Report Number 96-02L

MINNESOTA DEPARTMENT OF TRANSPORTATION

DFQL A Graphical Data Flow Query Language for Retrieval, Analysis, and Visualization of a Scientific Database

Research

QA 76.7 .D22 1994

00-35733351

This report was distributed to the following:

1	...... Mn/DOT Library - MS 155 - Transportation Bldg - 395 John Ireland Blvd - St. Paul MN 55155
1	.... Secretary of the Senate - 231 Capitol Bldg - 75 Constitution Ave - St. Paul MN 55155
1	.... Clerk of the House - 211 Capitol Bldg - 75 Constitution Ave - St. Paul MN 55155
6	.. Legislative Reference Library - 6XX State Office Bldg - 100 Constitution Ave - St. Paul MN 55155

Technical Report Documentation Page

1. Report No. MN/RC - 96/02U	2.	3. Recipient's Accession No.
4. Title and Subtitle DFQL A GRAPHICAL DATA FLOW QUERY LANGUAGE FOR RETRIEVAL, ANALYSIS, AND VISUALIZATION OF A SCIENTIFIC DATABASE		5. Report Date October 1994 6.
7. Author(s) Sait Dogru, Vijay Rajan, Keith Rieck, James R. Slagle, Bosco S. Tjan, and Yuewei Wang		8. Performing Organization Report No.
9. Performing Organization Name and Address Computer Science Department University of Minnesota Minneapolis, Mn. 55455		10. Project/Task/Work Unit No. 11. Contract (C) or Grant (G) No. (C) 69750 TOC #89
12. Sponsoring Organization Name and Address Minnesota Department of Transportation 395 John Ireland Boulevard Mail Stop 330 St.Paul Minnesota, 55155		13. Type of Report and Period Covered Final Report 1993-1994 14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract (Limit: 200 words)

This report describes the design, functionality, and implementation of a data flow query language (DFQL), that visually represents a query as a directed acyclic data flow graph. The language is well suited for application domains such as scientific databases, where data analysis and visualization need to be tightly coupled with data retrieval, and the relationships between the entities are ill-defined. Unlike most visual query languages that are declarative, DFQL is a functional language and thus provides the ability to combine data retrieval, analysis, and visualization, intuitively in a single query.

17. Document Analysis/Descriptors Database Query Language Visual Query Language Graphical User Interfaces Data Flow Graphs		18. Availability Statement	
19. Security Class (this report) Unclassified	20. Security Class (this page) Unclassified	21. No. of Pages 23	22. Price

G:\RESEARCH\T2\REPORTS\96REPDOC.PGS\RPTDOC96.02U

DFQL
A Graphical Data Flow Query Language for Retrieval, Analysis, and Visualization of a Scientific Database

Final Report

Prepared by

Sait Dogru, Vijay Rajan,
Keith Rieck, James R. Slagle,
Bosco S. Tjan, and Yuewei Wang

Computer Science Department
University of Minnesota
Minneapolis, Mn. 55455

July 1994

Published by

Minnesota Department of Transportation
Office of Research Administration
200 Ford Building Mail Stop 330
117 University Avenue
St Paul Minnesota 55155

Contents

List of Figures

Executive Summary

We describe the design, functionality, and implementation of a data flow query language (DFQL), that visually represents a query as a directed acyclic data flow graph. The language is well suited for application domains such as scientific databases, where data analysis and visualization need to be tightly coupled with data retrieval, and the relationships between the entities are ill-defined. Unlike most visual query languages that are declarative, DFQL is a functional language and thus provides the ability to combine data retrieval, analysis, and visualization, intuitively in a single query. We have implemented this language in a system being used by the Mn/ROAD project.

1 Introduction

To define a query in a conventional database query language, a user must be familiar with the logical design of the database and the query language. With the recent advances in graphical user interfaces (GUI), visual query languages have been proposed to ease the user's burden of constructing a query. Almost all the visual query languages developed so far are declarative and rely on prior knowledge of the database schema. We argue that such declarative languages are not well suited to applications involving large scientific databases. In this paper we present an alternative approach: a query is represented visually and functionally as a data flow graph. We have designed and implemented the Data Flow Query Language (DFQL) (previously published as SeeQL [1], for its ability to allow users to visualize queries) to provide data retrieval and analysis capabilities for a large-scale relational scientific database used in a cold-region highway pavement research project (Mn/ROAD) conducted by the Minnesota Department of Transportation.

A query in DFQL is represented as a directed acyclic data flow graph and may include data retrieval, analysis, and visualization operations. Operations are represented by nodes in the graph and are called *modules*. Modules that retrieve data from the database are the sources of the data flow graph. Data presentation modules are the sinks. Between the sources and sinks are transformation modules performing data selection, transformation, and analysis. There is only one kind of data, called a *data table*, that flows from the sources, via transformation modules, to the sinks. A query is represented by a graph, rooted at a sink module, in the data flow graph. Different queries may share common sub-graphs. Thus a single directed acyclic graph with multiple sinks represents a set of queries.

