XGI: A Graphical Interface for XQuery Creation and XML Schema Visualization

Xiang Li

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2006

Program Authorized to Offer Degree:
Department of Biomedical and Health Informatics

University of Washington

Graduate School

This is to certify that I have examined this copy of a master's thesis by

Xiang Li

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____
James F. Brinkley

_____
John Gennari

Date: _____

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature_____

Date_____

University of Washington

Abstract

XGI: A Graphical Interface for XQuery Creation
and XML Schema Visualization

Xiang Li

Chair of the Supervisory Committee:
Research Professor James F. Brinkley
Department of Biological Structure

XML (Extensible Markup Language) is used in many contexts of modern information technology to facilitate sharing of information between heterogeneous data sources and inter-platform applications. The prevalence of XML implementation in data storage and exchange necessitates a method to adequately query XML data. The World Wide Web Consortium (W3C) is proposing XQuery as the standard querying language for semistructured XML data. XQuery is designed for experienced database programmers, since its syntax and capabilities are analogous to the SQL relational query language. Therefore, the inherent complexity of formulating XQuery statements makes it an intimidating task for anyone, except an expert in the XQuery language, to construct queries.

The development of XQuery Graphical Interface (XGI), a visual interface for creating XQuery in a graphical format, is motivated by the need to simplify the query formation for unskilled users and speed up the query construction for expert users. The implementation of XGI is mainly inspired by three existing systems: Query and Reporting Semistructured Data (QURSED), XQuery By Example (XQBE), and XBrain. A review of these systems and many other systems has helped us understand the benefits and drawbacks of various system design approaches, and has assisted us in identifying a set of features for XGI that will successfully reduce the complexity of creating queries in the XQuery language. XGI provides a web interface for users to explore their own XML source data schema, search for specific schema elements, and visually create queries in

the XQuery language for the targeted XML data source.

A validation of the XGI system has verified its ability to efficiently and accurately create queries for various XML data sources. From the validation, we have recognized some strengths and weaknesses of the XGI system compared to other systems. We also recommend several areas in which XGI can be improved.

Table of Contents

Page

List of Figures

Figure Number Page

List of Tables

Chapter 1: Introduction

1.1 Statement of problem

The nature of modern biomedical research necessitates efficient methods for managing large sets of research data, collaborating among multiple research groups, and synthesizing results from data of various research trials and experiments[2,45-46]. However, heterogeneous data formats used by different research groups often impede the integration of research results[24-25]. Therefore, XML is becoming the de facto standard language for data exchange and representation in biomedical research[24]. Due to its self-describing nature, XML is a simple, powerful, and intuitive language for biomedical researchers to use to manage, share, and analyze research data[20]. XML has a strong advantage over the relational data model because XML can model semistructured data[29], i.e. data whose structure is flexible and is characterized by nesting, option fields, and high variability[3]. Many research groups have adopted XML to model their research data[23] and there are many ways to query those semistructured data.

The W3C (World Wide Web Consortium) recommends two textual methods, XSLT and XQuery, to transform and query semistructured XML data sources[43,47-48]. XQuery is gaining increasing popularity among informaticists in biomedical research because it is powerful and well-supported by the software development community. Informaticists with strong SQL backgrounds should feel comfortable creating XQuery queries, because the “programming” skills required for developing queries in XQuery and SQL are comparatively similar[3]. Unlike SQL, XQuery is a turing-complete programming

language[27]. However, the number of biomedical researchers who are experienced in SQL and/or XQuery is small compared to the vast number of biomedical research groups that have XML data. Additionally, writing complex XQuery statements is cumbersome and inefficient for informaticists who might not have an explicit knowledge of the source data schema and nuances of the XQuery language.

We have recognized the need for a more intuitive way for inexperienced biomedical researchers to query and integrate XML data sources using XQuery queries. We believe that tools to generate XQuery queries through a well-designed graphical interface will allow biomedical researchers to query XML data sources through XQuery more easily and efficiently. Therefore, we have developed the XGI system, a graphical tool for generating XQuery expressions, to help biomedical researchers create queries for a wide range of XML data sources. Our implementation of the XGI system is inspired by various existing informatics systems. We use an integrative approach that combines the best features from these existing graphical query systems as we perceive them.

1.2 Overview of approach

Our goal for the XGI system is to provide a graphical interface to generate accurate XQuery queries. In this section, we describe the approach we took to develop the XGI system. In our validation of XGI, we examine how well we follow this approach and how well the system actually meets our goal.

To reduce the costs of installing and maintaining software packages that invariably come with any application, XGI is designed to use a distributed resource model. With this model, only one XGI installation and set up needs to be shared among multiple users. Only a few informatics personnel will be required to manage this centralized system, thereby reducing the need for every user to manage his/her installation. XGI also utilizes the Asynchronous JavaScript and XML (AJAX) paradigm to deliver our tool as an interactive online application. The AJAX approach enables our application to be cross-platform and to behave similarly to rich client applications[21,50].

Because one XGI installation allows multiple users to simultaneously access the system, we believe time will be saved from having to install, configure, and maintain a dedicated installation for each user. Since the system only requires a typical web browser to access, most users will be able to access and utilize XGI without having to install any client software. Our implementation of XGI provides a set of features we believe can be useful for users to create specific XQuery queries. These features are compiled from our knowledge of existing graphical querying tools and our past experience building the graphical interface for XQuery. XGI also provides a look similar to rich client software that is familiar to most computer users. By eschewing the text, form, and button feel of traditional web sites, XGI's AJAX-enabled interface allows users to interact with the system in a more dynamic fashion.

1.3 Contributions of this thesis

This thesis first introduces the reasons for biomedical researchers to use XML and the need for providing tools that can increase the efficiency and accuracy in which XML queries (in the XQuery language) can be generated. This thesis discusses several of the informatics approaches in developing visual XQuery composition tools and reviews some of the existing systems that utilize these approaches. We also introduce a set of features for a visual XQuery system that we believe incorporates different approaches from the review of existing systems and our own personal experiences. This thesis also presents the XGI system's architecture and its approach to implement most of these features. We also demonstrate how a researcher can use the XGI system's features to generate XQuery statements for any XML data source.

An initial validation of XGI focuses on testing whether it can generate accurate XQuery statements for several XML data sources. To test the XGI system, we used XGI to recreate XQuery queries we have previously written for the Structural Informatics Group's XBrain application. Our validation examines XGI's ability to recreate all the previously written queries using its interface by comparing the queries from XBrain with the XGI-generated queries. Since XGI is designed to be an extensible system that can handle arbitrary XML data schema, we also tried to recreate queries that have been written for other XML data sources to help validate the generalizability of the XGI system.

Some of the system features were not implemented in the XGI system, suggesting areas in which our system can be further enhanced. Other system limitations and restrictions in the visual query paradigm lead us to conclude that a graphical query interface is not suitable for very advanced and complex queries, perhaps limiting XGI's usefulness. This thesis not only demonstrates the potential of a graphical query interface for generating queries in the XQuery language, but also generates ideas for future work on our system and for the field of biomedical informatics in general.

Chapter 2: Background

2.1 Existing systems: benefits and drawbacks

The amount of XML data sources in the biomedical research domain exceeds the number of qualified informaticists who are familiar with XML query language by a considerable margin[23,38,49]. Therefore, there is a growing need for tools, such as graphical query interfaces, that can help inexperienced users create simple and accurate queries for any arbitrary XML data source. Query By Example (QBE) was the first graphical query language that enabled users to query and modify data sources without having to learn all the complexities of the underlying query language[39]. The principle of QBE is to let users create queries as extensions of their thought processes, so the query formulation process is both quick and intuitive[51]. Even though QBE focuses on building the graphical query interface for the relational data model, its principle is applicable to the semistructured data model as well. The QBE paradigm shows that a graphical user interface abstraction layer built on top of a query language allows users with negligible knowledge of the underlying source data schema and the query language to sufficiently design and execute complex queries[39]. A useful graphical query tool for semistructured data should simplify and expedite the creation of XML queries.

Many informaticists have attempted to apply the QBE paradigm to semistructured data sources. Several informatics tools already exist to assist users in formulating queries for XML data sources through graphical interfaces. Many of these tools use divergent approaches, both structured and unstructured query approaches, toward designing and

implementing the graphical query interface for XML. We have examined these various tools and have detailed many of the benefits and limitations of each system as we perceive them. From exploring these options, we have identified a set of requirements for a graphical query system that we believe will help inexperienced users create XML queries more effectively. These requirements are the goals that direct us in our approach and design of XGI. In this section, we first classify the different interface design approaches into several general categories and elucidate a few representative informatics tools from each category. Then, we introduce the set of requirements we perceived as necessary for implementing a graphical query interface for semistructured data.

2.1.1 Structured query approach

Structured query approach, a type of graphical query interface design approach, is defined by the lack of arbitrary, hierarchical structures in the XML-querying results. Many of the informatics tools that implement this type of design approach are often a part of a larger information retrieval and management system. The main purpose of these tools is to provide to the users of the larger system a quick and uncomplicated, thus unsophisticated method of retrieving data from the source XML documents. These tools usually present the source data schema to users so they can select elements to query. Users have the option to choose which elements from the source schema to be included in the query output but have limited or no decision on how those elements will be displayed. The output format is usually pre-defined either by the system or a formatting template before users even create the query schema. Also, these tools generally do not expose

users to the underlying querying language.

2.1.1.1 QSByE

Querying Semistructured Data By Example (QSByE), an interface for querying semistructured data, was developed at Federal University of Minas Gerais[18]. QSByE is the visual interface component of Data Extraction By Example (DEByE), a system that is designed to extract information from unstructured data sources on the Web using examples provided by users. QSByE utilizes a QBE-like querying language to query its XML-based data sources, which are called DEByE Textual Object Repository (DTOR). QSByE implements many concepts from QBE, such as type coercion and path expression. The Query Specification Module of QSByE displays the XML source schema in nested tables. On the other hand, the Result Display Module of QSByE displays the querying output as a flat-table of elements users have selected from the source schema to query. Overall, QSByE is fairly effective at extracting XML-based data from DTOR.

Several design features of QSByE are notable. First, it represents and displays the XML source schema in a nested hierarchically format. Secondly, it allows users to choose which elements from the source table are to be included in the query result. Thirdly, it allows users to create constraints for these table elements through the query selection operation. However, QSByE has many drawbacks that hinder its generalizability. The biggest weakness of QSByE is that it uses a proprietary query language to query data sources in a proprietary format. QSByE uses its own QBE-like language to query the

DEByE-specific, XML notation-based data source (the DTOR). So it does not use W3C-compliant XML querying language like XQuery. Another downside of QSByE is the lack of variation in the output format. Query results are always presented to users in flat tables, so users are not able to redefine the structure, among other attributes, of the outputs. Many newer informatics systems have since surpassed QSByE in areas where it is deficient.

2.1.1.2 QURSED

Querying and Report Semi-structured Data (QURSED) is a query forms and reports (QFRs) generator developed at the University of California, San Diego[33-36]. It was designed for the purpose of creating a web-driven XML querying interface. The QURSED system consists of three components: the QURSED editor, the QURSED compiler, and the QURSED run-time engine. The graphical query interface of QURSED is divided into two parts: the QURSED editor and the QFRs. The QFRs are the web-based front-ends of the query interface. The QURSED editor is the user interface for developers to create QFRs, which are then used by end-users to query the XML data source. Users can query the XML data source by instantiate the form elements in the QFRs, which are referred to as query set specifications (QSS).

The QURSED editor first takes in a source XML schema and displays it as a labeled order tree objects (lotos) tree to QFRs developers[33-36]. Then developers can choose which elements from the source lotos tree end-users can query and which of these query-able

elements will be displayed in the query results. After developers have selected which elements from the source schema to query and display, the QURSED editor will generate the appropriate QFRs, which are dynamic web pages containing various input boxes and selectable lists to be instantiated with different values by end-users. Once the output schema (QFRs) is instantiated with values users have chosen (QSS), the QURSED compiler will take the QSS and translate it to XQuery query. The QURSED run-time engine will execute the XQuery query and retrieve the correct results from the XML data source.

The QURSED system has several useful features. The QURSED editor displays the source schema as a hierarchical tree to developers during the constructions of QFRs. Developers can access a great deal of useful information, such as relations among source elements or individual element's hierarchy and value, from the source schema tree. QURSED is also web-oriented. The QFRs, dynamic web pages which can be easily deployed on any server, allow end-users to access the query interface through a simple web browser. Users can use the QFRs to query the XML data source without bothering with complex system installations. QURSED does have a few limitations. Developers must first create the QFRs using the QURSED editor and deploy them online, and then end-users can use the QFRs to query the XML data source. Even though QURSED allows developers to create QFRs arbitrarily with any query-able elements from the source schema, every time end-users want to query a different element, the developers must update and deploy the QFRs again. The types of QFRs and the QSS that can be

generated are restricted by the source schema. Therefore the query results do not contain any user-created elements. QURSED allows developers to load a HTML template for each QFR to define which query elements will be displayed in the query result. But the format of the query result is inflexible because all the queried elements are presented in the predefined HTML template. In addition, whenever the developers change the QFRs they have to modify the output templates as well. The QURSED compiler also prevents end-users from changing the queries created from the QSS. Therefore, the types of queries end-users can execute are limited by the QFRs and the QURSED compiler.

2.1.1.3 XBrain

The XBrain project is developed by the Structure Informatics Group (SIG) at the University of Washington as a part of the integrated Human Brain Project (HBP)[45-46]. XBrain is a distributed data query and analysis system that allows users to query multiple data sources with a single XQuery statement[40]. XBrain incorporates a sophisticated relational-to-XML mapping middleware, Silkroute, to publish data from a relational database as XML over the web[19,44]. The distributed XQuery statement is processed by an integrated collection of web services that can query a relational database (Cortical Stimulation Mapping - CSM), an ontology (Foundational Model of Anatomy - FMA), and a XML file (Image Manager - IM) simultaneously. The current query interface for XBrain is a list of selectable saved queries in a Java Server Page (JSP). Each saved query also always has an explanatory name that will describe the query result. For example the saved query with the name “Semantic Paraphasia” can retrieve the data of patient who

has the semantic paraphasia speech error from the Cortical Stimulation Mapping (CSM), which is a relational database storing the speech error data from the brain surgeries of numerous patients. When users want to execute a query, they will need to select the query from the list of names. Once users select the query, XBrain displays the actual XQuery statement to users, so users can modify the query at their discretion. Then when the query is executed, XBrain will return the query result from one or multiple data sources.

Saved queries are akin to search terms for search engines. They are easy for users to execute and their self-explanatory names convey the meaning of the query result to users before the query is executed. The XBrain query interface allows users to modify the XQuery statement before submitting it to the distributed query system, so expert users can use the saved queries as templates to create even more complex queries. XBrain also allows users to save any newly created query with a self-explanatory name. However, XBrain's saved query approach limits the ability of novice users to create new queries themselves. For expert users who are comfortable at writing XQuery, they can easily modify the saved queries to suit their specific needs. But for novice users, if they want specific queries that are not in the saved queries, they will need knowledgeable informaticists to create the queries for them. And then they can execute the queries through the XBrain system. Also, all the saved queries have defined output formats, so it is not possible for novice users to graphically change the structure of the output format without having to manually modify the XQuery statement by themselves. The saved

queries approach provides a straightforward way of associating a query statement with an expected result, but it limits the possibility for inexperienced users to visually create arbitrary queries.

2.1.2 Unstructured query approach

Informatics tools that use the unstructured query approach can create a wider range of query expressions than tools that use the structured query approach. Unlike structured query tools, unstructured query tools can be a part of a larger information retrieval system, or as stand-alone applications, or as a feature in an Integrated Development Environment (IDE). Therefore, in order to accommodate all these types of applications unstructured query tools tend to be more powerful and diverse at generating XML queries. One of the defining features that separate informatics tools using a structured vs. an unstructured query approach is that unstructured query tools allow users to arbitrarily define the structure and other attributes of the query output. The concept of creating arbitrary output formats depends on the ability of the system to map the source element to the output element. Mapping is possible between the source and the output schema if the source elements and their corresponding elements in the output schema maintain equivalent hierarchical relationships. As long as these relationships are not violated, mapping allows users to arbitrarily construct the output format.

Unstructured query tools can generate a wider variety and more expressive of XML queries through their graphical interfaces. Unstructured query tools can create new

structural elements for the output schema and also impose sophisticated constraints statements on existing elements in the output schema. They can also perform joins of elements from multiple XML data sources. Compared to structured query tools, unstructured query tools allow intermediate to advanced XQuery users to construct complex queries using specific constraints and place the query results in sophisticated formats.

2.1.2.1 BBQ

Blended Browsing and Querying (BBQ), a graphical user interface for browsing and querying XML data sources, was developed at the University of California, San Diego[32]. BBQ allows users to construct queries in a QBE-like fashion. BBQ is the client graphical user interface for the Mediation of Information using XML (MIX) system. BBQ takes in a source schema in Document Type Data (DTD) format and creates a directory-like tree structure for the source schema. Then users can drag and drop elements from the source tree to the query result schema. BBQ strictly enforces that the query result schema must be a tree, so users must define a root element for the query result before they can drag and drop elements from the source schema. Once the query result tree construction is completed, the query is sent to the MIX mediator to retrieve results from the XML data source.

