ABSTRACT

Today there is a plethora of data accessible via the Internet. The Web has greatly simplified the process of searching for, accessing, and sharing information. However, a considerable amount of Internet-distributed data still goes unnoticed and unutilized, particularly in the case of frequently-updated, Internet-distributed databases. In this paper we give an overview of WebFormulate, a Web-based visual continual query system that addresses the problems associated with formulating temporal ad hoc analyses over networks of heterogeneous, frequently-updated data sources. The main distinction between this system and existing Internet facilities to retrieve information and assimilate it into computations is that WebFormulate provides the necessary facilities to perform continual queries, developing and maintaining dynamic links such that Web-based computations and reports automatically maintain themselves. A further distinction is that this system is specifically designed for users of spreadsheet-level ability, rather than professional programmers.

Categories and Subject Descriptors
D.1.7 [Programming Techniques]: Visual Programming

General Terms
Languages

Keywords
Continual query, visual query system, visual programming language

1. INTRODUCTION

Today there is a plethora of data accessible via the Internet. The Web has greatly simplified the process of searching for, accessing, and sharing information. However, a considerable amount of Internet-distributed data still goes unnoticed and unutilized, particularly in the case of frequently-updated, Internet-distributed databases. Four infrastructural constraints contribute to this problem: (1) the Internet-distributed computing world was developed primarily for the one-time distribution of information rather than the continuous flow of process communication, (2) the heterogeneous nature of database systems makes it difficult to have a single user interface to databases that utilize different connectivity protocols, schema metadata, and SQL syntax, (3) the standards infrastructure for data exchange is underdeveloped, and (4) “public programmers” (i.e., users of spreadsheet-level ability who are untrained as professional programmers) lack the necessary skills to access, query, and analyze data from heterogeneous, Internet-distributed databases.

For example, suppose that a biologist in California is researching the apparent disappearance of the plains leopard frog (Rana blairi) in Douglas County, Kansas. She suspects that it might be due to predation by or competition with the bullfrog (Rana catesbeiana). To investigate this problem, each week for nine months graduate students from a Kansas university go to areas throughout Douglas County to count the number of the two species of frogs, and enter the information into a database. The biologist would like to graph the counts of the frogs (such as the graph in Figure 1) to monitor the population changes in that area over the nine-month period. The biologist would also like to share this information (as it is updated) with his/her colleagues located throughout the world.

Figure 1. A graph to analyze the populations of two species of frogs.

In Specify\(^1\), a biological collection database that is typically used to record specimen locality information of this nature, the actual SQL statement for such a query would be the following:

\[
\text{SELECT TaxonName2.FullTaxonName, Locality.Latitude, Locality.Longitude FROM (CollectionObject) INNER JOIN ((CollectionObject AS CollectionObject2) INNER JOIN Specify1 AS CollectionObject2) INNER JOIN Specify1 AS CollectionObject3) AS CollectionObject4 \}
\]

\(^1\) http://www.usobi.org/Specify
and discuss how this system addresses the previously mentioned

In this paper we give an overview of the

than professional programmers.

specifically designed for users of spreadsheet-level ability, rather

themselves. A further distinction is that this system is

Web-based computations and reports automatically maintain

continual queries, developing and maintaining dynamic links such

WebFormulate

information and assimilate it into computations is that

analyses of Internet-distributed data. The main distinction

that facilitates the development of graphical and statistical

The biologist could more easily address this research problem

perseverance) to perform this procedure 36 times.

In addition, it would require that the biologist had the time (and

connection, what database driver to use, etc.),

connect to a remote database (knowing how to make the

construct a complex SQL statement (knowing the names

of the tables and fields, and the relationships between

tables in order to construct the appropriate joins),

download the query results,

import the data into a spreadsheet (assuming the data

are in a format that the spreadsheet program can interpret) and construct a graph, and

make available to each of his colleagues the updated spreadsheet or a static image of the graph via email or a

Web page.

In addition, it would require that the biologist had the time (and perseverance) to perform this procedure 36 times.

The biologist could more easily address this research problem using WebFormulate, a Web-based visual continual query system that facilitates the development of graphical and statistical analyses of Internet-distributed data. The main distinction between this system and existing Internet facilities to retrieve information and assimilate it into computations is that WebFormulate provides the necessary facilities to perform continual queries, developing and maintaining dynamic links such that Web-based computations and reports automatically maintain themselves. A further distinction is that this system is specifically designed for users of spreadsheet-level ability, rather than professional programmers.

In this paper we give an overview of the WebFormulate system and discuss how this system addresses the previously mentioned infrastructural constraints to formulating temporal ad hoc analyses over a network of heterogeneous, frequently-updated data sources.

