Tioga: A Database-Oriented Visualization Tool [*]

Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, Alan Su, Jiang Wu

Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

Abstract

In this paper we present a new architecture for visualization systems that is based on Data Base Management System (DBMS) technology. By building on the mechanisms present in a next-generation DBMS, rather than merely on the capabilities of a standard file manager, we show that a simpler and more powerful visualization system can be constructed. We retain the popular "boxes and arrows" programming notation for constructing visualization programs, but add a "flight simulator" model of movement to navigate the output of such programs. In addition, we provide a means to specify a hierarchy of **abstracts** *of data of different types and resolutions, so that a "zoom" capability can be supported. The underlying DBMS support for this system, Tioga, is briefly described, as well as the current state of the implementation.*

1 Introduction

Scientific visualization applications often deal with data objects of very large sizes. Examples include large regular arrays such as those found in global atmosphere and ocean circulation models[7] and in remote sensing applications[1]. In addition, users often wish to keep large numbers of such objects in an on-line store. For example, the various Earth Science participants of the Sequoia 2000 project[13] wish to keep about 100 terabytes of easily accessible information. In order to manage this large repository, they wish to use the services of a Data Base Management System (DBMS). Sequoia 2000 is utilizing the next-generation DBMS, POSTGRES[12], for this purpose.

[*]This research was sponsored by NSF Grant IRI-9107455, ARO Grant DAAL03-91-G-0183, DARPA Contract DABT63-92-C-0007. Additional support was provided by the University of California and Digital Equipment Corporation under Research Grant #1243.

Popular visualization systems such as AVS, Explorer, and Khoros provide only primitive data management support. In particular, they can only read or write data from files, and they are geared toward manipulating a fixed set of data types. It is certainly possible to build an interface between an existing DBMS and an existing visualization system by adding a "query" box to the visualization system. This approach, which we will call **loose coupling,** is exemplified by the interface between several visualization systems and POSTGRES[6]. However, much more power and flexibility can be obtained by a tighter integration of DBMS and visualization services, and the basic purpose of of Tioga is to exploit this possibility.

Conventional relational DBMSs lack the data modeling flexibility to support scientific data adequately. However, recently several "next-generation" DBMSs have been built with enhanced data models, including POSTGRES[8], IRIS[15], ORION[5] and Starburst[3]. Although we are building Tioga for the POSTGRES DBMS, it could be easily interfaced to any next-generation DBMS with the following three characteristics. First, Tioga requires a DBMS which can be extended with user-defined data types. Such types can either be new base types which augment the standard collection of integers, floating point numbers and character strings, or they can be composite data types. Second, Tioga requires a DBMS which allows users to store, or **register**, previously written functions. Lastly, Tioga requires a DBMS with a multi-dimensional access method such as R-trees[2] or grid files[9].

In addition to tight DBMS integration, Tioga also offers an enhanced browsing capability so users can interact with the output of visualization programs. Specifically, it offers a "flight simulator" user interface so the user can "navigate" in information space. Also, we allow visualization programs to be **abstracts** for other programs. Using this capability, a user can "zoom" into information space to obtain more detailed information. To support this construct, the run-time system will "stack" the currently running visualization program and run a second one.

This paper is organized as follows. In Section 2 we discuss the manner in which the boxes and arrows programming paradigm is integrated into POSTGRES. Then Section 3 discusses the way Tioga requires the DBMS to interact with the user-space (client) portion of the system. This interface is a generalization of both traditional SQL cursors and database portals[10]. Section 4 indicates the run-time support provided by POSTGRES for execution of Tioga boxes and arrows diagrams. In Section 5 we describe how Tioga supports additional functionality in the areas of abstracts of data, browser synchronization, versions, data entry and interfaces to foreign systems. Lastly, in Section 6, we conclude with an update of our current status and a look at future issues.

2 The Tioga Programming Paradigm

Existing scientific programming systems allow the user to create visual programs by connecting modules written in a conventional programming language. The modules are depicted on the screen as **boxes** with connections for inputs and outputs. The user connects the boxes with **arrows** to create a directed graph which represents the final program. One or more boxes in the diagram are input nodes which read data from named files. Executing a diagram entails running the read boxes and progressively running each box as its inputs are available. Normally, the final box in the graph is a rendering engine which displays the result of the computation on the screen. The user can interact dynamically with the diagram by changing the parameters of the boxes, and the diagram is automatically rerun to produce the new rendered output. In this way, a user can iteratively produce the desired visualization effect.

