) \in E(G)$
  $B = \text{KBESTSTEINER}(G, S_r, C_{r-1}, K)$
  $\mathbf{w}^{(r)} = \arg\min_{\mathbf{w}} \left\| \mathbf{w} - \mathbf{w}^{(r-1)} \right\|$
  s.t. $C(T, \mathbf{w}) - C(T_r, \mathbf{w}) \geq L(T_r, T) \; \forall T \in B$
    $\mathbf{w} \cdot \mathbf{f}_{ij} > 0 \; \forall (i,j) \in E(G)$
end while
Let $C(i,j) = \mathbf{w}^{(r)} \cdot \mathbf{f}_{ij} \; \forall (i,j) \in E(G)$
Return C

The learning process proceeds in response to continued user feedback, and finally returns the resulting edge cost function.

The edge features used in the experiments of the next section are simply the identities of the source and target nodes, plus a single *default* feature that is on for all edges. The default feature weight serves as a cost offset that is automatically adjusted by Algorithm 3 to ensure that all edge costs are positive.

6. EXPERIMENTAL RESULTS

Our Q prototype consists of four primary components. The k-best Steiner tree algorithm uses the MOSEK 5.0 integer linear program solver, run on a dual-core Linux machine (2.6.18.8 kernel) with 12GB RAM. Query refinement is provided by a Java servlet running on Apache Tomcat 6.0.14. Query answering with data provenance is performed by the ORCHESTRA system [19], implemented in Java 6 and supported by IBM DB2 9.1 on a Windows 2003, Xeon 5150 server. (Note that we do not evaluate ORCHESTRA in this paper; only the quality of the query results). Finally, the machine learning component is also implemented in Java 6.

6.1 Experimental Roadmap

In this paper, we answer the following questions experimentally:

- Can the system start with default costs on all edges, and based on limited feedback over query answers, generalize the feedback to learn new rankings that enable it to produce "gold standard" (i.e., correct and complete according to expert opinion) queries? How many feedback iterations are necessary?
- How long is the response time (1) in processing feedback and generating new top-k queries, and (2) simply in generating top-k queries from the schema graph?
- How does our performance scale to large, real cross-database schema graphs?

We note that our evaluation focuses purely on the tasks of learning and generating queries. Our interest is not in measuring the response times of the query engine, which is orthogonal to this work. The Q System returns the top k queries in pipelined fashion, and most modern data integration query processors begin pipelining query answers as soon as they receive a query [10, 24]. We also do not duplicate the work of [6, 33] by performing user studies comparing with existing keyword search systems: that previous work already demonstrated that query answers need to be ranked by source authority/quality and not according to keyword search metrics like path length or term similarity.

Data Sets and Methodology

We conducted our experiments (except for the final experiment focusing on very large schema graphs) using data from a previous biomedical information integration system, BioGuide (`www.bioguide-project.net`) [6]. BioGuide is, to a significant extent, a baseline for comparison, as it provides ranked answers over a schema graph, given weights set by biological domain experts. The data that the BioGuide developers kindly supplied includes schema graphs with record linking tables between bioinformatics data sources, edge costs determined by a panel of experts based on reliability and completeness judgments, and expert queries. An example of an expert query is, "What are the related proteins and genes associated with the disease narcolepsy?" From such queries, a set of keywords on concepts can be easily extracted. These form our query workload.

Since BioGuide does not support the kind of learning from feedback we describe here, we used the BioGuide schema graph and set the expert-determined edge costs to create a "gold standard" against which to compare automatic learning. For a given query, the lowest-cost Steiner tree according to expert costs is taken to be what the simulated user prefers, and is used both as feedback in the learning process and as the gold standard for evaluation. Our goal in the near future is to work with our bioinformatics collaborators to deploy the Q System in real applications, and to conduct user studies in this context to confirm our preliminary results.

6.2 Learning against Expert Costs

We first investigate how quickly (in terms of feedback steps) the Q System can learn edge costs that yield the same query rankings as the gold standard obtained from expert-provided costs. Note that this is stronger than simply learning, based on feedback, which query the user prefers: our goal is to take feedback over a subset of the queries, and generalize that in a way that lets the system correctly predict which future queries are preferred.

