Tioga: Providing Data Management Support for Scientific Visualization Applications *

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

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

Abstract

We present a user interface paradigm for database management systems that is 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, so that a "zoom" capability can be supported. The underlying DBMS support for this system is described and includes the compilation of query plans into *megaplans*, new algorithms for data buffering, and provisions for a guaranteed rate of data delivery. The current state of the Tioga implementation is also described.

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

Other industrial and government partners include the State of California Department of Water Resources, United States Geological Survey, Construction Engineering Research Laboratory (CERL) of the U.S. Army Corps of Engineers, the National Aeronautics and Space Administration (NASA), Epoch Systems, Inc., Hewlett-Packard Corp., Hughes Aircraft Company, MCI, Metrum Corporation, PictureTel Corporation, Research Systems Inc., Science Applications International Corporation, Siemens Inc., and TRW Space and Electronics.

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[12] and in remote sensing applications[5]. Large data structures used to model roads, drainage networks, and vegetation patterns 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 they are geared toward manipulating a fixed set of data types.

We are building a next generation visualization system, Tioga, which improves dramatically on current technology. Our architecture is motivated by the fact that many objects visualized by the scientific community are very large and complex and would be best managed by a database management system (DBMS). Scientific data are not well served by conventional relational DBMSs; however, the DBMS research community has constructed a collection of next generation DBMSs which support such objects more effectively. Example data managers in this class are POSTGRES, IRIS, Starburst, and Orion. Our architecture assumes the presence of a next generation DBMS, and we are building Tioga for the POSTGRES DBMS.

Two features of POSTGRES are important in the design of Tioga. First, POSTGRES supports a facility through which a user can define new 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, POSTGRES allows users to **register** a previously written function. The user must specify the number and types 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 C or in the query language POSTQUEL[13].

Although Tioga uses 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. [7, 8, 11, 14].

Tioga differs from other work on supporting scientific users of database systems. Previous efforts have tended to concentrate on broad requirements[4], representing scientific data[15], and statistical computations on large databases[2]. 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[1, 16].

This paper is organized as follows. In Section 2 we explain the "boxes and arrows" visual programming paradigm used by Tioga. Section 3 discusses the way Tioga requires the DBMS to interact with user-space (client) code. This interface is a generalization of both traditional SQL cursors and database portals[20]. Section 4 indicates the run-time support provided by POSTGRES for execution of Tioga boxes and arrows diagrams. This includes the definition and optimization of extended query plans. In Section 5 we describe how Tioga supports additional functionality in the areas of guaranteed data delivery, abstracts of data, browser synchronization, and visual updating of data. 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, using an easy-to-use graphical user interface. 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.

Consider as an application example the detection of wildfires using images from the Advanced Very High Resolution Radiometer (AVHRR) satellite. A fire in the mixed terrain of the California Sierra is hard to identify because of the interspersion of forest and crop land. An earth scientist begins with a composite, cloud-free satellite image and a landuse map. The map is first processed to block out areas of crop cultivation, since harvested crops and wildfires both reduce the amount of vegetation evident in a satellite image. Then the altered landuse map is superimposed on the satellite image to produce a new image with the crop areas eliminated. The earth scientist then calculates the "greenness" of a given pixel in the new image using two image bands, in order to locate the forest regions of interest. After cropping the image to just forest areas, wildfires are identified by comparing fall and late spring forest size. The earth scientist finally renders the resulting wildfire images onto the screen. Figure 1 illustrates the complete diagram for this application.

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 it that a collection of ingredients is "cooked" into a final visualization output.

The cornerstone of the Tioga architecture is 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 can 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. 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.

As a recipe is being constructed by the user, the editing program automatically performs 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 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,