The Tioga architecture generalizes this boxes and arrows user interface from commercial packages. Specifically, Tioga supports the definition, manipulation and execution of boxes and arrows diagrams, which we term **recipes**. Individual boxes in a recipe are called **ingredients**. The term recipe is used because a collection of ingredients is "cooked" into a final visualization output.

One key to the Tioga architecture is that each function registered with POSTGRES is automatically an ingredient, and is thereby in the menu of recipe building blocks. The menu of building blocks can thus be constructed by simply reading the catalog of POSTGRES registered functions.

In a boxes and arrows diagram, a one-way connection between two boxes indicates that the result of the first ingredient is to be passed as input to the second ingredient. In order for such a connection to be valid, the data type returned by the first function must be compatible with the type of one of the arguments of the second function. Existing visualization systems support a small collection of types and perform the above compatibility check for each arrow.

POSTGRES supports a very sophisticated type system, and Tioga must perform compatibility checking in this environment. Specifically, it must ensure that the output type exactly matches an input type of the subsequent function, or the output type is a set of the input type of the second function. In this latter case the second function will have to be called multiple times, once per element of the set. Types in the same inheritance hierarchy are also compatible. For example, if `EMP` is a subtype of `PERSON`, then outputs of type `EMP` can be passed as input to a function expecting an input of type `PERSON`.

As a recipe is being constructed by the user, the editing program automatically performs type-checking, since the input and return types of all functions are known. The user is told if a connection is invalid, so that he or she can correct it. In addition, the editor supports the use of optional icons to represent types. We plan to encourage type creators to design icons which give visual clues concerning the relationship of the type to other types. For example, icons of types within the same inheritance hierarchy might have similar graphical features. In this way, the user can be given visual clues concerning the compatibility of types, allowing a kind of visual type checking.

When the user finishes editing a diagram, the editor notes which function inputs are missing. Inputs are considered missing if they are not specified by an incident edge from some other function. Function inputs which are not connected are treated as **run-time parameters**. At recipe execution time, the user will be interactively prompted to supply the missing values.

There are two semantically different kinds of recipe building blocks. The first are conventional POSTGRES functions as noted above. As will be explained in Section 4, the code for these functions is executed inside the POSTGRES DBMS when the recipe is run. The second kind of building blocks are browsers.

These visualization boxes render screen images and run as DBMS application programs. As such they adhere to the client-server communication protocol described in the next section. There can be an arbitrary number of browsers in a recipe. Sophisticated users can define new kinds of browsers to meet specific rendering needs.

Using the diagram editor, the user constructs a recipe consisting of ingredients and browsers attached together into a directed graph. Such a recipe can be saved in the DBMS in two different ways. The recipe can be stored as a graph-like structure in a **cookbook**, a collection of recipes in the database. A user can then search this cookbook using any DBMS query capabilities.

Alternately, a recipe can be encapsulated or **canned** into a new ingredient. In order for a recipe to become an ingredient, it must be a legal POSTGRES function, meaning it can only have a single output, and it cannot have a browser. Once the recipe is compiled into a single ingredient, its original structure is lost and it becomes opaque to the user. Canned recipes are added to the collection of POSTGRES functions and hence, automatically augment the collection of ingredients for future recipes.

If a user wishes to run a previously constructed recipe, he can do so from the diagram editor. The appropriate ingredients are loaded, any missing input parameters are prompted for at run-time, and a window for each browser is generated. To run the recipe, the browsers communicate with the DBMS using the protocol described in the next section.

3 Browser-DBMS Protocol

As noted in the previous section, a recipe consists of a collection of interconnected functions, and it may contain one or more browsers. Each browser is run as a DBMS application program which interacts with the recipe engine. The engine manages the execution of the ingredients in the recipe. In this section we describe the protocol for communication between a browser and the DBMS. The interaction between the human user and the browser is unconstrained; however, the protocol to be described is most natural for a **flight simulator** paradigm, in which the user has a joystick by which he can navigate in a data space.

Although it is possible to support an interface between the browser and the DBMS which allows browsing of an arbitrary collection of DBMS types, we chose a different approach. Each database object may be of an arbitrary type, but it must have associated with it a **geometry**. The geometry of an object describes its location in an **application coordinate space**. All objects in an application are located in this common N-dimensional coordinate system, whose dimensions are appropriate to the specific application. The geometry of an object may be either an N-dimensional polyhedron or a point. It is the job of the human recipe designer to ensure that the recipe produces the geometry representation expected by some browser. Failure to provide this will result in a type mismatch.

