Tioga*: Providing Data Management Support for Scientific Visualization Applications

Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson

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

Abstract

We present a user interface paradigm for database management systems motivated by scientific visualization applications. Our graphical user interface includes a "boxes and arrows" notation for database access and a flight simulator model of movement through information space. We also provide means to specify a hierarchy of abstracts of data of different types and resolutions. In addition, multiple portals on data may be related as master and slaves. The underlying DBMS support for this system includes the compilation of query plans into *megaplans*, new algorithms for data buffering, and provisions for a guaranteed rate of delivery.

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 circulation models[Mech91] as well as in remote sensing applications[Dozi89]. Also prevalent are large data structures modeling roads, drainage networks, vegetation patterns, etc. that are represented as collections of arcs, polygons, or points. Popular visualization systems such as AVS, Explorer, or Khoros offer scientific users a visual programming environment and powerful visualization tools to manipulate and display scientific data. Most existing systems, however, provide only primitive data management support. In particular, they can only read or write data from files, and are geared toward manipulating a fixed set of data types.

We propose that visualization systems be supported by database management systems rather than file systems. We introduce DBMS support for the needs of scientific users. With these users in mind, we support the diagrammatic, "boxes and arrows" visual programming paradigm used by existing scientific programming systems. We call such visual programs *recipes* because they specify how a collection of inputs are to be "cooked" to produced a desired visualization output. With DBMS support, we are able to execute such recipes more efficiently and manage the creation, storage, and retrieval of recipes better. By exploiting a close coupling with the underlying data manager, we are also able to provide additional features of automatic abstracts, synchronization of browsers, guaranteed data delivery, and visual update of data.

*Tioga is an Iroquois word meaning "where it forks." Tioga Pass is the mountainous entrance to Yosemite National Park from the east and is the highest automobile pass in California. The Tioga system is part of the Sequoia 2000 Global Change Research Project.

This paper is organized as follows. In Section 2 we explore the architecture which we are implementing as part of the Sequoia 2000 project[Ston92]. In this architecture, certain services are performed within the DBMS and others in a separate front-end rendering engine. Section 2 also describes the protocol by which these two components communicate and indicates how this protocol is different from previous ones such as cursors and portals[Ston84]. Then, in Section 3 we describe how our framework supports additional functionality in the areas of abstracts of data, browser synchronization, and visual updating of data. Section 4 discusses the run-time support provided by the DBMS for recipe execution. This includes extending query plans, optimizing recipe execution, and providing a guaranteed rate of data delivery. Lastly, in Section 5, we conclude with an update of our current status and a look at future issues.

2 A New Recipe Management Architecture

2.1 Introduction

Existing scientific programming systems allow the user to create visual programs by connecting modules with an easy-to-use graphical user interface. The modules typically represent functions or programs written in a conventional programming language. The modules are depicted on screen as boxes with connections for inputs and outputs. The user connects the boxes with arrows to create a directed graph that represents the final program, which we call a *recipe*. 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. Most visualization systems supply the user with a collection of modules that perform common visualization and rendering functions. 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.

Consider as an example a recipe for the detection of wildfires using satellite images produced by an Advanced Very High Resolution Radiometer, or AVHRR. Although the ash produced by wildfires in dense boreal forests is relatively easy to detect from an image, a fire in the mixed terrain of the California Sierra is much harder to classify. An earth scientist begins with a composite, cloud-free satellite image. He then calculates the "greenness" of a given pixel using two image bands. This calculation would be the first box in a recipe to study wildfire detection. Next, actual current wildfires are identified by locating areas of the image saturated in the thermal infrared band. This would form the second box of the recipe, with input from the first box, which produced "vegetative index maps". Since wildfires and areas of harvested crop look similar on an image, another box might superimpose these wildfires over a map which indicates crop areas to eliminate the crop areas from consideration. Finally, the earth scientist would like to store a series of these results and play them forward in time to watch wildfires emerge and spread. He would also like to zoom in on these images for more detail as they are playing, perhaps noting that certain areas are not wildfires and requesting that the system recalculate the images. Figure 1 illustrates this example.

2.2 Requirements and Related Work

Our architecture is motivated by the fact that many objects visualized by the scientific community are very large and would be best managed by a database. Since such objects are complex and are not well served by conventional relational DBMSs, the DBMS research community has constructed a collection of next generation DBMSs that support such objects much more effectively. Example data managers in this class are POSTGRES [Ston90], IRIS[Wilk90], Starburst[Haas90], and Orion[Kim90]. Our architecture assumes the presence of a next generation DBMS.

Two features of POSTGRES were important in our design of Tioga. First, POSTGRES supports a facility through which a user can define new data types. Such types can either be additional base data types which augment the standard collection of integers, floating point numbers and character strings, or they can be composite data types. Second, POSTGRES supports a facility through which a user can *register* a previously written function. The user must provide the types and number of the input arguments and the type of the function result as well as the location of the code for the function. Currently, POSTGRES supports functions written in the query language and functions written in C. See [Mosh91] for more information.

Although Tioga is oriented towards POSTGRES, our proposal can be readily adapted to any system that supports an extendible type system, user-defined functions, and a multi-dimensional access method, e.g. [Robi81, Niev84, Gutm84, Rous85, Ston86, Gree89, Kolo91, Ston91]. Note that our architecture differs from other work that sought to support scientific users of database systems. Previous efforts have tended to concentrate on broad requirements[DeWi82], representing scientific data[Ozso85], and statistical computations on large databases[Baru84]. Little attention has been addressed to the programming needs of the scientific user of a DBMS. Instead, work on programming language integration with DBMSs has focused on the seamless integration of general purpose languages, such as C++, with data base systems[Rich87, Agra89].

Although the main motivation for using a DBMS for scientific data is storage management for large data objects, a DBMS provides other useful features as well for the scientific community. A DBMS has the ability to scale up gracefully for these large data sets. Access to data in a DBMS is semantic; queries, rather than file names, identify data, and data may be accessed at a finer grain than files. Finally, the customary DBMS features of transactions, security, views, and multi-user consistency provide to the earth science community features it has long had to do without.