Several graphical query languages have been proposed to bridge the gap between non-programmer users and traditional text-based query languages such as SQL [2] and QUEL [3]. A classical approach, IBM's QBE (Query By Example) [4], uses table templates to represent a query. While useful in constructing simple queries, its requirement of using constraint variables to relate different attributes from different tables makes it difficult to use for complicated queries. PICASSO [5] is a recently proposed system that uses hypergraphs to represent queries in a universal relation model [6]. It can be thought of as a generalization of QBE that overcomes some of QBE's problems. However, the novelty of the universal relation model and the unfamiliarity and complexity of the hypergraph representation of a query may present significant conceptual difficulties to non-programmer users. The language VQL [7] represents a query as a conceptual graph, allowing quantification, recursion, and variable-imposed constraints. The use of variables makes it hard to visualize a complex query in VQL, and its reliance on a novel data model (SSONET [8]) makes it difficult to use VQL with databases that use the traditional Entity-Relationship (ER) data model [9]. Other visual languages like QBD* [10], ERC [11], and VKQL [12] represent a query by an ER or Enhanced ER (EER) [13] diagram. Unlike the above-mentioned system, VisualReasoner [14] does not use a data model representation as a query language. Instead, a query is mapped to a visual space, and query construction is seen as an incremental refinement of the initial query by navigating in the visual space[14]. The drawbacks of VisualReasoner are that (1) a VisualNet has to be determined (manually) from the content of a database before any refinement operations can be performed on a query, and (2) it is unclear if any complex queries, such as a nested or quantified query, can be constructed using the refinement operations.

All these languages are declarative and rely on prior knowledge of the database schema. They are not suitable for a large-scale scientific database for the following three reasons.

- Most of the data in a scientific database are sensor readings and experimental results; although data intensive, the database is schema-poor. The relationships between the sensor readings are the subject of the on-going scientific investigations supported by the database, and are thus unknown at the database design time.
- It is difficult to understand and verify a declarative query in pieces because the meaning of one part of a query in general depends on the other parts.
- Scientific research requires a close integration of data retrieval (query) with analysis and visualization; however, data analysis and visualization are often represented functionally, and it is difficult to integrate these operations in a declarative query language.

An alternative to a declarative query language is to express a query functionally. Using a data flow paradigm to support functional programming has been well studied in the literature. Recent work on data flow programming languages include [15, 16, 17, 18, 19]. Some of these are general purpose programming languages, while the others are related to specific domains such as signal or image processing. Functionally oriented data flow languages [20, 21, 22] have also shown promise in scientific visualization.

The rest of the paper is organized as follows. Section 2 discusses the requirements of a visual query language for scientific applications. Section 3 describes the data flow query paradigm and the specific design of DFQL. Section 4 presents the data visualization capability of DFQL. Section 5 discusses various implementation considerations. Section 6 describes the user interface of the system. Finally, Section 7 concludes by describing the significance of this work.

2 Design Issues

In this section we discuss issues that are relevant to the design of a visual query language suitable for scientific databases. The most important reason for having a visual query language for a scientific database application is to provide a simple but integrated environment for data retrieval, analysis, and visualization. In contrast to a typical business application, the data model of a scientific database cannot specify all the relationships that exist between the data. The data collected from a series of experiments may be used to support several research projects, each defining and identifying different relationships between the data entities. Therefore, a suitable query language should be flexible and not rely on the database having a precise data model.

Three principles guided our design of a visual query language: conceptual simplicity, expressiveness, and provision of user assistance.

2.1 Conceptual Simplicity

Because the language is targeted for non-programmer users, it should be conceptually simple. To achieve conceptual simplicity, we identified five requirements:

- *Adoption of a functional orientation.* We think that a functional query language is conceptually simpler than a declarative one for scientific applications because (i) it is a paradigm that our users are already familiar with in other domains such as data analysis and visualization, (ii) it allows natural intermixing of data retrieval operations with data analysis and visualization, and (iii) a query specified in a functional query language can be constructed or understood in pieces, since the meaning of an operation is independent of the other operations in the query, and intermediate results from part of a query can always be made available.

- *Maintaining compatibility with the relational data model and query operations.* We want to avoid introducing unnecessary new concepts to our users. We chose to use the relational data model for its uniformity, simplicity and popularity.

- *Use of a single data type for all the operations in the language.* Ideally, the query language should be type-free so that our users are not burdened by having to match different data types for different operators. We use a rich data type, called a *data table* (see Section 3.1), to represent all data to be operated on by the language.

- *Use of understandable, explicitly stated operations in a query.* No operation should have implicit functionality. The meaning of an operator should be clearly defined and should not be context-dependent.

- *Avoid iteration and recursion.* To allow iteration or looping mechanisms at the data flow level in a functional query language requires the introduction of complicated new concepts such as synchronization and explicit flow control. While recursion provides an elegant implementation of looping in a functional language, introducing this concept to a non-programmer user is difficult. Avoiding iteration and recursion at the data flow level in our language does not significantly affect its expressiveness.

2.2 Expressiveness

Since queries to a scientific database are unconstrained in their form, our query language must be powerful enough to represent all queries a user may need. DFQL implements all the basic operators of relational algebra, and is therefore relational-complete. It provides all of SQL's data retrieval capabilities as defined by the SQL SELECT statement. DFQL does not allow explicitly nested queries. However, as demonstrated in Section 7, nested queries can be "flattened" and expressed in DFQL. Since the primary function of our language is data retrieval and analysis, other capabilities of SQL, such as table creation and update, are omitted from DFQL, but remain accessible via the DFQL system environment.

Iteration and recursion at the data flow (i.e., table) level is not allowed in DFQL. However, iteration on the rows of a table is embedded in the operations themselves. An implication of this limitation is that like in SQL, a DFQL query cannot be formulated to obtain the transitive closure of a relationship (e.g. find all descendants of Joe Smith, given a PARENT_OF relationship).