To achieve a common polyhedron representation, we have defined a standard N-dimensional polyhedron, `N-D-polyhedron`. The generic tuple passed to the browser from a recipe will have the form:

{value, type, location}

The value can be an instance of a base type or a composite type, and its location is represented by the N-D-polyhedron as indicated. For example, the value might be a satellite image; its type might be AVHRR, and the location associated with it might be a rectangle representing one of the quadrangle of a U.S. Geological Survey map.

With these preliminaries, the protocol between the browser and the recipe execution engine consists of the following commands:

MARK (N-D-point) with identifier
ERASE identifier
MOVE to identifier
MOVE to (N-D-point)
MOVE along $(\Delta_1, ..., \Delta_N)$ until F(value) <operator> <constant>
FETCH (number)
FETCH $(\Delta_1, ..., \Delta_N)$

The browser can mark any position in N-dimensional space with an identifier, so that it can return to that point at a later time. This is useful in marking points of interest.

The browser has three ways to relocate its position in N-space: it can move to a previously designated identifier, it can move to a specific N-D-point which it calculates in some fashion, or it can move in some direction, denoted by $(\Delta_1, ..., \Delta_N)$ until some condition

F(value) <operator> <constant>

is true. This third relocation command is useful, for example, if a user is browsing Hurricane Hugo, and wishes to **fast-forward** the hurricane, i.e. skip or skim through images sorted by time, until Hugo hits land. If landfall of the hurricane can be expressed as a predicate, then the appropriate MOVE command would look like

MOVE along (0,0,...,+1) until hits_land(Hurricane.hugo) = TRUE

The +1 means a movement along the positive time axis, assuming time is the last dimension in this coordinate system. Note that recipes may be fast-forwarded in this fashion in any dimension.

There are two ways to fetch data: first, the browser can request a fixed number of instances; second, it can request all the instances within a specific N-dimensional rectangle. In the first case, the number of instances requested is returned by running the recipe forward from its current position. Since the recipe determines the ordering of instances, it implicitly specifies what the "forward" direction of instance production is. In the second case, the rectangle is specified by a collection of offsets from the current position in the application coordinate system.

As the user moves through N-space with a joystick-like interface, it is the responsibility of the browser module to issue the appropriate move and fetch commands to support the user. It is also the browser's responsibility to display appropriately the values which are returned from the recipe in a fashion similar to that of SDMS[4].

To assist the browser, each type implementor is expected to define a display function in POSTGRES of the form:

display(object,location,screen-resource)

The location of the object is an N-dimensional polyhedron. The screen-resource argument specifies the screen resources which are available for the display of this object such as the dimensions in pixels of the area and the number of bits of color available. Given these parameters, the display function returns to the browser a screen representation for a given data object.

The display function can return either a **renderable object** or a set of sub-objects which individually need to be passed to display functions. The latter mechanism allows for a hierarchical decomposition of a complex object into simpler objects to be displayed. For example, a browser could display information about employees by calling the display function with the appropriate instances and locations. This function would either be a generic one or one written by the designer of the `EMP` class. The display function could return an image of the employee's face, or the display function could return separate data objects which make up an `EMP` instance, such as the employee's salary, department, name, and picture. These can then be separately rendered by calling the display function again for each one.

4 Recipe Execution

At first glance, Tioga may seem to be merely a convenient query tool for a next-generation system. Compiling (or cooking) a recipe entails converting the graph into a collection of queries on the DBMS, resulting in one or more query plans. This is similar to compiling the output of any other query tool. However, recipes differ from queries in four crucial ways.

First, when a recipe is executed, the Tioga optimizer receives a directed graph of ingredients, each of which corresponds to a query. This should be contrasted with a traditional DBMS which accepts a single query.

In order to support Tioga recipe execution, we are extending the POSTGRES executor so it can run a **megaplan**, which is a directed graph of nodes, each of which is a query plan. Specifically, we have introduced a plan node which is a **tee**, or fork, that connects the output of one plan to the input of one or more other plans. Megaplans are query plans with tee nodes in them.

When a recipe is inserted into a cookbook, each ingredient can be optimized by a traditional DBMS optimizer. The resulting megaplan is stored for subsequent execution by an extended execution engine. An optimization available on megaplans is to **coalesce** multiple query plans into a single composite query plan. Tioga will optimize by coalescing queries when coalescing results in more efficient execution.