We converted each expert query into a *query template* in which each keyword picks out a single table. For instance, for the narcolepsy query mentioned above, the template would be *"What are the related proteins (in [DB1]) and genes (in [DB2]) associated with disease Narcolepsy in [DB3]?"*. Here, *[proteins], [genes] and [disease]* are entities while *[DB1], [DB2]* and *[DB3]* are table names that need to be filled in. Using substring matching on table names, we populated these query templates by filling in the *[DB]* slots; each such instantiated template forms a query. For the experiments reported in this section, we generated 25 such queries and matched them over the BioGuide schema graph.

We created a feedback stream by randomly selecting a sequence in which the queries will be posed. For each such stream, we paired each query with the corresponding lowest-cost Steiner tree over our schema graph according to expert edge costs. We then applied Algorithm 3 to each stream, with the goal of learning the feature weightings that returned top query. At each point in the stream, our goal is to measure how well the top k algorithm's results for *all of the 25 queries* agree with the gold standard for those queries.

Thus, we simulate the interaction between the algorithm and a user who poses successive queries, examines their answers, supplies feedback about which is the best answer to this query, and moves on to the next query. However, to measure the quality of learning at each point, we need more than just the current query. We also need all the queries that could have been posed, both past and future ones, since the algorithm may change its weights in response to a later interaction in a way that hurts performance with previously submitted queries. The system behavior we aim for is that as this process continues, the queries preferred by the system

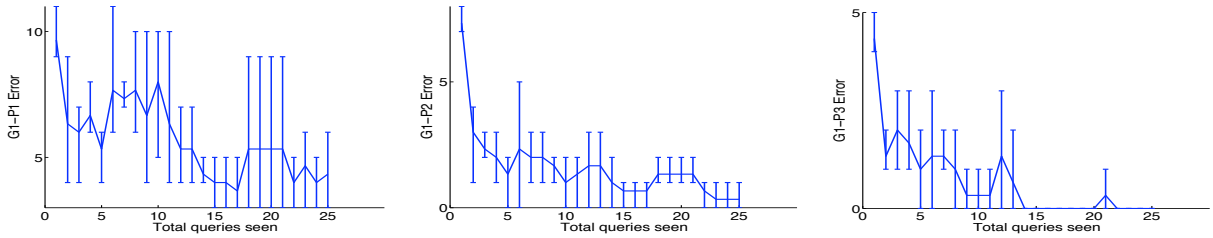


Figure 7: Learning curves of top k trees, $k = 1, 2, 3$ against gold standard as feedback is provided, with error bars showing best/worst performance, based on different feedback orders. There are 25 expert queries and the results are averaged over 3 random permutations of the queries.

will agree better with the user's preferences.

The results appear in Figure 7. For $k = 1, 2$ & 3, we plot the mean and min/max error bars (across the different random query-feedback stream sequences; note these are not confidence intervals) of how many of the 25 queries fail to have the gold standard tree within the top k trees computed with current weights. We conclude that the learning algorithm converges rapidly: that is, it quickly learns to predict the best Steiner tree consistent with the experts' opinion. After 10-15 query/feedback steps, the system is returning the best Steiner tree into one of the top three positions, and often the top position: Q begins to return the correct queries for *all* queries, given feedback on 40-60% of them.

6.3 Feedback and Query Response Time

Given that our goal is to provide an interactive query interface to the user, it is vital that our feedback process, as well as the creation of new top queries based on the feedback, be at a rate that is sufficient for interactivity.

To evaluate the feedback and query generation times, we fix the schema graph and measure (1) the time to process a "typical" user feedback given over a set of top answers, and (2) the time to create a set of queries based on that feedback. Assuming a search engine-like behavior pattern, the user will not look at answers that are beyond the first page of results; moreover, the user will only provide feedback on a few items. Hence, we measured the time taken to retrain and regenerate the set of top queries based on feedback. This took an average of 2.52 sec., which is easily an interactive rate.