In addition to data retrieval operations, a query language for a scientific database also needs to be expressive in data analysis and visualization operations. The approach we have taken in DFQL is to allow the user to *define* new operators in terms of external programs and to integrate them into the language as new operators on data tables. These external programs can be viewed as generalized aggregate functions (see Section 3.3.2) and can appear anywhere in a query.

2.3 User Assistance

User assistance means more than on-line documentation. A query should be visualized graphically, and assistance for query construction and troubleshooting should also be provided.

By using a data flow representation of a functional query language, a query in DFQL is clearly visualized as a network of operations performing data retrieval, analysis, and visualization. An operation is displayed as an iconic module. The column names of the input data tables of a module are always displayed in the module (see Section 6.3). Any intermediate result can be accessed by connecting a *Presentation* module (see Section 3.4) to the output of any module within a query.

Context-sensitive on-line documentation is also provided in the DFQL environment.

3 Data Flow As A Database Query Paradigm

A query in DFQL is an acyclic data flow network whose nodes are called *modules.* Data flows from a *source module* to a *sink module* through zero or more intermediate *transformation* modules. Every module provides one or more *input ports* and an *output port.* An arc can be attached to a module only at these ports.

3.1 A Single Data Type

In DFQL, every module operates on a single data type, a *data table.* Each arc in the graph carries a data table. A data table is similar to a relational database table in that its content is organized in rows and columns. Each column of a data table will have its own atomic type, but this type is invisible to DFQL at the *flow* level. Possible column data types include numeric, character, boolean, time, and date. In a relational database table, all the columns are physically stored together and (implicitly) qualified by the same table name. In a data table, however, each column has its own table identifier, which may or may not be the same as the other columns in the table. A data table also allows duplicate rows, i.e., two rows might be the same in a data table. For simplicity, we will refer to a data table simply as a table for the rest of this paper.

3.2 Source Modules

A source module in DFQL *retrieves* a user-specified table from the underlying database. The selected table is then propagated through every out-going arc of the source module. DFQL has three source modules.

Query: This is the simplest source module. It allows the user to retrieve data from any accessible database table. The module also lets the user select only the desired columns and/or specify a condition to restrict the rows being retrieved.

Sensor: This source module provides an easier interface for retrieving application-specific sensor data. Sensors may be chosen by their type and location. The module also allows the users to specify a time range for the data selected.

Time Points: The final source module generates simple tables of *time points.* A time point consists of a date and a time value. For example, 25 minutes after 10 AM on May 5, 1994 can be represented as the following time point: *1994/05/07 10:25:00.* These tables are typically used to synchronize joins on time points.

3.3 Transformation Modules

A transformation module receives one or more tables from its input arcs, processes them to produce a new table, and then passes the new table out. How the input tables are processed depends on the particular transformation module. We divide the transformation modules into two classes: *relational* and *extra-relational.*

3.3.1 Relational Transformation Modules

Modules in this class have correspondences in relational algebra and relational query languages like SQL. We have adopted the same or similar names for each.

Select Column: This module does projection, which is one of the standard relational operators. It may have only one input arc. Only the selected columns from the input table will be included in the output table. This module can also be used to reorder the columns. If needed, a column may also be duplicated.

Cross Product: This module generates the Cartesian product of all its input tables. Each row in a Cartesian product contains all the columns from all input tables. There will be one row for each possible combination of input rows. For instance, given a table with m rows and n columns, and another with p rows and q columns, the product would have $m \times p$ rows and $n + q$ columns.

Select Row: The selection operation is handled by this module. The module allows the user to enter a condition. The output will contain only those rows for which this condition is true. This module includes an option for making the output *distinct.* That is, only unique rows will be passed through. If more than one input arc is attached to it, the module will form a Cartesian product of all the input tables before applying the condition. This operation is better known as a *join.*

Union and Intersect: The union and intersect set operations are handled by these modules. Each takes multiple inputs and combines tables by union or intersect.

Set Difference: This module is similar to Union and Intersect, except that it has two input ports. Its output represents the table entering its top input port minus the table entering its bottom input port.

Time Join: The database for which DFQL was developed consists primarily of sensor readings, so this project is very concerned with manipulating time points. Of particular interest to the users is the ability to match sensor readings that occurred at about the same time, since the readings are never perfectly synchronized. The *Time Join* module can join sensor tables on their time points, where two time points that are within a user-defined period apart are considered to be *aligned.*

3.3.2 Extra-Relational Transformation Modules

An important advantage of DFQL over SQL and other visual query languages is its ability to offer an integrated framework that combines standard as well as non-standard operators and transformations in a natural and intuitive way. Although it is impossible to envision and list all such transformations, we describe the general framework in the User Functions module. Group Functions are similar to User Functions, but are offered as aggregate functions by SQL.

Arithmetic: This module allows the user to specify an arithmetic expression and have the value of that expression propagate out as a new column in the output table. Note that the numerical value is generated for each row from the values in the same row.

