A Simple Visualization Management System: Bridging the Gap Between Visualization and Data Management

Peter Kochevar
Digital Equipment Corporation

Zahid Ahmed
Jonathan Shade
Colin Sharp
San Diego Supercomputer Center

April 30, 1993

Abstract

A prototype *visualization management system* is described which merges the capabilities of a database management system with any number of existing visualization packages such as AVS or IDL. The prototype uses the Postgres database management system to store and access Earth science data through a simple graphical browser. Data located in the database is visualized by automatically invoking a desired visualization package and downloading an appropriate script or program. The central idea underlying the system is that information on how to visualize a dataset is stored in the database with the dataset itself. As a result, scientists can concentrate more on their science rather than on the process of doing it since visualization programs do not have to be created or searched for each time a dataset is to be viewed.

1 Introduction

State-of-the-art visualization systems, as exemplified by AVS [2], IDL [11], and Data Explorer [6], are quite sophisticated in their abilities to manipulate and render data. However, they are still lacking as complete *visualization management*

systems (VMS). Most visualization systems in and of themselves are poor data managers in that they give little built-in support for finding pertinent datasets other than offering a simple file browsing mechanism. This minimal capability is not so much a problem when there are few datasets residing locally on disk and one knows the names and locations of the files containing the needed information. But, if one is not familiar with the names, locations, or even the contents of files containing the data then serious problems arise. What is needed is a database management system (DBMS) to complement visualization packages along with an effective browsing and storage tool to allow easy navigation and interaction with the information in a database.

The desire for a more complete VMS is driven by the requirements of certain segments of the scientific community. In particular, Earth scientists in the Sequoia 2000 Project have expressed a need for sophisticated data management to go along with scientific visualization [8]. Sequoia is a unique blend of Earth and computer scientists who are working together to create a computing infrastructure that will foster a better understanding of global environmental change [5]. The entire field of global change is characterized by the massive amounts of information pertaining to Earth that are and will become available. The ready search for and subsequent access to this information is as important to a global change researcher as is its actual visualization.

The following paper describes a very simple VMS constructed as an early prototype within the Sequoia project. The VMS was formed by linking the Postgres DBMS [12] with several off-the-shelf visualization systems. The heart of this system is the *Visualization Executive* which is primarily responsible for locating information using a graphical database browser tailored to the Earth sciences. Once a dataset is located, an appropriate visualization system is started as a separate process, a program or script which actually visualizes the data is instantiated within the visualization package, and finally the raw data composing the dataset is ingested and converted to the internal format used by the visualization package. The key idea which makes the operation of the Visualization Executive possible is that information as to how to visualize a dataset is stored in the database along with the dataset itself.

In the balance of the paper, the Postgres DBMS will be briefly described followed by an outline of how Postgres was used to create and manage a simple Earth science database. Next, a description of the design and implementation of the Visualization Executive will be provided. Finally, a discussion will ensue about the functionality and usability of the prototype VMS and a number of ways of improving its design will be offered.

2 The Postgres Database Management System

Postgres is an *extended* relational system so called because it possesses a number of features that distinguish it from most other relational DBMSs. Some of these features include user-defined types, inheritance, user-defined functions, and support for storing large non-relational data such as satellite images. To use these features, Postgres comes equipped with its own set-oriented command language, Postquel, through which one creates, alters, and queries for information.

2.1 Basic Features

The fundamental concept in Postgres is that of a relation which Postgres calls a *class*. A class is a named collection of objects all of which have the same record structure. These objects, referred to as *instances* of the class, consist of a set of named *attributes* each of a specific built-in or user-defined type. The attributes correspond to fields in a record whose structure characterizes the definition of a given class.

Inheritance

Postgres supports a limited form of inheritance whereby all the fields in the record structure of class A are appended to the fields comprising the record structure of class B. In this case, class B is said to *inherit* from class A and class A is called class B's *parent*. In general, a class can inherit from multiple parent classes provided that the attribute names of each parent are unique.

