

Citation: Tjan† B.S., Breslow L., Dogru S., Rajan V., Rieck K., Slagle J.R., & Poliac M. (1993). A data-flow graphical user interface for querying a scientific database. In *IEEE Symposium on Visual Languages*, (pp. 49-54). IEEE Computer Society.

A Data-Flow Graphical User Interface for Querying a Scientific Database *

Bosco S. Tjan, Leonard Breslow, Sait Dogru, Vijay Rajan,
Keith Rieck, James R. Slagle, and Marius O. Poliac
Computer Science Department
University of Minnesota, Minneapolis MN 55455

Abstract

We describe the design principles and functionality of a visual query language called SeeQL that represents data retrieval and analysis operations as a data-flow graph. A query is viewed as a sequence of relational algebra and other data transformation operations applied to database tables. The language is well-suited for large-scale scientific database applications, where data analysis is a major component and the typical queries or data retrieval patterns are unrestricted. The language provides a flexible yet easy-to-use environment for database access and data analysis for non-programmer research scientists. We have implemented this language in a system being used in a long-term data-intensive highway pavement research project (MnRoad) conducted by the Minnesota Department of Transportation.

1 Introduction

The database query system described here was developed to provide data retrieval and analysis capabilities for a large-scale cold-region highway pavement research project (MnRoad) conducted by the Minnesota Department of Transportation. The intended users of the system are non-programmer research engineers. As non-programmers, such users require an easy-to-use interface that does not demand prior knowledge of database concepts or query languages. But as researchers, such users need a system that is powerful enough to perform complex queries and flexi-

ble enough to accommodate the changing demands of their research.

Since data retrieval and analysis capabilities are crucial to the success of the MnRoad project, we considered it important to provide a graphical query interface to a widely-used database management system. The relational model is currently the most popular database model and SQL is the most commonly-used relational query language. Our system provides a graphical query interface for the SQL language and extends SQL's power and flexibility with a variety of data analysis and presentation capabilities.

Previous approaches to providing user-friendly interfaces to databases have included English-like natural language query systems [1] and form-based templates. More recently, graphical query languages [2, 3, 4, 7, 8] have been introduced to provide greater ease-of-use and power than these early methods. Most of these languages, like SQL, are declarative in nature.

While declarative languages are considered suitable for data retrieval tasks, they may not meet all the needs of scientific applications. Scientific research requires a close integration of data retrieval with data analysis, which is normally conceived as a sequence of operations, each operating on the output of the previous operation. A procedural language allows researchers to clearly represent and incrementally construct such sequences of operations according to their individual needs. In scientific visualization applications that do not involve database retrieval, procedurally-oriented languages [6, 10] have already shown promise. These languages represent sequences of operations visually as data-flow graphs.

We developed SeeQL as a procedure-oriented graphical query language using data-flow graphs. Targeted for scientific database applications, SeeQL combines easy data retrieval, analysis, and visualization capabilities. It provides a rich graphical and procedural vocabulary that allows the user to construct com-

*This research is supported by the Minnesota Department of Transportation (MnDOT) as part of the MnRoad Project. We wish to thank Dave Johnson and Joe Cornell of MnDOT. The views expressed herein do not necessarily represent the views of MnDOT. We are also grateful to Steve Ratering and Albert Esterline for their contributions to the early design and implementation of the query system.

plex queries in an incremental fashion and to easily extend the language's capabilities to meet the user's individual needs. The approach evolved during the course of extensive consultation with our end users.

Queries in SeeQL, which include data retrieval, analysis, and visualization operations, are represented as acyclic data-flow graphs. Operations are represented by nodes of various kinds called *modules*. Modules that retrieve data from the database are the sources of the directed paths. Data presentation modules are the sinks. Between the sources and sinks are functional modules performing data selection, transformation, and analysis. There is only one kind of data, namely database tables, that flow from the sources to the sinks. A directed arc between two modules carries a table from one to the other. Each functional module receives input from one or more modules and transforms its input table(s) into an output table. Each module is represented by a descriptive icon. A query is represented by a tree in a data-flow graph rooted at a sink module with source modules as leaves. Different queries may share common subqueries. Thus the directed acyclic graph represents the set of queries a user is currently working with.