2.3 Recipe Construction

Our recipe architecture preserves and generalizes the boxes and arrows user interface from commercial packages. We assume a diagram editor which supports construction of recipes out of a menu of building blocks which we term *ingredients*. Our architecture has as its cornerstone that each function registered with POSTGRES is automatically an ingredient, and is thereby in the menu of building blocks. Thus, the menu of building blocks will be constructed by the diagram editor from reading the appropriate POSTGRES system catalog. Figure 1 gives an example of the user interface for the recipe editor. The large scrollable area represents the *workspace* where the user places building blocks chosen from the *palette* below. Choosing an ingredient adds a single box; choosing a recipe adds an entire diagram, which can then be modified. Browsers, which are used for data visualization, are represented by round-cornered rectangles. The rightmost column supplies building blocks for constructing specialized queries on recipes themselves. The results of such queries are additional recipes that can be loaded into the Recipes menu.

In a boxes and arrows diagram, a one-way connection between two boxes indicates that the result of the first registered function is to be passed as input to the second function. 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. Either 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`. The details of this coordination and other aspects of recipe execution will be covered in Section 4.

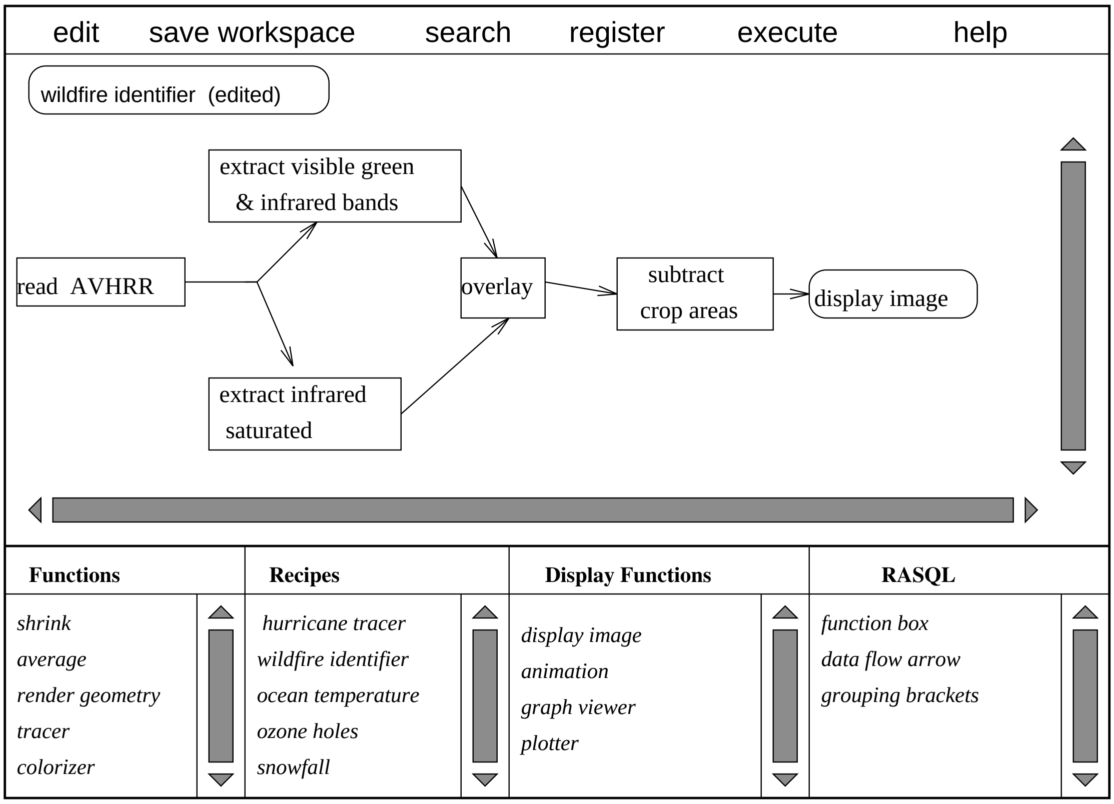

Figure 1: Recipe Editor

As a recipe is being constructed, the editor can automatically perform appropriate 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. Although not shown in Figure 1, the editor will support the use of icons to indicate types. We would encourage type creators to design icons which give visual clues as to the relationship of the type to other types. For example, icons of types within the same inheritance hierarchy may have similar graphical features. In this way, the user will be able to note easily the compatibility of types by the appearance of their icons, thereby performing a kind of visual type checking. In addition, the editor notes which function input parameters are missing, i.e. not provided by an incident edge from some other function. Such parameters can be associated with a pop-up window at recipe execution time, at which time the user can supply the missing parameters.

The diagram editor allows interactive construction of recipes from a menu of ingredients. There are two semantically different kinds of building blocks. The first are conventional POSTGRES functions as noted above. The second are rendering boxes which take input arguments just like conventional functions; they may optionally produce an output which is the data type `set-of image`. This output can be used as the input to subsequent boxes in a recipe, so that processing on screen images can be supported.

We call such rendering boxes *browsers*. There can be an arbitrary number of browsers in a recipe. The diagram editor will disallow type mismatches between browsers and ingredients. Sophisticated users can define new kinds of browsers to meet specific rendering needs. Both browsers and normal ingredients are extendible in a natural way. There are two aspects of rendering boxes that distinguish their behavior from normal ingredients. First, they are assumed to output to a window on a screen and to interact with a human user. Hence, architecturally, they run as a DBMS application program and do not run as user-defined DBMS functions in the DBMS address space. Second, communication of results from an ingredient to a browser is handled by a generalization of the cursor protocol used by an application program to communicate with the DBMS.