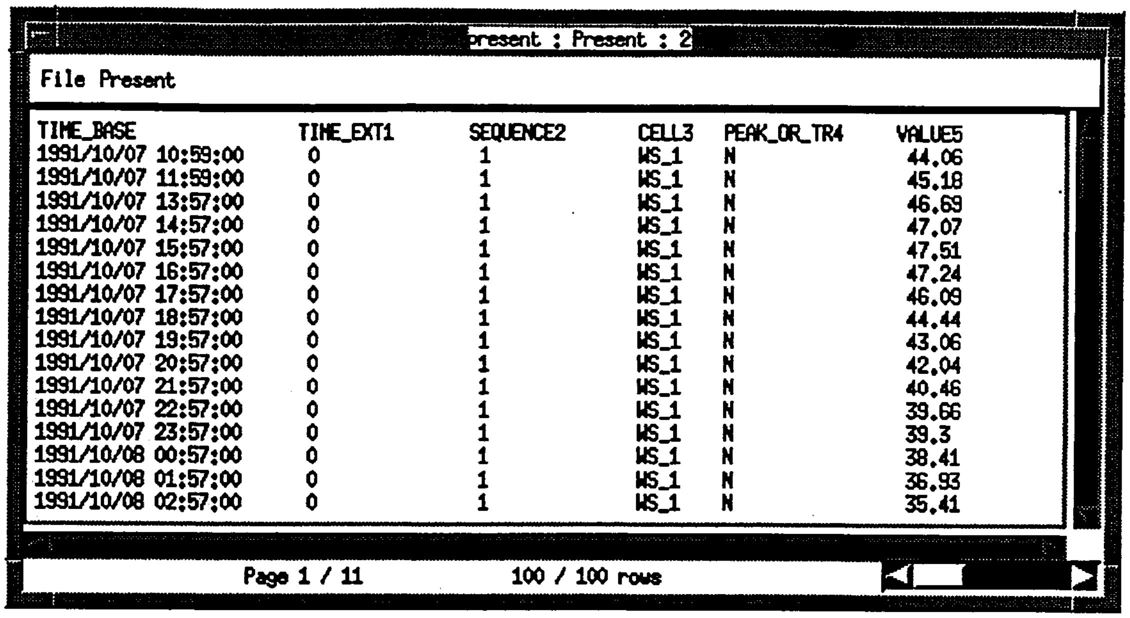
TimeToNum and **NumToTime:** Sometimes it is preferable to convert a time point value to a numeric one to perform analysis or visualization on the data collected at that specific time. This is also necessary as not every analysis tool has the capability to handle time and data values. These two modules are provided to convert time point values to numeric values and back.

Order: This module imposes an ordering on a table.

Group Functions: These are standard aggregate functions that operate on *logical groups* of data to compute a single value for the group. More formally, a group function is a mapping from m rows to 1 row (or from an $m \times 1$ matrix to a 1×1 matrix). For example, one might wish to find the coldest temperature recorded by a particular sensor in a specific period of time. These functions are handled in much the same way as User Functions, described below.

User Functions: The main purpose of data retrieval is analysis, more than likely involving complex transformations of the data. However, the standard relational modules are only useful for the retrieval of the data. As was mentioned earlier, DFQL is more powerful than the relational algebra and SQL derivatives. DFQL offers an integrated framework where non-standard operators and transformations can be applied to tables easily. A Fast Fourier Transformation (FFT), for instance, can be implemented by the user in any programming language such as Fortran, C, or C++. Users can also implement their own statistical analysis routines in SAS or SPSS, data/signal processing in IDL, and even complex queries in a query language like SQL. By following a simple integration step provided by the framework, the user is able to perform any customized operation within DFQL. This framework allows the user to integrate a multitude of such useful utilities into DFQL, for each of which a new module performing the corresponding transformation is created.

Formally, a user-defined function is a mapping from an $m \times n$ matrix to a $p \times q$ matrix, where there is no constraint imposed on m, n, p, and q. Data from a table are partitioned into groups by a mechanism similar to "GROUP BY" in SQL, and the user function is applied to each group separately.



```
TIME_BASE            TIME_EXT1   SEQUENCE2   CELL3  PEAK_OR_TR4   VALUE5
1991/10/07 10:59:00   0           1          WS_1   N             44.06
1991/10/07 11:59:00   0           1          WS_1   N             45.18
1991/10/07 13:57:00   0           1          WS_1   N             46.69
1991/10/07 14:57:00   0           1          WS_1   N             47.07
1991/10/07 15:57:00   0           1          WS_1   N             47.51
1991/10/07 16:57:00   0           1          WS_1   N             47.24
1991/10/07 17:57:00   0           1          WS_1   N             46.09
1991/10/07 18:57:00   0           1          WS_1   N             44.44
1991/10/07 19:57:00   0           1          WS_1   N             43.06
1991/10/07 20:57:00   0           1          WS_1   N             42.04
1991/10/07 21:57:00   0           1          WS_1   N             40.46
1991/10/07 22:57:00   0           1          WS_1   N             39.66
1991/10/07 23:57:00   0           1          WS_1   N             39.3
1991/10/08 00:57:00   0           1          WS_1   N             38.41
1991/10/08 01:57:00   0           1          WS_1   N             36.93
1991/10/08 02:57:00   0           1          WS_1   N             35.41

            Page 1 / 11          100 / 100 rows
```

Figure 1: The Presentation window

3.4 The Sink Module

A sink module may be thought of as a receptor for all the data that flow down from the sources of the graph. Currently, there is only one sink module.

Presentation: The Presentation module provides a simple, plain text rendition of the data it receives, as shown in figure 1. Each column in the data is listed under a title generated from the name of the output column, by appending a unique number to the column name. The data is then displayed in a window in *pages*. The window also provides a menu of useful options to the user. The user can update the display after changes to the query, save the result to a text file or as a table, or view the result using a visualization package (see Section 4).

A Presentation module can be attached to any module in a query graph to view the partial results, without affecting the semantics of the graph being constructed: it provides only a *peek* into the data at that point in the query.