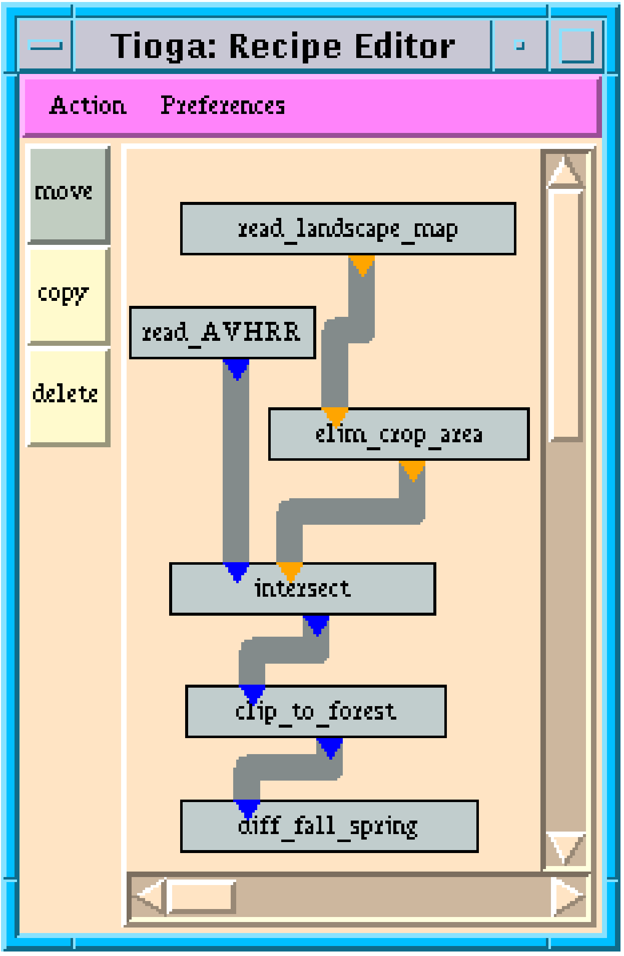

Figure 1: Recipe Editor

thereby allowing a kind of visual type checking.

When the user finishes editing a diagram, the editor notes which function inputs are missing, i.e. not provided 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. Browsers produce an output which is the data type `setof image`. This output can be used as the input to subsequent boxes in a recipe, so that processing on screen images is supported. 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. We provide a query tool to support browsing the cookbook. This tool, RASQL, is described in a companion paper[3]. 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 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. It is for this reason that ingredients may not contain browsers: browsers must be directly manipulated by 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. In this case 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 subsection.

3 Browser-DBMS Protocol

As noted in the previous section, 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 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 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 a polygon[1] or a point. It is the job of the human recipe designer to ensure that the recipe produces the geometry representation (polygon or point) expected by some browser. Failure to provide this will result in a type mismatch.

To achieve a common polygon representation, we have defined a standard N-dimensional polygon, `N-D-polygon`. 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-polygon 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 quadrants 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:

[1] In this document, "polygon" refers to a general N-dimensional polyhedron, not merely a two-dimensional polygon.

MARK (N-D-point) with identifier
ERASE identifier
MOVE to identifier
MOVE to (N-D-point)
MOVE along (Δ_1, ..., Δ_N) until
F(value) <operator> <constant>
FETCH (number)
FETCH (Δ_1, ..., Δ_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. Such marks can be permanent if they are defined as part of the data type of the object. 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 N-D-point which 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>

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[9].

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 polygon. The screen-resource argument specifies the screen resources which are available for the display of this object such 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.

The N-dimensional browser-DBMS interface is a generalization of the one-dimensional interface available for the traditional DBMS cursors found in SQL. 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[20]. 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 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.

Our browser interface has points in common with previous user interface work. For example, Cattell and Rogers [17] describe an interface which uses an entity-relationship data model constructed for a given data base. 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. In Tioga, one can decompose an E-R relationship into two functions and then browse a recipe containing something akin to an E-R diagram. On the other hand, Tioga is not bound to the E-R model but can implement many kinds of relationships between records. Also, multiple kinds of browsers can be included in our architecture.

USD[10] 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 5. In a sense, Tioga is a generalization of USD.

4 Recipe Execution

4.1 Introduction

At first glance, Tioga may seem to be merely a convenient user interface for specifying views for a next generation system. Or, one might think of Tioga as a convenient query specification tool since each box of a recipe corresponds to a query for the DBMS. Compiling a recipe entails converting the graph into a series 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 views or query plans in four crucial ways.

First, when a recipe is inserted into a cookbook, 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 is advantageous.

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 re-requesting of information that 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 remainder of this section considers these two issues. We will first describe these two tactics separately, and then show how to combine them into a single overall optimization strategy.

4.2 Buffering

A recipe may need to be re-executed in two different circumstances. First, when a user reexamines records which he has previously fetched, the browser must request them again. Second, the user may change run-time parameters for one or more functions and then re-run the recipe. In both cases, buffering the output of ingredient boxes can save recomputation.

