Extending a Graphical Query Language to Support Updates, Foreign Systems, and Transactions

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

ABSTRACT

In [STON93] we proposed a new user interface paradigm called Tioga for interacting with database management systems. Tioga simplifies the task of building database applications and is geared especially towards the needs of scientific users. We borrow the "boxes and arrows" visual programming notation of scientific visualization systems and allow users to graphically construct applications by using database procedures as building blocks.

This paper extends the Tioga paradigm to a general database programming environment. In particular, we address three shortcomings of graphical query languages. First, we define a mechanism for allowing general programs---not just database procedures---as building blocks. This extension allows better handling of general data entry and data visualization needs and provides an interface to foreign systems. Second, we permit database updates. Third, we define a transaction semantics for graphical query languages. Unlike traditional transactions, Tioga transactions contain a directed graph of queries instead of a linear sequence of queries. We explore concurrency control techniques to promote both intra-transaction and inter-transaction parallelism.

Finally, we present query processing strategies for graphical queries with general building blocks, updates, and transactions. We show how to efficiently execute a Tioga application by decomposing the application into components that are individually optimized.

1. INTRODUCTION

The design of user interfaces for database systems is an area in need of more attention[BERN89, STON93b]. Existing database user interfaces are unfriendly and difficult for nonexperts to use. Most database interfaces take the form of textual programming languages or forms-based interfaces oriented towards business applications. In [STON93], we presented Tioga, a new paradigm for user interaction with a DBMS. Tioga is motivated by the needs of scientific DBMS users in the SEQUOIA 2000 project [DOZI92, STON92, STON93a]. Tioga has a "boxes and arrows" programming model in which user-defined database procedures are depicted as boxes and edges between boxes represent flow of data. This paradigm allows nonexperts to build visual programs called **recipes** by interactively connecting boxes together using a graphical user interface. We introduced a browsing paradigm that allows the user to visualize data results in a multi-dimensional data space. A browser is simply another type of box that can be attached to a recipe wherever data needs to be rendered.

While the idea of graphical query language such as Tioga is attractive, current systems have a number of serious shortcomings that limit their usefulness in realistic database applications. For example, like many graphical query languages, the first version of Tioga focuses on retrieving data from the DBMS and displaying it on the screen. However, in addition to the ability to query data, users need to enter data into the database. Although the first Tioga prototype contains a mechanism for changing certain parameters at run-time, no mechanism is provided for general data entry. Another important need for many users is the ability to interface with existing programs and systems. Most users have heavy investments in code that does not directly work with a database, but would nonetheless like to integrate that code with database applications. Finally, the original Tioga model contains no transaction mechanism for controlling interactions between components of Tioga recipes or between Tioga recipes and other database transactions.

This paper shows how a graphical query language can be extended for developing general database applications. To this end we enhance the original Tioga model in three ways. First, boxes may not only query data but may also perform updates to the database. Second, we permit boxes which are stand-alone programs to be used in recipes. Finally, we add transactions to control the visibility and durability of database updates.

The addition of asynchronous, concurrently executing components to Tioga requires careful consideration of the semantics for Tioga recipes. We define the semantics of what it means to change **run-time parameters** (i.e., user inputs) during execution of a recipe with asynchronous components. The Tioga transaction semantics provides a simple mechanism for controlling the visibility of updates between concurrently executing components of a recipe.

Unlike traditional transactions, Tioga transactions can contain concurrent queries within a single transaction because Tioga recipes are directed graphs of queries. Finally, our extensions also require new recipe execution strategies. We present a recipe execution strategy based on dividing a recipe into pieces and choosing an eager or lazy evaluation strategy for each piece.

Section 2 of this paper briefly reviews our previous proposal from [STON93]. Section 3 extends this model with new constructs to handle data entry and interfacing to foreign systems. In Section 4, we discuss the query processing implications of our new constructs. Lastly, we define the role of transactions in the Tioga environment in Section 5.