An important addition to the Presentation module is the ability to do *transorder* manipulations. This allows you to group data by some value and then arrange the groups horizontally. For instance, since a sensor table may contain readings for many different cells, there may be several readings for each time point. A transorder on time will reduce this table to just one row for each time point, but with separate columns for each cell. This is an important new feature that is not available in other analysis packages.

4 Data Visualization

Although plain text rendition of the results of a query is very useful, especially when used for intermediate results, it is nonetheless a cumbersome method of visualizing the data. A trend in the values of a column, for example, is hard to observe by looking at the individual row values of the column in page after page. It is even harder to correlate and analyze trends involving various columns. With only the plain text presentation utility at their disposal, end users would thus be forced to manually move their results outside the system and use a separate visualization application. This was incompatible with the initial design goals of DFQL.

It was therefore important to integrate a complete visualization package within the framework of DFQL. Interactive Data Language (**IDL**) by the Research Systems, Inc. (**RSI**) was chosen as the underlying platform on which to support and provide the above-mentioned utilities to the end users [23].

IDL is a complete package for the interactive analysis and visualization of scientific and engineering data. IDL is a powerful array-oriented programming language with numerous mathematical functions built in. It supports an OpenLook or Motif widget interface to the X-Window system through a few useful widget classes.

We have implemented a general purpose graphical visualization interface to DFQL that we call *Mnplot*. Mnplot is accessible through the Plot menu items from Presentation modules. It also can be used separately from DFQL as a stand-alone visualization package.

Mnplot consists of a *control window* and one or more *draw windows*. The single control window is used to manage the draw windows. The control window features extensive options and utilities in a menu-driven, point-and-click interface. Although the control window can manage an arbitrarily large number of draw windows, at any given moment, it operates on a specially designated one, called the *current* draw window. Figure 2 shows Mnplot draw windows with a 2D line plot and a 3D surface plot. The current draw window is specially marked to distinguish it from the others. The user can use either the control window or the mouse to navigate through the draw windows. Some of the features available to the user through the control window are:

- Create and name graphs by defining its X, Y, and Z components.
- Plot the graphs, possibly overlaid with a previous plot in a number of styles, including business-style charts, 2D plots, and 3D wire-mesh and surface plots.
- Customize any of the graphs in terms of their line style, thickness, color, title, etc.
- Change the margins and axes ranges.
- Print the contents of a draw window in various formats, including PS, EPS, and HPGL.
- Annotate the plots.

In addition, a draw window also lets the user navigate, pan the plots, or zoom in on the plots using a mouse.

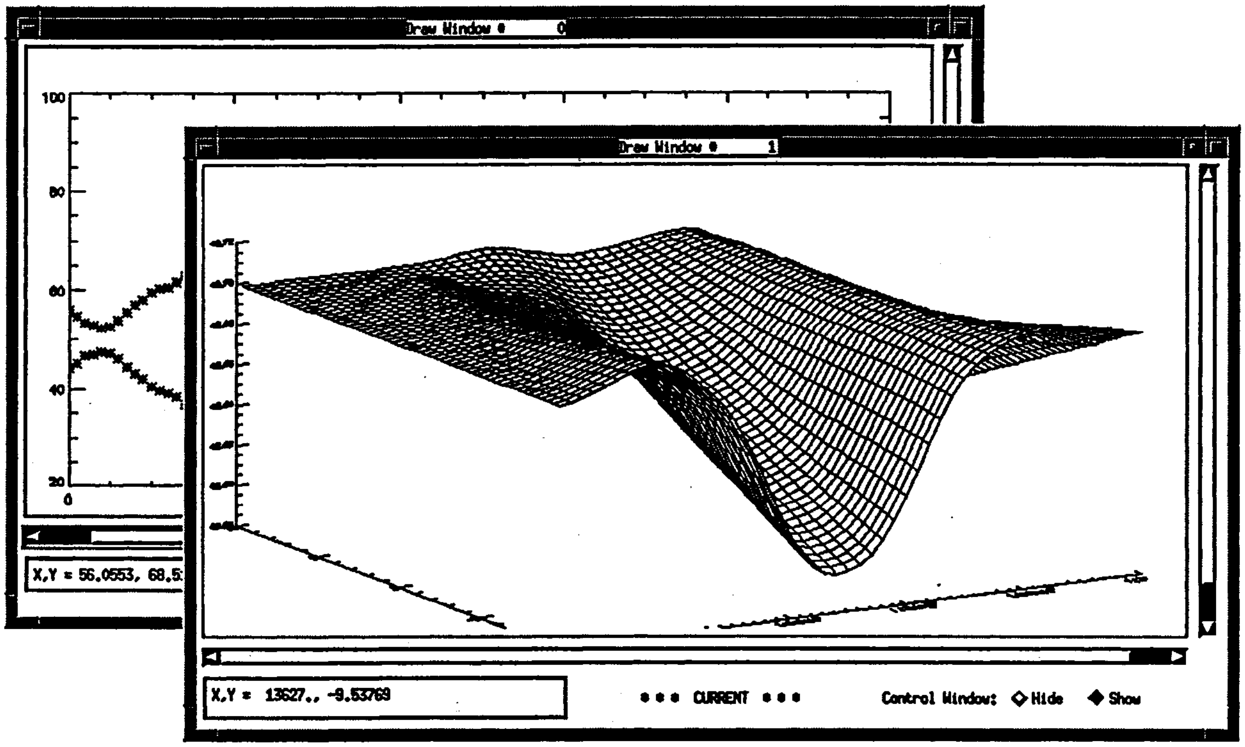


Figure 2: Mnplot draw windows

5 Implementation Issues

DFQL has been implemented using various languages, tools, and software packages. Figure 3 shows what systems have been used and how they all fit together.