A separate but related question is how quickly we can generate queries, given an existing set of weight assignments. Such a case occurs when a user retrieves an existing Web form and simply poses a query over the existing schema graph. We term this the *decoding time*, and Table 2 shows the total time it takes to generate the top 1, 5, 10, and 20 queries over the BioGuide schema graph (whose parameters are shown). In general, 5-10 queries should be sufficient to return enough answers for a single screenful of data — and these are returned in 2-4 seconds. Particularly since query generation and query processing can be pipelined, we conclude that response rates are sufficient for user interaction.

Test K	Graph (G) Size $(Nodes, Edges)$	Decoding Time (s)
1	(28, 96)	0.11
5	(28, 96)	2.00
10	(28, 96)	4.02
20	(28, 96)	8.32

Table 2: Average per-query decoding times (sec.) for requesting top-1 through -20 results over BioGuide schema.

6.4 Schema Graph Size Scalability

We have shown that the Q system scales well to increased user demand for answers. A second question is how well the system scales to larger schema graphs — a significant issue in the life sciences. Given that the Steiner tree problem is NP-hard, we will need to use our SPCSH algorithm (Sec. 4.4.3), but now the question is how well it performs (both in running time and precision.) To evaluate this, we used a different real-world schema graph based on mappings between the Genomics Unified Schema (www.gusdb.org), BioSQL (www.biosql.org), and relevant portions of Gene Ontology (www.geneontology.org). We call this the GUS-BioSQL-GO schema. The schema graph had 408 relations (nodes) and a total of 1366 edges. The edge weights were set randomly.

K	KBEST-STEINER (s)	SPCSH (s)	Speedup	Approx. Ratio (α)	Symm. Loss
1	1.2	0.1	12.0	1.0	0
2	43.8	3.0	14.6	1.0	0
3	111.8	5.5	20.3	1.0	0
5	1006.9	13.9	72.4	1.0	0

Table 3: Decoding times (sec.) of KBESTSTEINER and SPCSH with K ranging from 1 to 5, and m from 1 to 3. Also shown are the speedup factors, the approximation ratio, α, between the cost of SPCSH's and KBESTSTEINER's top predictions (α = 1.0 is optimal) and the symmetric loss (Equation 3) between the top predictions of the two methods. Results were averaged over 10 queries, each consisting of 3 keywords.

We use SPCSH to compute top-K Steiner trees on the GUS-BioSQL-GO schema graph. SPCSH is an approximate inference scheme, while KBESTSTEINER performs exact inference. Hence, the top prediction of KBESTSTEINER is always optimal. In Table 3, SPCSH's decoding time and inference quality is compared against KBESTSTEINER on the GUS-BioSQL-GO schema. Table 3 demonstrates that SPCSH computes the k-best Steiner trees (for various values of k) at a much faster rate than the previous method, while maintaining quality of prediction (α is 1.0). In fact, in our experiments, SPCSH's predictions were always optimal. We believe this demonstrates that our approach scales to large schema graphs without sacrificing result quality. We also reiterate that the time to generate the first query is of primary importance here: other queries can be pipelined to execution as they are produced.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed the problem of helping scientific users author queries over interrelated biological and other data sources, *without* having to understand the full complexity of the underlying schemas, source relationships, or query languages. In general, such scientists would like to rely on expert-provided *weightings* for data sources and associations, and use the best of these to guide their queries. However, experts are not always available to assess the data and associations, and moreover, the utility of a given association may depend heavily on the user's context and information need.

Our approach is based on matching keywords to a schema graph with weighted edges, allowing the user to refine the query, then providing answers with data provenance. As the user provides *feedback* about the quality of the answers, we *learn* new weightings for the edges in the graph (associations), which can be used to refine the query and any related future queries.

We have demonstrated that our approach balances the task of finding the top-k queries with the ability to *learn* new top queries based on feedback. The Q System learns weights that return the top answers rapidly — both in terms of number of interactions, as well as in the computation time required to process the interactions. Using real schemas and mappings, we have shown that we can use an exact top-k solution to handle schema graphs with dozens of nodes and hundreds of edges, and we can easily scale to hundreds of nodes and thousands of edges with our SPCSH approximation algorithm, which in all cases returned the *same* top answers as the exact algorithm.