An *inheritance graph* is a directed acyclic graph that reflects the inheritance relationships among the classes in a database. Nodes in the graph correspond to classes and a directed edge points from a node representing class A to a node representing class B if B inherits from A. All classes corresponding to nodes that can be reached by traversing a chain of edges that begin at the node representing class A are called *sub-classes* of class A. In general, queries in Postgres can be directed toward a class and all its sub-classes, or just the class itself.

User-defined Functions

Postgres allows the definition of two kinds of functions distinguishable by the way they are implemented. One kind of function is a macro containing a sequence of Postquel commands while the other type is implemented in a general-purpose programming language such as C. Both kinds of functions can have parameters which may be values of any Postgres type or they may be instances of some

class. The return values of functions are likewise either of some Postgres type or they are a set of class instances. Functions defined within Postgres can be used as part of most any database query. Of special note is the fact that object files of functions implemented in a general-purpose programming language are stored within the database itself. When such functions are referenced within a query, Postgres dynamically loads and then executes them.

User-defined Types

Postgres lacks most type constructors of the kind available in general-purpose programming languages. In particular, there is no convenient way of creating a type that has a record structure. Instances of user-defined types are treated by Postgres as unstructured strings of bytes of known sizes—Postgres can store and retrieve these instances but it cannot "peer" inside them. All manipulation requiring access to individual bytes within an instance must be performed through functions defined within Postgres. Therefore, user-defined types in Postgres are somewhat like abstract data types with the caveat that their implementation details are also hidden from Postgres itself.

Postgres requires at least two functions to be created in order to define any user-defined type. One function converts "constants" of a user-defined type into the Postgres internal format of an unstructured string of bytes. The other function performs the opposite transformation, converting strings of bytes into constants. These constants are character strings which represent the way a user wishes to perceive the type as output on a terminal, say. For instance, a user might create a type that implements a record containing one text and one integer field. A suitable constant for this type would be a text string of the form `"foo,4"` where the comma is a delimiter. To convert this constant into Postgres' internal format, a function must parse it enroute to creating a sequence of bytes which is the concatenation of the string `"foo"` and the byte-encoding of the integer four.

Large Objects

Large, non-relational datasets, such as satellite images or AVS networks, can be stored and retrieved in Postgres using the concept of a *large object*. Like instances of user-defined types, large objects are sequences of bytes whose structures are completely unknown to Postgres. Since large objects may be hundreds of megabytes in size, they are treated differently by Postgres than instances of user-defined types.

To handle large objects, a file system has been implemented *within* Postgres. This idea contrasts with the traditional view where a DBMS is implemented on

top of a file system. For this reason, Postgres calls its large object management facility an *inverted* file system. As with most file systems, Postgres' inverted one uses a directory structure to help manage the storage and retrieval of large objects. Utilities are provided to create and delete large objects, and to make and remove directories or subdirectories. Large objects are located within the inverted file system using path names that appear exactly like those in a Unix file system. Within Postgres, large objects are always referred to by their path names which act as pointers. User created functions can be defined on and can return large object pointers hence user-defined types can be created that are based on large objects.

2.2 Interfacing to Postgres

Postgres adheres to the client-server model of computing where the DBMS is the server and application programs, such as the Visualization Executive, are the clients. Postgres provides an application programming interface through a set of C-callable functions known collectively as LibPQ. The functions in LibPQ provide a means for sending Postquel commands to the Postgres server and receiving data in return.

A database query in Postgres is satisfied by forming unnamed, dynamically created classes called *groups* which contain instances known as *tuples*. These groups have a record structure formed by concatenating together select fields from the record structures of different named classes defined within the database. Postgres sends to an application program query results packaged into a data structure called a *portal buffer*. These buffers are structured conceptually as a sequence of tables in which each table corresponds to a group. The rows of a table that is part of a portal buffer correspond to tuples and the columns to group attributes. Associated with a portal buffer is additional information which describes the gross structure of the data that is returned. This information includes the number of groups returned, the names and types of attributes for each tuple, and the number of tuples in each group.