BBQ has several useful features for XML querying. It has a DTD inference mechanism that can augment loose or incomplete schemas in XML data sources. BBQ also displays

the source schema in a hierarchical tree format that is familiar to most users. In addition, unlike the proprietary, XML-based source data format used by QSByE, BBQ is actually XML-driven. So its source schema and query instances are represented in DTD and XML respectively. BBQ also supports joins of source elements and functions across multiple source documents. It allows users to drag and drop elements from different source tree windows to the same query result tree. Lastly, BBQ provides a mechanism similar to Querying-in-place[7] called filtering. The filtering process automatically evaluates the query result tree to provide users with dynamic feedback of the query as they are creating the query. However, BBQ also has a few shortcomings. BBQ can not generate W3C-compliant query language such as XQuery because the MIX mediator supports its own proprietary query language. BBQ only generates queries in XML Matching And Structuring (XMAS) query language[31-32]. The XMAS query language limits users from using more advanced features, such as FLWOR (For Let Where Order-by Return) expressions and other functions, which only the XQuery language supports[48]. BBQ allows users to access the query result trees from previous *n* number of queries, but BBQ does not allow users to assign descriptive names to these query results.

2.1.2.2 XQBE

XQuery By Example (XQBE) is a graphical query system developed at the Politecnico di Milano[1,3-5]. XQBE was designed to provide a graphical query interface for creating arbitrary XQuery statements. XQBE consists of two components: the XQBE client and the XQBE server. The XQBE server translates the query result tree to the XQuery

statement, executes the query over any arbitrary XML data source, and returns the result to the XQBE client. The XQBE client is a stand-alone, Java-based graphical query editor that allows users to construct the query result tree from any arbitrary XML source schema. XQBE accepts the source data schema in XML, DTD, and XML Schema formats. Once users have loaded the source schema, they need to explicitly define both the source tree and the query result tree. Users construct the source tree with only elements expected to either have direct mapping relations to elements in the query result tree or become constraints for the query result. Therefore users would not need to put any elements in the source tree if they do not to expect them to either augment or restrict the query result. XQBE can create new structural nodes in the query result tree, and it can also rename nodes that have mappings with source tree nodes. XQBE is also very expressive. It can generate a wide variety of types of queries supported by the XQuery language. Although XQBE does not implement the entire XQuery language graphically, the scope of its ability to formulate XQuery queries is considerably larger than any previous XQuery graphical query interface. Lastly, XQBE also allows users to save queries on their own computers.

Although XQBE is a fairly robust and competent graphical interface for XQuery, it has a few shortcomings. First, XQBE requires users to have an absolute knowledge of the XML source data schema. XQBE does not display the entire source schema to users as a nested tree, so users need to know the names of all the elements and their hierarchical relations in the source schema. XQBE offers one feature called the schema-assisted query

construction that can help users construct the source schema through gradual expansions of the source tree, but the interface is not user-friendly and the feature is not available by default[3]. XQBE also has elaborate syntax and semantics. Individual types of nodes such as source/target nodes, newly generated nodes, attributes and constraints, have their unique symbols and different restrictions on how they can be used. This complex system of symbols, lines, and expressions makes it cognitively more difficult for users to use the XQBE query interface. Lastly, XQBE is a stand-alone Java application, so it is not web-enabled, therefore XQBE would not be an ideal fit for a distributed computing model where all resources are expected to be shared among different users.

2.1.2.3 Stylus Studio

Stylus Studio is a comprehensive XML editing, publishing, and querying Integrated Development Environment (IDE)[52]. Stylus Studio is a very powerful integration tool for creating, deploying, and searching XML data sources. The XQuery graphical interface of Stylus Studio is considerably more robust and impressive when compared to other graphical query systems. Stylus Studio's query interface combines BBQ's ability to display the source schema in a hierarchical tree and XQBE's ability to create complex, arbitrary XQuery queries. It allows users to drag and drop nodes from the source tree to the query tree. Stylus Studio also supports Query-in-place, where users can preview query results from query fragments, and multi-document joins. Stylus Studio creates predicates in a unique way as well. It first requires users to attach a FLWOR expression to the node in the query result tree. The FLWOR expression is represented by a flower

icon in the interface. It acts as an anchor where users can attach multiple XQuery statements to its branches: “for”, “where”, “let”, “order-by”, and “return”. Overall Stylus Studio is a well-rounded commercial product for expert users who are experienced in the XQuery language to create queries.

Stylus Studio has several weaknesses. First and foremost is the cost. An individual license for Stylus Studio is expensive. So in a biomedical research setting where multiple researchers want to query their own XML data sources, the cost of supplying each user with a copy of Stylus Studio to create and execute XQuery statements is prohibitively expensive. In a collaborative environment such as a biomedical research laboratory, a distributed computing model is more preferable because resources are shared among everyone. Therefore, Stylus Studio, as a stand-alone application that needs to be installed and maintained by dedicated informatics personnel, is not completely suitable for the biomedical research domain. Also, Stylus Studio requires users to understand the concept of FLWOR expressions in XQuery. Therefore, for novice users who are inexperienced in XQuery, it will be difficult for them to create queries through Stylus Studio’s interface by themselves.

2.2 Requirements for XGI

The structured and unstructured query approaches each have their own advantages and disadvantages. Informatics tools that use the structured query approach generally have simpler query interfaces. Novice end-users can learn these tools quickly and begin to

retrieve information from the XML data source. For example, the QURSED system allows users to use simple web-based query forms (QFRs) to query XML data. The XBrain system enables users to execute saved queries whose names describe the meaning of the query results. Structured query informatics tools also have pre-defined query result formats. The query results always have the same structures, and the types of data returned are always consistent. In many structured query tools, the developer and the user of the query are not the same person. The developer creates the query interface for the end-user, and then the end-user uses the interface to obtain information from the targeted XML data source.

Informatics tools that use the unstructured query approach have more complex query interfaces, but the complex query interface also provides more features and options for users to create query statements graphically. Unstructured query tools are generally used by query developers who are also query users. For unstructured query tools, users need to spend more time learning the query interface. After getting past the initial learning curve, users will find that unstructured query tools can create more expressive XQuery expressions. These tools can also generate a larger and more complex set of queries than structured query tools. Unstructured query tools enable users to dictate the format of the query result by allowing them to construct any arbitrary format for the query result. Users create individualized query result formats for various reasons, such as to make the result more human-readable or to create a specific format for the query result so it can be imported into another application.

The structured and the unstructured query approach each offers different features for different users. Features that are easy for novice users to use may not generate desirable queries for expert users. Also, features that enable some users to create more expressive queries may be too complex for other users to learn. Therefore, trade-offs must be made between simplicity and expressivity when designing a graphical query generation tool[11-12]. We have taken into account these trade-offs between structured and unstructured query approaches when we designed the XGI system. We decided to combine the best features from both query approaches into our system. Our goal for the XGI system is to implement an XQuery query construction platform that is easy enough for inexperienced users to learn and expressive enough for advanced users to use.

We performed evaluations on all of the structured and the unstructured query informatics tools we have profiled in the previous section. We compared each system based on its most recent update, implementation of novel features, and generalizability of the system. At the end of our review process, we chose to focus our evaluations mainly on QURSED, XQBE, and XBrain. From our reviews of these systems, we believe a useful graphical query system should satisfy a few basic requirements. In the subsequent section, we will outline a list of requirements we perceived as necessities for XGI. We believe that for XGI's integrative design approach, which incorporates the best aspects of the structured and the unstructured query approach, these requirements should be followed and implemented.

Reduced cost of implementation

Commercial products, such as Stylus Studio, can be too expensive for an individual biomedical researcher to license and use. Dedicated solutions such as XBrain, which need informaticists to develop complex queries, require considerable efforts and expenses to maintain. Other informatics tools, such as QURSED and XQBE, are freely available and not difficult to acquire. However, they might have additional hardware and personnel requirements that still might prove too costly for an individual biomedical researcher to obtain. Reducing the cost of acquisition and maintenance should be a priority for any graphical query system. Internet technology is a logical solution to the problem of installing and maintaining local software clients. Using the new AJAX paradigm, web-only thin-clients can look and behave similarly to any rich-client on the desktop. Web-based applications can also support multiple users with only one installation of necessary software on the web server. Therefore, an online-based approach can greatly reduce the cost of software installation and maintenance for biomedical researchers.

Support for multiple schema formats

A complete graphical query interface system for XQuery should allow users to upload the source schema in different W3C-standardized formats. This feature gives users the flexibility to define the XML source data model in different ways. The three standardized and most commonly used representations for semistructured data are XML, DTD, and XML Schema. All three of these formats have different level of richness, and sometimes

it is easier to generate one type of schema format than another for a particular XML data source. Therefore, the graphical query interface should not limit its generalizability by limiting the type of schema format it accepts. Stylus Studio and XQBE allow users to upload source schema files in XML, DTD, and XML Schema format[3,52]. QURSED only supports XML and DTD format[33]. XBrain does not accept uploads of XML source schema, because XBrain is designed to create and execute queries against XML data sources whose schemas are already known to the XBrain system.

Navigable source tree

The source tree is a direct representation of the source schema. Like the source schema, the source tree is built as a hierarchically tree. For large and complex source schemas, such as the Cortical Stimulation Mapping (CSM) database used in XBrain, it is impossible for users to remember all of the elements and all of their hierarchical relations in the source schema. The source tree allows users to browse through the source schema. Users can then graphically choose which element in the source schema to be included in the query schema. The source tree eliminates the need for users to remember the entire source schema. Informaticists whom we have cooperated with on the XBrain project agree that a navigable source tree will improve the speed of query construction.

Powerful query creation

For a graphical query interface, its ease of use and its expressivity are inversely related. The easier the query interface is to use the less likely users can use it to create complex

queries graphically, and vice versa. However, a well-designed graphical query interface should be easy for novice users to use, but it should also be expressive enough that skilled users can use it to create as many types of queries as possible. The graphical query interface would use an integrative approach that is a combination of the structured and the unstructured query approach. The query interface should resemble the unstructured query tools so users have a lot of powerful functions to create complex queries. However, the query interface should also provide easily accessible functions for novice users to execute pre-defined queries.

Support collaboration

Collaborations are common among biomedical researchers because many current research efforts require integration of multidisciplinary expertise across different knowledge domains. Collaborators need to share data and research questions more easily and efficiently. An ideal solution would be implementing a data management system that can help an individual user organize his/her data source and queries, and allow multiple users to share their data source and queries among each other. Many laboratory data management systems are developed to support research collaborations by specifically using web technologies. Therefore, the graphical query interface should also strongly incorporate web technologies, so it can be integrated into any web-enabled laboratory data management system.

Easy integration with other applications

Most of the graphical query interfaces we have reviewed are implemented as stand-alone applications. For some systems, the query construction and the query processing are so tightly integrated that it is impossible to decouple the graphical interface from the query generator. Therefore, graphical query interfaces of these systems cannot be easily integrated with other applications. The only extensible graphical query interface of these systems is the one in XBrain. Informaticists can implement the interface of XBrain upon another system, because XBrain's query selection, formulation, and execution are implemented using web-based technologies. An extensible graphical query interface can be seamlessly implemented as the query interface for other semistructured data querying systems.

Chapter 3: XGI System Overview

In the previous chapter, we have proposed a list of requirements for XGI. Those requirements are gathered from our review of existing informatics solutions and from our past experience of building the query interface for the XBrain system. We feel those requirements are the most suitable for a simple, unobtrusive XML graphical querying interface that is also powerful enough to generate most XQuery expressions. The current implement of the XGI system does not satisfy all of the requirements we have listed. The importance of the requirements that XGI does not implement will be emphasized in a later chapter. In this chapter, we provide a detailed description for every component of the XGI system. We describe how we implemented each component and we also explain why we made those design choices.

The Figure 1 demonstrates how every aspect of the XGI system interacts with each other. The figure demonstrates that XGI loads a source data schema to create a source data model (DM). Then XGI uses the source data model to create the query data model and the query schema, which is translated into XQuery. XGI allows users to save both the source and query schemas as well.

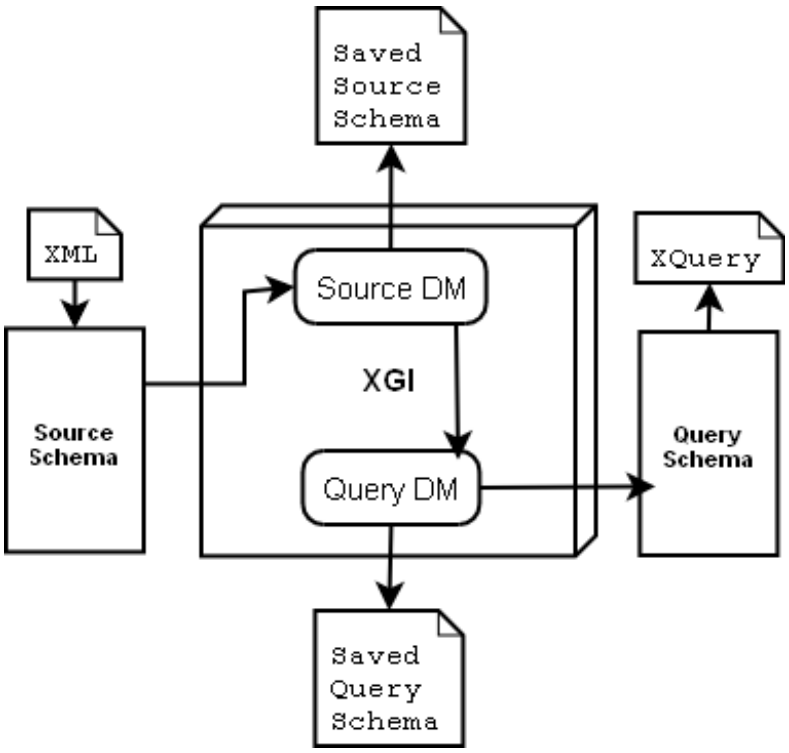

Figure 1. Schema relationships in XGI

The XML source schema is a set of rules that define the validity of the XML document[14]. It contains the hierarchical structure, the elements, the element properties, and other information that are in the targeted XML data source, which is the data source users want to query. XGI uploads the source schema from a file and creates the source data model (DM), the XGI local representation of the source schema. The source data model allows users to query and select elements in the source schema. After the data model is created, the XGI graphical query interface displays the data model in the query interface as a hierarchical tree. The source tree contains source tree nodes (Figure 2-1), which represent the XML elements of the source schema. The source tree nodes can also have attribute nodes, which are denoted by their red color (Figure 2-2).

Figure 2. Source tree node and attribute

The query schema defines rules for the generated XQuery statement. The query schema contains the output structure, the elements, the constraints, and other information about the XQuery statement that can be produced from the schema. Users can define the query schema through XGI or upload a query schema file that already describes the schema information for the targeted query result. The query data model, the local XGI representation of the query schema, is created as soon as the query schema is initialized. The query data model allows users to change the schema by adding/deleting elements and adding/deleting properties of each element. The XGI graphical query interface also displays the query data model in the query interface as a hierarchical tree. The query tree contains two types of query tree nodes: the schema node, which is denoted by its black color (Figure 3-2), and the user-defined node, which is denoted by its blue color (Figure 3-1). The schema node is the source tree node users have added to the query schema and there is an implicit mapping edge between the source tree node and the schema node. The user-defined node is a unique node that is created by users and does not map to any source tree node. Only schema nodes can have attribute nodes. The attribute nodes serve

as containers for users to return the query results as attributes of the schema nodes. Also, only schema nodes can have predicate nodes. The predicate nodes are used to store constraint statements for the schema nodes. The constraint statements act as filters that allow users to place parameters and conditions on the query results.

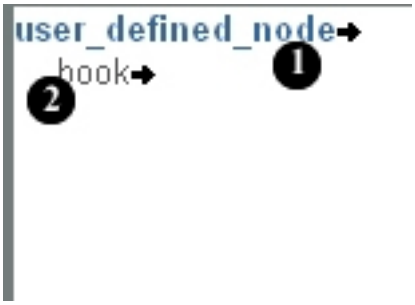

Figure 3. Query tree nodes

3.1 System architecture

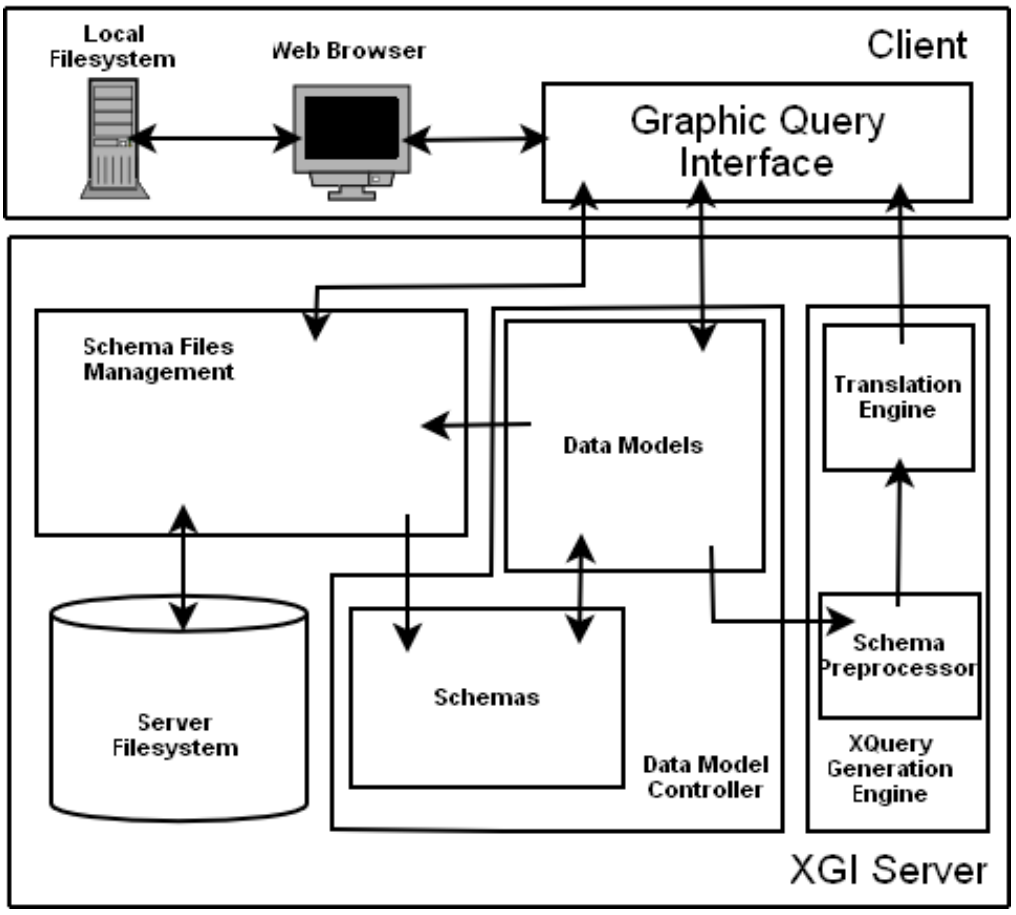

Figure 4. XGI system diagram

The implementation of the XGI system adheres to the Model View Controller (MVC) paradigm (Figure 4). MVC advocates that in building an application, the representation, presentation, and manipulation of its data model should be separately implemented. The XGI system consists of the graphical query interface, the schema files management, the data model controller, and the XQuery generation engine. The front-end of the XGI

system is the graphical query interface. It is what users utilize to inspect the source data schema, interact with the XGI system, create the query schema, and retrieve the XQuery statement. XGI's data model controller, schema files management, and the XQuery generation engine form the back-end of the XGI system. The schema files management component allows users to both load and save the source and the query schema file. The data model controller provides functions for the XGI system to create the appropriate data models from the source and the query schema. The data models manage users' access to the source and the query schemas. The data model controller provides functions to interpret AJAX requests for either retrieving schema information from the source data or modifying information on the query schemas. The XQuery generation engine will convert the query schema users have created to its corresponding XQuery query. The XQuery engine translates the query schema to a XQuery statement based on predefined XQuery grammar, so it does not verify whether the XQuery statement is valid (although it should always be) or whether the query will return, if any, useful results from the targeted XML data source.