Second, ingredients have run-time parameters which are changed frequently. For this reason, it is advantageous to **buffer** the output of some (or all) ingredients, so that changes in downstream parameters do not require recalculation of upstream ingredients. Where to buffer is a second decision which must be optimized. Buffering and coalescing decisions are interrelated, because coalescing two ingredients into a single query plan removes the opportunity to buffer at the output of the first ingredient. Hence, both kinds of optimization must be performed in a unified manner.

Third, the browser interface allows repeated requesting of information which has been previously retrieved. Hence, it is advantageous to buffer the output of the ingredient immediately preceding a browser. This output must be indexed using a multi-dimensional access method, such as an R-tree, in order to allow re-requested information to be located quickly.

Fourth, Tioga is **demand-driven.** A megaplan can have several browsers attached to it, each independently requesting records. Current query plans have a distinguished root node which outputs records to an application. In Tioga, each browser requests one or more records from a node of a plan, which responds by requesting records from its descendent nodes. The process completes when a node in the plan can deliver records, which then flow up the plan to satisfy the outstanding request.

When two browsers operate on a megaplan, then a tee must be present. If one browser requests records and the second one does not, then recipe execution will continue the evaluation of the megaplan to generate the records required by the first browser. The state of the tee junction will advance to that required by the first browser, and the second browser will thereby lose its place. Buffering at the tee will allow recipe execution to avoid the subsequent recomputation of the state of the second browser when it resumes requesting records.

To optimize a megaplan, we therefore must decide when to coalesce two ingredients in a megaplan and where to insert buffers. The details of our algorithms are beyond the scope of this paper and are discussed in[14].

5 Extensions to Recipe Management

By using a DBMS to support the data needs of recipe management, we are able to provide additional functionality for Tioga. In the following subsections, we present the Tioga approach to abstracts, synchronization of browsers, data entry, and interface to foreign systems.

5.1 Abstracts

A crucial capability of Tioga is user control over the resolution of the visualized information. For example, the user interface must allow the user to zoom in on recipe output to obtain more detail or to zoom out to coarser granularity. To satisfy this requirement, the recipe execution system must be capable of producing recipe output at varying levels of detail.

The zoom in/zoom out capability is reminiscent of SDMS[4], where additional detail appeared automatically and was hard-wired into the system. In Tioga we are implementing a much more flexible scheme. We allow every recipe to have one or more children, which will be termed **abstracts** for the given recipe, since they contain less information. Conceptually, they are analogous to textual abstracts for a conventional document. Note that an abstract need not produce the same type of information as does its parent. For example, an abstract for an image of Hurricane Hugo could be a hurricane icon and an abstract for the icon could be the character string "hurricane".

We organize recipes into a directed graph of abstracts so that an edge from one node to another in this graph indicates "is abstracted by." If there is an edge from P to C, then C is an abstract of P. P is also the parent of C, and P contains more information than C. Each edge in this directed graph is labeled with a notation concerning how the abstract loses information. Example notations include "lower resolution," "lower precision," and "lower accuracy."

Each recipe in the graph of abstracts has two associated constants. These are the maximum and minimum screen window sizes, specified in the application coordinate system, that this recipe can tolerate. When the browser executes a MOVE or FETCH command, it checks whether the window size currently requested is between the two bounds noted above. If so, it processes the request using the active recipe. The display functions for the objects that appear on the screen are assumed to scale their objects appropriately to fit in the available screen real estate. Put differently, the maximum and minimum window sizes must be chosen to assure that the display functions can perform this scaling.

If the window size is smaller than the minimum, then the browser is being called on to display more detail than is possible using the current recipe and it must move to a parent recipe that can display more information. Alternately, if the window is too large, then the current recipe displays too much detail, and the browser must move to a child of the active recipe. If the directed graph has multiple parents or children, then the browser will prompt the user with the labels on the arcs, so he can choose the recipe that is appropriate to his needs.

Lastly, all recipes in the directed graph of abstracts must have the property that the maximum window size for a parent is larger than the minimum window size for any of its children. Moreover, the maximum window size of any parent must be smaller than the maximum window size of each of its children. In other words, the maximum window size of a parent must be within the legal window size range of its children. In this way, when the parent recipe is zoomed out of, a child recipe with less detail can be used to accommodate the current window size. Similarly, the minimum window size of a child recipe must be within the legal window size range of its parent. This facilitates zooming into a parent recipe for more detail.