3 Implementing an Earth Science Database

A simple Earth science database was implemented via the Postgres DBMS to test ideas about implementing and using a VMS. A portion of the inheritance graph which forms the backbone of this database is shown in Figure 1. As can be seen from the graph, all classes in the database are sub-classes of the `Base` class whose instances contain two attributes: `ClassName` and `InstanceName`. `ClassName` refers to the name of the class to which an instance belongs while

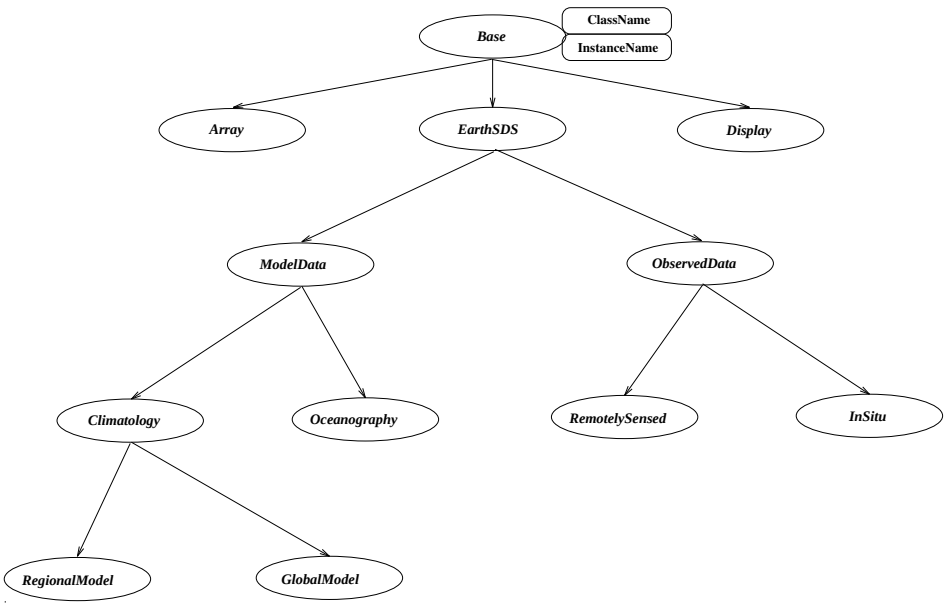

Figure 1: A partial inheritance graph for an Earth science database.

`InstanceName` is a name that is unique among all instances in a given class.

The sub-tree rooted at `EarthSDS` in the inheritance graph represents a partial taxonomy of Earth science data types. Attributes of class `EarthSDS` include fields for listing a dataset's source, its date and time of generation or observation, its name, and so on. Aside from this bookkeeping information, the `EarthSDS` class also contains an attribute which references information as to how a dataset is to be visualized.

If a sub-class of `EarthSDS` corresponds to a particular kind of dataset, then that class has attributes which reference the locations where the actual raw data resides. An assumption is made that all raw data is in the form of a multi-dimensional array whose array elements consist of a list of numeric values. These arrays are stored within the database as large objects. The path names for these objects are encapsulated within a class called `Array` which also contains attributes describing the array's dimensions and the data types of the values associated with each of the array's elements.

Pointers to instances of the `Array` class are the data references that are stored as attributes of the various `EarthSDS` sub-classes. Fields which contain these references are of the user-defined type `ClassPointer` whose values are text strings of the form "<*class-name*>:<*instance-name*>". The first component of

such a value is the name of some class in the database, e.g. `Array`, while the second component is an instance name within that class. Since instances of all classes inherit the `ClassName` and `InstanceName` fields from the `Base` class, the class and instance names encoded in values of type `ClassPointer` single out one instance from among all those in the database.

Note that two levels of indirection exist between an instance of an `EarthSDS` sub-class and the location of its raw data. In general, the `ClassPointer` mechanism was introduced to circumvent a serious flaw in Postgres, namely, there is no easy way to define a class attribute that has a record structure. Postgres does not allow *composite* classes, classes which have attributes that are instances of other classes, and trying to define a record-like attribute as a user-defined type is tedious with its requirement to define input and output functions. It is for this reason that Postgres is not an object-oriented DBMS even though it has notions of classes and inheritance.

