PQL: A Declarative Query Language over Dynamic Biological Schemata

P. Mork, M.S.[1], R. Shaker [2], A. Halevy, Ph.D.[1], P. Tarczy-Hornoch, M.D.[2,3]
[1]Computer Science & Engineering, [2]Pediatrics and [3]Biomedical & Health Informatics
University of Washington, Seattle, WA

We introduce the PQL query language (PQL) used in the GeneSeek genetic data integration project. PQL incorporates many features of query languages for semi-structured data. To this we add the ability to express metadata constraints like intended semantics and database curation approach. These constraints guide the dynamic generation of potential query plans. This allows a single query to remain relevant even in the presence of source and mediated schemas that are continually evolving, as is often the case in data integration.

1. INTRODUCTION

The benefits of data integration techniques in simplifying querying across heterogeneous databases are significant. The key strategy in data integration is to map the underlying sources to a common model or ontology, known as the *mediated schema*[1]. Advantages of data integration relevant to online biomedical data include a consolidated view of the underlying data and a single point of entry[2]. Most importantly, no one database contains complete information. Many queries can only be answered by integrating overlapping results from multiple sources.

The techniques for designing data integration systems are relatively well understood: A user query is formulated against the mediated schema, usually in a declarative query language like SQL. Using knowledge contained in a *source catalog*, a *reformulator* converts this query into a query plan over the actual sources. A *query execution engine* optimizes this plan, retrieving data from the underlying sources. This retrieval is done via a *meta-wrapper*[3], which is responsible for converting to and from the source-specific syntax to a common syntax. This paper focuses on the interaction between the reformulator and the source catalog.

One significant advantage of a mediated schema is that it provides a level of indirection. Application developers do not need to interact with a collection of heterogeneous sources; instead they describe what data they want the system to retrieve. The integration system maps these queries to the appropriate sources.

This approach works well in many domains. However, one important limitation is that complex relationships (a.k.a. paths) must be specified precisely. For example, assuming that one wanted to find all proteins closely related to a given disease, a traditional data integration system would allow you to retrieve all of the proteins that cause a given disease and combine that with the result of retrieving the proteins coded for by genes related to that disease.

In a complex or rapidly evolving domain (like genetics), this limitation becomes a bottleneck. There are numerous possible ways to answer the sample query. In addition, sources (and paths dependent on them) may come and go as their schemas evolve. Ideally, we would like to be able to express the types of paths that are valid, without enumerating them explicitly.

We propose a new language, PQL (pronounced pickle), which generalizes StruQL[4], a query language (QL) for semi-structured data (such as XML[5]) that allows one to construct arbitrary paths over the relationships in a single document. We extend this feature to paths involving multiple sources by allowing the user to express constraints about how paths can be formed (e.g., a metadata constraint limiting the relationships to those which are directly causal).

The contributions made by this paper are:

- First, we identify a class of queries that cannot be answered by any current declarative QL.
- Then, we propose PQL a high-level QL capable of answering these queries; this language is built on an existing QL for semi-structured data.
- Finally, we describe one possible implementation of PQL, which is one component of the GeneSeek architecture.

The rest of this paper is organized as follows. In Section 2 we provide an overview of the GeneSeek architecture and related work, identifying the limitations of current query languages. Section 3 describes StruQL, which serves as the basis for our work. In Section 4 we introduce PQL, a language that addresses the limitations in Section 3. In Section 5 we describe how we implemented PQL. In Section 6 we discuss our approach. Section 7 concludes.

2. SYSTEM OVERVIEW

GeneSeek[2] provides a single interface to a collection of online genetic databases distributed across the Internet. Live data is guaranteed by leaving all of the data at the sources, retrieving only what is necessary to answer a given query. The architecture that supports these features is displayed in figure 1; a brief summary follows.

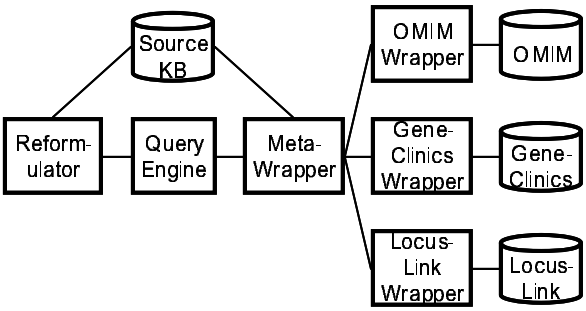

Figure 1 (PIP Architecture): Queries flow from left to right and results from right to left.