Using the diagram editor, the user can construct a recipe consisting of ingredients and browsers attached together into a directed graph. Such a recipe can be saved in two different ways. The recipe can be stored as a graph-like structure in a *cookbook*, a collection of recipes in the database. We provide a query tool to support browsing the cookbook. This tool RASQL is described in a companion paper[Chen92]. RASQL is integrated with the diagram editor, so a user can retrieve a recipe from the cookbook, modify it with the diagram editor, and then install his new recipe back into the cookbook. Alternately, a recipe can be encapsulated or **canned** into a new ingredient. In order for a recipe to be transformed into a legal POSTGRES function, 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. No browsers are allowed because browsers necessitate running a separate process for visualization. The single output restriction is compatible with the notion of normal building blocks, which have only one output. At encapsulation time, certain optimizations occur as discussed in Section 4. Encapsulated 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. In this case the appropriate POSTGRES commands are retrieved, any missing input parameters are prompted for at run-time, and a window for each browser is generated. To run the recipe, the browsers interface to the DBMS using the protocol described in the next subsection.

2.4 The Browser-DBMS Protocol

As noted in the previous subsection, a recipe consists of a collection of interconnected functions, and may contain one or more browsers. Each browser is run as a DBMS application program that interacts with the recipe engine which manages the execution of the ingredients of the recipe as database commands. In this section we indicate the protocol for communication between a browser and the DBMS. The interaction between the human user and the browser is unconstrained; however, we expect that browsers will provide intuitive graphical user interfaces that allow users to browse the output of the recipe easily.

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 choose a different approach. Each 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 a global coordinate space. All objects in the system are located in a common N-dimensional coordinate system where the number and characteristics of dimensions are appropriate to the specific application. The geometry of an object may be either a polygon[1] or a point. It is the job of the human recipe designer to ensure that his recipe produce the geometry representation (polygon or point) expected by some browser so that the browser receives pairs of the form (object, geometry). Failure to provide this will result in a type mismatch. Our protocol will operate effectively for any browser that accepts one of these two data representations or any subclass of them as input. Whether other geometry formats are needed for effective browsing remains to be investigated.

To achieve a common polygon representation, we have defined a standard N-dimensional polygon, `N-D-polygon`. This polygon representation has vertices which are N-tuples of integers. If the data is polygonal in nature, the generic tuple passed to the browser from the recipe will have the form:

{value-of-any-type, type, instance-of-N-D-polygon}

The value-of-any-type can be an instance of a base type or a composite type, and its location is represented by the N-D-polygon as indicated. For example, the value-of-any-type might be a satellite image. Its type might be AVHRR, and the polygon associated with it might be a rectangle representing one of the quadrangles of a U.S. Geological Survey map. An N-D-point is a merely degenerate N-D-polygon; hence, this representation works for point as well as polygon data.

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 to F(value) <operator> <constant> along (Δ_1, ..., Δ_N)
FETCH (number)
FETCH (Δ_1, ..., Δ_N)
FETCH (Δ_1, ..., Δ_N, angle)

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. Such marks can be permanent if they are defined as part of the data type of the object. In this case saving marks will require an update to the database. Usually marks will be local to a specific browsing session.

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 point that it calculates in some fashion, or it can move in some direction, denoted by

Δ_1, ..., Δ_N

until some condition

F(value) <operator> <constant>

[1]In this document, by polygon we mean a general N-dimensional polyhedron, not merely a two-dimensional polygon.

is true. This latter 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 it hits land. If landfall of the hurricane can be expressed as a predicate, then a MOVE command can express this desire. The command would look like

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

The +1 means move along the positive axis of time, assuming time is the last dimension in this global coordinate system. Note that recipes may be fast-forwarded in this fashion in any dimension. Moving forward along a non-time axis in the coordinate system implies that values are non-decreasing. Thus fast-forwarding in a non-time dimension will display records in non-decreasing order.

There are three 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, and lastly, it can request all the instances within a specific **frustum**, or truncated cone. In the first case, the number of instances requested is returned by running the recipe forward from its current position, wherever that happens to be. 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 global coordinate system. In the last case, the frustum is specified with its origin at the current position, a specific N-dimensional offset as its centerline, and an angle in degrees, which is measured from this centerline. Thus the frustum defines a cone of view onto the data, with the apex of the cone corresponding to the viewer's eye position.

The browser is expected to locate and display objects in N-dimensional space using whatever algorithms it wishes. The scientific visualization of the data is done by the browser. As the user moves through N-space with a joystick-like interface, it is the responsibility of the browser module to run the appropriate move and fetch commands to support the user. It is also the browser's responsibility to display appropriately the values that are returned from the recipe, using a display system similar to SDMS[Hero80]. To assist the browser, the DBMS will store functions of the form

display (object, N-D-polygon, screen-requirements)

which returns a screen representation for a given data object, locating it within a specific polygon of the global coordinate system in a way consistent with the screen requirements. The screen requirements might include, for example, the dimensions in pixels of the area to display into and the number of bits of color which the screen supports. Display() functions will be type-specific with a generic display() function supplied to take any data type. It can be overloaded using the standard inheritance mechanism by a class designer who wishes to specialize display() functions. The display() function can return either a *renderable object*, a abstract data type encapsulating sufficient information in order for the browser to do rendering, 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. The display() function may also return additional information to the browser for abstracting purposes; this will be discussed later.

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 the generic one or one written by the designer of the EMP class. The display() function might 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 manager. These can then be separately rendered by calling the display function again.

2.5 Relationship to Other Browsing Paradigms

Cattell and Rogers [Roge87] describe a user interface which uses an entity-relationship data model constructed for a given data base. With such a model in place, the user is given a browsing paradigm whereby he can navigate the E-R diagram by following "next" and "previous" links in an identified set of records as well as by following an E-R link to an associated record. This navigation paradigm is similar to recipe management: with recipes, the user can browse through a collection of records in a browser. Moreover, a "boxes and arrows" diagram can be constructed that is similar to an E-R diagram, by merely decomposing a relationship into two functions and then implementing both as additional boxes. On the other hand, recipe management is not bound to an E-R model but can implement many kinds of relationships between records. Also, multiple kinds of browsers can be included in our architecture.