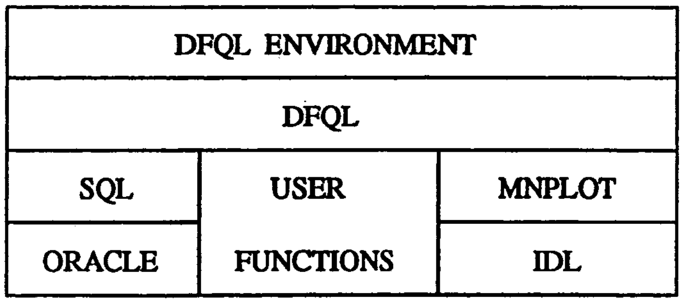


Figure 3: The architecture of DFQL

5.1 From Graphs to Data: Tracing The Execution

Viewed at the conceptual level, what flows along an arc from one module to another is a table. A source module introduces a database table into the graph. The table goes through zero or more transformation modules. Each transformation module processes its input tables in its own way and creates as output a new table. This process terminates eventually in a sink module. The execution of the graph can be thought of as a translation from each relational DFQL module to an equivalent SQL statement and each extra-relational DFQL module to an external function call.

Although this is a conceptually accurate description, it is a naive approach for the implementation. Building a new database table for each module would be extremely inefficient both in terms of query execution time and the use of database space. A solution is to combine as many DFQL operations as possible into a single SQL statement and delay the execution until needed. Consider for example, a column selection module. Independent of its position in a graph, the execution of this module can be delayed until a sink module is reached. We need only to create a suitable schema for the resulting table, which should include only the selected columns of its input. There can be many such modules in a query graph. An important implication of this method is that a DFQL graph ending in a single sink and constructed completely of such modules, would correspond to a single, albeit complex, SQL statement.

However, there are some modules that cannot be treated as above. These are exactly the modules that make DFQL more expressive than SQL or other visual query languages. An example is a user-defined function module performing FFT on its input. We can still create a schema for the resulting table, but we can no longer hope to maintain the continuity of a single query, mainly because of the limitations imposed on us by SQL. We distinguish such modules from the rest and call them *Xforms*, or Execution Form modules.

Xform modules are the key to another important implementation decision. Since creation of temporary tables is expensive, we wish to minimize the number of times they are recreated. Processing of DFQL graphs is divided into three passes: flagging, updating and executing.

When some parameters are changed on a module's dialog window, those changes are made to the module's data structure. The module, and all modules downstream, are flagged as requiring update. Also, all downstream Xforms have their temporary tables flagged as needing update. Flagged modules are then updated in topological order, moving downstream. To update a module, DFQL builds a list of columns entering the module, the module's SQL statements, and then the list of outgoing columns.

Execution rebuilds the temporary tables. This pass isn't performed until the user opens a sink module. DFQL finds all Xforms on which the sink depends. These modules are again processed in topological order and the temporary tables are recreated.

5.2 Intermediate Xform Tables

The discussion of Xforms above does not describe how the temporary tables created by Xforms are handled under the abstraction principle of DFQL. DFQL makes an effort to keep such intermediate tables invisible to the user. For instance, all columns in a temporary table that can be traced back to an original database table retain their original names. The new columns that are generated as a consequence of the particular transformation are given default names, although the user is allowed to explicitly name them. Thus, as far as the user is concerned, no intermediate table is created but the input table is changed by the transformation, i.e., new columns are added and/or existing columns removed.

5.3 Aliasing

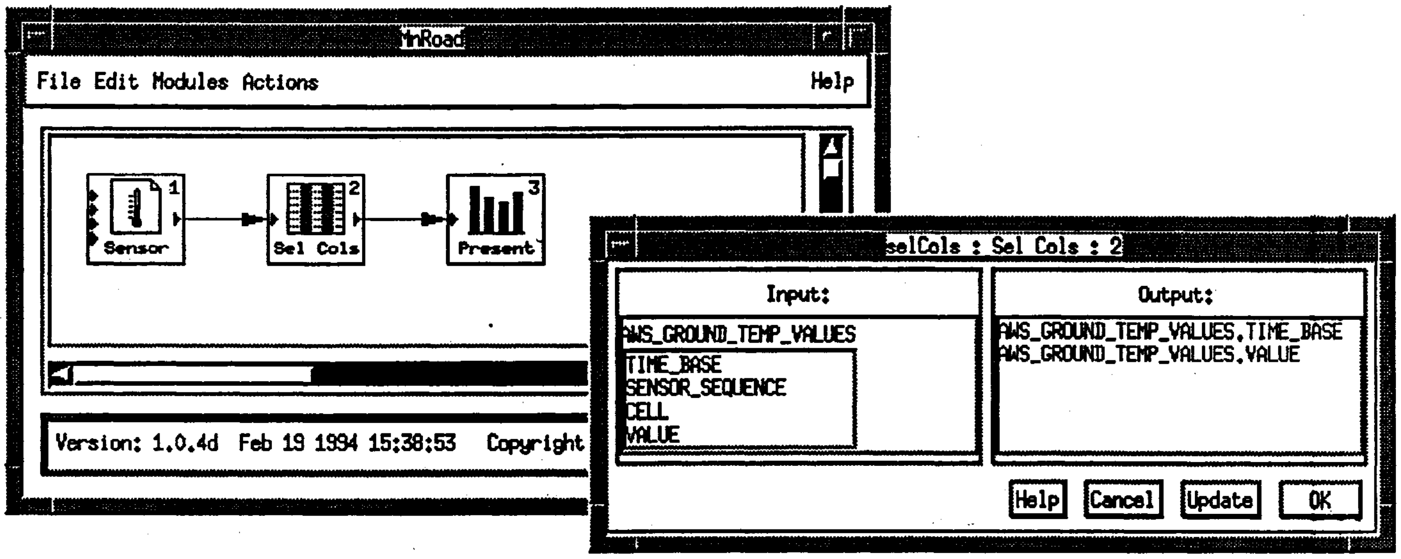
DFQL also addresses another kind of naming conflict. It is possible in a query graph for paths to fan out and then fan back in, or for different paths, each with its own source module but accessing the same database table, to merge at a later point. Either of these situations results in a module getting inputs from two or more tables with the same name. It is important to realize that each input arc carries a different *copy* of the same table and that they must be treated as though they were different tables. DFQL uses *aliases* to resolve this table name conflict.