The source knowledge base (SKB) contains the mediated schema, against which the user poses queries. (See figure 2 for a simple sample mediated schema.) The mediated schema contains the entities and relationships in the domain (e.g., proteins and genes). The SKB also describes how the underlying sources map to the mediated schema. This tightly couples the mediated schema to the source descriptions. Finally, the SKB contains metadata about entities and relationships used for specifying more complex constraints.

The user query is passed to a reformulator. Using source descriptions[6] that describe what can be retrieved from which sources, the reformulator determines all of the ways in which the user query can be answered. This plan is passed to a query execution engine: Tukwila[7] in our case. Note that the execution engine knows nothing about the mediated schema or source capabilities. This facilitates swapping one engine for another.

The query execution engine retrieves data from the meta-wrapper[3], via a URL. This URL indicates which entities should be retrieved from which sources. The meta-wrapper is a single, re-usable component capable of translating from the mediated namespace to the source name-space, using information contained in the SKB. The meta-wrapper passes the request to the appropriate wrapper, whose only responsibility is to retrieve data from a specific source and return valid XML.

The meta-wrapper translates these results back into the mediated name-space, passing the results to the query execution engine. The engine processes the data based on the user query, ultimately returning results to the reformulator. At this stage, the reformulator can attempt to match similar results, or pass the results back to the user.

We refer to this process as the PIP Integration Pipeline (or PIP). Data requests flow in one direction, and results flow in the other direction. These results are returned to the user as XML, the basic entities of which will adhere to the mediated schema.

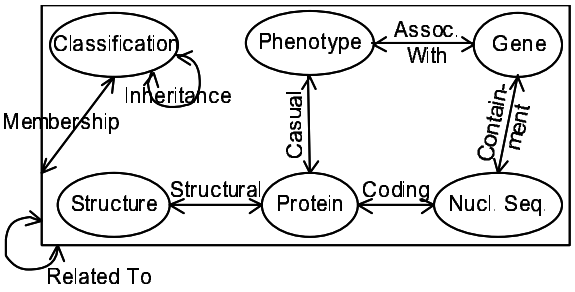

Figure 2 (Mediated Schema): Ovals indicate leaf concepts and arrows potential relationships. Note that "related to" can relate any entity to any other; "membership" relates a classification (set) to its elements.

This paper focuses on the way in which queries are posed and answered. A user query must be able to reference entities and relationships in the mediated schema; any QL for XML will suffice in this regard. In addition, the user query needs to be able to treat the network of interrelationships among the underlying sources as data, over which paths will be constructed dynamically.

There are several query languages already defined for XML. A prominent example is XQuery[8], which has been accepted as a de facto standard and is used by Tukwila. Other examples include XPath[9], Lorel[10], XML-QL[11] and XSL[12]. These languages share two important features. First, paths that span multiple data sources must be expressed explicitly. Second, the languages support some form of regular expressions (e.g., follow any number of "subclass" relationships). It is generally possible to bind a variable to the result of following a path, but not to the path itself.

As we will show, PQL generalizes these capabilities by allowing the query author to specify constraints that determine which paths are valid using metadata about the relationships (including name, curation, etc.). Before discussing the specifics of PQL in section 4, we introduce the basics of StruQL, which we have chosen for its simple syntax and use of edge variables (i.e., StruQL allows one to bind variables to edges or paths).

3. STRUQL

StruQL[4] is a declarative query language originally designed for web-site management. It can be used to transform any graph (like the semantic network model in the GeneSeek project) into any other. The nodes in the graph correspond to entities (e.g., proteins or genes) and the edges correspond to relationships between these entities (e.g., "caused by").

The basic StruQL syntax is:

```
WHERE Class1(X), Class2(Y), ...
      X->Path->Y, ...
```

```
CREATE Root(), Node(X,Y), ...
LINK Root()->"Link"->Node(X,Y), ...
```

The `WHERE` clause is used to select some sub-graph. There are two possible constraints in this clause. Class restrictions are used to limit the types of nodes, using standard inheritance semantics. Path expressions are used to indicate relationships that must hold between the nodes. These paths can include any regular expression operators, including optional relationships (?), alternation (|), concatenation (.) and closure (*). This latter operator allows for the construction of paths of arbitrary lengths.