USD[John92] has a similar "boxes and arrows" diagram notation, and each box can be a function as in our proposal. However, USD enforces a semantic net data model on the diagram, whereas we make no such restriction. Also, USD is not closely integrated with a DBMS and has none of the extensions covered in Section 3. In a sense, recipe management is a generalization of USD.

3 Extensions to Recipe Management

By using a DBMS to support the data needs of recipe management, we are able to provide additional functionality for recipe management. In the following subsections, we discuss how the recipe manager can be extended to handle abstracts, synchronization of browsers, computational steering, and visual update of data.

3.1 Abstracts

A crucial capability of recipe management is user control over the resolution of the visualized information. For example, the user interface must allow the user to zoom in to 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[Hero80], which used it in a browsing context. In SDMS additional detail appeared automatically and was hard-wired into the system. For recipe management, we propose the following much more flexible scheme. Let every recipe optionally have one or more children, which will be termed *abstracts* for the given recipe, since they contains less information. Conceptually, they are analogous to textual abstracts for a conventional document. Note that an abstract need not produce the same kind 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".

Our proposal includes organizing recipes into a directed graph 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 directed graph has a layout preference function which returns the minimum and maximum size screen representation that particular recipe can generate. The browser user interface can then perform zooming by interacting with the recipe management system as follows. The browser begins at a specific node in the abstract graph and determines the minimum and maximum size screen representation that recipe can produce. If the user zooms between those limits, then the display() function for this particular recipe is applicable. If the user zooms in beyond the level of detail provided by the maximum size screen representation, then one of the parents of the recipe should be run instead because the parents of the recipe are presumably abstracts with greater detail. Similarly, if the user zooms out beyond the coarsest level of detail provided by the minimum size as returned by the layout preference function, one of the children of the node in the abstract graph should be chosen to provide less detail. In this way, the recipe management system can be directed to move among the different nodes of the abstract graph by the user interface.

A node in the abstract graph is usually a recipe, as just explained, but it may also be a function. If it is a function, then the recipe execution engine will run the function on the existing data from its child node to produce a more detailed representation. If the node is a recipe, then the recipe execution engine must change to an entirely new recipe. The execution engine will position the new recipe at the appropriate current location and the browser can then perform a FETCH command, to refresh the screen with objects from the new recipe.

Using abstracts, the user interface can control the screen resolution for scientific visualization. However, abstracts have many other uses. For example, in a conventional business data processing application, the coarsest granularity might display an icon for each employee, the next granularity might indicate his relevant personnel information, and the finest granularity might indicate all personnel and biographic information. As such a very general retrieval system can be implemented by a browser with a "move and zoom" capability.

3.2 Synchronization of Browsers

A traditional user interface has a single cursor through which the result of a query or a view can be delivered to an application program. However, a user of recipe management might want to put several browsers in his diagram and visualize the data at several points in the diagram simultaneously. We propose that an arbitrary number of named browsers be available in a recipe. The reason that browsers should be named is to support the zoom capability. If the user zooms in and activates a new recipe in the graph, then his display should seamlessly change to the output of the correspondingly named browsers in the new recipe.

With the existence multiple browsers, the users may wish to constrain 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. More generally, we permit a **translation function** to be defined that 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 wished to view simultaneously two portions of a map, separated by a fixed distance. Synchronizing one browser to another produces the desired behavior, namely that moving the viewing window on one portion would result in a corresponding change in the viewing window on the other portion.

3.3 Computational Steering

Computational steering is a term used in scientific visualization to indicate that the course of a computation can be "steered" in real time, interactively, by a user. Traditional custom-designed modeling programs do not allow this flexibility, requiring instead that the scientist edit and recompile a computer program in order to effect a change in parameters of the program. The recipe system we propose here allows for computational steering by allowing the user to change parameters to any box of the recipe as the recipe is running. The user gains two advantages in this way: the effect of changes to the model parameters is visible immediately; and the user can begin to account for errors in visualized data by changing parameters to see their effects on the error.

3.4 Visual Update of Data

We support visual updating of data if the creator of a type has defined a update() function associated with that type. The update() function is, in effect, a type-specific on-screen editor. These editors are invoked by the browser when the user selects a object on the screen to edit. Recall that the browser allocates screen real-estate to various display() functions. Therefore, the browser can determine, from the user's screen selection, which data object has been chosen. The browser then invokes the update function for that object. Users may register update functions of the following form with the DBMS:

update (object, N-D-polygon, screen-area, open-portal)

The update() function will typically use the screen-area allotted to draw a dialog box for input from the user. The new value from the user is sent to the database via the portal through a normal database update command. The update function will also return the new value to the browser so that it may replace the current display of the object with the newly updated representation.

4 Recipe Execution

4.1 Recipes, Views, and Query Plans

At first glance, the proposed recipe management architecture may seem to be merely a convenient user interface for specifying views for a next generation system. Alternatively, one could think of a recipe management system simply as a convenient query specification tool, since each box of the recipe corresponds roughly to a query on the DBMS. Compiling a recipe, that is, converting the graph of boxes and arrows into a series of queries on the DBMS, results in one or more query plans, which is indeed the same result produced from compiling the output of any other query tool. Here we discuss several crucial ways in which recipes have expanded features relative to views and query plans.

First, recipes have an N-dimensional browser-DBMS interface, which is a generalization of the one-dimensional interface available for views and query plans. Traditional DBMS cursors such as those found in SQL fetch single records along a linear ordering in one dimension. SQL-2 and SQL-3 generalize this interface so that multiple records can be fetched in either a forward or reverse direction. In this way, they include some of the constructs proposed in portals, which allow an application program to retrieve multiple records in a variety of ways along a single dimension[Ston84]. Our browser-DBMS protocol generalizes portals to operate in an N-dimensional space. Recipes do not include explicit update commands; rather they rely on the browser to issue separate POSTQUEL commands for this purpose. Because a unique identifier (OID) is automatically returned with each object, the browser can easily perform a separate update, if it desires. In this way, recipe management follows the lead of portals, which include the same capability.