2. BACKGROUND AND RELATED WORK

2.1 Visual Query Systems

Visual query systems (VQS) [1, 2] are typically designed for users with limited technical skills. The technology for the construction of VQS is generally well researched and the choice of approach is primarily one of matching other language concepts to provide a clean, conceptually consistent interface. Adaptive systems such as the one developed by [3] even allow the user to select from several visual representations and interaction mechanisms for expressing the query and visualizing the results.

Like WebFormulate, many visual query systems provide a Web-based user interface. The primary differences between WebFormulate and other visual query systems are:

(1) Most VQS have been developed specifically to query geographic and image databases, a more restricted problem domain than that of the WebFormulate research.

(2) The WebFormulate query interface was designed to be consistent with an existing form-based visual programming language interface which had been designed for users of spreadsheet-level ability, and which had the necessary underlying evaluation model to make it extendable to continual query processing with automatic updating of visual and computational objects dependent upon query results.

(3) To date, research in VQS and continual queries has not been successfully integrated. Unlike VQS, user interfaces for most continual query processing systems are not designed for non-programmers. Typically the user is required to know the names of tables and fields in the database, to be able to construct SQL statements, and to use other applications to analyze query results.

2.2 Continual Query Systems

A continual query (CQ) as defined by [4] "...is a standing query that monitors updates of interest using distributed triggers and notifies the user of changes whenever an update of interest reaches specified thresholds or some time limit is reached." It is expressed in terms of a normal, SQL-like query, a trigger condition, and a stop condition. Some continual queries may also include a start condition and a notification condition (i.e., when the condition for notification of results is different from the trigger condition). The concept of continual queries was first introduced by [5] for append-only databases.

Active database management systems [6, 7, 8, 9, 10] are restricted implementations of continual query systems. Unlike the traditional, passive, program-driven database management

((Determination AS Determination2) INNER JOIN
(TaxonName AS TaxonName2) ON
TaxonName2.TaxonNameD =
Determination2.TaxonNameD) ON
Determination2.BiologicalObjectD =
(CollectionObject2.CollectionObjectD) INNER JOIN
((CollectingEvent AS CollectingEvent2) INNER JOIN
(Locality) ON Locality.LOCalityID =
CollectingEvent2.LOCalityID) ON
CollectingEvent2.CollectingEventID =
CollectionObject2.CollectingEventD) ON
CollectionObject2.CollectionObjectD =
CollectionObject.CollectionObjectD WHERE
(((TaxonName2.FullTaxonName IN('Rana blairi', 'Rana catesbeiana'))) AND (((Locality.Latitude BETWEEN 37.8 AND 39.4 AND Locality.Longitude BETWEEN 94.8 AND 96.0))))

To monitor the results of this study weekly over the nine-month period, the researcher would have to repeatedly run this query on the database, each time downloading and importing the latest results into his/her analysis application (perhaps a spreadsheet program) to regenerate the graph. In reality, it is unlikely that a biologist would have all the requisite skills to perform this activity, namely to:

(1) connect to a remote database (knowing how to make the connection, what database driver to use, etc.),

(2) construct a complex SQL statement (knowing the names of the tables and fields, and the relationships between tables in order to construct the appropriate joins),

(3) download the query results,

(4) import the data into a spreadsheet (assuming the data are in a format that the spreadsheet program can interpret) and construct a graph, and

(5) make available to each of his colleagues the updated spreadsheet or a static image of the graph via email or a Web page.

In addition, it would require that the biologist had the time (and perseverance) to perform this procedure 36 times.

The biologist could more easily address this research problem using WebFormulate, a Web-based visual continual query system that facilitates the development of graphical and statistical analyses of Internet-distributed data. The main distinction between this system and existing Internet facilities to retrieve information and assimilate it into computations is that WebFormulate provides the necessary facilities to perform continual queries, developing and maintaining dynamic links such that Web-based computations and reports automatically maintain themselves. A further distinction is that this system is specifically designed for users of spreadsheet-level ability, rather than professional programmers.

In this paper we give an overview of the WebFormulate system and discuss how this system addresses the previously mentioned infrastructural constraints to formulating temporal ad hoc analyses over a network of heterogeneous, frequently-updated data sources.
systems, active database management systems are data-driven. They actively monitor the arrival of desired information and provide it to the interested users as it becomes available. However, such systems often depend heavily on extensions specific to the database management system such as the built-in triggers in Informix [11]. Another limitation is that the trigger mechanisms of many of these systems will only work on active tables (i.e., append-only tables in which existing records are never updated and new records are appended to the end of the table). Active database management systems are simply not an efficient or scalable solution for a large number of concurrently running continual queries on a variety of Internet-distributed data sources.