4 The Visualization Executive

The operation of the Visualization Executive was intended to be somewhat analogous to the operation of the Apple Macintosh folder browser. On the Macintosh, a user traverses the folder tree until a document of interest is eventually located. The browsing of the folders proceeds visually by pointing and clicking on iconic representations of folders and documents. When a desired document is located, the user double clicks on the representative icon and the document is then "visualized" within the application which created the document.

In a similar fashion, a user of the database browser in the Visualization Executive navigates through the database by pointing and clicking on iconic representations of the database's contents. Once a dataset of interest is located, a user double clicks on the dataset's icon and the corresponding dataset is then visualized. The name of the visualization program to run and the parameters to pass to it are stored with the dataset in the database.

In general, the Macintosh analogy cannot be pushed too far. For instance, in the Visualization Executive an "application" is not the program which creates a dataset but rather is the method for its visualization. In addition, the same dataset can be visualized in many different ways using a variety of different visualization packages whereas on the Macintosh a single application is associated with only one document. In practice, information about each of the various ways a dataset is to be visualized should be stored with the dataset and a way of choosing from among the possibilities should be presented to a user. But, the VMS prototype described here makes the simplifying assumption that only one visualization method is associated

with a given dataset.

4.1 Database Browsing

The general idea underlying the browser is that queries are made of the database and the results are displayed graphically as icons. The icons are active in that they have behaviors associated with them that can be elicited by certain picking operations. The database browser in the current incarnation of the Visualization Executive comes with two interaction tools both of which were implemented using the X-Windows-based Tcl/Tk graphical user-interface toolkit [9, 10].

Query Formation

The simplest yet most powerful database interaction tool is the *text-tool* which allows any legal Postquel command to be sent to the Postgres server for execution. The text-tool is essentially a visually oriented text editor with a built-in history mechanism. All commands that are submitted for execution by the tool are saved in a buffer which can be stepped through either forward or backward by clicking on appropriate stepper buttons. At system startup, the history buffer is loaded with commands from a file so that frequently used Postquel commands do not have to be entered by hand each time the system is initialized.

Below the menubar is a scrollable workspace where Postquel commands are created and edited. Commands can either be built up from scratch by entering them from the keyboard or they can be fetched from the history buffer. Each time the history buffer is stepped through, a stored command is retrieved overwriting whatever is currently in the workspace. Once in the workspace, commands can be altered by using cut-and-paste operations. When a command is in a desired form, it is sent to the Postgres server by explicitly clicking on a "submit query" menu button.

The text-tool offers the utmost flexibility in interacting with the database, but it is really meant to be used only as a last resort. The tool is difficult to use in that textual commands must be entered from the keyboard and a knowledge of the Postquel query language is mandatory. The preferred method of interacting with the DBMS is through a point-and-click interface whose look is partially determined by the kind of information that is to be visualized. An example of such a domain-specific database interaction tool is provided by the Visualization Executive in the form of the *map-tool*.

The map-tool is an exceedingly simple geographical information system that allows limited forms of queries to be submitted to Postgres. The tool is built around a 2-D world map which forms a backdrop for specifying latitude and longitude

ranges (see Figure 3). As the cursor is moved across the map, the cursor's position in units of latitude and longitude are continually displayed in a sub-window. The mouse is also used to specify a latitude/longitude range which is depicted on the map as a rubber-banded box aligned with the lines of latitude and longitude. The coordinates of the corners of the range box are depicted in a sub-window where they can be edited by hand if needed. Along with a latitude/longitude range, the map-tool is used to enter a date and time range as well. The start and end of a temporal range are displayed in editable sub-windows which appear below the world map.

Once a latitude/longitude and date/time range have been entered, a query for all Earth science datasets in the database whose positional and temporal attribute values fall within the specified constraints is formulated. Like the text-tool, this query is submitted to the Postgres server by depressing an appropriate pushbutton on the map-tool's front panel. All Earth science datasets whose latitude/longitude and date/time range non-trivially intersect those given in the query are retrieved from the database. The DBMS does not clip datasets to the requested ranges although such a subsetting capability could have easily been added.