3.2 XQuery generation engine

In this section, we discuss how the XGI system creates the XQuery statement from the query schema. We first describe how the XQuery generation engine is implemented and how it is operated. And then we will define the grammar used by the XQuery engine to generate queries.

The XQuery generation engine is implemented as a collection of Java classes and Java Servlets. The XQuery engine is functional only if users have loaded a source schema and have also instantiated a query schema, which can be created from the XGI query interface or uploaded from a query schema file. Users need to initiate the query generation request from the graphical query interface. The XQuery engine retrieves the query schema from the query data model, and preprocesses the query schema using several Java helper classes to prepare it for the translation process by transforming the schema to conform to the grammar used by the engine. The Java helper classes create a copy of the query schema first to prevent the transformation process from permanently modifying the original query schema. Then the transformation process removes any invalid schema elements, resolves any ambiguities in the schema, and creates the translation-ready schema. Finally, the translation-ready schema is processed by the XQuery engine to generate valid query statements.

The Java Servlets of the XQuery generation engine recursively generate the XQuery statement from the preprocessed query schema. The translation engine uses an Extended Backus-Naur Form (EBNF) grammar which we have defined later in this section. The translation process is built through nested FLWOR expressions. XGI's implementation of XQuery FLWOR expression is limited to only the clauses "for", "where", and "return". Starting with <query>, each <query> element is translated to one XQuery FLWOR expression or an empty element. The "for" clause of the FLWOR expression contains a list of variable bindings. Each variable binding contains the variable for the schema node

and its path expression. Path expressions of the node and the node attribute are extracted from variable bindings in ancestor schema nodes. The “where” clause contains a list of constraint bindings. Each constraint binding can be translated to an existential function. It can also be an expression of pair-wise comparisons of a node value (NV) vs. a NV, a node attribute value (NAV) vs. a NAV, a NV vs. a NAV, a NV vs. a constant, and a NAV vs. a constant. The “return” clause can generate several different types of values depending on the context in which the “return” clause is used. The “return” expression can be a computed node value or a new (and possibly) empty element. The table below shows a modified view of the query schema (the node and its node type) and the schema’s corresponding XQuery statement.

Table 1. Query schema and corresponding XQuery

Query schema (modified view)	**Corresponding XQuery**
result:new **patient**:node Predicates: stimulated=Y **pnum**:node **viq**:node **age**:renamed **age_at_registration**:text	<result> { for $p0 in $root/patient where $p0/surgery/csmstudy/trial/stimulated/text() = ‘Y’ return <patient> {$p0/pnum} {$p0/viq} <age> {$p0/age_at_registration/text()}</age></patient> }</result>

The EBNF grammar describes the subset of XQuery that was implemented for the translation of XGI queries. The complete grammar is adopted from the EBNF grammar of XQBE, and it is such in Table 2.

Table 2. XGI's EBNF grammar

<query>	**:= <flwor_exp> | <startTag> '{' <query> '}' <endTag> | <emptyTag>**
<flwor_exp>	**:= <for> <where>? <return>**
<for>	**:= 'for' <some_var> (',' <some_var>)***
<some_var>	**:= '$'<var_name> 'in' <path_exp>**
<where>	**:= 'where' <constraint>?**
<constraint>	**:= <predicate> ('and' <predicate>)***
<predicate>	**:= 'exists(' <path_exp> ')' | < pred_exp>**
<pred_exp>	**:= <exp_var> <operator> <exp_val>**
<exp_var>	**:= <variable>**
<exp_val>	**:= <const> | <variable>**
<variable>	**:= <path_exp> | a variable assigned to a previous schema element**
<operator>	**:= '=' | '<' | '>' | '<=' | '>=' | '!='**
<return>	**:= 'return' (<emptyTag> | <variable>) | 'return' <query>**
<startTag>	**:= '<'<name> <attr>? '>'**
<endTag>	**:= '</' <name> '>'**
<emptyTag>	**:= '<' <name> <attr>? '/>'**
<attr>	**:= (<name> '=' <path_exp>) +**
<name>	**:= valid name of schema elements**
<path_exp>	**:= XPath expression of a schema element**
<const>	**:= a constant value of the schema element**
<var_name>	**:= an automatically generated, non-duplicative name for a variable**

3.3 Graphic query interface

The XGI graphic query interface is the front-end of the XGI system. It implements

several graphic interface script libraries. These graphic interface scripts are computer programs that a client, usually a web browser, must obtain from the server in order to access the features of the XGI system. These graphic interface scripts are implemented in the JavaScript programming language. They are always stored on the XGI server, and they are usually downloaded from the server by the client when users request the appropriate URL address using their web browsers. After these scripts are downloaded, they will be automatically executed by the web browser. Other web resources, such as the Java Server Page, the Cascading Style Sheets, and the images used in the XGI query interface, are also obtained by the web browser from the server at the same time as the graphic interface scripts. These web resources help the browser initialize the preliminary layout of the query interface.

The graphic interface scripts perform several essential functions before users can interact with the query interface. These functions include setting up the query interface display, allocating global variables, requesting additional resources and data from the XGI server, and registering call-back functions with the web browser's event handlers. Overall, the main purpose of these graphic interface scripts is to manage and facilitate user interactions. When users interact with the XGI query interface by executing a command, the functions provided by these scripts will determine how the user action should be handled, whether it is updating some information in the locally stored data or retrieving additional information from the XGI server. Then the scripts will always update the query interface to reflect the results of users' actions.

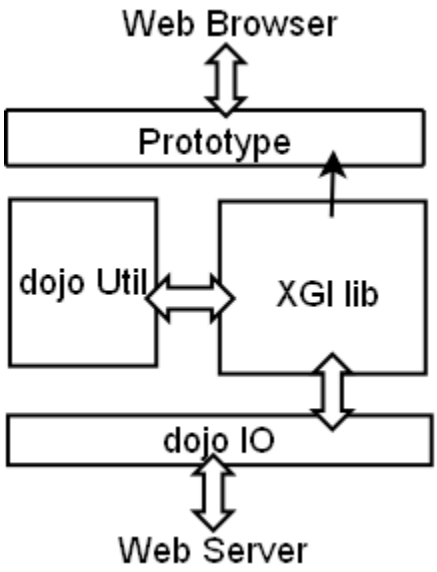

Figure 5. UI scripts interaction

The graphic interface scripts employ functions from three different user interface (UI) libraries (the Prototype libraries, the dojo toolkits, and the XGI libraries) to integrate AJAX features and functionalities into the XGI graphical query interface (Figure 5). The Prototype libraries and the dojo toolkits are freely available, open source UI programming libraries implemented in JavaScript. They make the tasks of managing and extending the XGI interface easier. We chose the functions from the Prototype libraries to provide simple Document Object Model (DOM) manipulations, such as retrieving/inserting DOM nodes, and showing/hiding DOM trees. The dojo toolkits provide additional functionalities that the Prototype libraries do not support. We specifically selected several IO libraries from the dojo toolkits to provide us with frequently used functions that can simplify sending asynchronous requests (especially

sending files) to the server and retrieving information from the server. We also used a few utility libraries from the dojo toolkits to register functions with browser's event handlers, and we implemented some of the animation capabilities from the dojo toolkits to deliver additional interactivity to the XGI interface.

We developed the XGI libraries to perform XGI specific tasks, such as initializing the XGI graphical interface environment, handling schema information from the XGI server, generating XGI interface contents, and providing abilities for users to formulate XQuery statement. These functions use a mix of dynamic DOM-tree manipulation and asynchronous requests to control the query interface, obtain data from XGI, and modify the data model on the server. The interface initialization process uses functions from the XGI libraries to register menus and buttons in the query interface with the browser's event handlers. These scripts also request the list of available saved schemas (both the source and the query schemas) from the XGI server, and then update the query interface with this list. Once users have uploaded a source schema, the scripts from the XGI libraries will automatically create the source schema tree from the source data model and display the tree in the query interface. The XGI libraries also provide the UI functions for users to create the query schema. After the query schema is created, the UI scripts will provide additional functions for users to modify the schema. Lastly, the XGI libraries allow users to retrieve and view the XQuery statement that is translated from the query schema.

3.4 Schema files management

The schema files management component is a significant part of the XGI system's back-end (Figure 6). The files management component is implemented as a collection of Java classes and Java Servlets. It performs several specific functions. The files management component reads the metadata from the saved-schema index files for the graphical query interface, handles asynchronous file upload requests, and stores both the source schema and the user-generated query schema in their respective saved-schema formats onto the XGI server.

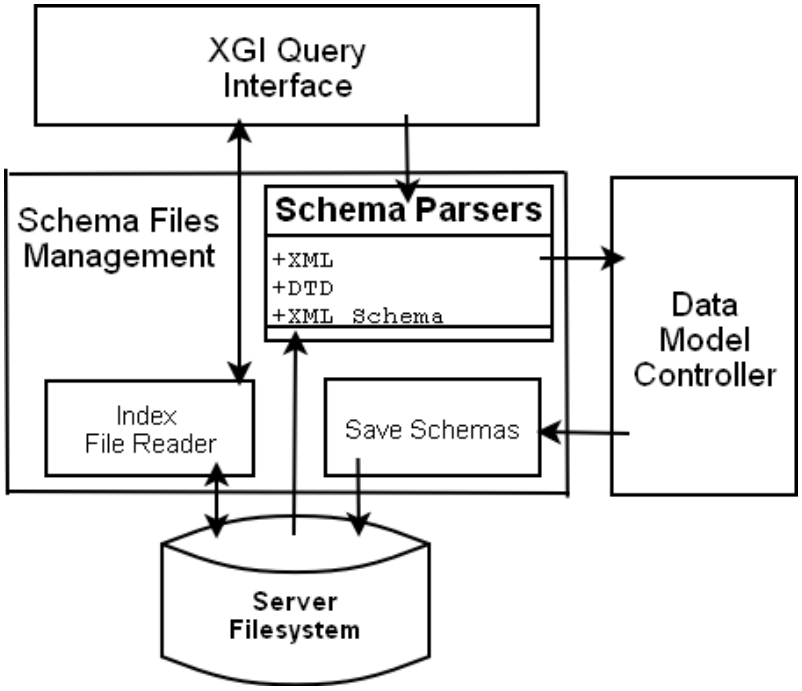

Figure 6. Files management component

When the XGI system's graphic interface scripts first initializes the query interface, it sends an AJAX request to the files management component asking it to read the metadata

from the saved-schema index files on the XGI server. These index files contain the path information of all the saved schema files on the server local filesystem. Once the files management component reads all the metadata contained in the source and the query schema index files, it will send the metadata back to the query interface. The XGI query interface will only display the list of available saved source schema files to the user at the beginning. The query interface will display the list of available saved query schemas after users have defined a saved source schema.

The files management component also handles uploading both the source and the query schema files. First, the user uses the interface to load either a saved schema on the XGI server or a schema file on the local filesystem of the user's computer. Then the query interface will initiate an asynchronous request to the files management component to indicate the mode of file upload (from the server or from the local filesystem). Different methods of file upload are handled differently by the files management component. The file on the local filesystem of the user's computer is transported via the HTTPRequest object to the server. We implemented the IframeIO library from the dojo toolkits to circumvent the HTML form element from submitting the file synchronously and automatically reloading the current query interface. The IframeIO library uses the hidden inline frame to upload the file. This method allows the file to be submitted in the background, away from the active query interface. Therefore, there is no disruption in the operation of the query interface because if the file was submitted through HTML form, the entire page will be forced to refresh. Servlets in the files management use the

Commons IO package to retrieve files that are uploaded from the local filesystem. The functions in the Commons IO package are used to parse the file stream from the HTTPRequest object and convert the file stream to the input stream for various schema parsers.

Users have options to upload the source schema in different file formats. The files management component can handle the source schema file in three formats: XML, DTD, and XML Schema. When users upload the source schema file, the files management component will determine the type of format and invoke the appropriate Servlets to process the file. If the file is in XML format, then the files management Servlets will use the Simple API for XML (SAX) parser to process the XML file; if the file is in DTD or XML Schema format, then the Servlets will use the open source DTD or XML Schema parser respectively to process the file.

Unlike the source schema file, the files management component can only handle the query schema file in JavaScript Object Notation (JSON) format. The files management component has a special set of Servlets to handle query schema files. The Servlets of the files management component that handle the query schema file also utilize the Commons IO package to parse the file stream from the HTTPRequest object. The Servlets incorporates an open source JSON package to process the query schema. The open source JSON package provides a JSON parser that can extract JSON objects from the file stream.

If users choose to load either the source schema or the query schema from the collection of saved schemas on the XGI server, the files management component only uses the XML SAX parser to process the saved source schema files, because we have defined our own XML-based notation for the saved source schemas. The files management component still uses the functions from the JSON Servlets to process saved query schemas files.

To load a saved schema, users need to select the schema file's HTML element in the graphical query interface. The HTML elements are constructed from the metadata of the index files that the query interface has requested from the files management component during the interface initialization. Saved source schemas and saved query schemas have separate index files. Each entry in the index files contains the given name of a file and its absolute path on the XGI server's filesystem. In the case of the query schema index file, each entry of the saved query schema also contains the name of the source schema which the query schema is associated with. When users select a schema file, its path information is sent asynchronously to the files management component. Since the saved schemas and the Servlets of the files management component all reside on the XGI server, the files management component can use a file reader to obtain the file stream from the saved schema.

All saved schemas are originally constructed from schemas uploaded to the XGI system.

From our experience collaborating with neuroscientists on the XBrain project, we learned that saved queries are useful for biomedical researchers. The XBrain system's saved queries approach allows informaticists to construct the XQuery statement that will answer a question posted by the user. Then informaticists will save the query with the question as its title, so the user can just select the question and execute the query. For example, neuroscientists prefer to ask specific questions such as "I want all patient information for patients who are over the age of 35 and have Semantic Paraphasia". This question is saved as a query in the XBrain system and it has the title "Semantic Paraphasia, patient over 35 years old". The neuroscientist is able to execute the query without needing to know what elements from the XML source schema are contained in the query result. They already know the meaning of the query result before they execute the query.

The XGI system was implemented with this saved query approach in mind. XGI allows users to save both the source schema and the query schema with specific names. This feature allows users to give queries names that will indicate the queries' answers to specific scientific questions. This feature also enables users to better manage their queries. For example, users (specifically neuroscientists) might want to repeat their previously queries to check if there are new updates, such as new entries of brain surgery data to the patient trial group since last time they ran the queries. However, if these queries have not been executed for awhile, users might forget what they are for. XGI's saved query approach allows users to easily recognize and load any query from the XGI

server and execute it, because users already know what the previous query is for from its name.

The XGI saved schema uses two different formats to standardize representations for both saved source and saved query schemas. The original uploaded source schema is always in either XML, or DTD, or XML Schema format. However, XGI does not preserve the original file format of the source schema when saving it. The XGI files management component will save the source schema in our own XML-based notation (Table 3), because we do not need to duplicate the variability of richness and expressivities in the different formats of the original source schema file. The files management component only stores information about the hierarchy of the schema elements, names of all elements, and names of each element's attributes. The XML-based notation of the saved source schema file looks similar to a regular XML document. It preserves the structure, elements, and attributes of the source schema and discards all other schema information from the original source schema file.

The saved query uses the JSON format (Table 3). We chose to use JSON format for the saved query mainly because of its simplicity and interoperability. Unlike the saved source schema, the saved query needs to have more information about the schema. The saved query needs to preserve every detail of the query schema, such as its structure, names of its elements, and nodal properties of each element. The saved query schema is implemented as a JSON array with each element saved as a JSON object. Each element's

nodal properties are stored as a mixture of JSON objects and arrays of element objects. Every JSON object is similar to an entry in a hash table, with a pair of keys and values. JSON also allows us to store and retrieve the saved query schema easily. The open source JSON package can conveniently parse the saved query file to produce JSON objects. Then XGI can automatically use these JSON objects as JavaScript objects in the query interface. JSON is also a portable data presentation format that can be transported between heterogeneous applications, which are applications that use different technologies. Therefore, the saved query can be easily used by different applications other than XGI.