2 Design principles

The most important reason for having a visual query language for a scientific database application is to provide easy and flexible access, analysis, and visualization of data. In contrast to a typical business application, the data retrieval and analysis patterns for a scientific database are often unknown during the design phase. The data collected in a series of experiments may be used to support an unanticipated number of differing research projects, each defining the relationship between the data entities differently and each using its own data processing techniques. The query language must permit scientific users to construct all the complex queries they will need.

Three principles guided our design.

- *Conceptual simplicity*: The query language must be user-oriented. It is designed for a non-programmer research scientist. The language must be conceptually simple to make it intuitive to learn and understand.
- *Expressiveness*: While maintaining simplicity, the language must be sufficiently expressive and extensible to represent all the queries a user will ever need, given only a loose specification of the application domain.
- *Provision of on-line help*: The programming environment must provide sufficient assistance at every step in the construction of a query to make the language easy to use.

In what follows, we shall elaborate these three design principles and discuss how they motivated our particular design choices.

2.1 Conceptual simplicity

A user-oriented query language should be conceptually simple. To meet this goal, we identified five requirements: (1) use of a single data type for all the operations in the language; (2) adoption of a procedural orientation; (3) maintenance of compatibility with the relational data model and query operations; (4) use of understandable, explicitly-stated operations in a query; and (5) prevention of looping and recursion.

To achieve conceptual simplicity we use a single data structure: a database table. The operations of our language, with few exceptions, operate on this data structure. A table is a rich data structure and yet is conceptually simple. It is organized into rows and columns. A row represents an individual record. A column represents a field (or attribute) of the records. Tables and their columns have names. An entry to a table can be text, a number, or a time stamp, but from the user's high-level perspective, the table is the only data type operated on.

To enhance its conceptual simplicity, we believe the language should be procedurally oriented. This is perhaps the most domain-dependent claim we shall make.

For a scientific research application, data analysis is typically performed as a sequence of operations, such as smoothing, thresholding, or performing a statistical test, each operating on the output of the preceding operation. Procedural representation is natural for this kind of application, since it conveys our users' (civil engineers) accustomed approach to data analysis.

Before any analysis is possible, data need to be obtained from the database. Relational query languages, such as SQL, are the current industrial standard for data retrieval. The procedural approach to data analysis contrasts with the primarily declarative nature of these languages. However, it is possible to express any data retrieval query procedurally as a series of operations in relational algebra [9].

In SQL, for example, a data retrieval query is expressed by the SELECT statement, which can be decomposed into seven relational algebra operations, namely, union, set difference, intersect, Cartesian product, selection, projection, and join. Conversely,

each of these relational algebra operations can be directly translated into SQL. Therefore, we can readily provide a procedural representation of a data retrieval query without introducing new concepts foreign to the relational data model. Further, the procedural representation of data retrieval operations is consistent with the representation of the data analysis operations, which we have argued are most naturally represented in a procedural form.

Representing the entire query (retrieval and analysis) as a sequence of operations on database tables provides a seamless integration of data retrieval and data analysis capabilities in a single query interface using familiar concepts of data retrieval and data analysis. Our experience suggests that a procedure-oriented query language is easier for our users to learn and use. We believe this may be the case for other scientific applications as well.

Although the operations provided in the language can be of high complexity, their meanings are always made explicit. Clever ‘undercover operations’ are avoided unless the context strongly implies their presence and their results are intuitively obvious to a user. Each operation is represented as an independent processing module, and side-effects are not allowed. Furthermore, the meaning of an operation does not vary with its position in a query.

To improve the readability of the query language, we also avoid flow control, loop structures and recursion. This can be done without sacrificing the expressiveness of the language since operations in the language are performed on a complex data structure, the table, taken as a whole. Loops that are needed to process each table row are embedded in these operations. Other operations that seem to require nested loops can usually be expressed in terms of the join and selection operators. However, looping used to process one table after another cannot be expressed in our language. We chose to forgo this rarely needed capability for the sake of representational clarity.