Visual Representation of Information

Upon receipt of a portal buffer after a query, the Visualization Executive invokes an icon generator to assign a graphic representation to each tuple that is returned. Each sub-class of `EarthSDS` has associated with it an icon stored as a pixmap. If a tuple in a portal buffer corresponds to an instance of an `EarthSDS` sub-class, then the icon generator assigns that class' pixmap to the tuple as its icon. In all other cases, a tuple must correspond to an instance of an unnamed, dynamically generated class which is always assigned the same catch-all pixmap by the icon generator.

The icons representing the tuples in a portal buffer are arrayed in groups on a scrollable canvas as depicted in Figure 2. Icons can be repositioned manually should a user wish to do so. When an icon is selected with the middle mouse button, a separate window appears which displays the attribute names and values for the tuple represented by the icon (see Figure 2). The attribute values can be edited within the window and the changes will be written back into the database whenever the window is closed. If an icon corresponds to an instance of an `EarthSDS` sub-class, then double-clicking on it causes the raw data referenced by the instance to be visualized according to the visualization information indicated within the instance. For example, Figure 3 shows the result of a double-click on an icon which caused the corresponding climatology dataset to be visualized within AVS.

4.2 Initiating Visualizations

The browser performs three actions after a dataset icon is double-clicked upon:

1. Create a temporary *dataset description file*.
2. Create a temporary *script file*.
3. Spawn a process which performs the actual visualization.

The dataset description file contains information describing the raw data which comprises a given dataset. The information in this file is just a listing of the attributes of certain instances of the `Array` class. In general, a dataset is a composite of several multi-dimensional arrays. For example, a dataset containing regional climate information typically consists of a 3-D grid of atmospheric variables, a 2-D grid of terrain elevations, and a 2-D grid of variables representing characteristics of the topography such as total precipitation.

The script file contains a program or script specifying how to visualize a given dataset. The browser makes the assumption that all visualization packages have a scripting language within which visualization programs can be written. These programs are stored in the database as large objects and their path names are encapsulated in a class called `Display`. Besides a field containing a path name, the `Display` class also has a field for recording the name of the visualization package to which the encompanying script or program belongs. Class `EarthSDS` has a field of type `ClassPointer` which references instances of `Display` therefore all sub-classes of `EarthSDS` inherit a pointer to visualization information.

The process that is spawned by the browser is a stand-alone executable module whose name has the form, `FireUp`<*ApplicationName*>. The symbol <*ApplicationName*> is a placeholder for the name of a visualization package that is to be used and this name is retrieved from an indicated instance of the `Display` class. The `FireUp` programs are created by the user and there must be one such program for each visualization package that is to be supported within the Visualization Executive.

In general, the `FireUp` modules are responsible for performing three actions: retrieving raw data stored as large objects in the database, converting them into appropriate forms for use by a visualization package, and executing the commands necessary to start a given visualization package. For the data retrieval and conversion tasks, a `FireUp` module must read in the information cached in the data description file. The large object path names that reside there are used by a particular LibPQ function to actually effect the transfer of bytes of raw data from the database into a memory resident buffer. Once in memory, the `FireUp` module

can do a data conversion into an internal format that is recognized by an indicated visualization package using the array dimension and element type information in the data description file.

After data retrieval and conversion, a `FireUp` module starts a given visualization package. The general procedure for doing this is to first execute the operating system level command which invokes the package and then somehow indicate to the package that the program in the script file is to be loaded and run. The details of this procedure vary greatly from one visualization package to another. In IDL, the `FireUp` module would do the following:

- Transfer the large object byte stream from the database into an internal buffer making use of the information in the data description file.
- Write out the buffer to a temporary file, `raw-data`, in a format readable by IDL.
- Start IDL with the command, `idl script-file`, where `script-file` is the name of the script file created by the browser which contains an IDL program.

Note that the IDL program contained in the script file must open the temporary file `raw-data` to read in the data that is to be visualized. This is a minor nuisance since developers of IDL programs must know the name of the temporary file that the `FireUp` routine creates to temporarily cache raw data.

For AVS, the `FireUp` module is constructed differently. First, it is assumed that an AVS module, `ImportData`, has been created which does the data retrieval and conversion. This module does the transfer of the large object byte stream into an internal buffer and then maps the data into an AVS field type which is then exported through the module's output port. The module has a single text input parameter which is interpreted as the file name of the data description file. An AVS network specification is stored in a script file and it is assumed that a single `ImportData` module is part of the network.