Second, each box in a recipe can have parameters which are expected to be filled in when the recipe is run. These correspond to run-time parameters in a compiled query plan. Such parameters are not allowed in conventional views. In this aspect, recipes resemble compiled queries but not views.

Third, the recipe model includes the idea of a hierarchy of abstracts of data, so that data can be viewed at multiple resolutions. Rather than being hardwired, the hierarchy may be built by the user. There may be more than one abstract of a recipe, and the type of the abstract is unconstrained.

Lastly, a recipe is really a directed graph of query plans to which are connected multiple browsers. A query plan is a tree to which a single cursor is attached. The multiple browsers may work independently, but they may also be synchronized using a master-slave relationship. Both of these extensions can be used in the query optimization process as discussed in the next section.

4.2 Extending Query Plans

Recipe generalize query plans in two different ways. First, it is a directed graph of boxes, each of which is a query plan. Second, multiple browsers can be connected to a recipe, whereas only one application program can be connected to a query plan through a cursor. It is straightforward to generalize POSTGRES query plans to support a directed graph of individual plans. All that is required is to introduce 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. We call a directed graph of plans a *megaplan*. Connecting multiple browsers to a megaplan is not difficult, because POSTGRES is designed to be *demand driven*. In other words, the browser requests one or more records from the root 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.

Supporting multiple nodes that request records presents a problem. Each node in a recipe has an internal state, namely a current record. The recipe executor can run the recipe forward if the data asked for in a FETCH command has not previously been requested. Since POSTGRES supports portals, data previously requested can also be fetched. However, if a node has a tee at its output, then records go to two places. If the browser downstream of one output branch makes a request, then the state of the node will change. In order not to change the state of the other downstream browser, we must cache the output at the tee. The second browser can then maintain its original state.

This caching problem brings us to the general problem of how to optimize recipe execution. The execution of a recipe can be made efficient in two ways. First, we can buffer the intermediate results from individual ingredients in the recipe. Second, we can collapse sequential ingredients into a single function which is potentially more efficient. We will describe these two methods as separate approaches, then discuss how we can construct an overall query plan for an entire recipe by combining the two.

4.2.1 Buffering

A recipe may need to be executed again after the browsers display the initial data. This may happen for two reasons. First, when a user wants to see a different portion of data, the browser must fetch new records. Second, run-time parameters for functions somewhere in the recipe may be changed. The user then wants to continue browsing the new result. In both cases, buffering the final and intermediate results can save computation.

In the first case, if the recipe manager buffers all previously observed output, obtaining data which has already been computed requires little extra effort. If a spatial index has been constructed on the N-dimensional geometries of the recipe output, then previously computed data can be rapidly looked up with the index. If the requested data in the fetch command is not found in the buffer, however, the recipe executor will have to run the recipe by advancing or backing up the cursor.

The second case is where a run-time parameter has changed. If we have buffered the input to the function farthest upstream whose run-time parameters may change, we do not need to reexecute upstream portions of the recipe. Recomputation is required only for boxes that are downstream from this function. The buffered data downstream is flushed before the recomputation.

The observations above suggests three possible locations for buffering in a recipe diagram.

1. Input to a browser.
   If the data for which the browser issues fetch commands is already in the buffer, there is no need to reexecute the recipe.

2. Output of a function which goes to more than one function
   This corresponds to a fork in a recipe diagram. It is likely that more than one browser is executing this recipe and each can request new records from this function independently. Buffering the output at forks reduces the need for recomputation that would otherwise be triggered by downstream browsers requesting different records.

3. Input to a function which has run-time parameters
   When a run-time parameter of a function changes, we must execute the recipe ingredients downstream of this function. If we buffer the input to this function, however, we can at least avoid reexecuting functions upstream of it.

Buffering in other locations bring little additional benefit, because the above three cases take into account all situations where changes could affect the value of the output.

The buffering of data is done by creating a temporary class for the output of a function. If space considerations preclude the buffering of all the desired data, then the following simple algorithm can be used to decide which tees, and also which ingredients in a straight sequence of ingredients, warrant caching. For each recipe, we can maintain the following statistics for each ingredient, I:

N(I) = the number of times the run-time parameter of this ingredient has been changed
S(I) = the average size of the output of this ingredient
C(I) = the average cost of running the recipe from the last cached place to this ingredient

For each browser, V, we maintain:

N(V) = the number of times this browser has backed up to previously calculated data
S(V) = the average size of the browser input
C(V) = the average cost of running the recipe from last cached place to this browser

If the recipe manager is allocated a fixed amount of buffer space, SP, then it can proceed as follows. Find node L from all ingredients and browsers where C(L)*N(L) is maximal. Allocate C(L)*N(L) of buffer space to node L. The overall buffer space, SP, is reduced by this amount. Recompute the cost of each browser and each ingredient by eliminating the cost of any nodes prior to and including the node that will be cached. Find another node L by maximizing C(L)*N(L) and continue this greedy algorithm until no additional buffer space remains. Although this algorithm is not optimal, we expect it will give good real-world performance. A simulation study is planned to test this hypothesis.

This caching of intermediate results has been advocated in[Sell88, Sell90]; however, this work is interested in the optimization of multiple queries in a query stream and hopes that a previous result can be useful as a part of a subsequent query. In our environment, when a recipe input is changed, we can avoid recreating the whole recipe by using this caching technique.

4.2.2 Compiling Recipes

Compiling recipes entails transforming the diagram of boxes and arrows produced by the user into a query plan. Initially, each ingredient in a recipe diagram is a separate POSTGRES function written in either C or POSTQUEL. Often, however, sequential ingredients can be collapsed together into one box. Sequences of POSTQUEL functions can be coalesced into a single POSTQUEL function using query modification. The new function has inputs of the first function and the output of the last, and the run-time parameters of all functions in the sequence. The reason for doing this is that the query plan for the combined POSTQUEL function may be more efficient than the query plans of the individual functions executed serially. The algorithm for collapsing two POSTQUEL functions is basically the query-modification technique for view composition discussed in [Ston75]. As [Ston75] notes, though, if either POSTQUEL function includes aggregate functions, this technique fails.