The `CREATE` clause is used to dynamically construct new nodes. Each class named in the clause corresponds to a Skolem function, which has the property that exactly one new node is constructed for every possible binding of the variables listed as arguments to the function. For example, `Root()` will return the same single node every time it is called because the parameters will always be the same, whereas `Node(X,Y)` will return a different value for each X and Y pair obeying the constraints in the `WHERE` clause.

Finally, the `LINK` clause is used to wire both old and new nodes together. The nodes indicated as the head and tail are related to one another using a labeled edge.

Limitations: The query must explicitly enumerate the relationships of interest. The closure operator provides a certain degree of flexibility, but the expression being closed must be precisely specified. Also, there is no mechanism for binding a path to a variable (although StruQL does allow a single labeled edge to be bound to a variable). The next section describes how PQL addresses these limitations and provides an illustrative example.

4. PQL

The PQL query language addresses the aforementioned limitations by allowing the query author to express assumptions that guide the construction of complex paths. Instead of manually enumerating all possible paths of interest, the query contains a collection of rules that are used to instantiate paths that adhere to the rules. For example, a common rule is that causality is transitive. This is expressed by adding a rule that states that the concatenation of two causal paths is itself causal.

Rules are expressed in a USING clause that precedes the StruQL query. These rules define a context-free grammar against which the contents of the SKB can be compared. Rules have the following form:

```
USING
X.Y!wellCurated == TRUE :-
   Y!wellCurated == TRUE;
Z!wellCurated == TRUE :-
   Z!curation == "human",
   Z!validation == "external"

WHERE
Phenotype(Ph), Protein(Pr),
Ph->Pa->Pr, Pa!wellCurated == TRUE,
Ph->"name"->"Cystic Fibrosis"

CREATE
PathProteinPair(Pa,Pr)

LINK
X->"causedBy"->PathProteinPair(Pa,Pr),
PathProteinPair(Pa,Pr)->"source"->Pa,
PathProteinPair(Pa, Pr)->"protein"->Pr
```

Figure 3 (Sample Query): This query retrieves all proteins related to cystic fibrosis by traversing a well-curated path.

```
<PATH-LIST>!<ATTR> <OP> <VALUE> :-
   (<PATH-VAR>!<ATTR> <OP> <VALUE>)*
```

The `<PATH-LIST>` contains a list of path variables. One sub-path must be bound to each path variable mentioned. The concatenation of these sub-paths is defined (by the rule) to have the property indicated by `<OP> <VALUE>`, where `<OP>` is some binary relation (e.g., equals or less than) and `<VALUE>` is any value, often true or false.

For example, `X.Y!isCausal == TRUE`, indicates that the concatenation of paths `X` and `Y` is causal. Note that the attributes (`<ATTR>`) can be any attribute defined in the SKB, or in another rule.

Following the ':-' token is a series of sub-path clauses that must be true to use the rule. Each `<PATH-VAR>` corresponds to one of the variables in the `<PATH-LIST>`. The sub-path bound to that variable must obey the `<OP> <VALUE>` relationship, or the rule cannot be used. For example, the previous rule might contain the following two clauses:

```
X!isCausal==TRUE, Y!isCausal==TRUE
```

This complete rule now states that two causal paths can be concatenated to produce a longer causal path. Note that these rules apply recursively, so it is possible to generate causal paths of any length.

As a simple illustration of how all of the pieces of PQL fit together, consider the query in figure 3. The `WHERE` clause indicates that we are interested in the relationships between phenotypes (variable `Ph`) and proteins (variable `Pr`), both of which are entities defined in the mediated schema. The query then specifies that there must exist a well-curated path between these entities and that the name of the phenotype must be "cystic fibrosis." The variable `Pa` is bound to each such path.

```
<Phenotype xid="@1">
  <name>Cystic Fibrosis</name>
  <causedBy xref="@2"/>
  <causedBy xref="@3"/>
</Phenotype>
<PathProteinPair xid="@2">
  <source xref="@4"/>
  <protein xref="@6"/>
</PathProteinPair>
<PathProteinPair xid="@3">
  <source xref="@5"/>
  <protein xref="@6"/>
</PathProteinPair>
<Path xid="@4">
  <edge>GeneTestsInternal</edge>
</Path>
<Path xid="@5">
  <edge>OMIM->LocusLink</edge>
  <edge>LocusLink(RefSeq)</edge>
</Path>
<Protein xid="@6">
  <name>CF transmembrane regulator</name>
</Protein>
```

Figure 4 (Sample Results): There are (at least) two possible ways to answer the sample query, but only a single protein.