CONQUER [4], OpenCQ [12], NiagaraCQ [13], and CQPD [14] represent the most extensive work to date on distributed, event-driven continual query systems that allow the specification of time-based or content-based trigger conditions. Prototypes of these systems have been developed for some small databases with very simple underlying data models (e.g., for monitoring weather conditions, stock prices, and bibliographic references). Although these systems are an improvement over the active-database approach, they have several issues that make their utilization problematic for widespread use.

One problem with some of these systems is that they claim portability simply because they are written in Java. Yet these Java-based systems rely upon platform-dependent resource management facilities (e.g., the UNIX cron facility, the Windows NT process scheduler, etc.). Such implementations do not have true portability, only interoperability through their Web front ends. By limiting a system to a particular platform, the benefits of utilizing the Java language are diminished.

Another problem with many existing continual query systems is the lack of external scalability (i.e., in terms of handling traffic from a large number of clients). By relying on platform-dependent resource management, and customized communication and dissemination protocols, these systems will likely suffer in performance when subjected to Internet-scale loads2.

Existing implementations of continual query systems lack not only external scalability, but also configurability and internal scalability (i.e., the ability to easily modify, expand, or contract the implementation). Although most of these systems have been implemented in the object-oriented Java language, the components are tightly integrated and require full deployment of all system components. This requires that an organization using one of these systems conform to a specific set of guidelines for the underlying storage and processing needs of the system. The proprietary, closed-source nature of such systems does not allow the modifiability and configurability that one might need to tailor for a specific processing environment; systems that lack internal scalability do not allow the system administrator to select which components to deploy and which available features to provide.

In addition, some of these systems have implemented their own database systems and query languages. It is unclear why the developers of these systems have chosen not to use off-the-shelf products that serve these purposes. The utilization, wherever possible, of reliable and highly stable components that are already available, would seem to more closely support the object-oriented paradigm of reuse of components.

To overcome the portability and scalability problems of other continual query systems, the WebFormulate continual query processor utilizes a distributed object infrastructure for heterogeneous enterprise computation of continual queries. A detailed discussion of the architecture of the continual query processor utilized by WebFormulate is beyond the scope of this paper; see [15] for a complete description of that system.

Yet another problem with existing continual query systems is that they do not provide integrated query and analysis functions that are usable and modifiable by non-programmers. For example, the user interfaces for these systems require that the user know the names of the database tables and fields to express the query. The NiagaraCQ system further requires that the user be able to express the query as an XML-QL [16] text file. In the CONQUER and OpenCQ systems users are notified of updated query results by email, and can then download a text file of the results or view the tabular results on a Web page (which must be created by the system developers, not the end-user). If the user wants to computationally or visually analyze the data further, s/he must download the text file containing the query results each time there is an update notification and import the data into another application such as a spreadsheet. The NiagaraCQ system allows the user to specify an action to be performed on the query results. However, such actions must be expressed as low-level system calls such as the "MailTo" UNIX command.

In summary, prior to WebFormulate, there has yet to be developed a Web-based visual query system, specifically designed for non-programmers, that can perform continual queries on internet-distributed databases, developing and maintaining dynamic links such that Web-based, user-specified computations and reports automatically maintain themselves.

2.3 The Formulate Visual Programming Language

WebFormulate is not simply an end-user application; it is a form-based visual programming language that allows end-users to construct programs in a Web browser environment. WebFormulate is based on an existing implementation of a visual programming language called Formulate [17, 18, 19, 20, 21, 22], which is the product of many years of visual and public programming research, and has been tested against users with little prior programming experience [19]. WebFormulate extends the functionality of the Formulate system by providing a Web-browser user interface and

---

2 This conclusion is based on the small-scale, limited performance results that have been reported for these continual query systems.
supporting continual query processing of distributed databases. A complete description of the Formulate language is beyond the scope of this paper; some of the highlights of Formulate that are also present in the WebFormulate system are:

- Object names are unnecessary because all objects are potentially visible and can be referenced by pointing.
- Assignment is avoided in favor of a functional approach more consistent with mathematics.
- Iteration is unnecessary. Recursion is supported, but often unnecessary as well.
- Definitions in the premeditated sense are unnecessary. Functions can be defined by a user observing that some combination of already existing objects and their associated equations might be abstracted. The system then develops the appropriated abstracted function.
- Evaluation ordering is unnecessary because the language definition guarantees that all evaluation orderings are equivalent.
- All programs are developed using live data. This mode of operation exploits the interactivity of the development environment and facilitates user understanding.
- Structured objects (e.g., arrays, lists, tables) are supported, but without the usual indexing mechanisms. Rather the system deduces indexing based upon higher-level user interactions. Determination of the size and shape of the resulting structure as well as all required indexing is left to the system. This also eliminates the need for developing loops to perform such indexing.