If a recipe ingredient is opaque to POSTGRES, such as a C function, it can still be coalesced with a preceding POSTQUEL box. One simply brackets the C function around the target list of the previous POSTQUEL command.

When a function, written in POSTQUEL, has outgoing edges to two or more subsequent POSTQUEL boxes, then the first function can be coalesced into each of the subsequent functions using the above query modification rules. Since the first function will be executed as part of each coalesced function, it will be executed repeatedly.

A function with more than one input can be combined with all its preceding functions by applying the above technique, one function at a time. In this way it is possible to collapse any recipe diagram into a diagram with only one node per browser.

Coalescing functions has a significant disadvantage. It is no longer possible to cache intermediate results because they have disappeared inside a single query plan. Hence, if the user changes a run-time parameter which was originally in the second ingredient of two, there is no longer any buffering between the ingredients, and the entire combined plan must be reexecuted.

4.2.3 Optimizing recipe execution

When we construct a query plan for a recipe, we must decide which functions will be coalesced and which outputs should be buffered to construct the most efficient plan.

Consider the sequence $\longrightarrow$ A $\longrightarrow$ B $\longrightarrow$, where A and B are POSTQUEL functions and the input to A is buffered.

Rule 1 If B has no run-time parameter and A's output goes only to B, always collapse this sequence. There is no gain in buffering between these functions.

Rule 2 If A's output goes to other functions as well as B, always buffer the output from A.

Rule 3 If A's output goes to only B, and B has one or more run-time parameters, then compute costs using the following formula to decide whether to buffer between A and B or coalesce A and B.
Cost of Buffering Method = cost(A) + cost(B)*(1+N(B))
Cost of Coalescing Method = cost(AB) * (1+N(B))

AB is the combined POSTQUEL function, and N(B) is the number of times the run-time parameters of B are changed and the recipe is reexecuted. The costs depends on how efficient the combined POSTQUEL function is and how frequently the parameters in B are changed. In the buffering method, each change to a parameter of B requires only the reexecution of B. This is more efficient if B's run-time parameters change often. On the other hand, with the coalescing method any change to B's parameters requires executing AB. AB may be cheaper than running A and B sequentially, but it is also probably more expensive than running just B. Note that the cost of A, B, and AB are known to the DBMS, but N(B) is usually unknown. This presents a problem with this technique. If buffer space is limited, we must take it into consideration in the calculation of cost as well. The estimation of N(B), the impact of limited buffer space on cost, and the effect of a distributed environment on these costs are all issues which we plan to study further.

When a recipe is compiled, decisions are made about where data will be buffered and which sequential ingredients will be collapsed into single ingredients. Compiling is easiest if the compiles assumes buffering at every node and collapsing of no nodes. Although this does not produce an optimized solution, it may be preferable for the user, since the actual execution of the recipe will correspond exactly to the diagram of boxes and arrows the user drew. Even if collapsing does occur and buffering is optimized, it may still be preferable to present the user with the original diagram layout when executing the recipe. Otherwise a new diagram layout must be generated by Tioga to indicate which functions have now been collapsed into one. This may be difficult, especially if the names of run-time parameters for to-be-collapsed functions conflict.

A recipe can be recompiled at run-time. At this time more accurate estimation of cost might be possible, since the system could monitor actual parameter changes to produce more accurate N(B) values. The disadvantage of recompiling is clear: compiling is a time-consuming process probably best done as a batch job. In addition, the original recipe layout must still be available, not a new diagram generated by the previous compile when ingredients were collapsed.

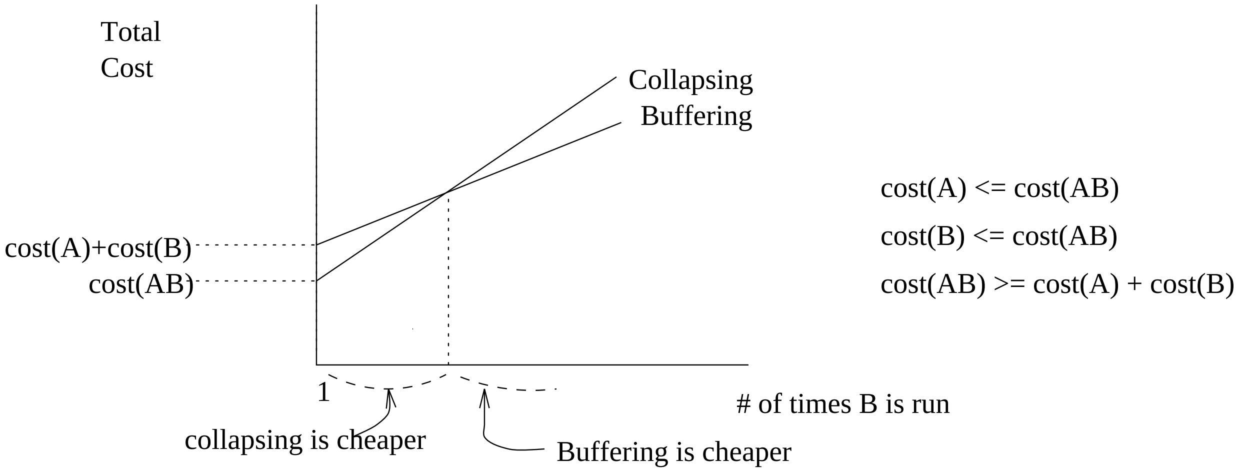

Figure 2: Cost Analysis of Buffering vs. Collapsing

4.3 Guaranteed Data Delivery

Many scientific visualization applications involve synchronized, interactive presentation of data that require input data at a predictable rate. For example, oceanographers need to view volume and surface data from the atmosphere and the sea surface simultaneously. Data from the two sources must be mapped to a common grid and displayed. Clearly the rate of arrival of data from both sources must be guaranteed so that it may be synchronized. The problem differs from standard real-time systems in several ways: the guarantee applies to a rate of data delivery, not a deadline for delivery; the visualization may start at an arbitrary time; the rate is determined by the scientist, not by the physical system; and the quantity of the data to be guaranteed is typically very high.