The CREATE and LINK clauses create a new node for each path and protein pair (note that a protein may be reachable by many paths and that a single path may lead to many proteins). The node corresponding to cystic fibrosis will be linked to all such pairs, which in turn are linked to their constituent pieces.

Figure 4 demonstrates (in XML) how these results are constructed. Note that there is only a single protein entity, even though multiple paths point to the same protein.

Finally, the USING clause defines how well-curated paths are constructed. The first clause establishes a base case; any single edge that is both human curated and externally validated will be considered well-curated. In addition, any path that ends in a well-curated sub-path is defined to be well-curated. In other words, the final edge of any well-curated path is well-curated. (The intuition behind this is that the last edge will prune irrelevant results.)

Sample single edges that meet these criteria include internal the GeneClinics[13] relationship between phenotype and gene and the RefSeq[14] relationship in LocusLink[15], which relates genes to proteins. Given the definition of well-curated, any path that ends with a RefSeq pointer will be considered well-curated.

5. IMPLEMENTATION

The USING clause defines a context-free grammar. The task of the reformulator is to enumerate all paths accepted by the grammar. The basic strategy is to dynamically construct a pushdown automaton.

At a high level, the reformulator performs a depth-first traversal over the contents of the SKB. At each step it tests the current path for membership in any of the rules defined in the USING clause. Whenever a valid path is found (resolved), that path is returned, and the traversal continues.

Testing for membership at each step is a potentially expensive operation. Instead, several pushdown automata are dynamically traversed during processing. Maintaining a collection of currently active rule facilitates this process. Each rule in the set is marked with a current position. Note that a rule can appear multiple times with different positions.

The process starts by adding one copy of each rule, in position 0. For example, the query in figure 3 begins with {*XY, *Z} as the active rule set. As each edge is traversed, each rule in the active set is tested to see if the edge matches the next path. If so, the position marker is moved. If not, the rule is removed from the active set. Finally, the active set is reseeded with a new copy of each rule.

For example, assume that three edges are traversed. The first two do not match the well-curated definition, but the third does. The following is the sequence of active rule sets:

Start: {*XY, *Z}

First edge: {X*Y}→{X*Y, *XY, *Z}

Second edge: {X*Y}→{X*Y, *XY, *Z}

Third edge: {XY*, Z*}

At this point, it is possible to conclude that this path is well-curated. Note that some additional bookkeeping is required with respect to singleton rules (rules that define paths of length 1) to avoid erroneously concluding that resolution of the singleton applies to an entire path.

A final filter is necessary to discard paths that rely on capabilities not supported by the actual sources. Most sources will not service requests for the entire database. Instead, queries must provide some initial information (e.g., the name of a gene or phenotype). A path cannot be used if the database at the head of the path requires information not present in the query. In the sample query any database willing to retrieve phenotype information by name can serve as a head.

In addition, the query may contain restrictions on the entity at the tail of the path. If so, the database at the tail of the path must be able to provide the information needed to resolve those restrictions. For example, if the query asked for only those proteins containing a certain amino acid sequence, there is no sense retrieving data from a database that does not contain sequence information.

The reformulator is responsible for discarding useless paths. The information needed to make this decision is contained in the SKB.

6. DISCUSSION

We have implemented the path enumeration algorithm in Java, using JavaCUP[16] and JLex[17] for query parsing and the Protégé-2000[18] API to retrieve information from the SKB. The enumeration process is rapid, assuming a limited number of valid paths (obviously a rule that returns all possible paths incurs a certain amount of overhead).