In both Formulate and WebFormulate, objects communicate via messages to request or announce changes in values. In particular, computations retain symbolic links to referenced objects. Using such links, computations request to be notified of value changes. When notified of a value change to a referenced object, an object recomputes and then notifies all objects that have previously requested to be notified. This notification process happens automatically and keeps all computations moment-to-moment up to date. This distributed object model is essential for the WebFormulate system where dynamic links to objects distributed across the Internet must be maintained so that computations and reports can automatically update themselves.

3. SYSTEM OVERVIEW

As shown in Figure 2, the WebFormulate system consists of several modules (shown as gray boxes), which encompass various internal objects (shown as inner white boxes). The client-side modules run as a Java applet on the client’s machine, while the four server-side modules run as individual applications on a server. Here we discuss the functionality of the components as well as the interface between components.

3.1 The User Interface Module

The WebFormulate User Interface Module contains two components: a Login Authorization Manager and a Graphical User Interface (GUI) that runs in a Web browser environment. To use the WebFormulate system, the user must first login through the
Client Login Manager. The login information is then sent to the Server Login Manager for validation.

From the Web-based visual user interface, a user can create a WebFormulate Web page (form) or open an existing form. All such forms are stored on the WebFormulate server, with access and storage managed by the server-side File Manager. When a form is created, the owner can designate that it can be viewed, but not modified, by other users. This access information is maintained by the File Manager. In the example given in the Introduction, this feature would allow the biologist to grant permission for his/her colleagues to view a WebFormulate form containing a graph of the localities of the frogs, and monitor the changes in the graph as query updates are processed.

WebFormulate is conceptually similar to the Xforms model in that various types of objects can be placed on a form by dragging a representation of the object from a palette in the user interface onto the form. Values and/or equations can then be assigned to attributes of those objects using dialog boxes in the user interface. WebFormulate exceeds the current capabilities of XForms-based development systems with regards to the types of computations (e.g., database query, arithmetic, Boolean, graphical, and statistical equations) that it allows to be specified at a "public programmer" expertise level. In this discussion, we will focus on the specification of database query equations.

To formulate a continual query, the user must first create a 'Database' object and assign as its ‘value’ attribute a reference to a database (i.e., a URL). All WebFormulate equations (including constants such as the string URL) are processed through a client-side Evaluation Engine. The role of the Evaluation Engine will be discussed in more detail shortly. The Evaluation Engine works in conjunction with the Database Access Manager to determine the schema for the referenced database. The XML schema information is then interpreted by the XML Database Schema Processor in preparation for display on the WebFormulate form as a hierarchical "tree" of the names of tables, related tables, and fields. An example of such a schema display is shown in Figure 3. Nodes representing tables in the database can be expanded to display the names of related tables and fields by clicking the mouse on the "+" button to the left of the table name in the tree display. Once expanded, clicking on the "-" button to the left of a table name contracts the display of the corresponding branch of the tree.

[Figure 3.a - A generic schema representation of a database schema; displayed as a hierarchical tree containing the names of tables, related tables, and fields.]

[Figure 3.b – Actual Specify database schema; displayed as a hierarchical tree containing the names of tables, related tables, and fields.]

The user can then create a ‘Database Query’ object on the form and assign it a query equation. In constructing such an equation, the user can simply click on a field name in the schema display to reference it within any part of a WebFormulate query expression. An internal representation of the selected field will then be inserted into the equation. For example, a WebFormulate continual query expression for the query discussed earlier in the Introduction would look like the following:

\[
(CQSELECT ([FullTaxonName] [Latitude] [Longitude])
\]

\[
(\text{AND} (\text{IN} [\text{FullTaxonName}]
\]

\[
("Rana blairi" "Rana catesbeiana")
\]

\[
(\geq [\text{Latitude}] 37.8)
\]

\[
(\leq [\text{Latitude}] 39.4])
\]

\[
(\geq [\text{Longitude}] 94.8)
\]

\[
(\leq [\text{Longitude}] 96))
\]

\[
((\text{ASC} [\text{FullTaxonName}]))
\]

\[
(\text{WEEKLY})
\]

\[
(\text{DAILY})
\]

\[
(\text{DATE} 10 31 2002)
\]

The parameters to the CQSELECT function are: (1) the list of fields to select, (2) the selection criteria for extracting the tuples of interest (e.g., here specifying the species names and locality ranges of interest), (3) the sorting criteria for the query results (e.g., here specifying ascending order by the species name), (4) the notification condition (e.g., this condition requests that the user be notified when new results are available).

---

3 http://www.w3.org/MarkUp/Forms
notified weekly), (5) the trigger condition (i.e., how often the database should be queried, in this case “daily”), and (6) the termination condition for the execution of this continual query (e.g., October 31, 2002).

The notation \([X]\) is meant to convey that the database field name \(X\) in the database schema tree display is selected (clicked on) with the mouse, and an image representing that selection appears in the equation. The field name is not actually typed as it appears in the expression.