Researchers are already attacking the problem of how to provide guaranteed network performance; however, it is clear that overall data delivery guarantees can only be met if all components of the system, from the I/O subsystem to the database to the network, agree to meet guarantees. Otherwise, the component that has not agreed to the guarantee can become a performance bottleneck that prevents the overall delivery guarantees from being met. In order better to support applications such as animation of scientific data, we propose to support guaranteed data delivery from the database so as to work in harmony with other delivery guarantees from other components of the system.

We assume an architecture as shown in Figure 3. In the diagram, the network boxes indicate either local or remote network connections. Local connections are assumed to be fast enough to meet delivery guarantees. The network manager is assumed to support delivery guarantees for remote connections using approaches such as [Ferr90]. Rates of data delivery will be specificed via contractual protocols that each subsystem will follow. Note that performance constraints must propagate throughout all components and agreements must be secured from all components before the contract can be reliably carried out.

Since the ultimate performance demands stem from interaction with the user, the visualization system must be responsible for initiating any performance contracts. The visualization system begins by proposing a *contract* which specifies data delivery rates of X bytes per second. The contract is then propagated to all underlying systems. If the network, data manager, and operating

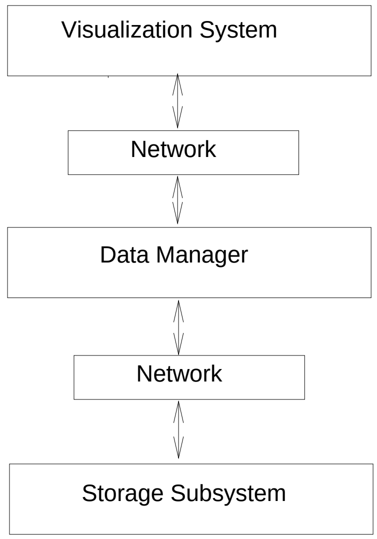

Figure 3: Architecture for Guaranteed Data Delivery

system all agree to deliver on the contract then the contract is considered *signed*. In cases where the underlying systems cannot deliver, they may respond with counter-offers and negotiations for a modified contract may occur.

Assuming that the network manager has agreed to deliver on the contract, we now consider how the DBMS can also provide a guarantee. Traditionally, DBMSs create an optimized plan for an ad-hoc query at run-time[Seli79]. The query optimizer attempts to find the lowest cost plan. Optimizers already estimate costs in terms of I/O and CPU resources. I/O resources are usually measured in disk pages fetched and CPU resources in estimated number of instructions. Given the throughput and computing power of the actual hardware platform, these estimates can be converted to elapsed time. In effect, the optimizer can consider the cost function:

$$Cost_{Time} = T_{I/O} + T_{CPU}$$

where $T_{I/O}$ and T_{CPU} are the times needed for I/O and CPU operations, respectively. During actual execution, the data manager may receive a very different allocation of resources, and in most systems today, the allocation of I/O and CPU resources can vary unpredictably.

In order to optimize correctly, the database needs to obtain guarantees of some fraction of total I/O and CPU resources available, $F_{I/O}$ and F_{CPU}, from the operating system. Given such guarantees, the query optimizer can then consider the cost function:

$$Cost_{Time} = T_{I/O}/F_{I/O} + T_{CPU}/F_{CPU}$$

Since the database knows the expected number of records returned for a given query, it can estimate the number of bytes, NB, that will be returned. If the operating system guarantees $F_{I/O}$ fraction of I/O time and F_{CPU} fraction of CPU time to the database, then the DBMS can attempt to find the query plan that will return NB bytes in the calculated amount of time $Cost_{Time}$. In other words, the DBMS can search for query plans which produce

$$X = NB/(T_{I/O}/F_{I/O} + T_{CPU}/F_{CPU})$$

where X is the bytes per second required by the original contract. If a plan can be found that satisfies this equation, then the database will agree to deliver on the contract. If this equation cannot be satisfied, then the database cannot deliver on the contract immediately; however, the database may still be able to meet the contract at a later time by first buffering the query results. This implies that contract specifications should include not only data delivery rate, e.g. X bytes per second, but also the required offset from the current time at which to start delivery, T_1. By intelligently buffering the results, the database may be able to retrieve the contents of the buffer at a later time offset T_2 at the required rate of X bytes per second. If sufficient buffering capacity is available and $B_{I/O}$ and B_{CPU} are the I/O and CPU costs in time associated with reading from or writing to the buffers, then the database needs only to satisfy the constraint:

$$X = NB/(B_{I/O}/F_{I/O} + B_{CPU}/F_{CPU})$$

in order to deliver data starting at time offset T_2 where

$$T_2 = ((T_{I/O} + B_{I/O})/F_{I/O} + (T_{CPU} + B_{CPU})/F_{CPU})$$

The database can start delivery at T_2 time units from the current time because T_2 is the amount of time necessary to compute and store the entire query result. If the database is not able to meet the original contract, it will respond to the client with the counter-offer of delivering the data at time T_2, as this may still be acceptable to the client. If buffering capacity is unavailable for some reason, then the database will respond negatively to the client and not propose a counter-offer since the desired data delivery rate can never be satisfied.

In the above description, we have assumed that the database is able to extract allocation guarantees from the operating system. This interaction is complicated by the fact that, for ad-hoc queries, the database must spend time calculating the plan. This planning time causes a lag between the time resources are requested and the time resources are actually needed from the operating system. Thus, contracts between the database and the operating system should also have a "starting at time T" clause. This would result in a more efficient use of resources during query planning. The interaction between the DBMS and the operating system is further complicated by the fact that the database may find at the end of the planning time that it is not able to deliver on the contract under any circumstances. If that occurs, then the database would like to release the resources it has reserved. Further investigation into how the database and the operating system can best negotiate these contracts is needed.