When the recipe engine switches to a new recipe, it must save the old one, load the new one and then position it at the correct location. The browser can then perform a FETCH command to refresh the screen with objects from the new recipe. This will be an overhead-intensive operation which will probably generate a pause in the zooming operation. To alleviate this "heavyweight" recipe switch, Tioga allows a node in the abstract graph to be a function. In this case, the recipe execution engine will run the function on the existing data from its child node to produce a more detailed representation. This reduces greatly the overhead of zooming.

5.2 Synchronization of Browsers

A traditional user interface has a single database **cursor** through which the result of a query or a view can be delivered to an application program. A Tioga user, in contrast, might put several browsers in his diagram and then visualize the data at several points in the diagram simultaneously. Multiple browsers must be synchronized when a recipe switch occurs due to zooming and abstracting. To support such synchronization, we use **named** browsers. If the user zooms in and activates a new recipe in the abstract graph, then his display should seamlessly change to the output of the correspondingly named browsers in the new recipe.

The user may also wish to constrain multiple browsers in some manner. For example, he may wish to specify that two browsers be **overlaid**. This means that the data that they display should be superimposed in the same visual window, rather than placed in separate windows. The user may also wish to specify that two browsers be synchronized so that one browser is a **slave** to a second one. In this case, whenever a move or fetch operation is performed by the **master** browser, the same operation would be performed by the slave browser.

Synchronizing a slave browser is accomplished by constraining the slave's input controls to those of the master. In other words, the slave's joysticks and input widgets, which allow the user to direct viewing, are controlled by the master. Any joystick commands given by the user to the master are identically dispatched to the slave browser. Thus, any move or fetch operation performed by the master browser would result in the same move or fetch operation in the slave browser. We also permit a **translation function** to be defined which translates the input controls of the master browser to the input controls of the slave browser. For example, a slave browser can be set up so that its controls are at a fixed offset away from the controls of the master browser. This may be useful, for example, if one wishes to view simultaneously two portions of a map, separated by a fixed distance.

5.3 Versions

POSTGRES supports the notion of **time travel** for data objects[11]. When an object is updated, the old value may be kept in the database, along with the new one. Both objects are time stamped, so either can be subsequently retrieved by specifying the desired logical time of a query. For example, the following query finds Mike's salary from the EMP class as of July 12, 1991:

```
retrieve (EMP.salary)
using EMP [July 12, 1991]
where EMP.name = "Mike"
```

This capability is also used in the system catalogs, where metadata for the database is stored. In particular, there is one catalog where information on user-defined functions is stored. Whenever any user-defined function is redefined, perhaps because the algorithm it implements has been improved, the old version of the function can be retained. The same time-versioning available for data is hence also usable for functions. As a result, whenever any function in a recipe is improved, a new version of the recipe is automatically defined. The user can decide which version of the recipe he wants to invoke by indicating the logical time for the recipe that he wishes to use.

More generally, if a user changes the structure of a recipe, then the previous version is automatically retained. Again, either version can be executed by indicating the desired logical time. Lastly, when a user runs a recipe and specifies a set of run-time parameters, the parameter combination is stored in a database class which is also time-versioned. The next time this user runs the recipe, his run-time parameters can be retrieved from the table instead of having to be specified again. If the user executes an older time-version of a recipe, the corresponding time-versioned parameters are used.

5.4 Data Entry

The recipes in this paper have focused on getting data out of the DBMS and onto the screen. However, there are applications where users wish to perform data capture with Tioga. For example, a user might wish to have a continuous data feed from a satellite which he wished to use as input to a recipe. This recipe might process the raw imagery and store a refined version in the data base. In this case, the recipe is capturing data from an external source and entering it into the DBMS. To support such recipes, we need two additional constructs in Tioga.

The first is another type of box, which we term a **hand**. Like a browser, a hand is a program which runs in user space, not inside the DBMS. However, a hand requires a different protocol than the one described in Section 3. A hand is a box which produces an output of some legal POSTGRES type, or more usually, a set of some type, and it may or may not have inputs. Hands can interact with subsequent boxes using the normal Tioga protocol for passing results of functions to subsequent functions, but with one key difference. Because a hand runs in user space, the results Tioga accepts from a hand come from a different process, whereas normal boxes interact with one another in the DBMS address space.

Fortunately, POSTGRES supports the notion of **untrusted** functions. The basic idea is that a data base administrator can decide whether a function is trusted and runs in the DBMS address space or is untrusted and runs in a separate address space. If a function is not yet thoroughly debugged, it should be considered untrusted, so that a crash will only bring down the address space in which the function is running and not the whole DBMS. Untrusted status is likewise appropriate if the data base administrator is convinced that the function might attempt unauthorized data accesses.