Table 3. Saved schemas

Source schema	Query schema
<elem_1 attr1="" attr2="" ….> <elem_2 attr1="" … > …… …… …… …… </elem_2> </elem_1>	{ "node" : [{ "rName" : "", "lName" : "", "parent" : "", "rename" : "", "relation" : "", "returnType" : "", "predicate" : [{ "source" : "", "sourceType" : "", "target" : "", "sourceType" : "", "operator" : ""},] "attribute" : [{ "name" : "", "value" : "", "type" : ""},] },] }

3.5 Data model controller

The data model controller is also a component of the XGI system's back-end (Figure 7).

The main purpose of the data model controller is providing functions for users to formulate XQuery statements by manipulating the source and the query data models. These functions are implemented in Java programming language as both Java classes and Java Servlets. The data model controller creates data models from the source and the query schemas, retrieves information from data models for the query interface, and modifies the query data model according to users' requests.

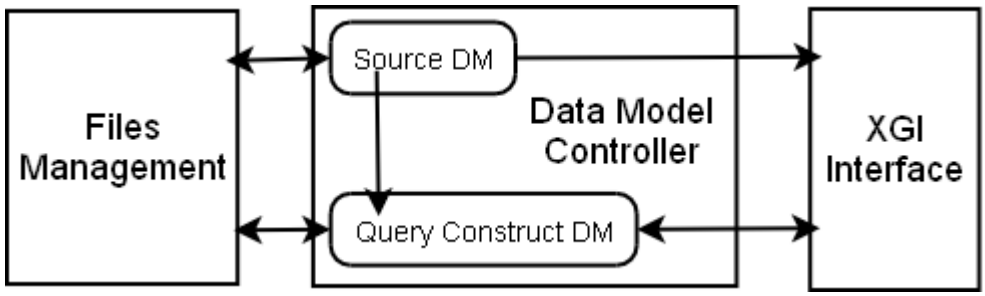

Figure 7. Data model controller

Data models are initialized after the schemas have been created. The source data model and the query data model have similarities and differences in their methods of initialization. Both the source and the query data model can be created by loading the schemas from files on the local filesystem of the user's computer or from the saved schemas on the XGI server. The data model controller's model creation function is integrated with the schema files management component. When the files management component is processing a schema file, it extracts the schema and then passes the schema to the data model controller to create the appropriate data model. The source data model can only be created by loading a schema file; the query data model, on the other hand,

can also be initialized when users create a custom query through the XGI query interface. To create a custom query, users must build the custom query schema from the beginning by using the query interface to define a root element.

The source data model, a representation of the XML source data schema for XGI, is constructed hierarchically in a tree-like fashion. The source data model is not modifiable; the XGI system can only read information from it. Once the source data model is initialized, the XGI query interface will attempt to create a navigable source schema tree in the interface from the schema information. The query interface first retrieves only the root element of the source schema from the source data model. Then the query interface initializes the source tree and displays the root element to users. The transmission of schema information between the data model controller and the query interface uses a send-per-request model. The send-per-request model is implemented so XGI will never send all elements of the source data model to the query interface at once. Every time users expand a source tree node that contains children elements (starting with the root node), the query interface will send an asynchronous request to the data model controller to only retrieve all the children elements of that tree node. When navigating a large XML source schema, the send-per-request model is useful at preventing transmission of large datasets between the XGI server and the query interface. For example, the schema for the CSM database has 474 elements. If the CSM database schema is uploaded and send-per-request is not implemented, users might need to wait for a very long time for the query interface to retrieve all 474 elements from the XGI server and display the source tree.

The query data model is always created after the source data model. The reason is because in order to construct a useful query for a given XML data source, users must formulate the query schema with elements from the source schema. Therefore, XGI requires users to load the targeted source schema before defining the query schema. XGI initializes the query data model in two different ways: upload a query schema file or define a unique schema. For file upload, the schema is parsed from the file as JSON objects. These JSON objects are sent to the query interface all at once. The query interface will evaluate each JSON object, and then submit it to the data model controller to construct the query schema. Users can also define a root element of the query schema through the query interface. The query data model and navigable tree are created simultaneously by the data model controller when users submit the root element.

The query data model is also organized hierarchically in a tree to reflect the nested nature of the XQuery query. The query data model does differ from the source data model in several meaningful ways. First, the query data model is modifiable. Users are allowed modify the nodal properties of each element in the schema, in addition to accessing information about each node. The query tree, on the other hand, also has a few visual differences from the source tree. The query tree does not have the option to expand or minimize the tree. Also, every query tree node has a node menu. Users will use functions in each node’s menu to modify nodal properties, such as children nodes, predicates, attributes, name, and nodal type or nodal relation.

The data model controller also provides users the ability to search for specific schema elements in the source schema. For large and complex schemas, such as the schema for the CSM database, the ability to search allows users to find schema elements by their names very easily. The search function is triggered when users enter a query string into the search box. The query string is automatically sent asynchronously to the data model controller. If the search function is used for the first time, the data model controller will construct a list of names for all the schema elements in the query data model. This list is saved to the current user session so it can be retrieved and used in any subsequent searches. The data model controller will compare the query string with the name of each element to determine if any element's name begins with the query string. Then the data model controller will return the list of source schema elements that match the query string to the query interface. The query interface will display these elements to users in the search result box.

Chapter 4: XGI Design, Feature, and Usage

Many of the graphical query interface applications we have reviewed require informaticists to download, install, and setup the necessary software components before they can be used. This process can be daunting and time-consuming for someone without previous system administration experience. With the XGI system, the only time intensive installation process is setting up the web server environment. The XGI system is designed to be a self-contained, web-only application, so it can be deployed on any web server. Once XGI is deployed successfully on a server, any user with access to the server can begin to use the system to create XQuery statements. However, like any new application, users must have an understanding of XGI's interface to use the system effectively. In this chapter, we first provide a description of XGI's interface layout. Then we will use an actual query as an example to explain each step to load the source schema, to create the query schema, and to save the XQuery statement. We hope this overview of the XGI system will demonstrate the generalizability and the many features of the system.

4.1 XGI interface layout

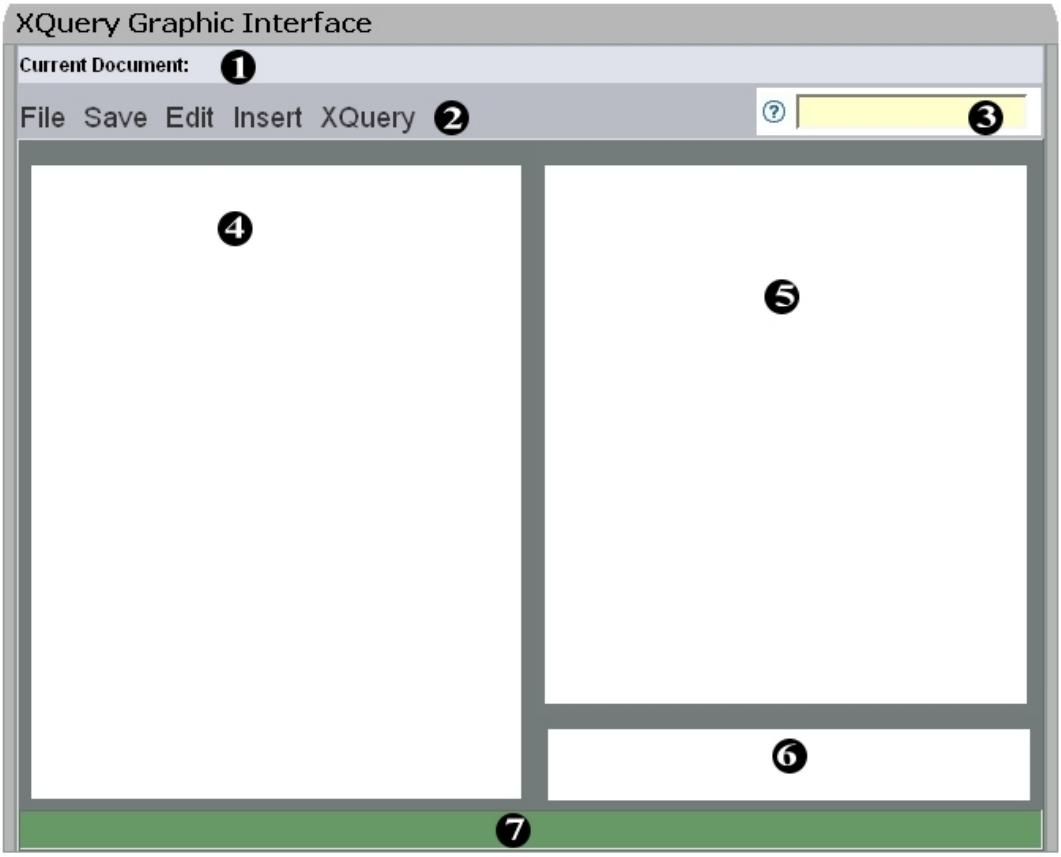

Figure 8. XGI query interface layout

The interface of the XGI system is divided into six major areas: the toolbar (Figure 8-1 and 8-2), the search box (Figure 8-3), the source panel (Figure 8-4), the query panel (Figure 8-5), the saved predicate panel (Figure 8-6), and the information panel (Figure 8-7). The toolbar consists of two panels: the document name panel (Figure 8-1) and the menu panel (Figure 8-2). The document name panel displays file names for the current source and query schema files. When users upload the source schema either from a file on the local filesystem of their computer or from a saved source schema on the XGI

server, the document name panel will display the name of the source file as "Source:…". The name of the query schema file will always appear after the source file as "Construct:…" (Figure 9). However, if users decide to create a custom query schema, the document name panel will not display any name for the query schema file.

Figure 9. Document name panel

The menu panel situates below the document name panel. It contains the menu buttons "File", "Save", "Edit", "Insert", and "XQuery". Every menu button has its own menus that are only accessible if users click on the button. The "XQuery" button is the only menu button that does not have any menus because it is used to send an asynchronous request to the XGI XQuery engine so the engine can translate the current query schema to an XQuery statement. Every other button's menu has specific functions. The "File" menu enables users to load and close the source and the query schema. The "Save" menu permits users to save the current source and query schemas on the XGI server. The "Edit" menu allows users to edit the selected query tree node. Lastly, the "Insert" menu allows users to insert the root node, user-defined query tree nodes, predicate nodes, and attribute nodes.

Users can search for specific elements in the source schema through the search function. The search box (Figure 8-3) is located on the right side of the interface, next to the

toolbar's menu panel. Users can enter either the full name or the partial name of a source schema element into the search box. XGI will assemble a list of source schema elements that match the query string, and then display the list in the search result box, which is located conveniently below the search box. The query interface normally hides the search result box from users' view. Users can toggle its visibility by selecting the text field in the search box.

The source panel (Figure 8-4) is the part of the query interface where users can navigate the source tree. The source tree provides a similar appearance and equivalent functionality compared to other expandable menu trees users might have encountered previously. Every source tree node has the same name as its corresponding element in the source schema. The non-leaf source tree node, i.e. the node that has children nodes, has a tree icon on the left side of the node's name. Users can use the tree icon to view the non-leaf node's children by clicking the tree icon to expand and collapse the node's sub-tree as indicated by the '+' and '-' symbol. Some source tree nodes might also have attributes. Users can view and select a node's attributes by using the menu icon, which is denoted by the '*' symbol and is located on the right side of the node's name (Figure 10), to open the attribute menu. The attribute menu contains a list of attribute nodes, which will be in red color to differentiate themselves from source tree nodes.

Figure 10. Source tree with attributes

The query panel (Figure 8-5) of the XGI interface is where users can construct the query schema. The query schema is displayed as a query tree. XGI does not allow users to expand and collapse the query tree unlike the source tree nor do the query tree nodes have attribute menu icons. However, every query tree node has a tooltip icon, denoted by an arrow symbol, on the right side of the node's name. Users click the tooltip icon to open the tooltip menu. The tooltip menu enables users to modify the query node using various functions, such as renaming the node, changing the node's relation or return type, deleting the node, and adding user-defined child nodes, predicates, and attributes to the node.

Situated directly below the query panel is the saved predicate panel (Figure 8-6). It is used to display and select saved predicates. The saved predicate is a unique feature of the XGI system. The motivation for its implementation stems from our observation of numerous instances in XBrain's saved queries where one query often uses the same

predicate statement in multiple schema nodes. In this situation, any other graphical query informatics tool we have profiled would require users to create the same predicate statement multiple times. Also, if the predicate statement requires fairly complex graphical construction, then using the graphical query tool will become increasingly burdensome for users. The process is unnecessary and redundant. Therefore, XGI will circumvent this problem by saving every predicate statement users have created for any schema node so users can reuse these predicate statements later, if needed.

The information panel (Figure 8-7), which is located on the bottom of the XGI query interface, is an unobtrusive way of displaying useful information about elements in the query interface to users without obfuscating the interface. The information panel can display many types of information and it is also extendable so users can add additional types of information. The panel can display the predicate statement of a saved predicate, information about a source tree node (such as its name and attributes), a query tree node (its name and nodal properties), and a node in the search result (the node's path).

In the following sections of this chapter, we will describe the various functions of the query interface by using an example question. This following sample question is directed at the CSM database used in XBrain and it is a typical question that a biomedical research might ask. *"Put into a tag 'result' of all the patient's pnum, viq, and age at registration only if the patient's brain during trial is stimulated and patient's data is public. Also, put the patient's sex as an 'id' attribute of the patient and rename to age_at_registration to*

age."

4.2 Load source and query schema

After users become familiarized with the interface, they first need to provide the system with the schema of the targeted XML data source. The query session starts when users define the source schema. XGI can handle any arbitrary XML data sources as long as their schema files are in XML, DTD, XML Schema, or XML-based notation format used in saved schemas. XGI affords users the option to simply load an XML instance of the source data. However, if users can not obtain the proper XML instance for the source data, they can generate the DTD or the XML Schema file for the targeted data source from freely available, open source programs or online DTD/Schema generators. To begin constructing the query for our example question, we first need to load the source schema of the CSM database.

4.2.1 Load file from local filesystem

When we first use the system, we do not have the CSM source schema saved on the server. We will need to upload the schema to the XGI server by choosing the "Source" option from the "Import" menu under the "File" menu. There are two options in "Import": upload either a source schema file or a query schema file. The choice for query schema file is not selectable initially to prevent users from uploading a query schema file without first defining the source schema. Users need to select the "Source" option in the "Import" menu to open the source file import dialog box. The file import dialog box

allows users to browse for the source schema file. Users need to select the targeted schema file and click the “Load Source” button to upload the file from the local filesystem (Figure 11).

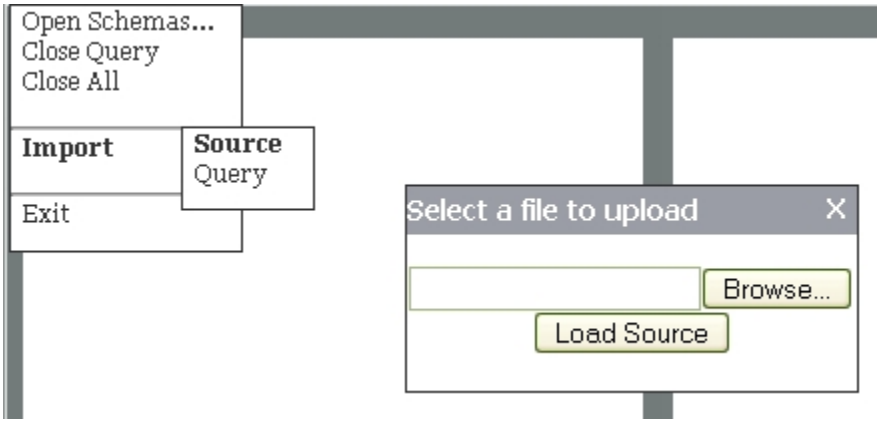

Figure 11. Load source schema file

The file upload option for uploading a “Query” schema will become available immediately after users define the source schema. Once users select the “Query” option, they can use the import dialog box to upload the query schema file. XGI prevents users from defining a query schema that is not derived from the source schema already in the system. When the data model controller adds the schema node to the query schema, it will validate whether a node with that name also exists in the source schema. If the node does not exist, then XGI will flag the query schema as incompatible, prompt users to verify whether the query schema file is correct, and restart the upload process.

4.2.2 Load file from server

If we have used the XGI system before and saved the CSM source schema onto the XGI server, we can reload the schema from the saved schemas. Also if we have created saved query schemas, they can also be reloaded from the XGI server. The list of available saved schemas is displayed under the “Source” of the “Open Schemas” menu in the “File” menu. The list of source schemas is the only list available to users in the beginning (Figure 12). The list of query schemas will remain empty if users have uploaded a new source schema file or chosen a saved source schema that is not associated with any saved query schemas. XGI will determine from the metadata it has extracted from the saved-schema index files if the saved source schema has any saved query schemas. If the saved source schema is associated with saved query schemas, then the query interface will populate the “Query” menu with the list of selectable names of saved query schemas.

In this example (Figure 12), we have already loaded and saved the CSM source schema as “CSM Public View”. So, we can select the “CSM Public View” element and XGI will load the source schema and construct the source tree.