There are three possible locations for buffering in a recipe.

1. At the output of an ingredient that connects to a browser.
   If the data required for the current fetch command is in the buffer, then an indexed lookup can replace recipe execution.

2. At the output of an ingredient directly upstream from one with a run-time parameter. When the run-time parameter changes, then upstream ingredients need not be reexecuted.

3. At the output of an ingredient which goes to more than one node.
   This corresponds to a tee in a recipe plan. If more than one browser is connected to the recipe, then buffering at tees reduces the recomputation that would otherwise be triggered by downstream browsers requesting different records.

If space considerations preclude buffering in all possible locations, then the following algorithm can be used to decide which outputs to cache. This algorithm assumes that the following statistics are available for each ingredient, I, in a recipe.

$P(I)$ = the number of times a run-time parameter for this ingredient will be changed

$S(I)$ = the amount of storage needed to effectively buffer the output of I (in bytes)

$C(I)$ = the cost of running the recipe from the "nearest upstream" buffer to I(in seconds). This includes the cost of running I. In the case where I has multiple inputs, $C(I)$ is the sum of the costs of executing portions of the recipe needed to produce each input. The cost corresponding to each input "branch" is the cost of running the recipe from the nearest upstream buffer along that branch.

Our algorithm requires one additional computed statistic:

$N(I)$ = number of change requests from nodes downstream from I, calculated as follows:

(1) if I is followed by the node J:

$$N(I) = \begin{cases} P(J) & \text{if J is buffered} \\ P(J) + N(J) & \text{if J is unbuffered} \end{cases}$$

(2) if I is followed by a browser B:

$$N(I) = P(B)$$

where $P(B)$ is the number of times the user of the browser causes a re-execution.

(3) if I is followed by a tee, then calculate $N(I)$ using method (1) and take the sum of nodes forked from the tee.

These statistics are gathered over time from previous executions of the given recipe. Periodic reoptimization allows fresh statistics to influence future megaplan execution. Our algorithm is based on statistics from a sequential execution model, i.e. the execution of ingredients is serialized. Optimization based on a parallel execution model remains a problem for future study.

If the recipe manager is allocated a fixed amount of buffer space, SP, then we use following simple greedy algorithm. Find the ingredient, I_1, which maximizes

$$\frac{C(I_1) * N(I_1)}{S(I_1)}.$$

Allocate $S(I_1)$ of buffer space to ingredient I_1 and reduce the overall buffer space, SP, by this amount. Recompute $C(I)$ for each remaining ingredient I by taking into account the buffer added after I_1. Find the next ingredient I_2 by again maximizing

$$\frac{C(I_2) * N(I_2)}{S(I_2)}$$

and continue this greedy algorithm until no additional buffer space remains.

Intuitively, $C(I) * N(I)$ is the amount of time that is saved in recomputation by buffering the output of I. The formula maximized is thereby the time savings per unit of buffer space, and the algorithm is a hill climbing one on this metric. Although not optimal, we expect the algorithm will give good real-world performance. A simulation study is planned to test this hypothesis.

4.3 Coalescing Ingredients

The ingredients in a recipe can also be coalesced into a smaller number of queries. For example, sequences of POSTQUEL functions can be coalesced into a single POSTQUEL function using the query modification technique for view composition discussed in [19]. The new function has the inputs of the first function, the output of the last, and the run-time parameters of all functions in the sequence. The query plan for the combined POSTQUEL function may be more efficient than the query plans of the individual functions executed serially. As [19] notes, though, if any POSTQUEL function in the sequence includes aggregate functions, this technique fails.

If a recipe ingredient is a C function and is opaque to POSTGRES, 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; however, since C functions cannot be rewritten by POSTGRES, no performance benefit is gained from coalescing them.

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 with no aggregates into a diagram with only one node per browser.

Coalescing two functions has a significant disadvantage. It is no longer possible to buffer the intermediate result of the first function because it has disappeared inside a single query plan. Hence, if the user changes a run-time parameter of the coalesced function which came from the second ingredient, the combined plan must be reexecuted. Uncoalesced plans with an intermediate buffer would have required only the second function to be re-executed.

The next subsection completes the Tioga optimization description by indicating how to choose between coalescing ingredients and buffering intermediate results.