We showed that our approach is highly promising for practical bioinformatics query authoring, and we believe the results extend to other domains as well. As future work, we hope to investigate approximation guarantees of SPCSH and look into other approximation algorithms for computing Steiner trees, to evaluate our system in real biological applications, and to more fully investigate settings where it is feasible to incorporate data-level keyword matching.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants IIS-0447972, 0513778, and 0713267. Thanks to the Penn Database Group, Ted Sandler, Ben Taskar and the anonymous reviewers for valuable feedback. Special thanks to Sarah Cohen-Boulakia for supplying the BioGuide schema, rankings, and expert queries.

8. REFERENCES

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.

[3] A. Bernal, K. Crammer, A. Hatzigeorgiou, and F. Pereira. Global discriminative training for higher-accuracy computational gene prediction. *PLoS Computational Biology*, 3, 2007.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[5] C. Botev and J. Shanmugasundaram. Context-sensitive keyword search and ranking for XML. In *WebDB*, 2005.

[6] S. C. Boulakia, O. Biton, S. B. Davidson, and C. Froidevaux. BioGuideSRS: querying multiple sources with a user-centric perspective. *Bioinformatics*, 23(10), 2007.

[7] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[8] M. J. Carey. BEA Liquid Data for WebLogic: XML-based enterprise information integration. In *ICDE*, 2004.

[9] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB '00*, 2000.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[11] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, 1998.

[12] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585, 2006.

[13] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[14] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[15] C. Duin and A. Volgenant. Reduction tests for the steiner problem in graphs. *Netw.*, 19, 1989.

[16] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1), 2007.

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

[18] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *WWW*, 2003.

[19] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[20] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[21] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[22] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *ICDE*, March 2003.

[23] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[24] Z. G. Ives, D. Florescu, M. T. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[25] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[26] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, 2008.

[27] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, 2006.

[28] J. C. Kissinger, B. P. Brunk, J. Crabtree, M. J. Fraunholz, B. Gajria, A. J. Milgram, D. S. Pearson, J. Schug, A. Bahl, S. J. Diskin, H. Ginsburg, G. R. Grant, D. Gupta, P. Labo, L. Li, M. D. Mailman, S. K. McWeeney, P. Whetzel, C. J. Stoeckert, Jr., and D. S. Roos. The Plasmodium genome database: Designing and mining a eukaryotic genomics resource. *Nature*, 419, 2002.

[29] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18, 1972.

[30] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[31] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2), 2004.

[32] R. McDonald and F. Pereira. Online learning of approximate dependency parsing algorithms. In *European Association for Computational Linguistics*, 2006.

[33] P. Mork, R. Shaker, A. Halevy, and P. Tarczy-Hornoch. PQL: A declarative query language over dynamic biological schemata. In *AMIA Symposium*, November 2002.

[34] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI '03*, 2003.

[35] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4), 2001.

[36] R. Tuchinda and C. A. Knoblock. Building mashups by example. In *IUI '08*, 2008.

[37] R. Varadarajan, V. Hristidis, and L. Raschid. Explaining and reformulating authority flow queries. In *ICDE*, 2008.

[38] P. Winter. Steiner problem in networks: a survey. *Netw.*, 17(2), 1987.

[39] P. Winter and J. M. Smith. Path-distance heuristics for the steiner problem in undirected networks. *Algorithmica*, 7(2&3):309–327, 1992.

[40] L. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

[41] R. T. Wong. A dual ascent approach for steiner tree problems on a directed graph. *Mathematical Programming*, 28(3):271–287, October 1981.

[42] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 18(17), 1971.