Our discussion above has dealt with ad-hoc run time queries. Database queries may also be compiled at an earlier time. To adapt compiled queries to the demands of guaranteed data delivery, we propose that compiled query plans be maintained in a table of plans. Each entry in the table contains a pointer to a plan and an equation for calculating the cost of that plan. The equations are of the form:

$$Cost_{plan} = I/O_{plan}/F_{I/O} + CPU_{plan}/F_{CPU}$$

I/O_{plan} and CPU_{plan} represent, respectively, the time required by this plan for page fetches and the time required for CPU instructions. The variables in the equation are the fraction of total CPU and I/O resources the operating system is willing to allocate to the database at run-time. Given an allocation, the best compiled plan can be chosen by calculating the cost of each plan in the table and choosing the least costly.

To generate such a table of plans, we use the following algorithm. Generate a plan. If both I/O_{plan} and CPU_{plan} are higher than some entry in the table, reject the plan. (It will always be more expensive than an existing plan in the table.) Otherwise, enter the plan in the table with its cost function, and generate another plan. Do this until the plan space has been exhausted. We assume that the number of plans possible for a given recipe is relatively small, so the size of this table is reasonable.

5 Conclusion

We have proposed here a system for database support of scientific visualization applications. Providing a natural user interface for the scientist has motivated work on multiple cursors for a query plan, intelligent buffering of computed data, and guaranteed delivery. We currently have a preliminary system, including an N-dimensional browser, working. Areas for further study include the simulation of buffering algorithms, and the effect on buffering of limited disk space. We plan work as well on the estimation and monitoring of the number of run-time parameter changes made by a user. Another goal is to study further the impact of a distributed environment on these ideas. Full implementation of the constructs presented in this paper is expected during 1993.

References

[Agra89] Agrawal, R. and Gehani, N., "ODE: The Language and the Data Model," *Proc. 1989 ACM-SIGMOD Conference on Management of Data*, Portland, OR, May 1989.

[Baru84] Baru, C. and Su, S., "Performance Evaluation of the Statistical Aggregation by Categorization in the SM3 System," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984.

[Chen92] Chen, J. "RASQL: A Graphical Query Language for Recipes," work in progress.

[DeWi82] Dewitt, D. et. al., "A Framework for Research in Database Management for Statistical Analysis," *Proceedings of the 1982 SIGMOD International Conference on Management of Data*, Orlando, FL, June 1982.

[Dozi89] Dozier, J., "Spectral Signature of Alpine Snow Cover from the Landsat Thematic Mapper," *Remote Sensing Environment*, March 1989.

[Ferr90] Ferrari, D., "Client Requirements for Real-Time Communication Services," *IEEE Communications Magazine*, November 1990.

[Gree89] Greene, D., "An Implementation and Performance Analysis of Spatial Data Access Methods," *Proc. 1989 Data Engineering Conference*, Los Angeles, CA, February 1989.

[Gutm84] Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984.

[Haas90] Haas, L. et. al., "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[Hero80] Herot, Christopher F., "Spatial Management of Data," *ACM Transactions on Database Systems*, December 1980.

[John92] Johnson, R.R. et. al., "USD - A Database Management System for Scientific Research," *Proceedings of the 1992 SIGMOD International Conference on Management of Data*, San Diego, CA, June 1992.

[Kim90] Kim, W. et. al., "Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[Kolo91] Kolovson, C. and Stonebraker, M., "Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data," *Proc. 1991 ACM-SIGMOD Conference on Management of Data*, Denver, CO.

[Mech91] 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.

[Mosh91] Mosher, C. ed., "The POSTGRES Reference Manual," Electronics Research Laboratory, University of California, Berkeley, CA, Memo 91/57, August 1991.

[Niev84] Nievergelt, J. et. al., "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems*, March 1984.

[Ozso85] Ozsoyoglu, G. et. al., "A Language and a Physical Organization Technique for Summary Tables," *Proc. 1985 ACM-SIGMOD Conference on Management of Data*, Austin, TX, May 1985.

[Rich87] Richardson, J. and Carey, M., "Programming Constructs for Database System Implementation in EXODUS," *Proc. 1987 ACM-SIGMOD Conference on Management of Data*, San Francisco, CA, May 1987.

[Robi81] Robinson, J., "The K-D-B Tree: A Search Structure for Large Multidimensional Indexes," *Proc. 1981 ACM-SIGMOD Conference on Management of Data*, Ann Arbor, MI, May 1981.

[Roge87] Rogers, T.R., and Cattel, R.G.G., "Entity-Relationship Database User Interfaces," *Proceedings of the ER Institute*, Baton Rouge, LA, 1987.

[Rous85] Rousoupoulis, N. and Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees," *Proc. 1985 ACM-SIGMOD Conference on Management of Data*, Austin, TX, June 1985.

[Seli79] Selinger, P. et. al., "Access Path Selection in a Relational Data Base System," *Proc 1979 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1979.

[Sell88] Sellis, T.K., "Multiple-Query Optimization," *ACM Transactions on Database Systems*, March 1988.

[Sell90] Sellis, T.K. and Ghosh, S., "On the Multiple-Query Optimization Problem," *IEEE Transactions on Knowledge and Data Engineering*, June 1990.

[Ston75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," *Proc. 1975 ACM-SIGMOD Conference*, San Jose, CA, May 1975.

[Ston84] 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.

[Ston86] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," *Proc. 1986 ACM-SIGMOD Conference on Management of Data*, Washington, D.C., May 1986.

[Ston90] Stonebraker, M. et. al., "The Implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[Ston91] Stonebraker, M., "Managing Persistent Objects in a Multi-level Store," Electronics Research laboratory Memorandum M91/72, University of California, Berkeley, February 1991.

[Ston92] Stonebraker, M. and Dozier, J., "SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research," SEQUOIA 2000 Technical Report No 1, Electronics Research Lab, University of California, Berkeley, March 1992.

[Wilk90] Wilkinson, K. et. al., "The IRIS Architecture and Implementation," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.