01 Jan 2001

A Visual Query System for the Specification and Scientific Analysis of Continual Queries

Jennifer Leopold
Missouri University of Science and Technology, leopoldj@mst.edu

A. Ambler

M. Heimovics

T. Palmer

Recommended Citation

J. Leopold et al., "A Visual Query System for the Specification and Scientific Analysis of Continual Queries," *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, 2001*, Institute of Electrical and Electronics Engineers (IEEE), Jan 2001.
The definitive version is available at https://doi.org/10.1109/HCC.2001.995260

A Visual Query System for the Specification and Scientific Analysis of Continual Queries

Jennifer Leopold, Allen Ambler, Meg Heimovics and Tyler Palmer
University of Kansas
Department of Electrical Engineering and Computer Science
{leopold, ambler, megheim, tyler}@designlab.ukans.edu

Abstract

The lack of a facility that would allow non-programmers to easily formulate temporal ad hoc analyses over a network of heterogeneous, constantly-updated data sources has been a significant impediment to research, particularly in the scientific community. In this paper we describe WebFormulate, an Internet-based system which facilitates the development of analyses using information obtained from databases on the Internet. 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 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.

1. Introduction

The revolution in computing brought about by the Internet is rapidly changing the nature of computing from a personalized computing environment to a ubiquitous computing environment in which both data and computational resources are network-distributed [1, 2]. However, the lack of a facility that would allow non-programmers to easily formulate temporal ad hoc analyses over a network of heterogeneous, constantly-updated data sources has been a significant impediment to research, particularly in the scientific community. There are many classes of research queries that cannot be addressed as a result of the n-way struggle that must go on between every researcher and every network database interface of interest. The process is tedious, inefficient, and subsequently leaves information unnoticed and unutilized. Furthermore, although common client-server communications protocols permit parallel ad hoc queries of ODBC databases, they do not provide the functionality to automatically perform *continual queries* to track changes in those databases through time. The lack of persistence of the state of data resources requires researchers to repeatedly query data sources and manually compare the results of searches through time.

For example, suppose that a biologist in California is researching the apparent disappearance of the plains leopard frog (*Rana blairi*) in Douglas County, Kansas. S/he 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.

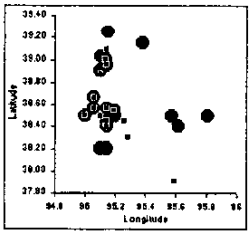

Figure 1. A graph to analyze the populations of two species of frogs.