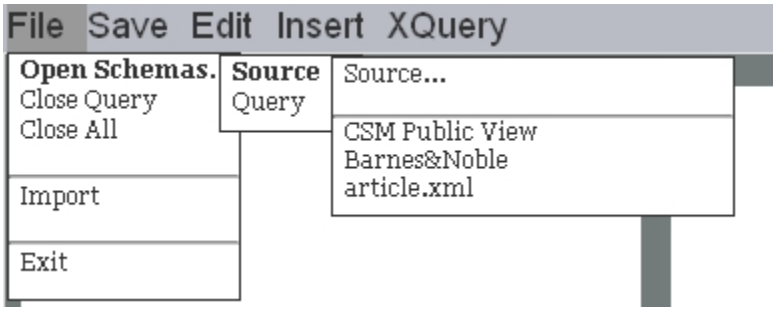

Figure 12. Load saved source schema

4.2.3 Change schemas loaded

Once schemas are loaded onto XGI, users can change the current schemas by either deleting them or defining new schemas. Users can either delete just the query schema or delete both the query and the source schema altogether. Users can remove the query schema by selecting the “Close Query” option under the “File” menu. XGI empties the query tree in the query interface first, and then sends an AJAX request to the data model controller to destroy the query data model as well. If the query schema has saved predicates, deleting the schema will not delete the saved predicates. Because as long as the source schema remains the same, the saved predicates can still be used in any new query schema. The saved predicates will be deleted if users choose to close both the source and the query schema. Users can remove the current source and query schema by selecting the “Close All” option under the “File” menu. XGI will reset the query interface and delete both the source and the query data model.

Users can also define new schemas for XGI without having to close the current schemas. Users can just select a schema they want to load either from saved schemas or from schema files on the local filesystem, then the current schema will be destroyed and the new schema will be created in its place. However, if users need to create a custom query schema, they have to use the “Close Query” option to delete the current schema first before they can define the new query schema.

4.3 Create the query schema

The query panel of the graphical interface allows users to create the query schema, which, if properly formulated, will translate into a valid and useful XQuery statement for the targeted XML data source. Users can add either schema nodes or user-defined nodes to the query tree. Schema nodes in the query tree are added from source tree nodes and they form an implicit mapping edge between them. User-defined nodes enable users to arbitrarily structure the query result and they do not contain any mapping edge to nodes in the source tree. Users can add a user-defined node anywhere on the query tree, as long as its name does not conflict with names of other schema nodes in both the source and the query schemas. Whenever users add a query node to the query schema, the query schema data model is updated first, and then the query interface will update the query tree to display the node to users.

Currently, XGI does not support adding multiple nodes to the query tree all at once. Users need to construct the query tree one element at a time. Users can create the query schema entirely out of user-defined nodes. There will be no mapping edge in this type of schema, but it will still produce a type of valid XQuery query called the empty query. The empty query will return a result that does not contain any data from the source XML document, so the result is not meaningful to users. The reason XGI requires users to load the source schema before they can define the query schema is to encourage users to create the query schema using elements from the source schema. Even though XGI still allows users to create empty queries after the source schema is initialized, users need to

be aware that the only way to retrieve useful information from the source XML data is to map source tree nodes to schemas nodes in the query tree.

4.3.1 Add root node

To construct a unique query schema, users first need to define a root element. The query schema and its data model are initialized as soon as users add a root node to the query tree. Users have two options: they can either add a schema node from the source tree or insert a user-defined node as the query tree root node. The XGI query interface provides separate methods of adding these two types of root node. To add a source tree node as the root, users only need to double click the source tree node. If users want to add a user-defined node as the query tree root node, they need to open the “Insert” menu by clicking the “Insert” menu button. If the query schema is not initialized, the “Insert” menu will have one option, “Root”, for users to select. Selecting the “Root” option will open the “Insert Root” dialog box for users to enter the name of the root node. Once users have entered the name and selected the add button, denoted by the “+” symbol, XGI will create the data model and the query tree with a root node bearing the name that users have entered.

For our example, we will create the “result” outer tag of the intended query result by inserting a user-defined node as the query tree root node.

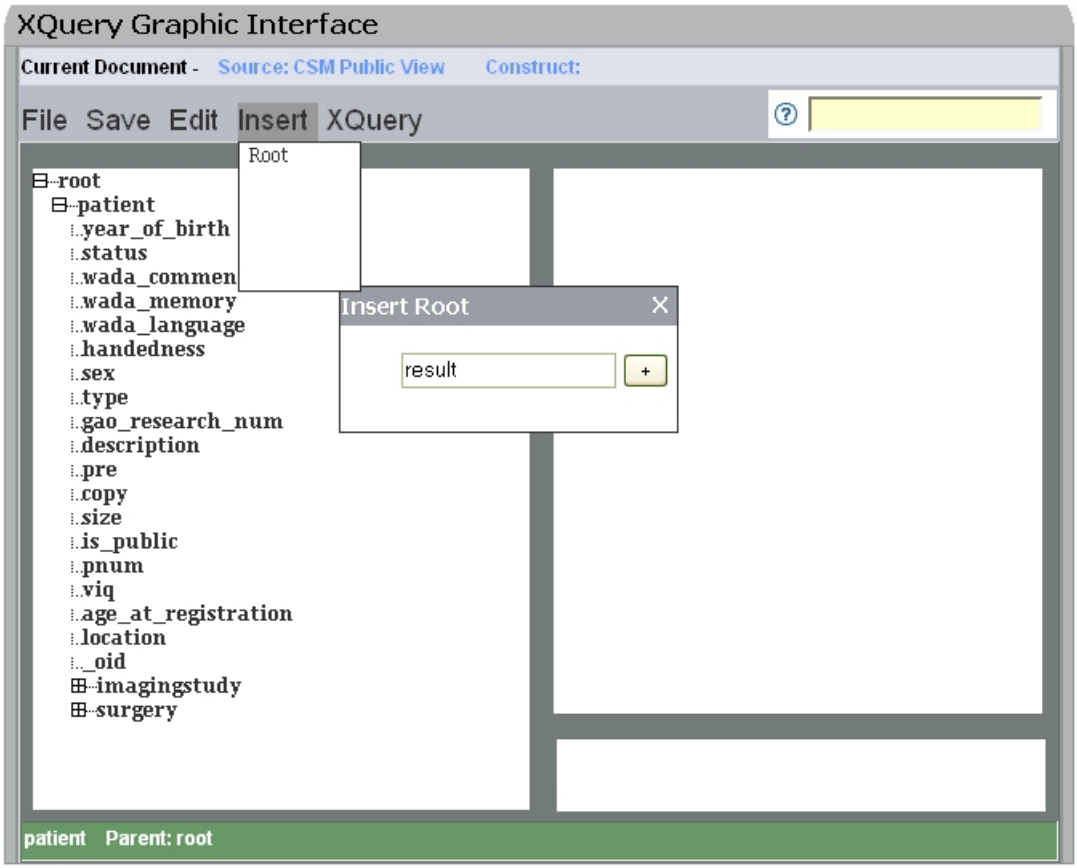

Figure 13. Add root node

4.3.2 Add schema node and user-defined node

To create more sophisticated queries, users need to add more elements to the query schema than just the root. Adding a child element to a query node requires users to select the node first. Once the query node has been selected, users can add a schema node or a user-defined node. If users want to add a schema node, they need to double click the source tree node and it will be added to the selected query node. If users want to add a user-defined node, users can insert the node through the query node's "Children" menu (Figure 11) or through the "Insert" menu. To access the "Children" menu of the query

node, users need to open the node's tooltip menu and select the "Children" option that is situated on the bottom of the menu. The "Children" menu contains an input field for users to type in the name of the child node. After users enter the name, they can select the add button, denoted by the "+" symbol on the right, to add the child node. Users can also insert the user-defined child node through the "Insert" menu. It should be noted that after the query schema is initialized, the "Insert" menu will no longer have the insert "Root" option. Users can select the "Child" option in the "Insert" menu to open up the "Insert Child" dialog box where users need to enter the name and select the add button to add the child node.

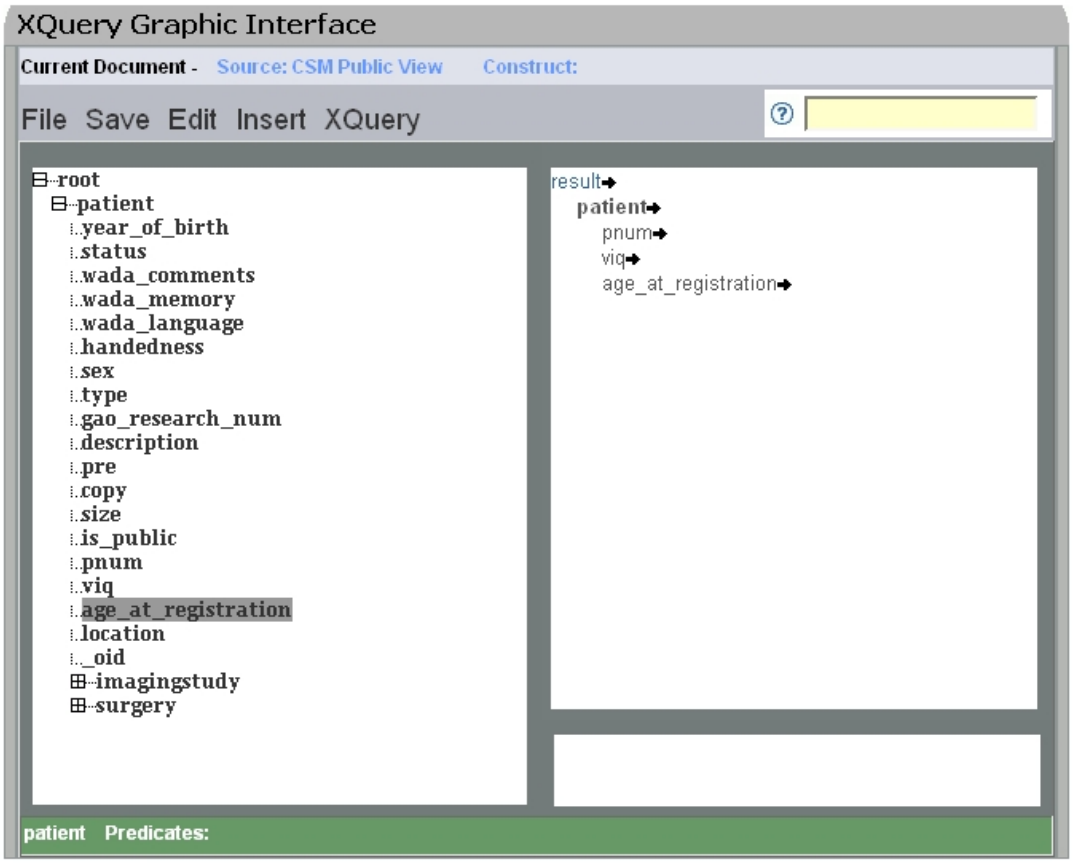

Figure 14. Insert child nodes

For our example, we select the “result” query tree node first to add the “patient” schema node from the source tree. Then we select the “patient” query node and add the “pnum”, “viq”, and “age_at_registration” schema nodes from the source tree node (Figure 14).

Schema nodes and user-defined nodes are added to the query data model through asynchronous requests. XGI will add the node to the query data model first, and then display the node in the query tree. The schema nodes and the user-defined nodes have many differences between them and the major difference is that only the schema nodes have mapping edges. The schema node allows users to change its nodal type and the user-defined node allows users to change its nodal relation. We will explain the meaning of the nodal type and nodal relation in the later “modify the query schema” section. Also, only the schema node has the options to add predicate nodes and attribute nodes. We also explain the procedure to add predicate nodes and attribute nodes to the schema node in the “modify the query schema” section.

Users can create a simple XQuery query by defining a query tree with only schema nodes and user-defined nodes. Also users can use advance functions provided by the query interface, such as adding constraint properties to schema node, to modify the query schema to create more complex queries. Predicates and attributes are two types of properties users can add to the schema node in the query tree. Users can also change the inherent properties of all query nodes to further define the query schema. In the following

sections, we will discuss the common modifications users can make to the schema node and the user-defined node.

4.3.3 Add/delete predicates and attributes

The purpose of schema nodes is to retrieve information from the XML data source. In order to construct complex queries, users need to further define the type of data that schema nodes will return. XGI allows users to define criteria for the returned data through assigning predicate nodes to the schema nodes. Also in order to construct more structurally varied query schemas, XGI allows users to return data as attributes of the schema nodes. Only the schema node has options to open the “Predicate” and the “Attribute” menu in its tooltip menu. Users need to use the “Attribute” menu to add attribute nodes to the schema node. Users can use the “Predicate” menu or the saved predicates to add predicate nodes to the schema node. User-defined nodes are used to provide structural elements in the XQuery statement, so they do not have options to add predicate or attribute nodes.

4.3.3.1 Add/delete attributes

In our sample question, we want to add the sex of the patient as an attribute with the name “id”. To do this, we need to open the “Attribute” menu (Figure 15) in the “patient” schema node’s tooltip menu in order to add an attribute node. The “Attribute” menu contains the attribute dialog box (Figure 15-4) and the attribute fields. The attribute fields are composed of the name text box (Figure 15-2), the assignment operator (denoted by

the “=” sign), and the value text box (Figure 15-3). To construct a new attribute, we need to enter the name of the attribute (“id”) into the name box and then add the source tree node (“sex”) into value box. Once both the name and the value boxes are completed, we select the add button, denoted by the “+” symbol on the right, to add the new attribute node to the schema node.

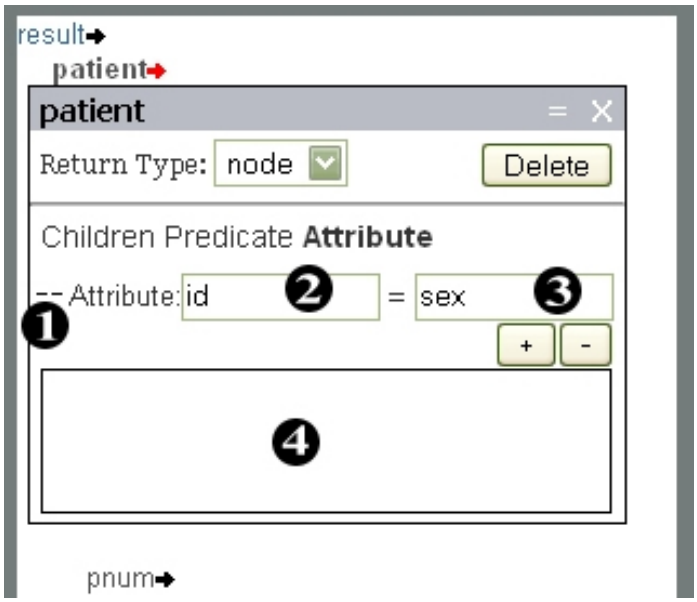

Figure 15. Add/remove attribute node

Users can also add an attribute of a source tree node into the attribute value box. There is a special case in adding the attribute node where users do not need to set both the name box and the value box. If users want to transport the attribute of the source tree node to the corresponding schema node, then only the value box needs to be set with the attribute before users can add the new attribute node to the schema node. For example, suppose a source tree node S has an attribute A and users added S to the query tree as node C. If

users want to transport the attribute A to node C, then users only need to set the value box of node C with A, submit the attribute node, and then node C will have attribute A. If users want to add an attribute from a source tree node to a non-equivalent schema node in the query tree, users need to enter a name into the name box and add the attribute to the value box in order to add the attribute node to the schema node.

To delete an attribute node from the schema node, users first need to open the attribute dialog box, using the "+" symbol on the left of the attribute menu (Figure 15-1), and select an attribute node. Then users can delete the attribute node by selecting the delete button, denoted by the "-" on the right of the attribute menu. XGI deletes the attribute node from the query data model, and then the query interface updates the attribute dialog box by removing the attribute node.

4.3.3.2 Add/delete predicates

There are two ways of adding a predicate node to the schema node: through the predicate menu or from the saved predicates. If there are no saved predicates, users need to open the "Predicate" menu (Figure 16) in the schema node's tooltip menu to add the predicate node. The "Predicate" menu contains the predicate dialog box (Figure 16-5) and the predicate fields. The predicate fields consist of the constraint box (Figure 16-2), the operator box (Figure 16-3), and the value box (Figure 16-4). The constraint and the value box can contain various types of value. They can both be any pairing of source tree node and attribute of a source tree node, although only the value box can contain an

alphanumeric value. To add a source tree node or an attribute to either the constraint or the value box, users need to select the box and then double click the node or the attribute in the source tree. The operator box contains the following logic operators: “==”, “>”, “<”, “>=”, and “<=”. To add a predicate node, users can select the add button, denoted by the “+” symbol on the right, to add the predicate node to the schema node.

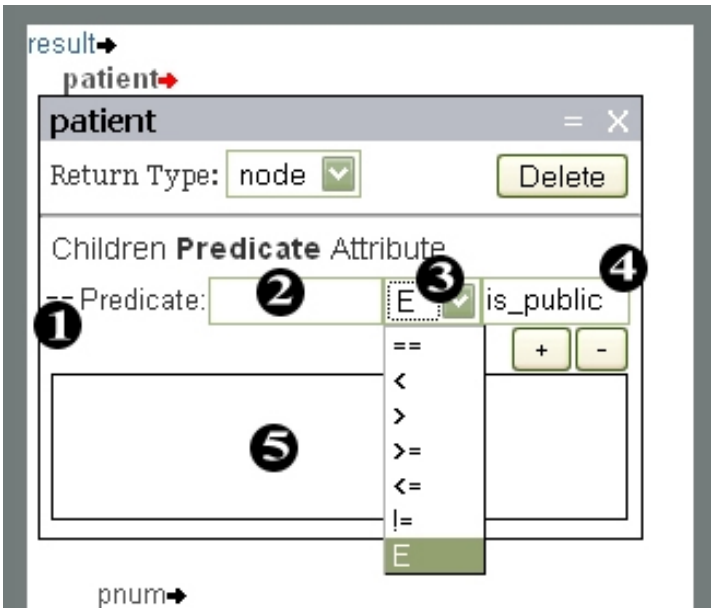

Figure 16. Add/remove predicate node

Figure 16 demonstrates one of the predicates we are going to add to our sample query. From our example, we need to constrain the “patient” schema node by placing two predicates on it: one, test whether the patient’s data is public, and two, find out if the patient’s brain is stimulated during the surgery. Figure 16 shows how to add the predicate to test whether the patient’s data is public.