It is important to note that in order to construct a query in the WebFormulate system the user need not type in field and/or table names, or construct complex SQL statements that require explicit specification of joins. Thus, as [23] advocate, “the user is released from syntactic and implementation details, and the query can be naturally expressed by pointing directly to objects and spatially navigating among them.”

Once a continual query expression is constructed for the ‘Database Query’ object, it is submitted to the Evaluation Engine. The processing that takes place for that equation will be discussed in the next section. The user interface will then display both the initial (and subsequent, updated) query results as a table in the ‘Database Query’ object on the WebFormulate form as shown in Figure 4.

![Figure 4. A ‘Database Query’ object’s tabular display of query results.](image)

To analyze these query results, the user can create other objects on the Web form that can directly reference these data in the equations for their ‘value’ attributes. WebFormulate includes many of the statistical and graphing functions found in spreadsheet programs. For example, to create the graph shown in Figure 5, the user would create a ‘Graph’ object on the form and assign it the following equation:

\[
\text{GRAPH} \left( \left[ 38.2 .. 95.8 \right] \left( \text{BLACK-CIRCLE} \right) \right) \\
\left( \left[ 37.9 .. 95.6 \right] \left( \text{BLACK-SQUARE} \right) \right) 
\]

The argument to the predefined function GRAPH is a list of lists, where each sublist specifies the source of the data value pairs to be graphed, and the color and/or pattern to be used to display the corresponding data points. In the above equation, the sources of the data values are referenced by dragging the mouse over the appropriate regions (the block of Latitude and Longitude values for each species of frog) of the ‘Database Query’ table. The notation \([38.2 .. 95.8]\) is meant to convey that the rectangular region bounded by the value 38.2 (the upper left hand corner of the desired rectangular block of values) to the value 95.8 (the lower right hand corner of the desired rectangular block of values) in the table display is being referenced. The selection is made by dragging the mouse over the desired region of the table, from upper left to lower right. The region representation is not actually typed as it appears in the example. For a more comprehensive discussion of the user interface for the selection and manipulation of regions within a table or array, see [18].

![Figure 5. A ‘Graph’ object that references data from a ‘Database Query’ object.](image)

Whenever the Evaluation Engine is notified of updated query results from the continual query processor, both the table display in the ‘Database Query’ object and the visual display of the ‘Graph’ object will automatically be updated.

### 3.2 The Evaluation Module

The WebFormulate Evaluation Module runs as part of the client-side applet. It functionally consists of two components: the Evaluation Engine and the Database Access Manager. The function of the Evaluation Engine is to evaluate all object attribute equations submitted through the user interface. However, when an equation is submitted for evaluation, it is first sent (via the Communication Module) to the server-side Evaluation Preprocessor for syntactic and semantic analysis. If a syntactic or semantic error is found, the Evaluation Preprocessor sends back an error message and the user is notified accordingly through a dialog in the user interface.

Otherwise, actual evaluation of the equation takes place in the Evaluation Engine, and the user interface is notified to update the display of that object, as well as other objects that are interested in the results of that evaluation (i.e., objects that have referenced that object’s attributes in their own attribute equations).

The WebFormulate equation evaluation process differs slightly for database queries in two respects: (1) in determining database schema information when a URL is specified as the ‘value’ attribute expression for a ‘Database’ object, and (2) in evaluating a continual query equation for a ‘Database Query’ object. When a ‘Database’ object is created on a form, the GUI requests from the Evaluation Module information about the specified database’s schema. The Database Access Manager handles this information...
request. Because the WebFormulate client-side applet has no file system access and cannot make connections outside the codebase on the server, it must forward this information request to the server-side Database Access Server. The process of determining the database schema is discussed in detail in the next section. The schema information is then communicated to the GUI and the schema is displayed in the ‘Database’ object.

Likewise, when a query is submitted to the Evaluation Module, actual evaluation of the equation must be deferred to another component in the WebFormulate system, the continual query processor (CQServer). The Evaluation Engine must construct from the WebFormulate CQSELECT query equation a CQServer query request. This first requires that a well-formed SQL statement be constructed from the CQSELECT equation, with substitutions for the actual table and field names, construction of necessary join operations, and appropriate formatting for the type of database to be queried⁴ (e.g., MySQL, PostgreSQL, Oracle, etc.). The query request is submitted to the CQServer as a string specifying the SQL statement, the trigger condition, the notification condition, and the termination condition, as well as the identification information for the client (i.e., the user id of the client, including the WebFormulate ‘Database Query’ object identification), the URL of the database to query, the user name and password for access to the target database, the WebFormulate client-side connection that will be accepting the results of the query, and the port number to connect to on the WebFormulate server.