0-7695-0474-4/01 $ 10.00 © 2001 IEEE

In *Specify*[1], a biological collection database that is typically used to record specimen information of this nature, the actual SQL statement for such a query would be the following:

```
SELECT TaxonName2.FullTaxonName,
Locality.Latitude, Locality.Longitude FROM
(CollectionObject) INNER JOIN (((CollectionObject
AS CollectionObject2) INNER JOIN ((Determination
AS Determination2) INNER JOIN (TaxonName AS
TaxonName2) ON TaxonName2.TaxonNameID =
Determination2.TaxonNameID) ON
Determination2.BiologicalObjectID =
CollectionObject2.CollectionObjectID) INNER JOIN
((CollectingEvent AS CollectingEvent2) INNER JOIN
(Locality) ON Locality.LocalityID =
CollectingEvent2.LocalityID) ON
CollectingEvent2.CollectingEventID =
CollectionObject2.CollectingEventID) ON
CollectionObject2.CollectionObjectID =
CollectionObject.CollectionObjectID  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 spreadsheet application to regenerate the graph. In reality, it is unlikely that a biologist would have all the requisite skills to perform this task, namely: (1) to connect to a remote database, (2) to construct a complex SQL statement, (3) to download the query results, and (4) to import the data into a spreadsheet and construct a graph. In addition, it would require that the biologist had the time (and perseverance) to perform this procedure 36 times.

The above example clearly demonstrates that a second obstacle to the synthesis of networked data is the lack of a programming interface which permits end-users to easily formulate and execute scientifically meaningful and often complex queries of distributed data sources and to be able to link the results of those queries to analysis applications. In short, we need to be able to empower researchers to build these kinds of automated retrieval and analysis functions through a network interface without any specific knowledge of programming tasks. The problem is, of course, that many scientists are untrained as programmers and to use today's programming languages generally requires significant programming training.

Herein we describe *WebFormulate*, an Internet-based system which facilitates the development of analyses using information obtained from databases on the Internet. 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 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.

2. Background and Related Work

2.1. Visual Query Systems

Visual query systems (VQS) [3, 4] 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 [5] even allow the user to select from several visual representations and interaction mechanisms for expressing the query and visualizing the results.

The primary differences between the *WebFormulate* query interface and other VQS are:

(1) Most VQS have been developed specifically to query geographic and image databases, which are not the problem domain 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. User interfaces for most continual 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

1 http://www.usobi.org/Specify

SQL statements, and to use other applications to analyze query results.

2.2. Continual Queries

A continual query as defined by [6] "...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 conditional 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 [7] for append-only databases.

Active database management systems [8, 9, 10, 11, 12] are restricted implementations of continual query systems. Unlike the traditional, passive, program-driven database management 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 [13]. Another limitation is that the trigger mechanisms 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 [6], OpenCQ [14], and NiagaraCQ [15] 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, stocks, and bibliographic references). 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 visual query system specifically designed for non-programmers that can perform continual queries on Internet-distributed databases, developing and maintaining dynamic links such that user-specified computations and reports automatically maintain themselves.

3. *Formulate*

The *WebFormulate* system is based on an existing implementation of a language called *Formulate* [17, 18, 19, 20, 21, 22]. *Formulate* is the product of many years of visual and public programming research, and has been tested against users with little prior programming experience [19]. A complete description of the *Formulate* language is beyond the scope of this paper; some of the highlights of *Formulate* which 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 interactiveness 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 *Formulate* 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 which 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 was essential for the *WebFormulate* system as well where dynamic links to objects distributed across the Internet must be maintained so that computations and reports can automatically update themselves.

4. A Visual Interface for the Specification of Continual Queries

The *WebFormulate* user interface is very similar to the *Formulate* user interface (which is described in detail in [17, 18, 20, 21, 22]) with the exception that it runs in a web browser environment. *WebFormulate* extends the functionality of *Formulate* allowing the user to perform continual queries of distributed databases and to use the results of those queries for subsequent computations and visualizations.