Appendix: Q System Screenshots

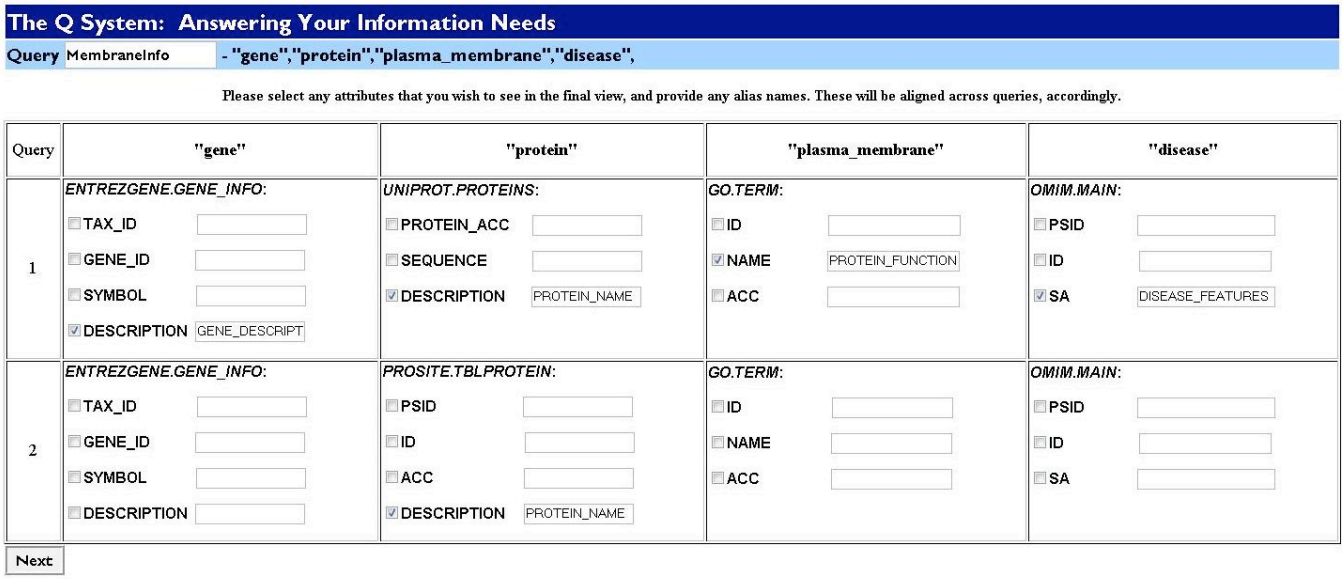


Figure 8: Query Refinement Interface

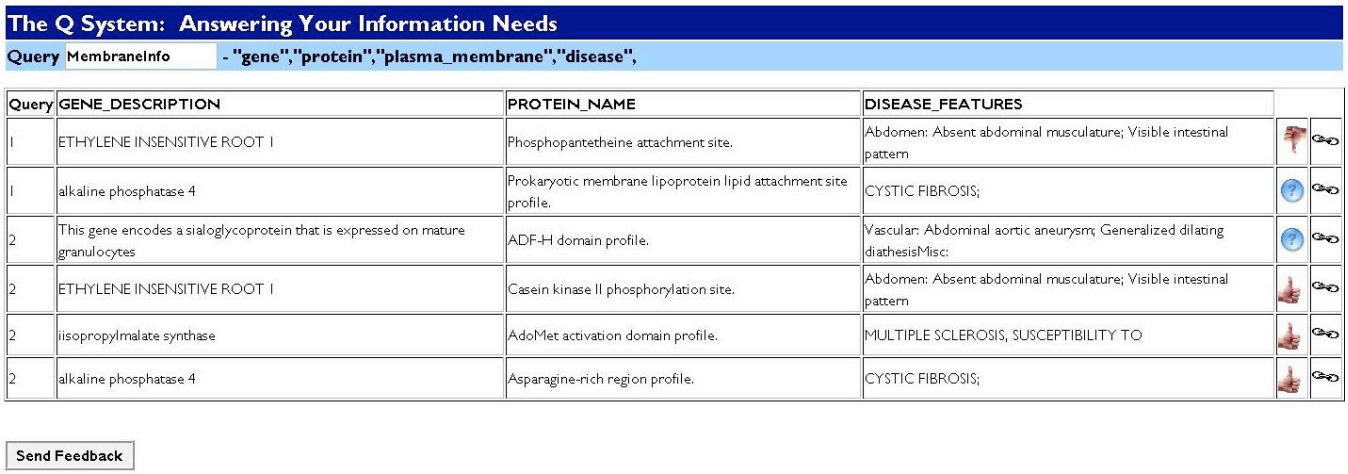


Figure 9: User Feedback Interface

Figures 8 (the query refinement interface that allows for final modifications to the view before it is converted to a Web form) and 9 (the feedback interface after query answers are returned) are included to give a better sense of the user interface and interaction.