The predicate node that tests for whether an element exists in the source schema is a special case where users do not need to set all of the predicate fields. The special predicate node implements XPath's exists() function. To use this special predicate node, we need to leave the constraint box empty, set the operator to "E", and then add a source node ("is_public") into the value box. Users can also add an attribute of the source node to the value box.

Figure 17. A predicate is saved

Once users have added a predicate node to the schema node, the predicate node is stored as a saved predicate and displayed in the saved predicate panel (Figure 17). If users want to use the saved predicate for another schema node, users need to select that schema node and double click the saved predicate to add the predicate node to the new schema node.

To delete a predicate node from the schema node, users need to open the predicate dialog box, denoted by the "+" on the left of the "Predicate" menu (Figure 16-1), and select the predicate node. Then users need to select the delete button, denoted by the "-" symbol on the right, to remove the predicate node. XGI will remove the predicate node from the query data model and the query interface. However, the predicate node's corresponding saved predicate is not deleted. Currently, XGI does not allow users to delete the saved

predicates.

4.3.3.3 Search for source schema element

To add the second predicate from our example where we need to find out if the patient's brain is stimulated during surgery, we will utilize the search function of XGI to find the "stimulated" source element.

XGI allows users to search for source schema elements using a query string. This is a unique feature of XGI and no other graphical query tools we have reviewed have it. The ability to search can minimize the amount of effort users have to spend to manually find specific source schema elements. The search function is especially useful to find elements in XML databases, which are becoming larger and more complex in the biomedical research domain. For example, the XML schema of the CSM data contains 474 elements. The schema also contains multiple elements with the same name and they can be situated at different depths and different branches of the source schema hierarchy. So if users want to add a specific element to the query schema, but they only know its name and not where it is buried in the source schema, they might have to browse through the entire source tree to find it. Searching can eliminate this tedious and time-consuming task.

The first time users select the search box, the search result box will appear directly below the search box. (Figure 12) The search function is initiated as soon as users begin to type

into the search box. The query string does not have to be the full name of the source schema element. It can just be a few beginning characters of the element's name, as long as the query string is over two characters long. The query string is sent to the XGI server asynchronously as soon as users stop typing. XGI will check the list of names of all source schema elements against the query string to determine if any schema element's name entirely matches to or begins with the query string. Then XGI will compile the list of source schema elements that match the query string and send the result back to the query interface. Finally, the result is constructed and displayed to users in the search result box. If users modify the query string, by either adding or deleting characters, the query string is resent to the XGI server. This process is made to be dynamic so users feel as if they are receiving feedback from the system almost instantaneously.

Result nodes in the search result box are not just names of source schema elements that match the query string. They are also pointers to the source tree node with the same name. Users can add any result node to as a child schema node to the selected query node, because when users add the result node, XGI is actually adding the result node's corresponding source tree node to the selected query node. To distinguish result nodes that might have the same name, XGI will display the result node's path information in the information panel when users position the mouse over the node (Figure 18). The path information of the result node, retrieved through AJAX request, is the path from the source tree root to the result node's corresponding source tree node. Users can distinguish multiple source schema elements with the same name by their different paths.

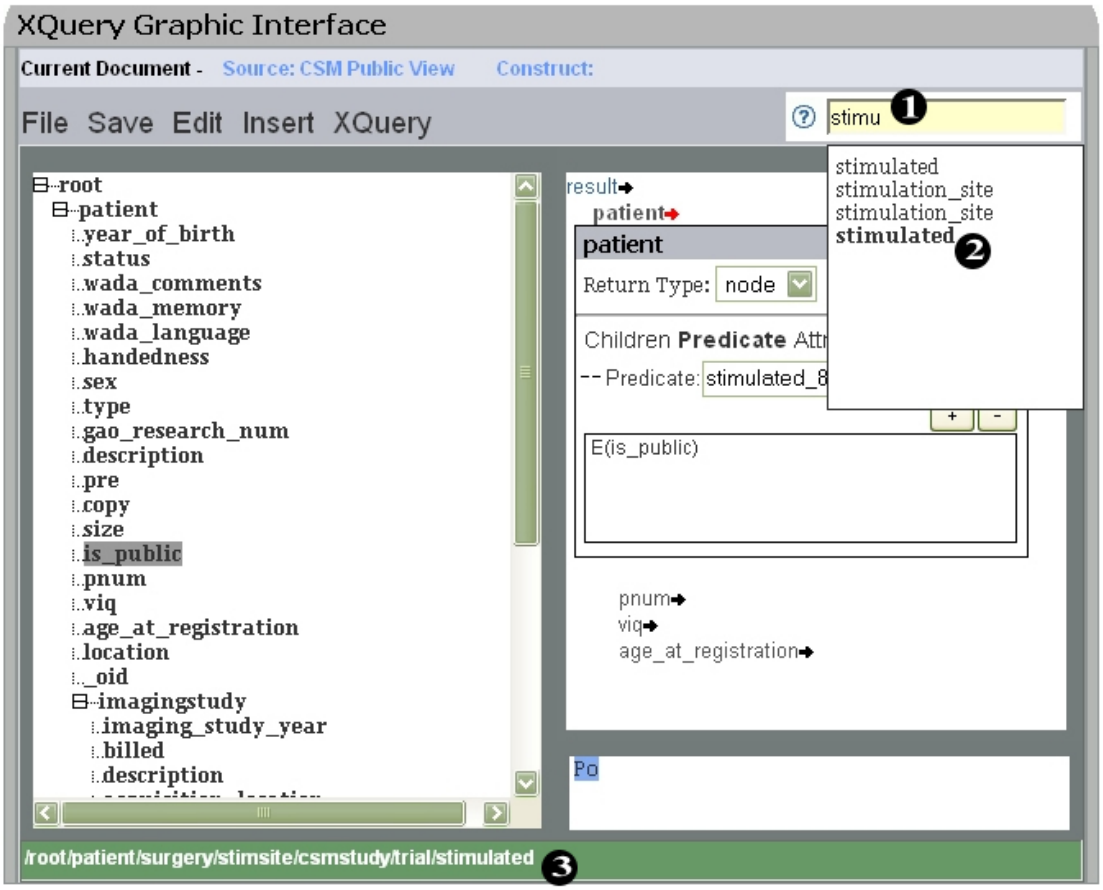

Figure 18. Search for schema elements

So for our example, we utilize the search function to find the source schema element "stimulated". We entered just the beginning few characters ("stimu") of the node's name into the search box (Figure 18-1). The search function automatically retrieves all the source elements that begin with "stimu" and display them in the search result box (Figure 18-2). We find the "stimulated" element we want by examining its path in the information panel (Figure 18-3). The "stimulated" element we need is under the "trial" element, so we add this element to the constraint field of the "Predicate" menu. We also

set the value box to be “Y” to represent that the trial (surgery) is stimulated. And finally, we add this predicate to our query schema.

4.3.4 Change name of the node

The last modification we will make to complete our query schema is to change the name of “age_at_registration” to “age”. Users can change the name of the query node so the same query result will be returned under a different tag name. The name of both the schema node and the user-defined node can be changed. For the schema node whose name is changed, it still maintains the mapping edge to the source tree node. To rename the node, users need to open the node’s tooltip menu and select the node’s name on the top left corner. The name of the node will be replaced by a text field for users to enter the new name. Once users finish entering the new name or decide not to change the name, users can close the text field by clicking on the space outside of the text field. We click on the “age_at_registration” schema node’s name to open the text field where we can change the name to “age”. Then once we click outside of the text field, the name of the schema node is changed to “age” automatically.

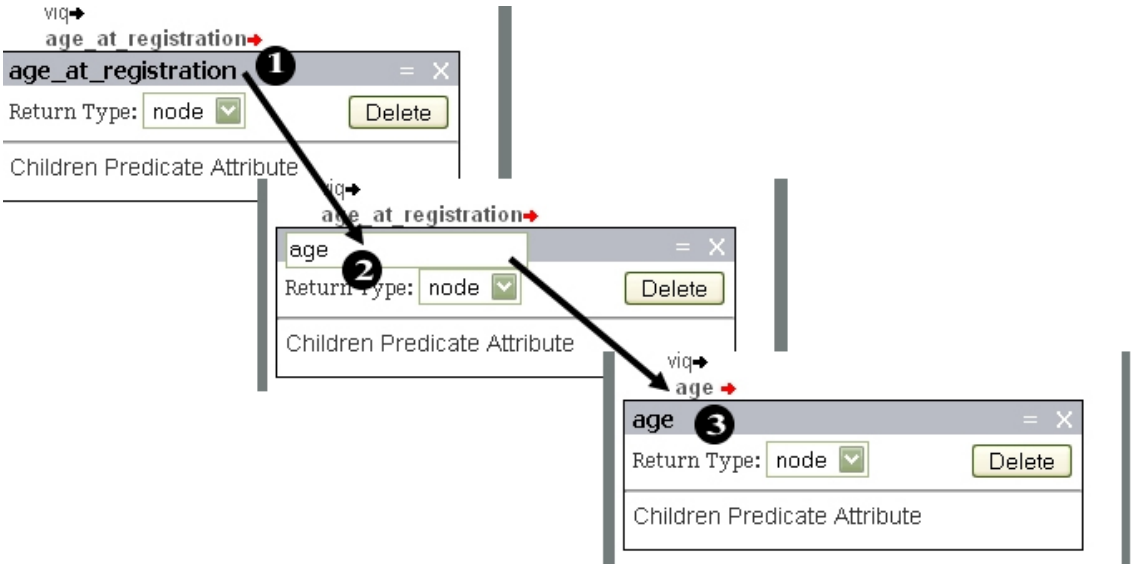

Figure 19. Change node's name

XGI distinguishes if users have entered a new, non-empty name or have left the name unchanged. If the name is not changed, then the text field will be replaced by the previous name. If users have entered a new, non-empty name, then the query interface will send the new name asynchronously to the XGI server. The data model controller will update the node's name in the query data model, and then the query tree will remove the text field and replace it with the name that users have selected entered.

4.3.5 Other schema modifications

After we add the appropriate schema nodes and user-defined nodes, add attribute and predicate nodes, search for schema elements, and change the name of the "age_at_registration" schema node, we have completed the intended query schema construction for our sample question. XGI's functions to modify the query schema is not

limited to the functions we have demonstrated, so in this section we will detail other functions of XGI that we have not used in our query schema construction.

4.3.5.1 Delete node

There are two ways to remove a node from the query schema. One, users can use the query node's self-delete button from the node's tooltip menu to remove it from the schema. Users can also delete a query node from the children dialog box of its parent. The children dialog box is opened by the "+" icon on the left of the parent's "Children" menu. The children dialog box contains references to all the node's children. Each child reference is also added to the parent's children dialog box when the child node is added to the query tree. Users can delete the child node by selecting the reference node in the dialog box, then clicking the delete button, denoted by the "-", to remove the child node and its children.

Users can delete any query node using either of these two options with only one exception. The root of the query tree does not have any parent, so it only can be deleted through the self-delete button. When a query node is deleted, an asynchronous request is sent to the XGI server to remove the node and all of its children from the query data model. Then the query interface will update the query tree by removing the node and its children.

4.3.5.2 Change other node properties

The schema node and the user-defined node each has a unique nodal property users can modify. The user-defined node has the option to change its node relation property (Figure 20-1). The default choice for all user-defined nodes in the query schema is “one-many”. Users can change the node relation property of the user-defined node to “one-one” in the node’s tooltip menu. The “one-many” and the “one-one” node relation dictate how the tag generated by the user-defined node in the XQuery statement will enclose the node’s children elements. If the user-defined node’s node relation is “one-many”, then there will be only one outer tag to enclose all of the node’s children elements. If the node relation is “one-one”, then there will be one outer tag for each of the node’s children element.

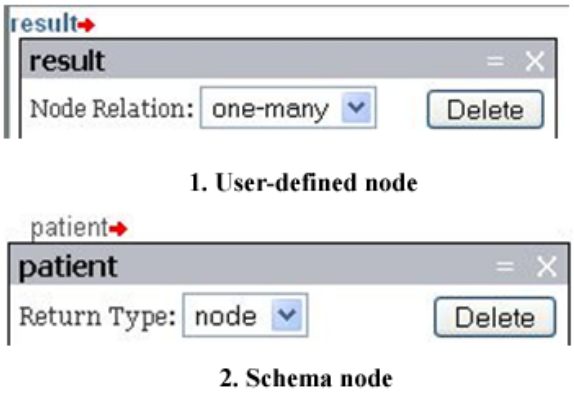

Figure 20. Node relation and node type

The schema node has the option to change its node type property (Figure 20-2) in the node’s tooltip menu. The schema node can change the node type to “node”, “text”, or

"none". If the schema node's node type is set to "node", which is the default node type, then the XQuery query will return the entire source element (with the element tag and the data contained within tag). If the node type is set to "text", then the query will return only the data contained within the source element. If the schema node's node type is set to "none", then the query will return an empty source element (with the element tag only).

4.4 Create XQuery statement

Once users have constructed the query schema, they can use the XQuery generation engine to translate the schema to the appropriate XQuery query. Users need to select the "XQuery" menu button to initiate the translation process. XGI retrieves the query data model, performs necessary transformations to make the schema conform to the XGI grammar, and finally generates the XQuery statement. The statement will be sent back to the query interface as a text string. XGI will create the XQuery dialog box with the query string inside and display the dialog box over the query interface. Users can modify the XQuery statement in the dialog box, but any change users make to the XQuery statement is not mirrored in the query data model. Users can edit the query statement and use it to query the targeted XML data source.

From the query schema we have constructed for the sample question in Table 4, we generate the XQuery statement (Figure 21).

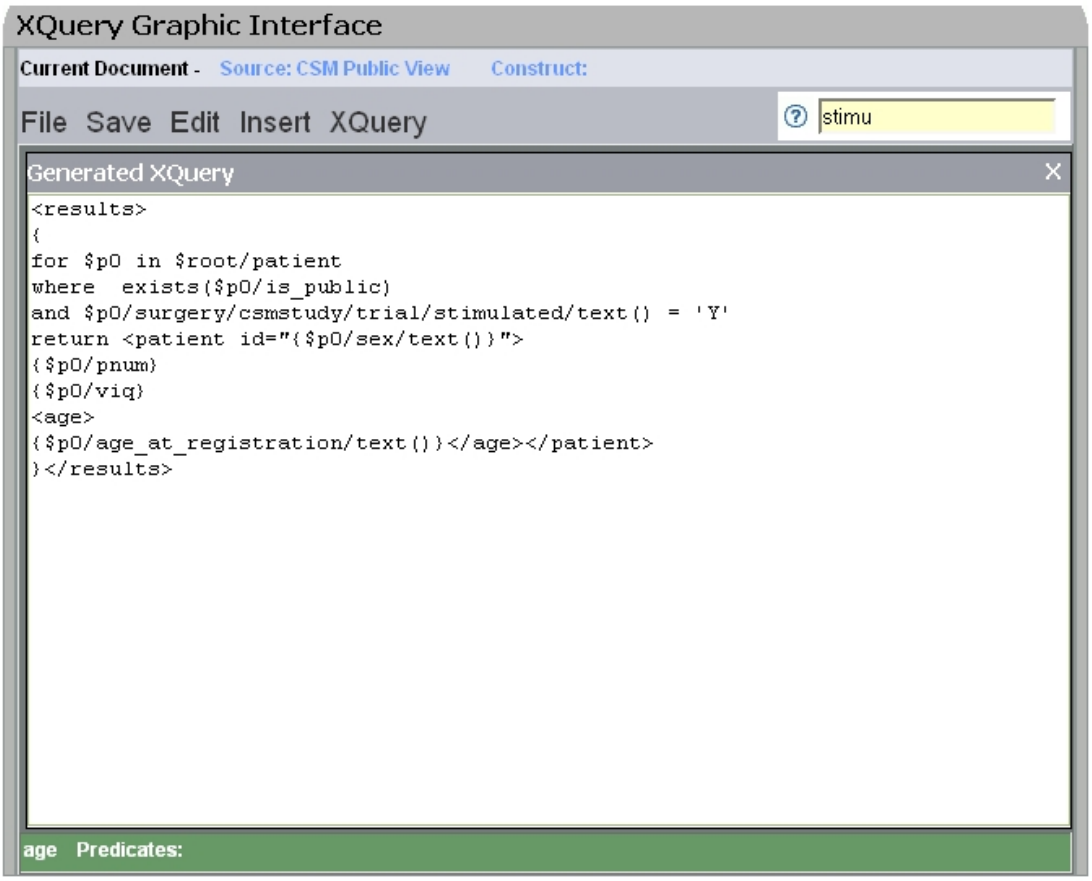

Figure 21. Generated XQuery statement

4.5 Save schemas

A typical query session begins with users defining the source and the query schema, changing the query schema to generate the appropriate query, and ends with users saving the source and/or the query schema onto the XGI server for later use. To save a schema, users need to open the “Save” menu and select the options of either “Save Source As..” or “Save Query As...” (Figure 22). Then, the saved schema dialog box will be opened for users to enter the name of the schema. Finally users select the “Save” button to store the schema onto the XGI server in the saved schema formats.