4.1. Specification of a Continual Query

In their taxonomic classification of visual query systems, [23] define paradigms for the scheme of both data and query representations. *WebFormulate* utilizes a hybrid paradigm: the diagrammatic paradigm to describe the database schema, and the tabular paradigm to visualize query results and to reference intermediate database query results in subsequent queries. According to the classification criteria for classes of users and type of visual query system given in [23], the diagrammatic paradigm should be particularly well-suited for scientists with little or no programming background; users who likely have significant knowledge of the semantic domain of the databases they are querying, but are required to perform structurally complex queries[2].

2 For example, the *Specify* specimen database mentioned in the example given in the Introduction contains over 70 tables and approximately 800 fields.

In the *WebFormulate* system continual queries are expressed in terms of an SQL-like query, a notification condition, a trigger condition, and a termination condition. The SQL-like query specifies the names of the fields to be included in the result set, the conditions to be applied to extract the tuples of interest, and the sorting order for the results. Once a database is specified (by a URL identifying the database or by referencing a *WebFormulate* object containing the results of another query), the database schema is displayed as a hierarchical "tree" of the names of tables, related tables, and fields. An example of such a schema display is shown in Figure 2. 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.

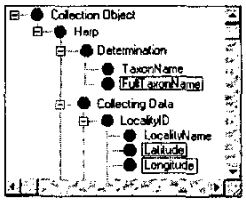

Figure 2. A database schema displayed as a hierarchical tree containing the names of tables, related tables, and fields.

The user can simply click on a field name in the tree display to reference it within any part of a *WebFormulate* query expression. For example, a standing (non-continual) *WebFormulate* query expression for the query discussed earlier in the Introduction would look like:

(SELECT ([*FullTaxonName*] [*Latitude*] [*Longitude*])
(AND (IN [*FullTaxonName*]
("Rana blairi" "Rana catesbeiana"))
(>= [*Latitude*] 37.8) (<= [*Latitude*] 39.4])
(>= [*Longitude*] 94.8) (<= [*Longitude*] 96)))

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.

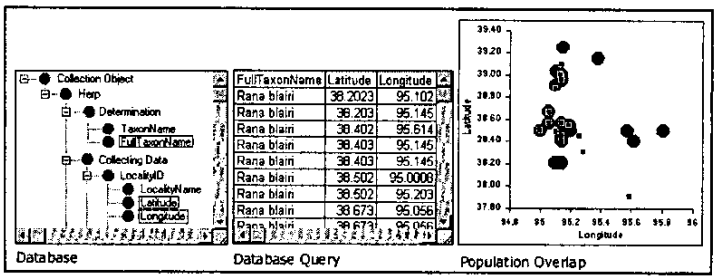

Figure 3. A *WebFormulate* form to track the populations of two species of frogs.

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 involving 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."

A notification condition for a *WebFormulate* continual query can be a combination of time-based conditions and content-based conditions. Time-based conditions include: (1) absolute points in time (defined by the system clock), (2) regular or irregular time intervals (e.g., weekly, the first of every month, etc.), and (3) relative temporal events (e.g, one week after event A occurred). Content-based conditions include: (1) relationships between a previous query result and the current database state (e.g., when the number of bullfrogs increases by 20% since the last reporting time), (2) trends (e.g., when the number of bullfrogs decreases over the last two months), and (3) absolute threshold values (e.g., when the number of frogs exceeds 1000). The trigger condition is used to determine how often the database should be queried and tested against the notification condition. It is restricted to regular or irregular time intervals. The *WebFormulate* system provides predefined functions for the expression of many such conditions. In addition, the user can define and utilize his/her own functions in these expressions. See [17, 20, 21] for a more detailed discussion of the creation of user-defined functions in *Formulate*.

The termination condition for a *WebFormulate* continual query must be an absolute point in time, functionally expressed in terms of a month, day, and year.

4.2. Example of Specifying a Continual Query

Here we demonstrate some of the salient aspects of the *WebFormulate* user interface with respect to the specification of a continual query using the example given earlier in the Introduction, slightly modified to use a more complex notification condition.

Figure 3 is a web page that a biologist could create in *WebFormulate* to analyze the population of the two species of frogs.

In brief, a *WebFormulate* web page (form) is created by dragging various types of objects from a palette in the user interface onto a clean form, and assigning values and/or equations to attributes of those objects. To create the form in Figure 3 the user would do the following:

1. Create a 'Database' object and set its 'Database ID' attribute to the URL of the web-accessible ODBC database containing the data of interest. The database schema will then be displayed as shown in Figure 3, organized hierarchically as a "tree" of the names of tables, related tables, and fields within the tables[3].

2. Create a 'Database Query' object to query the two frog populations using the following equation:

```
(CQSELECT ([FullTaxonName] [Latitude]
          [Longitude])
  (AND (IN [FullTaxonName]
           ("Rana blairi" "Rana catesbeiana"))
       (>= [Latitude] 37.8)
       (<= [Latitude] 39.4])
```

3 The database schema is determined each time the *WebFormulate* form is opened. Changes to the schema of the actual database (e.g., removal of the FullTaxonName field) may subsequently generate errors in equations that reference the 'Database' object.

(>= [*Longitude*] 94.8)
(<= [*Longitude*] 96))
((ASC [*FullTaxonName*]))
(OR (WEEKLY)
(> (COUNT [*FullTaxonName*])
(+ (PREVCOUNT [*FullTaxonName*])
1000)))
(DAILY)
(DATE 10 30 2001))

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 WEEKLY or whenever the total number of frog records (of both species of interest) has increased by 1000 since the last query. 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.

3. 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 region of the table[4] (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.

4. Create a graph object to plot the occurrences of the two species of frogs by latitude and longitude. The equation[5] for this object is:

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

The argument to the predefined function GRAPH[6] 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.

Each time the continual query returns updated results, the tabular display of the 'Database Query' object and the graph object which references that data will automatically be updated on this form.

It is of interest to note that the 'Database Query' object could now be referenced as the value for the 'Database ID' attribute of another 'Database' object. Its schema would simply consist of a single table containing the three fields: FullTaxonName, Latitude, and Longitude. This feature facilitates the querying of "intermediate" query results (i.e., database "views").

5. System Architecture

The evaluation model utilized by *Formulate* (and *WebFormulate*) is discussed in detail in [20, 21, 22]. Here we briefly describe the system architecture employed by *WebFormulate* for continual query processing and its interface to the evaluation engine.

5.1. Overview of the Continual Query Processing System Architecture

When the user submits a continual query expression to the *WebFormulate* evaluation engine, it is parsed and translated into a well-formed SQL expression with

4 For a more comprehensive discussion of the *Formulate* user interface for the selection and manipulation of regions within a table or array, see [18].

5 The notation [n..m] is meant to convey that the rectangular region bounded by value n to value m in a table object 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.

6 *WebFormulate* includes many of the statistical and graphing functions found in spreadsheet programs.

complete table and field name identifiers, required joins, etc. It is then submitted to a continual query processing server (CQServer) which has been implemented using Enterprise Java Beans (EJB), a standard server-side component transaction monitor architecture that automatically manages transactions, object distribution, concurrency, security, persistence, and resource management [24].

A CQBean process is created by the server and begins its cycle of connecting to and querying a particular database according to the specified trigger condition, and comparing the results against the notification condition. CQBeans are self-autonomous in that: (1) a CQBean can communicate with other CQBeans to share information on similar active queries, (2) CQBeans maintain their own schedule for querying the database, and (3) a CQBean determines whether or not to allow itself to be passivated by the server under various conditions.

If a client disconnects from a CQBean (e.g., the *WebFormulate* form is closed), the CQBean will be passivated by the CQServer unless: (1) the notification condition is a linear trend and passivation would result in a loss of information, or (2) the CQBean is actively gathering information from other CQBeans interested in the same query. The system allows the client to later reconnect to the CQBean (e.g., when the *WebFormulate* form is re-opened).

Most other continual query processing systems utilize web-based Java applet interfaces to achieve portability, but often rely on platform-dependent resource management facilities (e.g., the UNIX cron facility, the Windows NT process scheduler, etc.). The *WebFormulate* system is portable to any EJB implementation (which is available for most commonly used operating systems). This system is also more efficient and scalable than other continual processing systems due to the optimized resource and process management facilities of the EJB server, and the semi-autonomous processing capabilities of the CQBeans.

5.2. Example of Processing a Continual Query

We will use the sample problem given in section 4.2 to explain in more detail how the CQServer processes a *WebFormulate* continual query.

When the *WebFormulate* evaluation engine processes a CQSELECT equation, it connects to the CQServer, which in turn performs all necessary authentification and authorization checks of the client connection. The CQServer then adds to an Active Bean table a CQBean to process the query. That CQBean will first check the CQServer's Active Query table to see if collaboration is possible with another CQBean that is processing the same query (e.g., if there is more than one instantiation of the same *WebFormulate* form). If collaboration is possible, the CQBean will establish inter-bean communication with that other CQBean to negotiate information exchange.

When a CQBean is created, a clockCheck object within the bean is configured for the time-based trigger and termination conditions specified in the continual query. In the example given in section 4.2, the trigger condition is DAILY. Since "daily" is an imprecise time, a random time of day is selected and the CQBean process is scheduled to query the database every day at that time. The clockCheck object is also configured to check the system date once a day and notify the CQBean to terminate itself when the specified termination date is reached.