6 User Interface

6.1 Programming in DFQL

Programming in DFQL consists of building data flow graphs. A complex problem can be broken up into a sequence of simpler operations, each of which is performed by one of the transformation modules. The main program window of DFQL (figure 4) consists of a menu bar, a canvas on which the query is constructed, and an area at the bottom where diagnostics are displayed. The pull-down menu titled "Modules" lists all the available modules. When a particular module is selected from this menu, the module is placed on the canvas. Modules can be moved around the canvas as required. The user builds a query by placing modules and then connecting them to form a data flow graph.

DFQL provides features to easily construct and modify data flow graphs. A module or a subgraph consisting of a set of modules, can be added, deleted, copied, or pasted. Two modules are connected by creating an arc from one to the other. An arc between two modules can also be broken.



Version: 1.0.4d Feb 19 1994 15:38:53 Copyright

Input:

AWS_GROUND_TEMP_VALUES
TIME_BASE
SENSOR_SEQUENCE
CELL
VALUE

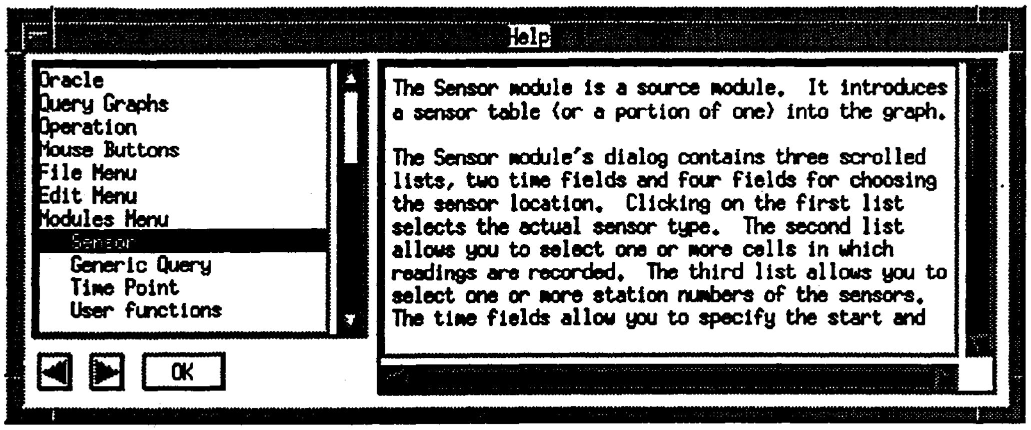
Output:

AWS_GROUND_TEMP_VALUES.TIME_BASE
AWS_GROUND_TEMP_VALUES.VALUE

Figure 4: The DFQL main window and Select Column dialog

6.2 Dialog Windows

Once a data flow graph is constructed, it is not yet ready to be executed. Most modules also require the user to specify some parameters. Such modules have a dialog window associated with them that can be opened using the mouse. Figure 4 shows the window associated with the Select Column Module. When the user specifies the required parameters, the module changes its color from red to blue. When every module comprising a query is blue, the query is ready to be executed.



The Sensor module is a source module. It introduces a sensor table (or a portion of one) into the graph.

The Sensor module's dialog contains three scrolled lists, two time fields and four fields for choosing the sensor location. Clicking on the first list selects the actual sensor type. The second list allows you to select one or more cells in which readings are recorded. The third list allows you to select one or more station numbers of the sensors. The time fields allow you to specify the start and

Figure 5: The Help window

6.3 Input list

Typically the user performs an operation on some of the columns of the tables that flow into the module. Dialog windows have a section called the *Input List*. All tables entering the module, along with their associated columns, are displayed in the input list, and the user can select the required columns with a mouse. In figure 4, the input list takes up the left half of the Select Columns dialog. The right half of the dialog shows the columns that have been selected by the user to be in the output table.

6.4 On-line documentation

The help menu on the main window provides on-line documentation on all the functionality provided by DFQL. Figure 5 shows the help window. The left part of the window lists the topics, and the right part the text for the particular topic. All dialog windows also have a help button. Pressing the help button on a dialog window will bring up the help window and go directly to the particular topic.