2.2 Expressiveness

Because we cannot predict at design time most of the queries (data retrieval and analysis) a user may need, the query language must be flexible and powerful enough to represent a large range of queries. As mentioned before, our language implements all of the seven basic operators in relational algebra. It provides all of the data retrieval and manipulation capabilities of the SQL SELECT statement, the vehicle for SQL’s data retrieval capabilities. Since the primary function of our language is data retrieval and analysis, other ca-

pabilities of SQL, such as data insertion and update, receive only limited support.

One of the design goals of this language is to integrate data analysis with data retrieval. Data analysis is domain specific, but even within a specific domain, there are a large variety of data analysis techniques. We allow a researcher to build his/her own data analysis program in any language, and then easily integrate the program into our visual language by entering a simple specification of the program in a configuration file. The system automatically constructs a module with full graphic user interface support for the user-defined program. This module functions like any other module in our visual language. Conceptually a user-defined function is a generalized function which maps m input rows of n columns to m' output rows of n' columns, where m and n do not necessarily equal m' and n' , respectively.

2.3 Provision of on-line help

Programming with a visual language is made easy by a programming environment that can render assistance at every step of query construction. This requires more than on-line documentation.

In our language, a query is visualized as a data-flow graph with operations of data retrieval, analysis, and presentation displayed as iconic modules. Any intermediate result can be accessed by connecting a presentation module to the output of any module within a query. Also, any module can participate in several queries at the same time.

A macro facility in the language allows a user to encapsulate a frequently-used sub-query as a single module. With macros properly defined, a user can construct and view queries at a proper level of abstraction.

Data in a query are identified by the names of their tables and columns, but a user does not need to memorize these names. An *input list* in each processing module displays the names of all the input columns directed into the module. When an input field of the module demands a column or table name, a user can select an item in the input list and copy the name using the mouse.

Finally, all available operations in the language are listed on-line in a pull-down menu, and the functionality of each operation is fully documented on-line.

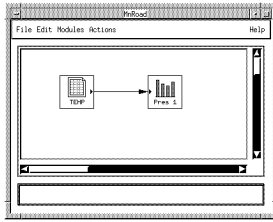


Figure 1: Main Window

3 SeeQL – a synthesis

SeeQL is a visual query language we have implemented according to the above design principles. Queries are built as directed acyclic graphs which describe the processing of data. The nodes on the graph are called *modules* and the lines between modules are called *arcs*. Conceptually, each arc carries one table between two modules.

The system runs under the X Window System. We use the Oracle Relational Database Management System [12] as our database engine and IDL (Interactive Data Language) [5] as our data visualization tool. IDL provides general two- and three-dimensional graphing capabilities, as well as certain data processing capabilities which can be used in User Function modules.

Starting the program brings up a single window with menus and a drawing area. Module icons are added to the drawing area using a pull-down menu. The module icons can then be moved around, singly or in groups. Arcs are drawn with the mouse. When modules are moved, the arcs redraw automatically. Subgraphs can be copied, pasted, or saved as files. Module icons can be ‘opened’ causing a dialog window to pop up, displaying the module’s parameters.

Wherever possible, the program enforces semantic rules. For instance, adding an arc will fail if that arc forms a cycle in the graph. In cases where errors cannot be prevented, the error condition is signaled by changing the color of the module to red during query execution.

3.1 Building simple queries

Figure 1 shows the main window, on which a trivial query has been drawn. The source module on the left is a Query module which retrieves the **TEMP** table containing average daily temperature data. This table flows down the arc into a Presentation module, which allows the results of the query to be displayed.

Figure 2 shows the dialog windows that pop up when these two modules are opened. The top win-

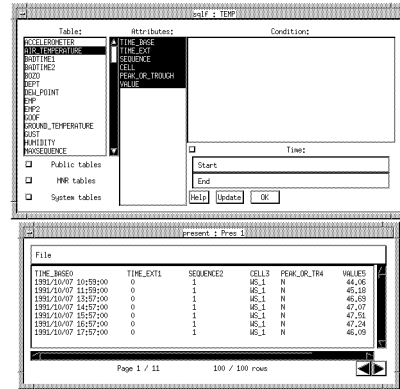


Figure 2: Module dialogs

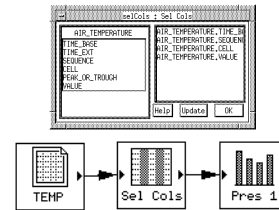


Figure 3: Column Select dialog

dialog belongs to the Query module. All tables available to this user are listed on the left. The Presentation module window shows the results of the query in a tabular format. This window also allows the text to be printed, plotted, saved as a text file or as a new database table.