A hand is simply a POSTGRES function that runs in untrusted mode. Hence, Tioga runs recipes that contain a mixture of trusted and untrusted functions.

As noted in Section 4, Tioga recipes are demand driven by requests from the attached browsers. However, in recipes without a browser, we require a mechanism to drive recipe execution. In such recipes there will be one or more boxes which are **sink** nodes. They do not generate output for other boxes to consume. Using the satellite feed example above, the sink would be a box which performed DBMS insert operations, but had no output arrows. At any such sink node, Tioga must attach an artificial browser. This browser simply executes a "fetch all" command, which will cause the sink box to be called repeatedly. This will iteratively prompt upstream boxes and serve as the source of demand to drive recipe execution. Consequently, the second extension for recipes which capture data is to attach artificial sink nodes demand data.

5.5 Foreign Systems

A desirable feature of Tioga would be to allow data to be received from foreign systems as well as sent to foreign systems. These include other DBMSs, statistical packages, and other visualization systems. Hands are a natural way to get data from other systems, as noted in the previous subsection. To send data to other subsystems, one can also use the hand mechanism. This simply requires a hand which takes input from elsewhere in a Tioga recipe, but produces no output visible to the recipe. Instead, it sends output to the foreign system. This hand will be a sink node, as explained above, and Tioga will attach an artificial browser to "pull" data through this node into the foreign system.

The last desirable extension would be to support an interface to a foreign system that allowed data to be received from the foreign system. The results produced by the foreign system would then be inserted back into the recipe. Such functionality would be appropriate to a package which performed data validation and discarded the outliers, for example. This capability is easily supported in two different ways. First, it can be supported by making the foreign system a hand. Alternately, the foreign system could be a browser. In this case, the browser definition must be extended to allow output arrows to other recipe boxes.

6 Conclusion

We have described a system for database support of scientific visualization applications. Providing a natural user interface for the scientist has motivated our work on multiple browsers for a recipe and intelligent buffering of computed data. At the current time, we have an N-dimensional browser, the diagram editor and the recipe storage system working. We are beginning work on the optimizer and executor extensions discussed in Section 4, and expect to have a complete system within six months.

We seek to extend Tioga in several different directions. For example, POSTGRES supports a sophisticated rule management system[12], and we require a mechanism to use these capabilities in the Tioga environment. POSTGRES also allows a user program to specify transaction boundaries, for which POSTGRES will guarantee standard transaction semantics. The integration of transaction support into Tioga is another area of current investigation.

References

[1] Dozier, J., "Spectral Signature of Alpine Snow Cover from the Landsat Thematic Mapper," *Remote Sensing Environment*, March 1989.

[2] Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984.

[3] Haas, L. et. al., "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[4] Herot, Christopher F., "Spatial Management of Data," *ACM Transactions on Database Systems*, December 1980.

[5] Kim, W., Garza, J.F., Ballou, N., Woelk D., "Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[6] Kochevar, P. et. al., "A Simple Visualization Management System: Bridging the Gap Between Visualization and Data Management," Sequoia 2000 Technical Report 93/27, University of California, Berkeley, CA, July 1993.

[7] Mechoso, C. et. al., "Distribution of a Coupled Atmosphere-Ocean General Circulation Model Across High-Speed Networks," *Proceedings of the 4th International Symposium on Computational Fluid Dynamics*, 1991.

[8] Mosher, C. ed., "The POSTGRES Reference Manual," Electronics Research Laboratory, University of California, Berkeley, CA, Memo 93/57, July 1993.

[9] Nievergelt, J. et. al., "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems*, March 1984.

[10] Stonebraker, M. and Rowe, L., "Database Portals - A New Application Program Interface," *Proceedings of the 10th International Conference on Very Large Databases*, Singapore, August 1984.

[11] Stonebraker, M., "The POSTGRES Storage System," *Proceedings of the 13th International Conference on Very Large Databases*, Brighton, England, August 1987.

[12] Stonebraker, M. et. al., "The Implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[13] Stonebraker, M. and Dozier, J., "SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research," SEQUOIA 2000 Technical Report 91/1 Electronics Research Lab, University of California, Berkeley, CA, July 1991.

[14] Stonebraker, M. et. al., "Tioga: Providing Data Management Support for Scientific Visualization Applications," *Proceedings of the 19th International Conference on Very Large Databases*, Dublin, Ireland, August 1993.

[15] Wilkinson, K. et. al., "The IRIS Architecture and Implementation," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.