The CQServer will notify the client connection when updated query results are available via a socket in the CQ Communication Module. A message processor in the CQ Communication Module notifies the Evaluation Engine, which, in turn, notifies the user interface to update its object displays. Query results obtained from the CQServer are formatted as an XML string. The user interface utilizes the WebFormulate XML Query Results Processor for interpretation of that XML string in order to display the data as a table in the ‘Database Query’ object.

3.3 The Database Access Server

The server-side Database Access Server is responsible for determining the schema of a database for subsequent display on a WebFormulate form. Because the WebFormulate user interface runs as a Java applet, it has no client-side file system access and cannot make connections outside the codebase on the server. Therefore, the request for database schema information is processed through the client-side Database Access Manager, with actual determination of the schema information performed by the server-side Database Access Server.

To determine the schema for an Internet-distributed database, the Database Access Server first goes to the location specified by the database’s URL and looks for a special XML-formatted schema file that contains the names of the tables and fields, as well as information about relationships between tables, the data type of each field, and an optional caption for each table and field name (i.e., in case the database owner prefers that a caption be displayed instead of the actual field or table name). This schema file is generated by a Java application made available by the WebFormulate Project to database owners who want to customize or restrict the display of their database schema. For example, the Specify database application mentioned in the example given in the Introduction allows end-users to customize their view to a Specify database, determining which of approximately 800 fields in the database will be used and what the caption should be for each table and field. This information is contained as metadata within the Specify database and must be analyzed so that the WebFormulate schema display will accurately reflect the database view that the Specify database owner intended to be seen.

Although the XML Query Working Group⁵ is working towards defining standards for XML-based specifications of data models and query languages, in practice there is neither a standard XML representation for a database schema nor a standard way of generating the schema for the variety of client-server database management systems in use today. Thus, the WebFormulate system uses the following XML DTD for database schema representation:

```xml
<?xml version="1.0" encoding = "iso-8859-1" standalone = "yes"?>
<!DOCTYPE Database [<!ELEMENT Database (table+)>
<!ATTLIST Database
 name CDATA #REQUIRED
 caption CDATA #REQUIRED
 type CDATA #REQUIRED>
<!ELEMENT table ( field | relatedField)+>
<!ATTLIST table
 name ID #REQUIRED
 caption CDATA #REQUIRED>
<!ELEMENT field (#PCDATA)>
<!ATTLIST field
 name CDATA #REQUIRED
 caption CDATA #REQUIRED
 type CDATA #REQUIRED>
<!ELEMENT relatedField (#PCDATA)>
<!ATTLIST relatedField
 name CDATA #REQUIRED
]
</DOCTYPE Database>
```

⁴The java.sql.Connection interface provides a method, nativeSQL, to return the native form of an SQL statement for a particular database driver. However, this method has not yet been implemented in the drivers for many of the commonly used client-server databases.

⁵http://www.w3.org/XML/Query
To overcome the portability and scalability problems of other continual query systems, the CQServer system was implemented. To provide this information. Instead, the Database Access Server attempts to connect to the database using the most commonly used database drivers. Once the correct driver is determined, this information is recorded in the WebFormulate system for subsequent use in the continual query processor, which must also make connections to the remote database.

If the Database Access Server is unable to locate the special XML-formatted schema file, it determines the schema information ad hoc by using JDBC functions to obtain the names of all tables and fields in the database. The heterogeneous nature of database systems has precluded the development of JDBC functions to determine relationships between tables. Thus, the Database Access Server must make a “best guess” at these relationships by looking for fields with identical names and data types across tables. This simple algorithm, as well as the hierarchical tree structure utilized for representing the schema, assumes that the database is in Third Normal Form.

When an ad hoc analysis of a database schema is necessary, the Database Access Server must first connect to the database using the URL that was specified for the WebFormulate ‘Database’ object. However, the Database Access Server has no a priori knowledge of the type of database (e.g., MySQL, SQL-Server, Oracle, PostgreSQL, etc.) to which it is connecting. Because the WebFormulate system is intended for use by people with limited technical skills and knowledge, the end-user is not expected to provide this information. Instead, the WebFormulate Database Access Server attempts to connect to the database using the most commonly used database drivers. Once the correct driver is determined, this information is recorded in the WebFormulate system for subsequent use in the continual query processor, which must also make connections to the remote database.

3.4 The Continual Query Processor

When the WebFormulate user submits a continual query expression for evaluation, the Evaluation Module parses and translates it into a well-formed SQL expression with complete table and field name identifiers, required joins, etc., and then submits a query request to the continual query processor (CQServer) via the CQ Communication Module. Here we present a brief overview of the WebFormulate continual query processor; for a more detailed discussion, see [15].