The advantages of path enumeration are compelling. The reformulation process frees the query author from needing to know the myriad of ways in which a query might be answered. Moreover, queries become more resilient to changes in the underlying sources and schema. When a new entity is added to the mediated schema, old queries will automatically incorporate relationships involving the new entity as a possible plan.

The separation made between query authoring and path enumeration introduces the possibility of informing the query author of the results of the path enumeration. This may be important if the user has provided a collection of assumptions that are easy to fulfill. For example, a query similar to the one in figure 3 that assumes that causality is transitive and requests the proteins causally related to some phenotype results in 41 separate paths. Evaluating all 41 of these queries may be too time-consuming.

We considered various ranking schemes for paths[2], but determined that such an approach forced users to accept our definition of "good" paths. We have adopted PQL as a mechanism to allow the query author to define his own definition of "good" paths. We also rejected solutions based on probabilistic weightings as too cumbersome and unintuitive.

We have been able to express many queries of interest using PQL, particularly with respect to the curation of the GeneClinics database. We must now seek additional users to determine if PQL meets their needs as well. In all likelihood this will also involve expanding and refining the mediated schema and the developing wrappers to new sources.

7. CONCLUSIONS

The benefits of data integration systems are numerous. These systems provide a user with a consolidated view of several heterogeneous, autonomous sources. The interface to these systems is usually a declarative query language for semi-structured data.

Traditional query languages for semi-structured data lack several features that are important in a complex, evolving domain, like genetics. We have developed the PQL query language to address a number of these limitations.

In particular, PQL allows the query author to express high-level constraints governing the kinds of relationship paths he is willing to let the system make on his behalf. A reformulator takes the query and enumerates all query plans that pass the given constraints. These query plans are passed to a query execution engine, which returns results integrated from several source databases.

We have implemented the path enumeration algorithm, which pulls relevant meta-data from a source knowledge base. The enumeration module can thereby be reused in any number of integration projects by providing a new SKB.

ACKNOWLEDGEMENTS

We would like to thank Matt Barclay for programming support and Dan Suciu for his expertise with StruQL. Joint funding was provided by NHGRI and NLM (1R01HG02288).

REFERENCES

1: Wiederhold G. Intelligent integration of information. Proceedings of the ACM SIGMOD Conference on Management of Data; 1993 May 26–28, Washington, DC, USA; 22(2): 434–437.
2: Mork P, Halevy A, Tarczy-Hornoch P. A model for data integration systems of biomedical data applied to online genetic databases. Proceedings of AMIA Annual Symposium; 2001 Nov 3–7, Washington, DC, USA: 473–477.
3: Shaker R, Mork P, Barclay M, Tarczy-Hornoch P. A rule driven bidirectional translation system remapping queries and result sets between a mediated schema and heterogeneous data sources. Submitted to: AMIA Annual Symposium; 2002.
4: Fernandez M, Florescu D, Levy A, Suciu D. A query language for a web-site management system. SIGMOD Record; 1997 Sep; 26(3): 4–11.
5: http://www.w3.org/XML/
6: Levy A, Rajaraman A, Ordille J. Querying heterogeneous information sources using source descriptions. Proceedings of the 22nd VLDB Conference; 1996 Sep 3–6; Bombay, India: 251–262.
7: Ives Z, Florescu D, Friedman M, Levy A, Weld D. An adaptive query exection system for data integration. Proceedings of the ACM SIGMOD Conference on Management of Data; 1999 May 31–Jun 3; Philadelphia, PA, USA; 28(2): 299–310.
8: http://www.w3.org/TR/xquery/
9: http://www.s3.org/TR/xpath/
10: Abiteboul S, Quass D, McHugh J, Widom J, Wiener J. The Lorel query language for semistructured data. International Journal on Digital Libraries; 1997; 1(1): 68–88.
11: http://www.w3.org/TR/NOTE-xml-ql/
12: http://www.w3.org/Style/XSL/
13: http://www.geneclinics.org/
14: Pruitt K, Maglott D. RefSeq and LocusLink: NCBI gene-centered resources. Nucleic Acids Research; 2001; 29(1): 137–140.
15: http://www.ncbi.nlm.nih.gov/LocusLink/
16: http://www.cs.princeton.edu/~appel/modern/java/CUP/
17: http://www.cs.princeton.edu/~appel/modern/java/JLex/
18: http://protege.stanford.edu/index.shtml