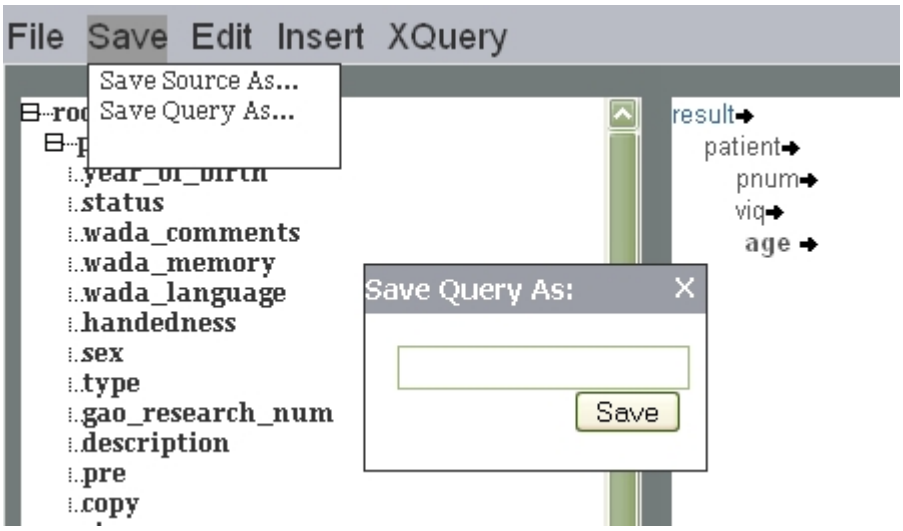

Figure 22. Save query schema

The ability to save query templates is very useful for biomedical researchers. From our experience designing the XBrain system, we found that users prefer to execute queries whose titles describe the query results. For example, in XBrain the query “Semantic Paraphasia” will return information on all patients who have semantic paraphasia. In XGI, users can give saved queries the same type of descriptive titles as XBrain, so biomedical researchers can select the query from the saved schemas and execute the query in the same fashion as XBrain. Saved query schemas also allow users to create generic query schemas. For example, we have observed in XBrain that most of the queries belong to a few query groups, such as content characteristics, temporal characteristics, and miscellaneous diacritics speech error. The queries in each query group differ slightly in their query schemas. Using XGI, users can create a generic query schema for each query group that will encapsulate the common schema elements of all

the queries in the query group. Then users can use the generic schema to create all the queries in the query group and save them with descriptive names onto the XGI server. Using the generic query schema approach, users can create many queries whose schemas do not differ very much from each other very quickly and effectively.

4.6 Information panel and its usage

The information panel is used to display information on various schema elements, such as the source tree node, the query tree node, the search result node, and the saved predicate. For each type of schema element, the information panel will display different information when users place the mouse pointer over the schema element. For the search result node, we have already demonstrated that the information panel can display the node's path information for users to distinguish nodes with the same name in the search result. If the path information is too long and overflow the space in the information panel, users need to select the search result node, and then use the scroll icons (denoted by the "<<" and ">>" symbols on either end of the information panel) to scroll the path information.

The information panel is also used to display nodal information for both the source tree node and the query tree node. The information panel will display the source tree node's name, attributes, and path information. It will also display the query tree node's name, attributes, and predicates. If needed, users can also use the scroll icons to scroll the nodal information after users have selected the query tree node.

The saved predicates are stored in “P#” format in the saved predicate panel. The first predicate node will be saved as “P0”, the second predicate node will be “P1”, and so on and so forth. We save the predicates in this format to prevent the saved predicate panel from overflowing with long predicate statements. If users want to add a particular saved predicate to the schema node, they can examine its predicate statement in the information panel to determine whether the saved predicate is appropriate to add to the schema node (Figure 23).

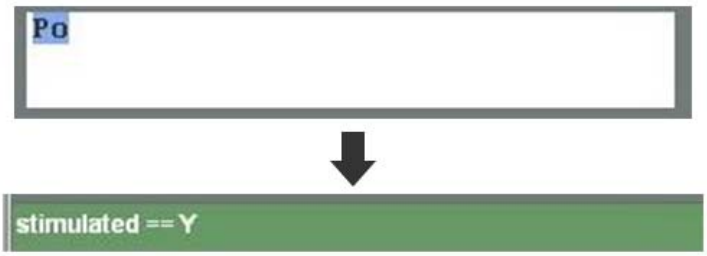

Figure 23. Predicate information

The information panel will revert to display the previous information in the panel once users move the mouse pointer out of the schema element. For example, if users have previously selected a search result node to examine its path information and then later examine a source tree node by hovering mouse pointer over it, the information panel will display the nodal information of the source tree node. But when users move the mouse pointer away, the selected search result node’s path information will return to the information panel.

Chapter 5: Validation and Result

We designed XGI to be a simple and effective graphical query interface for generating XQuery statements. Our goal is to implement the graphical query interface requirements that we have described based on our review of existing informatics solutions in Chapter 2. In order to test our implementation, we have used XGI to create queries for both the CSM database and the generic XML sample document used in the published XQBE paper. The advantage of using the generic XML sample document and the CSM database is that XQBE and XBrain contain an extensive amount of sample queries for two distinct XML data sources. Therefore we can validate whether XGI can generate the same types of queries as these sample queries and determine how well XGI is at recreating them. For the generic XML sample document, we provide a description of the types of queries that XGI can generate and compare these queries against the types of queries XQBE and XQuery are capable of generating. For the CSM database, we discuss how well XGI is able to recreate the saved queries that were constructed by experienced informaticists, listing saved queries that XGI can and can not recreate. We also describe the preliminary evaluation of the process to implement and apply XGI by an actual informaticist.

We realize that XGI might not satisfy all the requirements we have detailed for a graphical query interface. We have included a discussion of the goals that were met and the goals that were unable to be met, and we also discuss the limitations of our system and the compromises we had to make in our implementation.

5.1 Examples of query capability

Table 4. XGI query capability

Feature	XQuery	XQBE	XGI
Existential quantification	Yes	yes	yes
Conjunction	Yes	yes	yes
Breadth projection	Yes	yes	yes
Depth projection	Yes	yes	yes
Renaming	Yes	yes	yes
New element	Yes	yes	yes
Join	Yes	yes	partial
Cartesian product	Yes	yes	partial
Flattening	Yes	yes	yes
Nesting	Yes	partial	no
Filtering	Yes	yes	no
Negation	Yes	partial	no
Aggregates	Yes	yes	no
Arithmetic computations	Yes	yes	no
Sorting	Yes	yes	no
Universal quantification	Yes	no	no
Union	Yes	no	no
Differences	Yes	no	no
Querying schema order	Yes	no	no
Querying instance order	Yes	no	no
Disjunction	Yes	no	no
Grouping	No	no	no

We first compared XGI with XQBE. To demonstrate that XGI can implement the same types of queries that XQBE can generate, we have taken the comparison table and the sample queries given in the XQBE paper and used them to validate our system. In Table 4, we listed the subset of XQuery features that XGI implements. We also used the sample XML document from XQBE (Figure 24) to generate the sample queries. Each example query first notifies the feature the query is implementing, and then it describes the query question, and finally demonstrates the XQuery statement that is generated by XGI.

```
<bib>
<book year="1994">
<title> TCP/IP Illustrated </title>
<author>
<last> Stevens </last>
<first> W. </first>
</author>
<publisher> Addison-Wesley </publisher>
<price> 65.95 </price>
</book>
<book year="2000">
<title> Data on the Web </title>
<author>
<last> Abiteboul </last>
<first> Serge </first>
</author>
<author>
<last> Buneman </last>
<first> Peter </first>
</author>
<author>
<last> Suciu </last>
<first> Dan </first>
</author>
<publisher> Morgan Kaufmann Publishers </publisher>
<price> 39.95 </price>
</book>
</bib>

<!ELEMENT bib(book*)>
<!ELEMENT book        (title, (author+ | editor+),
publisher, price)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last, first)>
<!ELEMENT editor (last, first, affiliation)>
<!ELEMENT title       (#PCDATA)>
<!ELEMENT last        (#PCDATA)>
<!ELEMENT first       (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
<!ELEMENT publisher   (#PCDATA)>
<!ELEMENT price       (#PCDATA)>
```

Figure 24. Sample XML document (http://www.bn.com/bib.xml) and its DTD

Q1. Returning schema element

"*Return all books in the source document*"

```
<result>
        for $b0 in $root/bib/book
        return $b0
```

```
</result>
```

Q2. Existential quantification

"Return all books that have an editor"

```
<result>
        for $b0 in $root/bib/book
        where exists($b0/editor)
        return $b0
</result>
```

Q3. Conjunction

"Return the year and title of all books published by Addison-Wesley after 1991"

```
<result>
        for $b0 in $root/bib/book
        where $b0/publisher == "Addison-Wesley"
          and $b0/@year > 1991
        return <book year= "{$b0/@year}">
                { $b0/title }
                </book>
</result>
```

Q4. Breadth projection

"Return only the title and authors of all books in the source document"

```
<result>
        for $b0 in $root/bib/book
        return <books>
                { $b0/author }
                { $b0/title }
                </books>
</result>
```

Q5. Depth projection

"For each book, return the title and last name of the authors (maintain the books in the same order as the source document)"

```
<result>
        for $b0 in $root/bib/book
        return <book >
                { $b0/title }
                { $b0/author/last }
                </book>
</result>
```

Q6. Renaming

"Return only the title and author of all books in the source document; but change the title tag to 'name'"

```
<result>
        for $b0 in $root/bib/book
        return <book>
                { $b0/author }
                <name>{ $b0/title/text() } </name>
                </book>
</result>
```

Q7. New element

"Return the title and authors of all books, but put the list of authors inside a new 'authors' element"

```
<result>
        for $b0 in $root/bib/book
        return <book>
                { $b0/title }
                <authors>{ $b0/author } </ authors >
                </book>
</result>
```

or

“Return the title and authors of all books, but put the each author inside a new ‘by’ element”

```
<result>
        for $b0 in $root/bib/book
        return <book>
                { $b0/title }
                { for $a1 in $b0/author
                 return <by> { $a1} </by> }
                </book>
</result>
```

Q8. Join, intra-document

“Return all books where the authors have the same last name but different first name”

```
<result>
        for $b0 in $root/bib/book
        where $b0/author/last == $b0/author/last
                and $b0/author/first != $b0/author/first
        return $b0
</result>
```

Q9. Cartesian product, single document and flattening

“Create a flat list of all title-author pair and place them in a result element”

```
<results>
        for $b0 in $root/bib/book
                $a1 in $b0/author
                $t2 in $b0/title
        return <result>
                { $a1 }
                { $a1 }
                </result>
</results>
```

Q10. Set schema element as attribute

"Return the title and price of book. Rename the book element to 'books', place the title in 'books', and place price as attribute of 'books'"

```
<results>
        for $b0 in $root/bib/book
        where exists($b0/price/text())
        return <books price= "{ $b0/price/text() }">
                {$b0/title/text() }
                </book>
</results>
```

We demonstrate the types of queries XGI is capable of generating through the ten example queries. Users can use these features we have described in the example queries to create fairly complex and expressive queries. XGI only implements a subset of the XQuery language, and its features are also encapsulated by XQBE.

We modeled our queries after the example queries presented in the XQBE paper, because it provides the simplest way of validating that the XGI system can generate accurate XQuery statements. From the list of 23 types of queries capable by XQBE, XGI can generate 13 of them. The 10 types of queries XGI cannot generate are the direct results of XGI's limited XQuery features compared to XQBE. XGI does not implement the arithmetic computations, the multiple document-joins, and the hierarchical binding features that XQBE have. These three features account for an additional 6 out of 10 queries that XQBE can generate but XGI cannot.

Although XGI's XQuery implementation compares favorably against XQBE's, XGI does not implement several XQuery features that XQBE fulfills. There are some features of

XQBE that XGI can easily incorporate into its own set of features, such as filtering (or conditional), aggregate, negation, and sort. However, there are some features that will be considerably more difficult to implement with the current XGI graphical interface. This includes nesting (or hierarchical binding) and arithmetic computation. We intentionally limited XGI's feature set to achieve the balance between ease of use and expressivity. By limiting the features that XGI implements, we have created a simpler graphical query interface than XQBE. Also, even in its current state of implementation, XGI enables users to visually create XQuery statements easily and efficiently. Overall, we believe that with more time to develop XGI, the system is capable of being at least as expressive as XQBE.

5.2 Generated queries for the CSM Database

The XBrain system contains some language-error queries (19 in total) and custom queries created by our expert informaticists. We attempted to use XGI to graphically reproduce all of these queries by defining the query schemas so that the queries translated from the schemas are equivalent to the queries written for XBrain. We used both the XGI generated queries and the XBrain queries to query the CSM data base. Then we compared the query results from the XGI generated queries with the original XBrain queries. If the query generated by XGI is exactly the same as the one in XBrain, we indicate it with a 'Yes', noting that XGI can generate that particular query, and thus the queries we generated are equivalent to the queries in XBrain. If XGI is absolutely unable to generate the query, we put down a 'No' and indicate the reasons why XGI cannot

generate the query. If a large portion of the query from XBrain can be generated by XGI but users must edit the query to get the exact XBrain query, then we put down a ‘Partial’ and also indicate the reasons why XGI cannot fully generate the query.

Table 5. Using XGI to recreate saved queries

Saved Custom Queries	**XGI**
Language error queries (19 in total)	Yes
Sites with error codes 2 and 3	Yes (DXQ)
A view with stimsites under trials as a base for other queries. Results for P50	Yes (DXQ)
All trials & sites with error code 2 or 3 and color by viq (green for < 100, red for > 100)	Partial (if... then ... & return text string)
P Number, GAO Number, Exam Number	No (if... then ..., text, hierarchical binding)
Electrode - Sites 20 & 21	Partial (for $a in (20 to 21) return string($a))
Language Error - Sites 20-30	Partial (for $a in (22 to 23)...)
Memory - Sites > 30	Partial (text)
Speech Arrest - Sites 10 and 11	Partial (for $a in (10 to 11)...)
sites 1 to 9 with grid site filtering	Partial (for $a in (1 to 9)..., text)
Semantic (red) vs. Phonological (blue)	No (comment, let, if... then ...)
CSV for Ids, VIQ, Sex, Age, Wada, etc	No (concat function, order)
P50 all mapped sites	Partial (pathname/text() / filename/text())
All patients' trials and sites with Type 2 errors	Yes
P50 all site labels	Yes
All patients' venogram labels	Yes
Pnum, GAO num, Exam num ordered by Pnum	No (let, order, concat, if... then ...)
Patient demographics	No (if...then..., concat)
Average spread of high versus low VIQ	No (declare function, let, if...then...)
All CSM data for a single patient.	Yes
SUR_Summary: Summary view of SUR db	Yes
CSM_with_SUR: CSM patients with SUR data	Partial (let)
csm_inverse: Info under stimsites	Partial (let, hierarchical binding)
csm_inverse_code: csm_inverse filtered by EC	No (let, hierarchical binding, too complex)
csm_inverse_code_fma: add FMA filtering	No (let, hierarchical binding, too complex)
fma Parts of the temporal lobe	No
csm_inverse_code_fma_sur: add a SUR snip	No (let, hierarchical binding, too complex)
Patient counts	Partial (count)
fma_structure_parts: transitive closure in part-of link for an arbitrary structure	No (let, concat)
Preferred photo for P50	Yes
List all patients	Yes
All CSM and MRI info for patients whose pnum is in list	No (concat, DXQ)
All stimsites for patient 50 that are located in the postcentral gyrus (csm-fma query)	Partial (let, DXQ, concat)
Spatial distributed query (compare location of average type 2 vs. 3 error sites)	No (let, DXQ, too complex)
Do males or females have more of a tendency to give semantic paraphasias?	No (let, DXQ, count, too complex)
Do Male or females have a greater naming sites?	No (import, function, if...then...)
Sample SUR query	Yes (DXQ)
Sample transformation query	No
Within a given subject, are there regions that give rise to phonocological reductions that do not also show a semantic error?	No (let, DXQ, function, too complex)
For the 3 cases with morphological errors, what were the intended targets and exact responses?	Yes

XGI is able to recreate language-error queries, and it can also generate a portion of the custom queries (Table 5). XGI is able to fully or partially generate 22 out of 43 custom queries. The main reason that XGI cannot fully recreate some of these 22 custom queries is because XGI does not support some XQuery query constructs and functions, such as “for”, “if…then…”, “concat”, “count”, and “let”, which are in these custom queries. Also, there are 21 saved custom queries that XGI is incapable of recreating graphically. The main reasons are: one, XGI does not support user-defined functions; two, hierarchical binding is not implemented in XGI; and three, sometimes the saved query is just too large and too complex to recreate graphically through XGI. We used the saved schemas feature of XGI extensively because many of these queries share some common portions of the query schemas. Even though there is not one single generic query schema that we can use for all the saved queries, we were able to produce multiple query schemas that we used to generated most of the “Yes” and “Partial” queries.

We have also conducted a preliminary evaluation that includes an expert user of XQuery from the SIG group to use XGI informally. The expert user installed the XGI system, learned the interface, and used XGI to create several queries. In generally, the expert user found that XGI was fairly simple to install, although some specific implementation details of XGI require users to change the system settings and modify the source codes. The expert user found the interface easy to learn and functioned in an expected manner.