More complex queries can be built by inserting modules into the graph. A simple transformation is to eliminate columns from the input. Figure 3 shows that a Column Select module has been inserted between the original modules and then opened. The dialog above it allows attributes to be selected and ordered.

Another simple function is to select rows that satisfy a certain condition using the Row Select module. Figure 4 shows a Row Select query added to the graph. New modules can be added anywhere. In this

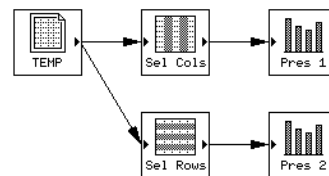


Figure 4: Adding a Row Select module

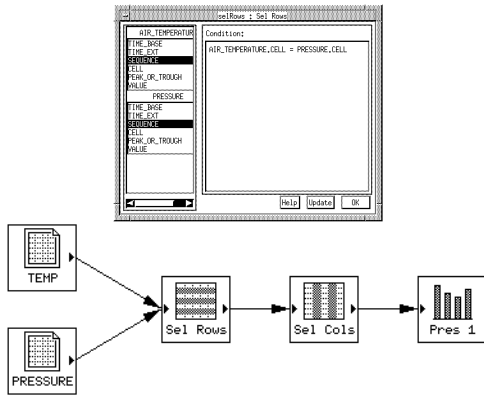


Figure 5: Join

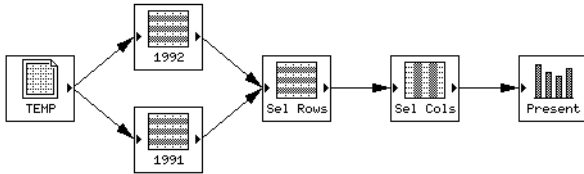


Figure 6: Self Join

case, rather than insert the module along the original flow, a new branch has been created. The output of any module can be split into any number of identical flows. The Row Select module is receiving the same table as Column Select. A particular strength of this program is that intermediate results can be tapped at any point in the query. One technique for debugging queries is to insert presentation modules into the middle of the graph.

Flows can also be combined. The Row Select module computes the Cartesian product of the incoming tables, before applying the selection condition. By specifying an equality condition, a database join can be performed. Figure 5 shows that a table containing temperature data has been joined to a table of pressure data using the condition `AIR_TEMPERATURE.CELL = PRESSURE.CELL`. The condition trims away all rows except those whose `CELL` fields match in both tables. A Column Select module has been added after the Row Select to delete unwanted columns.

Although arcs carry tables from module to module, the program remembers the database table from which each table column originated. Therefore, if the Column Select module in figure 5 were opened, temperature columns would still be listed separately from the pressure columns.

It is permissible for flows to diverge and then re-

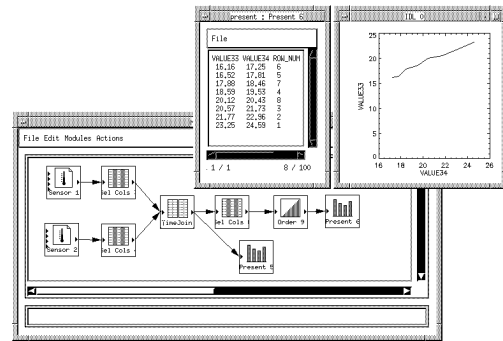


Figure 7: More Complex Query

converge, as long as no cycles develop in the graph. Figure 6 shows one table being passed to two Row Selects. The top one removes everything but the data from 1992 and the other restricts data to 1991. These two tables are then joined in a third Row Select. When this third Row Select module notices that it is receiving identical tables, it will create aliases for each flow before doing the Cartesian product. The condition that joins these two flows matches the rows from the 1991 and 1992 tables on their date values. The resulting rows allow one to compare the daily temperatures in the two years. The Column Select module then removes irrelevant data.