4.4 Buffering and Coalescing Together

When we construct a megaplan for a recipe, we must decide which functions will be coalesced and which outputs should be buffered to construct the most efficient plan. The following heuristic algorithm contains our first simple treatment of this problem. An optimal algorithm would need to take into account the complex interrelationship between coalescing and buffering benefits. It remains an area for future study.

Our heuristic solution performs a coalescing step followed by a buffering step followed by a second coalescing step. The first step coalesces all pairs of ingredients where coalescing is more beneficial than buffering. The second step allocates available buffer space according to the greedy algorithm in the subsection 4.2. A final coalescing step is necessary to combine ingredients which were not coalesced in the first step because buffering would have been more advantageous in those cases. Step two may not have allocated buffers to all possible outputs because total space available for buffering may have been limited. Therefore, the final coalescing step is necessary to find all remaining ingredient pairs where coalescing is advantageous.

Consider the case of two adjacent POSTQUEL ingredients, A and B, where A outputs to B. We ignore cases involving coalescing ingredients implemented in the programming language C because no performance benefit is gained from coalescing ingredients implemented as C functions. In the first coalescing step there are three possibilities to consider:

1. If B has no run-time parameter and A's output goes only to B, always coalesce this sequence. There is no gain in buffering between these functions. Coalescing the functions may allow the query optimizer to pick a more efficient composite plan.

2. If A's output goes to other functions as well as B, never coalesce A and B. A would need to be coalesced into multiple ingredients, and substantial duplicate execution is inevitable.

3. If A's output goes only to B, and B has one or more run-time parameters, then compute the following formulas and coalesce if coalescing is more beneficial than buffering.

$$\text{Benefit of Buffering} = C(A) * N(A)$$
$$\text{Benefit of Coalescing} = (C(B) - C(AB)) * (N(A) + P(A))$$

Here, AB is the result of coalescing A and B. $C(A)$, $C(B)$, $C(AB)$, $N(A)$, and $P(A)$ are statistics defined as in the subsection 4.2. The benefit of buffering is the cost that would be avoided if there is a buffer on the output of A. This is an optimistic estimation of buffering benefit because we are not considering the presence of other buffers. Other buffers upstream and downstream of A would decrease $C(A)$ and $N(A)$, respectively. In addition, if ingredients were coalesced upstream of A, $C(A)$ would also decrease.

Intuitively, the $C(B) - C(AB)$ term in the coalescing formula is the benefit gained each time ingredient AB is executed instead of running A followed by running B. This benefit is gained for each change request from nodes downstream from A. In addition, this benefit is also gained for each expected change in run-time parameters for ingredient A, namely, $P(A)$. The benefit from coalescing is underestimated because coalescing accrues benefit every time re-execution of AB occurs, not just when the re-execution is caused by changes in requests downstream. Re-execution of AB can also occur as a result of changes in run-time parameters of ingredients upstream of A. Since coalescing benefit is underestimated and buffering benefit is overestimated, using the formulas above will result in ingredient pairs where coalescing is unequivocally better than buffering.

In the case where ingredient B has multiple inputs from ingredients A_1 to A_k, use the above algorithm to determine the best A_i to coalesce with B. After coalescing, ingredient B becomes ingredient A_iB. Now repeat with all remaining input branches and the new ingredient A_iB until no more coalescing is possible.

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 guaranteed data delivery, abstracts, synchronization of browsers, and visual update of data.

5.1 Guaranteed Data Delivery

Many scientific visualization applications involve synchronized, interactive presentations of data which 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 have already attacked the problem of how to provide guaranteed network performance. 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 appropriate guarantees. Otherwise, the component that has not agreed to the guarantee will become a performance bottleneck and prevent the overall delivery guarantees from being met. In order 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 2. 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 [6]. Rates of data delivery will be specified via contractual protocols which each subsystem will follow. Since