With the creation of the `ImportData` module, the AVS version of the `FireUp` program would then do the following:

- Start AVS in server mode with the command, `avs -server`, and establish communications via sockets (see [1] for details).
- Send to the AVS server commands to:
  - Read the AVS network from the script file.

– Send the name of the data description file to the `ImportData` module thereby starting execution of the network.

Note that the above procedure has been simplified by off-loading the data retrieval and conversion steps to the visualization package but at the expense of complicating communications. Furthermore, data movement is more efficient with AVS since the raw data does not have to be temporarily cached in a file as with IDL.

5 Discussion

The prototype VMS described above was designed and built to test new ideas about linking scientific databases to visualization packages. Experience gained in implementing and using the system has provided a good understanding of what kind of methodology and functionality works, and which can be improved upon. The following are some observations about this system which have bearing on future research and development efforts:

Information presentation and the relational database model. In the Visualization Executive, icons are generated solely on a per tuple basis. This idea works satisfactorily when the sizes of classes are small but severe problems arise with large classes that typically reside in a scientific database. For instance, in one Sequoia database there is a class encapsulating station measurements of oceanographic variables whose size numbers several million instances. In this case, even restricting attention to relatively small regions on Earth still produces on the order of thousands of tuples which would easily overwhelm the present icon generator in the Visualization Executive.

In dealing with such large classes, what is needed is a hierarchical representation of retrieved data much like the concept of folders in a Macintosh system. Part of the problem in dealing with large relational datasets lies with the DBMS itself—relational systems do not easily support hierarchical encapsulation of data. Such a data structure would require attribute values to be themselves instances from another class and in general relational database systems do not directly support constructs of this kind.

Even so, hierarchical information presentation could have been enforced in the current implementation of the prototype VMS. If too many tuples are returned in a portal buffer after a query then the icon generator could start grouping them into "folders." As on the Macintosh, a separate icon would denote a folder to signify that information is being grouped hierarchically. However, the question still remains as to exactly how the retrieved data would be partitioned into folders.

Dynamic icon generation. In the current implementation, a separate icon is associated with each class in the database and every instance of a class uses that icon as its visual representation. It would be far better to dynamically tailor more telling icons based on the information content within each retrieved tuple. A more intelligent icon generator should take into account both the type and value of each tuple attribute when constructing an icon. To make this idea work, the type information should include more than just a structural description of the encoding of a value as, say, an integer. Semantic information should also be present that identifies a value as representing some physical quantity, such as temperature, which uses a particular unit of measurement and must lie within a certain range of validity. With a deeper understanding of values, an icon generator could, say, place an image of a thermometer, indicating the appropriate temperature level, in an icon representing a tuple having temperature as an attribute.

Icon placement. Currently, icons are merely arrayed on a blank 2-D canvas so that at best they give only a quick visual estimate of the number of tuples for each class of data that is retrieved. Icons could instead be placed onto an "informational landscape" so that their position on the landscape conveys meaning. For example, the query for datasets pertaining to a certain region of Earth could result in the icons being placed on a 3-D globe. Other kinds of relational data can be displayed graphically as scatter plots, graphs, histograms, etc. using the techniques discussed by Beshers and Feiner [3], Casner [4], and Mackinlay [7].

Database schema knowledge in the Visualization Executive. When constructing a browser, one should strive to make it as independent of the database schema as possible, that is, assumptions about the structure of the database inside the browser should be minimized. In this way, the browser would not have to be recoded each time the schema changes thus affording a more general and flexible utility. In order to achieve this goal, the database needs to be self-describing, in other words, the schema design and complete characterization of all data must be stored in the database along with the data. Having this information on-line allows a browser to "learn" about the structure of a database thereby giving it the flexibility to dynamically create graphical interfaces tailored to the information that a user seeks.