However, the expert user did find several features that could have been more easily accessible without having to read additional documentations. These features included searching, uploading schema files from the local filesystem, and accessing the sub-menus in the query node's tooltip menu. The query construction process was easy to execute, but the expert user preferred more features to be included in future implementations of the system. Also, the expert user brought up a suggestion that we had not yet taken into consideration, which was XGI should provide system feedbacks on the limitations of the query interface, such as preventing users from creating invalid query schema by validating the schema while users are constructing it

5.3 Summary of validation results

We have validated the XGI system successfully against an arbitrary XML document and the CSM database. We have demonstrated that for the generic XML document, the XGI system, even though it only implements a subset of the XQuery language, can produce a wide range of useful queries. We also have shown that for the CSM database queries from the XBrain system, XGI can improve the query construction process by allowing users to create generic query schemas and reusing the schemas to create more complex queries.

Table 6. Requirements satisfied by XGI

Requirement	Implemented	Comment
Reduced cost of implementation	Possible	Installation not streamlined
Support for multiple schema formats	Yes	
Navigable source tree	Yes	Useful at browse schema tree
Powerful query creation	Possible	Not as powerful as XQBE
Support collaboration	No	
Easy integration with other applications	Possible	Need to adopt AJAX in new applications

In Table 6, we demonstrate our implementations for the requirements we have described for the graphical query interface. The table indicates that XGI satisfies the requirements for supporting multiple schema formats and creating a navigable tree. XGI's ability to accept multiple schema formats increases its generalizability. Also, from our experience using XGI to recreate queries from XQBE and XBrain and from the evaluation process, the ability to navigate the source schema through an online tree-like interface proves to be very useful.

XGI is easy to install and set up. However, XGI does not have one executable package where users can just select it and it will install and configure the system environments automatically on the web server. XGI can also be integrated with other web-based applications. We did not attempt to incorporate XGI with XBrain by allowing users to use XGI as the main query interface instead of the saved queries pages. However, the modularity of XGI's design will allow us to integrate XGI into XBrain fairly easily in the future by modifying the original XBrain installation to add AJAX functionalities.

From our validation, XGI has demonstrated itself to be a useful graphical query tool for creating XQuery queries. We were able to create different types of queries and use the different features of the system to expedite the query construction process. XGI is able to satisfy most of the requirements for the graphical query interface we have detailed in the preceding chapter. From our validation process, we have discovered several limitations of the system. The first limitation is that XGI does not implement the entire feature set of XQuery and also XGI's features are limited comparing to XQBE. Therefore, XGI is unable to create some advanced queries from XQBE and XBrain. However, XGI does simplify the query construction process by balancing the complexity and the expressivity of the query system.

The second limitation of XGI is that it does not support collaboration. Currently, XGI does not have a login system that can identify different users and enable the individual user to manage his/her queries and schemas data. There are two different approaches to implement the user data management system for XGI. Firstly, we can create a data management system specifically for XGI and distribute it as part of the XGI system software package. However, some existing systems, such as XBrain, already have their own data management system. So if any existing system wants to implement XGI as its query interface, then that system can incorporate its own data management system into XGI. We encourage the second option because it is more flexible to incorporate XGI into other systems.

Lastly, the saved schemas feature of XGI is not as robust as XBrain's. Similar to XBrain, XGI allows users to modify the XQuery statement, such as adding advance query constructs, after the query is translated from the query schema. But the modification is not echoed in the query schema, because XGI does not have visual representations for the XQuery constructs that cannot be created from the query interface. Therefore, only the queries that the XGI query interface can generate are allowed to be saved. For a query that XGI is only able to reproduce partially, users can only save the portion of the query that the query schema would allow. They will not be able to save the more advanced query constructs, so when next time users load the saved query, they will have to enter these advance query constructs again.

Chapter 6: Discussion and Conclusion

The validation and preliminary evaluation helps us understand how well XGI satisfies our selection of requirements and also indicates whether XGI is an effective tool for generating XQuery queries graphically. It is important to evaluate XGI's usability in creating XQuery statements for any arbitrary XML data source and its generalizability in integrating with other applications in the future. The attempt to recreate the queries from XBrain, in particular, has shown the limitations of XGI and graphical querying systems in general. Although XGI implements many features of XQuery, it is unable to capture and duplicate the complexity and the variability of XQuery. Experienced informaticists can formulate very intricate and multifaceted XQuery queries that XGI, and any other graphical querying tools, can not duplicate. In this chapter, we indicate areas where XGI's implementation is insufficient and we provide our ideas on how they can be improved. At the end, we discuss the usefulness and the value of XGI and graphical informatics querying tools in general.

6.1 Future work

The top priority in our plans for future work is to augment and improve the system requirements that XGI does not completely satisfy. We need to further improve upon XGI's straightforward installation process and its compatibility with other querying systems, but user-collaboration and expressive querying ability are the more urgent system requirements we need to implement for XGI in the future.

To satisfy the user-collaboration requirement, XGI needs to be integrated with a data management system. When users save a schema on the XGI server, the schema is currently useable by everyone. The XGI system lacks a permission-based login system to associate specific saved schemas to users or user groups. Ideally, saved queries should only be accessible by the user who saved them. Also, the user should be able to share the queries with other members of the user group. There are many freely available, open source permission-based data management systems we can incorporate into XGI. However, we do not believe that implementing a XGI-specific management system is necessary. If XGI has its own data management system, then it will be difficult to integrate it into other querying systems that already have their own permission-based data management system. So we believe that in order to extract the most benefits from the saved schema feature, XGI needs a data management system, but not necessarily its own data management system.

In our future work, we want to improve XGI's abilities to generate more complex XQuery queries. First, we can implement the same set of features as XQBE by including nesting (hierarchical binding), aggregates, sorting, negation, filtering, and arithmetic computation. Hierarchical binding could be implemented as nodal properties where users can designate a schema node "hierarchically bound". Then the XQuery query generation engine will translate the schema node using the hierarchical binding "//" XPath expression. The negation and filtering features require XGI to modify how the predicate nodes are created. Users can designate a predicate node as "conditional" and they can

also select multiple predicate nodes and add them to a “negation” predicate node.

In XQBE, aggregates and sorting are distinctively separate construct elements. We can implement these features in XGI as special query tree nodes. In the future, we plan to implement the special query tree node as function node. For example, for the “sort” function, the function node will have two parameters in the format of “A is sorted-by B”. Then users can define which schema node in the query schema should be parameter “A” and which schema element should be parameter “B”. The function node also allows XGI to have arithmetic computation. Arithmetic computations can be easily implemented as functions that users can instantiate. The function node also enables XGI to incorporate query constructs, such as “for”, “if…then…”, and “concat”. These query constructs can enable XGI to recreate more of the saved custom queries from XBrain.

There are also many interface functionalities we can improve in XGI, such as support for adding multiple nodes to the query tree all at once, inserting a parent node to the selected query node, using special characters and regular expression in search, and implementing a Query-in-place feature similar to BBQ’s and Stylus Studio’s. We can also improve XGI’s saved schemas system to display queries that do not have an equivalent XGI query schema. We feel that in the future implementation of XGI, users should be allowed to save the manually created queries even if they contain query constructs that the XGI query interface does not support.

XGI also needs to implement the feature that will enable it to incorporate elements from multiple documents. BBQ, XQBE, and Stylus Studio all allow users to combine schema elements from several XML documents. We can add the same type of multi-document browsing and joining features into XGI. The BBQ's multi-document feature, where each document receives its own document browsing window, is the implementation most suitable for XGI. XGI would display each XML document in its own document tab (similar to tabs in a web browser), and then users can toggle between multiple documents by selecting their tabs.

Last but not least, in all of the XML graphical querying tools, only XBrain supports the distributed XQuery model. The distributed XQuery allows users to use one XQuery statement to retrieve information from multiple data sources through web services. XGI can be modified to support the distributed XQuery as well. To accomplish this, XGI first needs to implement the multi-document browsing and querying feature we have described above. So users can select elements from multiple documents to query. Also, XGI should generate two types of XQuery at users' discretion: a simple XQuery for a single XML data source or a distributed XQuery that can query multiple XML data sources.

6.2 Conclusion

XGI is intended to assist both novice end-users and experienced informaticists in creating expressive XQuery queries efficiently through a graphical query interface. XGI, in its

current state, has already demonstrated its usefulness in creating various types of queries for the sample XML document and the CSM database. All the potential areas of future work, from integrating XGI with a permission-based data management system to incorporating function nodes into XGI's feature set, can increase the usefulness and value of the XGI system. However, we have to remember to continually balance between the usability and the expressivity of the system. The more features we incorporate into XGI, the more complex it might potentially become. Informaticists can use XGI to browse the source schema, define the query schema, and visualize the query output graphically. In the end, XGI is inherently limited by its visual querying paradigm. XGI is designed to augment the query construction process, so it should never be a replacement for expert informaticists who are experienced in writing XQuery. We hope in future implementations of XGI that we will continue to make its query interface more accessible and its querying ability more powerful.

Bibliography

1. Augurusa, E., Braga, D., Campi A, Ceri, S. Design and implementation of a graphical interface to XQuery. Proceedings ACM symposium on Applied computing, 2003.

2. Bales, Nathan and Brinkley, James F and Lee, E. Sally and Mathur, Shobhit and Re, Chris and Suciu, Dan. A Framework for XML-based Integration of Data, Visualization and Analysis in a Biomedical Domain. In Proceedings, XSym, pp. 207-221, Trondheim, Norway, 2005

3. Braga, D., Campi, A., Ceri, S. XQBE (XQuery by example): A visual interface to the standard XML query language. ACM transactions on database systems, 2005.

4. Braga, D., Campi, A., Ceri, S. XQBE: A graphical interface for XQuery engines. Lecture notes in computer science, pp 848-850, Springer, 2004.

5. Braga, D., Campi, A. A Graphical Environment to Query XML Data with XQuery. Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003.

6. Bouganim, L., Chan-Sine-Ying, T., Dang-Ngoc, T.T., Darroux, J.J., Gardarin, G., Sha, F. MIROWeb: Integrating Multiple Data Sources Through Semistructured Data Types. In Proc. 25th VLDB Conference, Edinburgh, Scotland, 1999.

7. Carey, M.J., Haas, L.M., Maganty, V., and Williams, J.H. PESTO : An integrated query/browser for object databases. In Proc. of the Int'l Conf. on VLDB, pages 203-214, 1996.

8. Ceri, S. Comai, S. Damiani, E. Fraternali, P. Paraboschi, S. and Tanca, L. XMLGL: A Graphical Language for Querying and Restructuring XML. Proc. of WWW Conf., pp. 93-109, 1999.

9. Chamberlin, D. D. Robie, J. and Florescu, D. Quilt: An XML query language for heterogeneous data sources. In Proceedings of WebDB, 2000.

10. Chawathe, S.S., Baby, T., Yoo, J. VQBD: exploring semistructured data. Proceedings of ACM SIGMOD international conference on Management of data, 2001.

11. Cohen, S., Kanza, Y., Kogan, Y., Nutt, W., Sagiv, Y., Serebrenik, A. Combining the Power of Searching and Querying. Conference on Cooperative Information Systems, 2000.

12. Cohen, S. Kanza, Y. Kogan, Y., Nutt, W. Sagiv, Y. and Serebrenik, A.. EquiX -- Easy Querying in XML Databases. In 2nd ACM SIGMOD Int. Workshop on The Web and Databases, pages 43-48, 1999.

13. Comai, S., Damiani, E., and Fraternali, P. computing graphical queries over XML data. ACM TOIS, 19(4):371-430, 2001.

14. Comai S., Damiani E., Posenato R., Tanca L. A Schema-based Approach to Modeling and Querying WWW Data. In Proc. of FQAS, Roskilde, May 1998

15. Consens, M.P., Mendelzon, A.O. The graphlog querying system. In Proceedings of the ACM SIGMOD international conference on Management of data, page 388, 1990.

16. Cruz, I.F., Mendelzon, A.O., and Wood, P.T. A graphical query language supporting recursion, Proceedings of the ACM SIGMOD international conference on Management of data, 1987.

17. Deutsch, A. Fernandez, M. Florescu, D. Levy, A. and Suciu. D. XML-QL: a query language for XML. W3C Note, 1998.

18. Evangelista-Filha, I. M. R., Laender, A. H. F., and Silva, A. S. Querying Semistructured Data By Example: The QSByE Interface. In Proceedings of the International Workshop on Information Integration on the Web (WIIW'2001), April 9-11, pp. 156-163, Itaipava, Rio de Janeiro 2001.

19. Fernandez, M. Morishima, A. and Suciu, D. Efficient evaluation of xml middle-ware queries. In ACM SIGMOD Conf., 2001.

20. Fernandez, M., Simeon, J., Wadler, P., Cluet, S., Deutsch, A., Florescu, D., Levy, A., Maier, D., Mchugh, J., Robie, J., Suciu, D., Widom, J. XML query languages: experiences and exemplars, 1999.

21. Gehtland, J., Almaer, D., Galbraith, B. Pragmatic Ajax: A Web 2.0 Primer. Pragmatic Bookshelf; 1 edition, 2006.

22. Goldman, R., Widom, J. Interactive query and search in semistructured databases. In Proc. of the First Intl. Workshop on the Web and Databases (WebDB), pages 42-48, March 1998.

23. Gordon, P. XML for Molecular Biology. http://www.visualgenomics.ca/gordonp /xml/, June 2006.

24. Hastings, S., Ribeiro, M., Langella, S., Oster, S., Catalyurek, U., Pan, T., Huang, K, Ferreira, R., Saltz, J., Kurc, T. XML database support for distributed execution of data-intensive scientific workflows. ACM SIGMOD Record, 2005.

25. Ives, Z.G., Lu, Y. XML query languages in practice: an evaluation. Proceedings of the First International Conference on Web-Age Information Management, 2000.

26. Kay, M. XPath 2.0 Programmer's Reference. Wrox, 2004.

27. Kepser, S. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In Proc. Extreme Markup Languages, 2004.

28. Lakshmanan, Laks V. S. Ramesh, G. Wang, H. and Zhao, Z. On Testing Satisfiability of Tree Pattern Queries, In VLDB, page: 120-131, 2004.

29. Lau, H.L. and Ng, W. Querying XML Data Based on. Nested Relational Sequence Model. Proceedings of. Poster Track of WWW, Honolulu, 2002.

30. Levy, A., Rajaraman, A., and Ordille, J. Querying heterogeneous information sources using source descriptions. In Proceedings of the International Conference on Very Large Data Bases, 1996.

31. Ludaescher, B., Papakonstantinou, Y., Velikhov, P. Vianu V. View Definition and DTD Inference for XML. Workshop on Semistructured Data and Nonstandard Data Formats, 1999.

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

33. Papakonstantinou, Y., Petropoulos, M., Vassalos, V. Generating Query Forms and Reports for Semistructured Data: The QURSED Editor. Ninth Panhellenic Conference on Informatics (PCI), 2003

34. Papakonstantinou, Y., Petropoulos, M., Vassalos, V. QURSED: Querying and Reporting SEmistructured Data. ACM International Conference on Management of Data (SIGMOD), 2002

35. Petropoulos, M., Papakonstantinou, Y., Vassalos, V. Graphical query interfaces for semistructured data: the QURSED system. ACM Transactions on Internet Technology (TOIT), 2005.

36. Petropoulos, M., Papakonstantinou, Y., Vassalos, V. Building XML Query Forms and Reports with XQForms. Computer Networks Journal, Special Issue on XML, vol 39/5, pp 541-558, 2002

37. Quass, D. Rajaraman A., Sagiv, Y. Ullman, J. D. and Widom, J. Querying semistructured heterogeneous information. In Fourth International Conference on Deductive and Object- Oriented Databases (DOOD'95), Singapore, 1995.

38. Pittendrigh, S., Jacobs, G. NeuroSys: A Semistructured Laboratory Database. Neuroinformatics, Volume 1, Number 2, pp. 167-176 June 2003

39. Ramakrishnan, R., Gehrke, J. Query-By-Example (QBE), Database Management Systesms, Third Edition 1999.

40. Re, C., Brinkley, J.F., Hinshaw, K.P., Suciu, D. Distributed XQuery. In Proceedings, Workshop on Information Integration on the Web (IIWeb), Toronto, 2004.

41. S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 497-508, 2001.

42. S. Abiteboul, L. Segoufin, and V. Vianu. Representing and Querying XML with Incomplete Information. In Proc. of ACM PODS, 2001.

43. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi and L. Tanca, XMLGL: A Graphical Language for Querying and Restructuring XML. Proc. of WWW Conf., pp. 93-109, 1999.

44. Shanmugasundaram, J. Shekita, E.J. Barr, R. Carey, M.J. Lindsay, B. G. Pirahesh, H. Reinwald, B. Efficiently publishing relational data as XML documents. VLDB Journal 10(2-3): 133-154 (2001).

45. Tang, Z., Kadiyska, Y., Li, H, Suciu, D and Brinkley, J.F. Dynamic XML Based Exchange of Relational Data: Application to the Human Brain Project. In Proceedings, American Medical Informatics Association Fall Symposium, pp. 649-653, 2003

46. Tang, Z. and Kadiyska, Y. and Suciu, Dan and Brinkley, James F. Results Visualization in the XBrain XML Interface to a Relational Database. In Proceedings, MedInfo, pages 1878, San Francisco, CA, 2004

47. W3C. XPath Tutorial. http://www.w3schools.com/xpath/default.asp, November 2005.

48. W3C. XQuery Tutorial. http://www.w3schools.com/xquery/default.asp, November 2005.

49. Wong. R.K. and Shui. W.M. Utilizing Multiple Bioinformatics Information Sources: An XML Database Approach, IEEE International Symposium on Bio-Informatics and Biomedical Engineering, 2001.

50. Zakas, N.C., McPeak J., Facwett, J. Professional Ajax. Wrox, 2006.

51. Zloof, M.M. Query-by-Example: a data base language. IBM Systems Journal, 16(4):324-343, 1977.

52. Stylus Studio Website. http://www.stylusstudio.com. August, 2006.