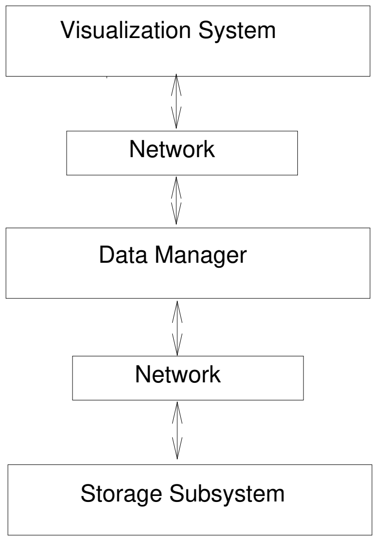

Figure 2: Architecture for Guaranteed Data Delivery

the ultimate performance requirements stem from interaction with the user, the visualization system must be responsible for initiating any performance demands. The visualization system begins by proposing a **contract** which specifies data delivery rates in bytes per second. The contract is then propagated to all underlying systems. If the network, data manager, and operating 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, a DBMS query optimizer minimizes a weighted sum of I/O cost and CPU cost[18]. Given the throughput and computing power of the actual hardware platform, each of these estimates can be converted to expected elapsed time. In effect, the optimizer should optimize:

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

where $T_{I/O}$ and T_{CPU} are the elapsed time needed for I/O and CPU operations, respectively. This assumes the DBMS is allocated all of the machine's resources. During execution, the DBMS may receive less resources, and in most systems today, the allocation of I/O and CPU resources can vary unpredictably.

In order to provide a service guarantee to the visualization system, the DBMS must obtain a guarantee for a certain fraction of total I/O and CPU resources, $F_{I/O}$ and F_{CPU}, from the operating system. Given such a guarantee, the query optimizer can then use the cost function:

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

Since the DBMS 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 DBMS, then the DBMS must find query plans for which:

$$\frac{NB}{\frac{T_{I/O}}{F_{I/O}} + \frac{T_{CPU}}{F_{CPU}}} > X$$

where X is the bytes per second required by the original contract. If a plan can be found that satisfies this equation, then the DBMS can agree to deliver on the contract. If more than one plan can be found, then the DBMS should choose the one with least total resource consumption, as in the Selinger model.

If this equation cannot be satisfied, then the DBMS cannot meet the contract immediately; however, it may still be able to guarantee the contract delivery at a later time, by buffering query results in the meantime. 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 DBMS can execute the entire query into a buffer in time T_1 where

$$T_1 = \frac{T_{I/O} + B_{I/O}}{F_{I/O}} + \frac{T_{CPU} + B_{CPU}}{F_{CPU}}$$

If the DBMS can then satisfy the constraint:

$$\frac{NB}{\frac{B_{I/O}}{F_{I/O}} + \frac{B_{CPU}}{F_{CPU}}} > X$$

then it can respond with a counter proposal containing an offset T_1 from the current time at which to start delivery.

If sufficient buffering capacity is unavailable for some reason, then the DBMS must respond negatively to the client since the desired data delivery rate can never be satisfied.

In the above description, we have assumed that the DBMS can extract allocation guarantees from the operating system. This interaction is complicated by the time the DBMS must spend calculating the optimal 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 avoids the over-allocation of resources during query planning.

The discussion above has dealt with the compilation of plans at runtime when immediate resource requests can be made. When query optimization occurs prior to execution, resource requests must be deferred until runtime. In this case we require the optimizer to construct a table of compiled query plans. Each entry in the table contains a plan and the I/O and CPU time for that plan, namely, $T_{I/O}$ and T_{CPU}. At run time, a resource allocation can be requested from the operating system and the best plan chosen according to the above formulas.

At compile time, a plan can be rejected if both $T_{I/O}$ and T_{CPU} are higher than some entry in the table. Otherwise, enter the plan in the table. Further heuristics will be needed if this table becomes too large.

5.2 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[9], 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 a **sizing function** which returns the minimum and maximum size screen representation for objects which that particular recipe can generate. The browser begins at a specific node in the abstract graph and determines the minimum and maximum size screen representations that a 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 must be run, 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 sizing function, one of the children of the node in the abstract graph must be chosen to provide less detail. In this way, Tioga recipe management can be directed to move among the different nodes of the abstract graph by the user interface.

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 "heavy-weight" 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.

5.3 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. 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 are using **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.

5.4 Visual Update of Data

We support visual updating of data if the creator of a type has defined an 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 resources 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,location,screen-resource)

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.

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, intelligent buffering of computed data, and guaranteed delivery. 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.

Areas for further study include the simulation of buffering algorithms in the presence of limited disk space. In addition, we plan to work on the estimation and monitoring of the number of run-time parameter changes made by a user. Lastly, further tuning of our guaranteed delivery system is anticipated.

References

[1] 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.

[2] 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.

[3] Chen, J. "RASQL: A Graphical Query Language for Recipes," (in preparation)

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

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

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

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

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.