When the CQBean queries the database (based on the trigger condition), the query results are stored in XML format on the server and the CQBean checks the notification condition. The notification condition in our sample problem is WEEKLY or whenever the number of records returned by the query has increased by 1000 since the last time the query was issued. Since the notification condition contains a content-based condition involving an increase in the size of the result set, the CQBean maintains a count of the number of records returned when the database is queried. For the notification condition "weekly", the CQBean is also configured to return query results on the same day of the week that the query was originally issued. For example, if the query was originally issued on a Monday, the CQBean interprets "weekly" as every Monday. Therefore, the CQBean for this example checks to see if the current date is Monday or if the size of the result set has increased by 1000 since the last time the database was queried. If either condition is true, the CQBean sends a message to the *WebFormulate* evaluation engine to notify it that updated query results are available. Any other CQBeans which are interested in these query results and have established inter-bean communication with this CQBean will also be notified.

If the *WebFormulate* web page is currently open, every *WebFormulate* object that is interested in the query results will be sent a message from the evaluation engine to recompute and/or redisplay itself. In our example, the 'Database Query' object will be notified to update its table display. The update of that object in turn will cause an update message to be sent to the graph object whose equation was specified in terms of values contained within the 'Database Query'

object. Thus, the graph object will also recompute and redisplay itself based on the new data.

If the *WebFormulate* form is not currently open, the next time the form is opened, it will request the most recent data from the evaluation engine. The evaluation engine will then retrieve the most recent query results from the XML file that is maintained on the server by the CQBean.

6. Future Work

In the future, we plan to continue consulting with scientists to determine what additional functionality needs to be included in the *WebFormulate* system in order to 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.

7. 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. The incomplete, dynamic, and "unknowable" structure of databases, and the inability of non-programming scientists to formulate queries against multiple Internet databases has been a significant impediment to research. *WebFormulate* was specifically designed as an efficient and scalable application that would enable non-programmers to perform continual queries on Internet-distributed databases, and to automatically update user-specified computations and reports. 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.

Acknowledgements

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

References

[1] K. Kavi, J.C. Browne, and A. Tripathi, "The Pressure Is On," IEEE Computer, January 1999, pp. 30-39.

[2] W. Bolosky, R. Draves, R. Fitzgerald, C. Fraser, M. Jones, T. Knoblock, and R. Rashid, "Operating System Directions for the Next Millennium," Microsoft Research, Redmond, Washington, http://research.microsoft.com.

[3] T. Catarci and M. Costabile (eds.), "Special Issue on Visual Query Languages", *Journal of Visual Languages and Computing*, 6(1), 1995.

[4] T. Catarci, M. Costabile, S. Levialdi, and C. Batini, "Visual Query Systems for Databases: A Survey", Technical Report SI/RR-95/17, Dipartimento di Scienze dell'Informazione, Universita' di Roma "La Sapienza", 1995.

[5] T. Catarci, M. Costabile, A. Massari, L. Saladini, and G.Santucci, "A Multiparadigmatic Environment for Interacting with Databases", *SIGCHI*, 28(3), July 1996.

[6] L. Liu, C. Pu, W. Tang, and W. Han, "CONQUER: A Continual Query System for Update Monitoring in the WWW", Special edition on Web Semantics, *International Journal of Computer Systems, Science, and Engineering*, 1999.

[7] D. Terry, D. Goldberg, D. Nichols, B. Oki, "Continuous Queries over Append-Only Databases", *ACM SIGMOD International Conference on Management of Data*, 1992, pp. 321-330.

[8] U. Dayal, B. Blaustein, A. Buchman, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarvin, M. Carey, M. Livny, and R.Jauhari, "The HiPAC Project: Combining Active Database and Timing Constraints", *ACM-SIGMOD Record*, 17(1), March 1998, pp. 51-70.

[9] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS", *ACM SIGMOD International Conference on Management of Data*, 1991, pp. 469-478.

[10] M. Hsu, R. Ladin, and D. McCarthy, "An Execution Model for Active Database Management Systems", in *Proceedings of the 3rd International Conference on Data and Knowledge Bases - Improving Usability and Responsiveness*, 1988.

[11] J. Widom and S Ceri, Active Database Systems, Morgan-Kaufman, 1996.

[12] D. McCarthy and U. Dayal, "The Architecture of an Active Database Management System", in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, May 1989, pp. 215-224.

[13] Informix Software, Inc., *Informix Guide to SQL: Syntax (Version 6.0)*, 1994.

[14] L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery", Special issue on Web Technologies, *IEEE Transactions on Knowledge and Data Engineering*, January 1999.

[15] J. Chen, D. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", in *Proceedings ACM SIGMOD International Conference on Management of Data*, May 1999.

[16] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML", http://ww.w3.org/TR/NOTE-xml.ql.

[17] A. Ambler and A. Broman, "Formulate Solution to the Visual Programming Challenge," in *Journal of Visual Languages and Computing*, 9(2), April, 1998, pp. 171-209.

[18] J. Leopold and A. Ambler, "A User Interface for the Visualization and Manipulation of Arrays", in *Proceedings of IEEE 12th Symposium on Visual Languages*, 1996, pp. 54-55.

[19] J. Leopold, "A Multimodal User Interface for a Visual Programming Language", Ph.D. Thesis, University of Kansas, Lawrence, Kansas, 1999.

[20] G. Wang and A. Ambler, "Invocation Polymorphism", Proceedings of IEEE Symposium on Visual Languages, Darmstadt, Germany, September 1995, pp. 83-90.

[21] G. Wang and A. Ambler, "Solving Display-Based Problems", in *Proceedings of IEEE 12th Symposium on Visual Languages*, 1996, pp. 122-129.

[22] G. Viehstaedt and A. Ambler, "Visual Representation and Manipulation of Matrices", in *Journal of Visual Languages and Computing*, Volume 3, 1992, pp. 273-298.

[23] C. Batini T. Catarci, M. Costabile, and S. Levialdi, "Visual Query Systems: A Taxonomy", in Visual Database Systems II (E. Knuth and L. Wegner, eds.), Elsevier Science Publishers, North-Holland, 1992, pp. 153-168.

[24] R. Monson-Haefel, Enterprise JavaBeans, O'Reilly & Associates Inc., Sebastopol, CA, March 2000.