To overcome the portability and scalability problems of other continual query systems, the CQServer system was implemented using Enterprise JavaBeans (EJB), a standard server-side component transaction monitor architecture that automatically manages transactions, object distribution, concurrency, security, persistence, and resource management. EJB applications utilize components known as beans, which can be one of two types: a session bean or an entity bean. A session bean is an extension of the client application and is responsible for managing processes or tasks. In contrast, an entity bean is used to model a business concept that can be expressed as a noun (e.g., the representation of an underlying object such as data in a database) [24].

When the application server running the CQServer initially receives a query request, it performs all necessary authentication and authorization checks of the client connection. The CQServer then performs its own syntactic and semantic analysis of the continual query request. If errors are found, the client connection is notified; otherwise, the continual query is persisted to a CQBean that represents the continual query as a whole, and a ScheduleBean that represents the scheduling aspects of the continual query such as the time-based trigger, termination, and notification conditions.

During the validation of the continual query request, certain ambiguous information contained in the query specification is converted to more concrete representations. For example, if the trigger condition is “daily”, an imprecise time, a random time of day is selected, and execution of the continual query is scheduled to query the database every day at that time.

When the continual query executes a query against a remote database (based on the trigger condition), the query results are stored as an XML-formatted document in a ResultSetBean on the CQServer. If the condition for notification has been met, a NotificationBean is instantiated to process the client notification. For example, if the notification condition is “weekly” (an imprecise time), the client would be notified of query results on the same day of the week that the query was originally issued. So if the query was originally issued on a Monday, the CQServer would interpret “weekly” as every Monday, and each Monday a NotificationBean would send an XML-formatted document containing the query results to the client connection in the WebFormulate CQ Communication Module. If the notification condition had been content-based (e.g., “whenever the number of records returned increases by 100 since the last time the query was run”), the CQServer would parse both the current and previous XML-formatted query result documents using the Java API for XML Parsing (JAXP)7 and the Apache Xerces

---

6 The current implementation of the WebFormulate system uses the Orion EJB application server (www.orionserver.com). However, the CQServer component can be used in conjunction with any EJB application server.

XMLParser, a standard XML parsing component, in order to analyze whether the specified condition had been met.

As with the XML representation for a database schema, there is no standard XML representation for database query results (although the XML Query Working Group is working towards establishing such standards). The CQServer utilizes the following DTD:

```xml
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<!DOCTYPE query-results [
<!ELEMENT query-results (header-row, result-row+)> 
<!ATTLIST query-results sql-query-string CDATA #REQUIRED 
number-of-results CDATA #REQUIRED> 
<!ELEMENT header-row (header-column+)>
<!ELEMENT header-column (#PCDATA)>
<!ATTLIST header-column num CDATA #REQUIRED 
name CDATA #REQUIRED 
type CDATA #REQUIRED 
sql-type CDATA #IMPLIED> 
<!ELEMENT result-row (result-column+)>
<!ATTLIST result-row num CDATA #REQUIRED> 
<!ELEMENT result-column (#PCDATA)>
<!ATTLIST result-column num CDATA #REQUIRED 
value CDATA #IMPLIED>]> 
```

The DTD above provides mark-up for the results returned from the database. The document as a whole is marked as query-results and is further identified by the SQL statement of the original query (i.e., the attribute sql-query-string). Also specified is the size of the result set (i.e., the attribute number-of-results). The basic identifying information of the result set, obtained from a Java ResultSetMetaData object, is formatted as the sub-element header-row, which consists of several header-column sub-elements that list the result set data. These header-column elements include information about the various result set columns, including the particular column number (i.e., num), the name given to that column (i.e., name), the database-specific data type (i.e., type), and the SQL defined data type (i.e., sql-type). The actual set of result data follow, marked up individually as result-row sub-elements. A result-column sub-element of a result-row contains the column number for this particular portion of this result row (i.e., num), and the specific value contained in that column of a result set row (i.e., value).

As mentioned earlier, this XML-formatted data set is sent to the client connection each time the specified notification conditions are satisfied. The CQServer also maintains the most recent query results as a file on the server as long as the continual query is still installed (i.e., until the continual query reaches its specified termination date). Thus, the client application can obtain the most recent results from the CQServer at any time, not just when a notification message is sent. This facilitates keeping `WebFormulate` computations and displays up-to-date, even if the form is closed and re-opened at a later time.

4. AN EXAMPLE OF SPECIFYING A CONTINUAL QUERY

To more clearly illustrate the capabilities of the `WebFormulate` system from a user’s viewpoint, we will construct a `WebFormulate` solution for the sample problem given in the Introduction, slightly modified to use a more complex notification condition.