Figure 7 shows a query that examines the correlation between air and pavement temperatures. The Sensor 1 module retrieves air temperature values obtained by an automatic weather station over a one-day period in December 1992, while Sensor 2 retrieves pavement temperature values on the same day. The Column Select modules eliminate the irrelevant columns. Then, the Time Join module aligns the rows on their time values within a user-specified tolerance. Another Column Select module removes the time columns, leaving air and pavement temperatures which are then sorted, listed and plotted. The plot of pavement temperature *vs* air temperature shows that the variables are linearly correlated.

3.2 Modules

The following modules are available in SeeQL:

- *Query*: A generic source module for introducing any database table into the graph. It functions like a SQL `SELECT` statement.
- *Sensor*: A module for selecting sensor database tables.
- *Column Select*: Select from the list of columns in its single-table input. Corresponds to the Projection operation in Relational Algebra.

- *Row Select*: Forms a Cartesian product of all its input tables. Then selects rows that satisfy a logical condition. Corresponds to Selection and Join relational operations.
- *Group/Order by*: Lets rows of its input be ordered by one or more attributes. Also defines the grouping criteria for SQL group functions and user functions.
- *Cross Product*: Forms a Cartesian product of all incoming tables.
- *Union*: Performs the union set operation on tables. All tables must have similar columns.
- *Intersect*: Another set operation. Outputs only those rows common to all input tables.
- *Difference*: Performs a set difference on tables.
- *Arithmetic*: Allows a new numeric column to be added, its value determined by an arithmetic expression using other columns as input.
- *Rename*: Allows column names to be changed.
- *Presentation*: Displays the results of its input as a tabular list or as 2- or 3-D graphs.
- *Macro*: A module which represents an entire query subgraph.
- *Time Join*: A module for aligning input tables on their time columns, with user-specified starting times, duration, and alignment tolerance.
- *Group Functions*: Performs SQL functions on groups of rows, such as MIN, MAX, etc.
- *User-Defined Functions*: Modules which execute a user's UNIX programs for transforming tables. The user can write such programs in any language or with any software package and then incorporate them into the system.

4 Conclusion

We have described SeeQL, a visual query language for data retrieval and analysis. Although it was conceived with a particular application in mind, SeeQL can easily be customized for an array of applications using a relational database. It was designed to provide expressiveness, consistency, flexibility, and ease of modification in constructing complex queries.

SeeQL provides a suitable environment for a researcher to perform a wide range of data retrieval and analysis techniques. The uniform data representation is conceptually intuitive. The user can construct complex queries with virtually no knowledge of the underlying database organization or the underlying query language.

References

- [1] E. F. Codd, "How About Recently?," *Databases: Improving Usability and Responsiveness*, Academic Press, 1978.
- [2] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, *G+ : Recursive queries without recursion*, Proceedings of the 2nd International Conference on Expert Database Systems, 1989.
- [3] R. Elmasri and G. Wiederhold, "GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model", in P. Chen (ed.), *Entity-Relationship Approach to Information Modeling and Analysis*, North-Holland, Amsterdam, 1983.
- [4] *Idiot Proof your SQL Queries*, InfoWorld, Vol 38, Number 7, April 01, 1992.
- [5] Interactive Data Language, Version 2.2, Research Systems, Inc, Boulder CO, 1991.
- [6] J. R. Rasure and C. S. Williams, *An integrated data flow visual language and software development environment*, Journal of Visual Language and Computing, Vol 2, Issue 3, September 1991.
- [7] M. Schneider and C. Trepied, *Extensions for the graphical query language CANDID*, IFIP Transactions, Vol A-7, 1992.
- [8] M. Senko, "DIAM II with FORAL LP: Making Pointed Queries with Light Pen," *Proceedings of the IFIP congress 77*, Toronto, Canada, 1977.
- [9] J. D. Ullman, *Principles of Database Systems*, Computer software engineering series, Computer Science Press, second edition, 1982.
- [10] C. Upson, T. Faulhaber, Jr., D Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, *The Application Visualization System: A Computational Environment of Scientific Visualization*, IEEE Computer Graphics and Applications, 1989.
- [11] Moshe M. Zloof, "Design Aspects of the Query-By-Example Data Base Management Language," *Databases: Improving Usability and Responsiveness*, Academic Press, 1978.
- [12] ORACLE Relational Database Management System, Version 6.0, Oracle Corporation, 1989.