2. THE ORIGINAL TIOGA PROGRAMMING MODEL

Existing scientific programming systems like AVS[UPSO89], Khoros[RASU92], and Data Explorer[LUCA92] allow users to create visual programs by connecting modules, written in a conventional programming language, through an easy-to-use graphical user interface. The Tioga system preserves and generalizes the boxes and arrows user interface from commercial packages. Tioga supports the definition, manipulation and execution of boxes and arrows diagrams. We call these visual programs **recipes.**[1] A Tioga recipe is a directed acyclic graph of modules. In the original Tioga model, there are two semantically different types of recipe modules: **ingredients** and **browsers.** Ingredients in Tioga are user-defined database procedures that are registered and stored in POSTGRES. Ingredients can be implemented either in the POSTQUEL query language or in a standard programming language like C. Therefore, an ingredient can be a sequence of queries or it can encapsulate arbitrary transformations of inputs to outputs. Browsers are rendering modules that interact with the screen and run as application programs. They adhere to a client-server communication protocol described in [STON93]. Our protocol supports a joystick-oriented browsing style through multi-dimensional space.

Our original Tioga prototype includes a graphical recipe editor and a recipe execution engine. The editor allows interactive construction, modification, storage and retrieval of recipes. The executor schedules the actual executions of the ingredients. A screen dump of the prototype editor is shown in Figure 1.

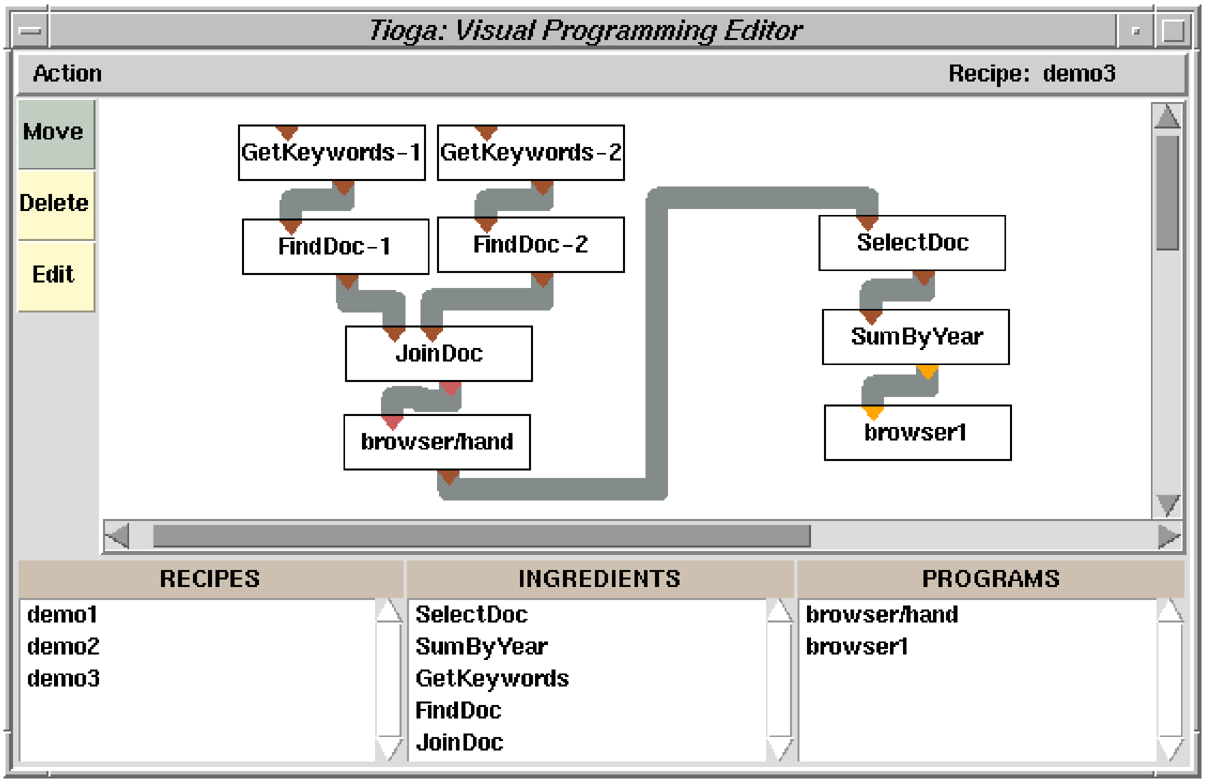

Figure 1: The Tioga User Interface

[1]The term "recipe" is used because it indicates how to "cook" a collection of ingredients into a final visualization output.