6.5 Execution

Whenever a Presentation module is opened, the system begins executing the queries as described in Section 5. During this time, the cursor changes to a watch to indicate that the user must wait. As modules are processed, their borders are temporarily expanded. This allows users to watch execution move across the graph. Frequently, query results will be larger than expected and execution will spend too much time on one or more modules. If execution runs too long, users may cancel the query and go back to editing the graph.

7 Conclusion

The design and successful implementation of DFQL demonstrates the effectiveness and practicality of a functional approach to tightly integrating data retrieval from a database with data analysis and visualization. The functional approach lends itself naturally to a visual representation of a query with the use of a data flow graph.

This functional approach to querying a database does not rely on prior knowledge of the data model of the database. It is, therefore, particularly suitable for a database that is schema-poor or for which the data model cannot be precisely determined, as is the case for most data-intensive scientific databases.

DFQL can be used to access traditional schema-intensive databases, such as those used in a business application. However, an additional tool to browse the database schema or to constrain query construction based on the schema may prove helpful to a user in this case.

All the current users of DFQL are civil engineering researchers involved with the Mn/ROAD pavement research project. We found productive use of the system by our users after the three system demonstrations totaling about five hours. We observed no conceptual difficulties on the user's part in understanding DFQL. Further training is nevertheless needed for the users to make the best use of DFQL.

References

[1] B.S. Tjan, L. Breslow, S. Dogru, V. Rajan, K. Rieck, J.R. Slagle, and M.O. Poliac. A data-flow graphical user interface for querying a scientific database. In *1993 IEEE Symposium on Visual Languages*, pages 49–54. IEEE Computer Society, Aug 1993.

[2] D.D. Chamberlin and R. F. Boyce. Sequel: A structured English query language. In *Proc. ACM SIGDIFET Workshop*, 1974.

[3] P. Kreps M. Stonebraker, E. Wong and G. Held. The design and implementation of INGRES. *ACM TODS, vol.1, no.3*, pages 189–222, 1976.

[4] M. Zloof. Query by Example. In *Proc. National Computer Conf.*, pages 431–438, May 1975.

[5] H.J. Kim, H.F. Korth, and A. Silberschatz. PICASSO: a graphical query language. *Software - Practice and Experience, vol.18, no.3*, pages 169–203, Mar 1988.

[6] D. Maier and J. D. Ullman. Maximal objects and the semantics of universal relational databases. *ACM TODS, vol.8, no.1*, pages 1–14, 1983.

[7] L. Mohan and R.L. Kashyap. A visual query language for graphical interaction with schema-intensive databases. *IEEE Transactions on Knowledge and Data Engineering, vol.5, no.5*, pages 843–858, Oct 1993.

[8] L. Mohan and R. L. Kashyap. A framework for building knowledge-intensive data models. In *Proc. 2nd Int. Conf. Software Engineering and Knowledge Engineering*, pages 33–38, Jun 1990.

[9] P. P. Chen. The Entity-Relationship model: toward a unified view of data. *ACM Trans. Database Systems, vol.1, no.1*, pages 166–192, 1976.

[10] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: a fully visual system for E-R oriented databases. In *1989 IEEE Workshop on Visual Languages*, pages 56–61, Oct 1989.

[11] B. Czejdo, M. Rusinkiewicz, D. Embley, and V. Reddy. A visual query language for an ER data model. In *1989 IEEE Workshop on Visual Languages*, pages 165–170, Oct 1989.

[12] K.L. Siau, H.C. Chan, and K.P. Tan. Visual knowledge query language. *IEICE Transactions on Information and Systems, vol.E75-D, no.5*, pages 697–703, Sep 1992.

[13] M. Junet. Design and implementation of an Extended Entity-Relationship database management system. In *Proc. 5th International Conference on Entity-Relationship Approach*, pages 199–216, 1986.

[14] S.K. Chang. A visual language compiler for information retrieval by visual reasoning. *IEEE Transactions on Software Engineering, vol.16, no.10*, pages 1136–1149, Oct 1990.

[15] M.B. Gokhale. Programming in a very high level data flow language. *ICS 88. 3rd International Conference on Supercomputing, vol. 3*, pages 366–71, 1988.

[16] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL: a data flow oriented language for real time signal processing. *Computational and Combinatorial Methods in Systems Theory*, pages 419–33, 1986.

[17] N. Hunt. IDF: A graphical data flow programming language for image processing and computer vision. In *Proceedings of 1990 IEEE International Conference on Systems, Man and Cybernetics*, pages 351–60, Los Angeles, CA, USA, Nov 1990.

[18] A. Knoll, A. Schweikard, and M. Freericks. Data-flow oriented functional programming language for real-time data processing. In G. Hommel, editor, *Prozessrechensysteme '91. (Process Computer Systems '91)*, pages 226–35, Berlin, Germany, Feb 1991. Springer-Verlag and Berlin, Germany.

[19] T. Shimada, S. Sekiguchi, and K. Hiraki. Data flow language DFC: design and implementation. *Transactions of the Institute of Electronics, Information and Communication Engineers D, vol.J71D, no.3*, pages 501–8, Mar 1988.

[20] J. Barber. LabVIEW: an implementation of data flow programming in a graphical language. In *Proceedings of the ISA/89 International Conference and Exhibit, vol.3*, pages 1259–66, Philadelphia, PA, USA, Oct 1989. ISA; Research Triangle Park, NC, USA.

[21] 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*,, Sep 1991.

[22] C. Upson, Jr. T.A. Faulhaber, 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, vol.9, no.4*, Jul 1989.

[23] *Interactive Data Language, Version 2.2.* Research Systems, Inc., Boulder CO, 1991.