Queries via the visual interface. A graphically-oriented query tool is far preferred to one which requires the use of text to enter DBMS commands. With a textual tool, a user is required to know both the DBMS command language and the underlying database schema to even be able to formulate a query. Although graphical query tools do not have these problems, they may suffer from another problem in that general queries may not be expressible using the tool. For example, the map-tool of the present database browser only allows a visual query for Earth science data

that lies in a specified latitude/longitude and date/time range. In other words, it produces a query that cannot be further constrained by specifying desired ranges of values for other attributes as well. If scientists only want to see AVHRR satellite imagery, they must nonetheless query for all other Earth science data types that might reside in the database.

In addition, there will be times when a user would like to apply successive queries to information already retrieved rather than always against the full database. Frequently when making a query, more information is retrieved than was intended; in other words, the query was too general and additional constraints need to be applied. In the current system, the only alternative is to formulate a new, more fully constrained query based on the original. If the original query was created by an elaborate sequence of interactions with a graphical user-interface, then the formulation of the new query would be quite tedious and time-consuming.

Information modification. Non-relational information that is stored in the database as Postgres large objects cannot be altered from within the Visualization Executive so that the changes are reflected back in the database. For instance, visualization programs must either be written by hand or they must be created within a visualization package. In either case, the resulting program must be dumped to a temporary file from where it can then be ingested into the database and stored as a large object from outside the Visualization Executive. A similar criticism holds for raw, non-relational datasets such as satellite images.

This drawback is partly the fault of relational DBMSs such as Postgres which do not adequately deal with large datasets that are not inherently relational in character. In Postgres, classes come with a wide variety of manipulation and interrogation operators while large objects are given virtually no built-in support. To Postgres, large objects are simply an unstructured string of bytes that are shunted about in their entirety as a single unit.

What is needed is some kind of generalized editor that understands the internal structure of large objects. Preferably such an editor would be visually oriented rather than being text-based. It would be quite useful, say, if the entire AVS Network Editor could be subsumed by the general large object editor in a VMS. In that way, AVS networks could be manipulated more naturally without having to know the details of the AVS scripting language, CLI. Visual editors for other kinds of large objects, analogous to the Network Editor, would round out the design of a generalized large object editor.

6 Conclusion

The VMS that has been constructed is an early prototype for a much more elaborate, next-generation visualization system that is currently under development as part of the Sequoia 2000 Project. The prototype demonstrates the utility of merging a DBMS with most varieties of off-the-shelf visualization systems which by themselves do not provide adequate data management services. The prototype makes use of one example of a *domain-specific* graphical browser which is much easier and more intuitive to use than textual DBMS command interfaces. Finally, a powerful concept was introduced where visualization information is stored with datasets themselves. This idea implies that scientists do not have to waste time creating or searching for a visualization program to run in their preferred visualization package each time they wish to view a dataset—scientists can concentrate more on their science rather than on the process for doing it.

Acknowledgements

This work was supported by the Digital Equipment Corporation, the University of California, and the San Diego Supercomputer Center. Special thanks to Len Wanger for his help in the preparation of this paper. Thanks also to Mike Bailey, director of the SDSC visualization group, for his tremendous support.

References

[1] Advanced Visual Systems, Inc. *AVS Developer's Guide*, May 1992.

[2] Advanced Visual Systems, Inc. *AVS User's Guide*, May 1992.

[3] Clifford Beshers and Steven Feiner. Automated design of virtual worlds for visualizing multivariate relations. In *Proceedings Visualization '92 Conference*, pages 283–290, 1992.

[4] Stephen M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, April 1991.

[5] Jeff Dozier. How Sequoia 2000 addresses issues in data and information systems for global change. Technical Report 92/14, University of California, August 1992.

[6] Bruce Lucas et al. An architecture for a scientific visualization system. In *Proceedings Visualization '92 Conference*, pages 107–114, 1992.

[7] J. D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[8] Mike Olson et al. Visualization benchmark specification for the Sequoia 2000 project. unpublished report, November 1992.

[9] John Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[10] John Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[11] Research Systems, Inc. *IDL User's Guide*, January 1993.

[12] Michael Stonebraker et al. The implementation of POSTGRES. *IEEE transactions on knowledge and data engineering*, March 1990.