Figure 6 is a `WebFormulate` Web page (form) that the biologist could create to analyze the population of the two species of frogs. To generate this Web page (form), the user would first login to the `WebFormulate` system, open a `WebFormulate` library (i.e., file system directory on the server), create a new `WebFormulate` form, and then do the following:

(1) Create a 'Database' object and set its ‘value’ attribute to the URL of the Web-accessible ODBC database containing the data of interest. The user will then see the database schema displayed within the ‘Database’ object, organized hierarchically as a "tree" of the names of tables, related tables, and fields within tables.

(2) Create a 'Database Query' object to query the two frog populations and assign it the following equation:

---

8 http://xml.apache.org/xerces2-j/index.html

9 User accounts are created and maintained by the `WebFormulate` server administrator.
The first argument to the CQSELECT function is a list of the fields to be displayed in the query results. The second argument is the conditional expression that will be used to identify the tuples to be retrieved by the query. The order by which the results are to be sorted is specified by the third argument (i.e., FullTaxonName in ascending order). The fourth argument specifies that the user wants to be notified of the query results whenever the number of records returned exceeds the previous number of query results by 100 (not simply “weekly” as was specified in the original problem statement). The fifth argument designates that the database is to be queried DAILY (i.e. the trigger condition). The last argument is the stop condition for the continual query, specified here as a particular date.

Once the user submits this equation for evaluation, the query is evaluated and the (initial) results are displayed as a table in the 'Database Query' object. This table display can be further sorted using any sequence of columns. To do so, the user simply drags the mouse over a rectangular region of the table (possibly the entire table) and sets the ‘sort order’ attribute of that region to a list of column names and sort order indicators (ASC for ascending order or DES for descending order), for example:

```
(((FullTaxonName) ASC) ([Latitude] ASC) ([Longitude] ASC))
```

As in the previous equations, the column names referenced in this equation are not typed, but rather are selected (clicked on) with the mouse in the column headings of the 'Database Query' table display.

Create a graph object to plot the occurrences of the two species of frogs by latitude and longitude. As was discussed in section 3.1, the equation for this object is:

```
(GRAPH
  ([38.2..95.8] (BLACK-CIRCLE))
  ([37.9..95.6] (BLACK-SQUARE)))
```

The argument to the predefined function GRAPH a list of lists, where each sublist specifies the source of the data value pairs to be graphed, and the color and/or pattern to be used to display the corresponding data points. In the above equation, the sources of the data values are referenced by dragging the mouse over the appropriate regions (the block of Latitude and Longitude values for each species of frog) of the 'Database Query' table.

Each time the continual query processor returns updated results, the user will see the automatic update of the tabular display of the 'Database Query' object and the ‘Graph’ object’s display which references the query result data.

If the form is closed, the next time it is opened, the client connection to the server-side continual query processor will be re-established and the most recent data will be requested. Even if a client connection is closed, the CQServer maintains an XML file of the most current query results for each continual query that is currently installed (i.e., each continual query that has not yet reached its specified termination date). Accordingly, the
biologist's colleagues will be able to open a copy of this form\textsuperscript{10} at any time in their own Web browser and monitor the progress of the analysis.

5. FUTURE WORK
The Formulate system, upon which WebFormulate is based, has previously been empirically tested for usability; see [19]. In the future, we intend to conduct additional formal usability studies to evaluate both the usability and usefulness of the WebFormulate system by “public programmers.” We will specifically focus those studies on non-programmers' use of the WebFormulate Web-based user interface and the continual query processing functionality. In response to feedback from those studies, the WebFormulate software will be modified and/or enhanced as necessary to better facilitate meaningful visual and computational analyses of query results. We will also be conducting simulated experiments to evaluate the performance of the continual query processor for a variety of scenarios, varying the complexity of the queries, the update frequency, and the size of the distributed databases that will be queried, as well as the number of client connections.

Also in the future, as the standards and use of XML-based specifications for queries, query results, and database schemas mature, we will incorporate those standards into the WebFormulate system, replacing our currently used DTDs.

6. SUMMARY
It is important not only that information be accessible to the public, but that the same public be able to combine this information into effective analyses. However, to date both academic and commercial use of Internet-distributed databases has been hindered by: (1) the incomplete, dynamic, and "unknowable" structure of those databases, (2) the lack of XML-based standards for data model and query representations, and (3) the inability of non-programmers to be able to formulate queries against multiple Internet databases. WebFormulate was specifically designed as an efficient and scalable tool that would enable non-programmers to perform continual queries on Internet-distributed databases and to automatically update user-specified computations and reports in a Web browser environment. We believe that the technology developed for the WebFormulate system will contribute to the general advancement of the Internet and its impact on industry, government, and education.

7. ACKNOWLEDGMENTS
This work was supported by NSF under award DBI-9905760.

8. REFERENCES

\textsuperscript{10} "Copies" of a WebFormulate form are updated with recomputed data in the same manner and timeframe as the "original" form.