Using the recipe editor, users can construct recipes by connecting ingredients and browsers together into a directed acyclic graph. A directed edge between two boxes indicates that the result of the first box is to be passed as input to the second box. In order for such a connection to be valid, the data type returned by the first ingredient must be compatible with the type of the connected argument of the second ingredient. Since POSTGRES supports type inheritance, type compatibility means that either the output type exactly matches an input type of the subsequent ingredient, or the output type is a subtype of the input type. For example, if EMP is a subtype of PERSON, then outputs of type EMP can be passed as input to ingredients expecting an input of type PERSON. As a recipe is being constructed, the editor automatically performs static type-checking, and the user is alerted by the editor of any type mismatches. In addition, the editor notes which boxes have missing input parameters, i.e. inputs not provided by an incident edge from some other box. These parameters are considered **run-time parameters**. As the name implies, run-time parameters take on values entered by the user at the time the recipe is run. Dialog boxes pop up at recipe execution time to prompt the user for the necessary values.

In Tioga, recipes themselves are stored as data in the database. They can be queried, manipulated, and efficiently accessed just like other data. A recipe can also be **canned** or encapsulated into a new ingredient as a means of abstraction. Canning a recipe is analogous to compilation; once a recipe is canned, its original structure is lost and it becomes opaque to the user. Canned recipes are added to the collection of legal ingredients and can later be used in all respects as an ingredient.

When a recipe is executed, the appropriate POSTGRES functions are retrieved, any missing input parameters are prompted for at run-time, and each browser is spawned as a separate process. The recipe executor then supervises the execution of the recipe by generating data required to satisfy the requests from each of the browsers. Ingredient are executed as their inputs become available.

Tioga benefits users by giving them a easy to use interface for building database programs. The ability to encapsulate database queries into ingredients supports modularity and re-use. The graphical programming paradigm is more accessible to nonexperts than textual programming languages. Currently many scientists develop applications by relaying requirements to expert programmers on their staff, resulting in a slower development cycle. With Tioga, scientific users themselves can develop applications by building recipes out of ingredients implemented by expert programmers. They can then readily execute recipes, make changes to run-time parameters, browse results, and modify recipes in a rapid, interactive manner.

3. EXTENSIONS TO THE TIOGA MODEL

3.1. Introduction

In this section, we propose several extensions to our original Tioga programming model. The Tioga programming paradigm described above did not provide for adequate handling of many data entry needs. In subsection 3.2, we outline some example data-entry applications. To handle these data-entry applications, we describe in subsection 3.3 the generalization of browsers into modules that can handle input or output. The original Tioga model specified that unconnected inputs to ingredients are to be treated as run-time parameters. In subsection 3.4, we clarify how changing a parameter affects recipe execution. In doing so, we extend our model to include a new type of box called a **parameter generator**. Finally, in subsection 3.5 we point out the need to distinguish between ingredients with side-effects and those without.

3.2. Data entry applications

There are at least three different types of data entry applications we would like to support in Tioga. The following examples are indicative of what our users need.

First, users often wish to interact with a forms-based input screen to enter data that will be used in the rest of the recipe. Although it is possible to treat each piece of data as a missing parameter in the original Tioga scheme, this is awkward if a collection of data elements is to be entered. Rather than using a simple run-time parameters dialog box, it would be more convenient to allow data entry through arbitrary user interfaces, such as screen painters provided by fourth generation language (4GL) products. Screen painters cannot be ingredients because they are interactive processes that must be run in user space, asynchronous to the rest of the recipe.

Second, SEQUOIA 2000 scientists often wish to capture data from external devices such as satellite feeds. Such a feed is a continuous data stream, often producing data at a high rate. Scientific users would like to write a program to read the external device, process the arriving data and then store it into the database. We would like to be able to implement this entire capture process as a Tioga recipe. The data capture component must keep up with the external device. Hence, it must run asynchronously as a separate program and not as a user-defined function inside the DBMS. In addition, the recipe should buffer the incoming data in the event that the data is being produced at a higher rate than it is being consumed.

Third, some SEQUOIA 2000 scientists wish to incorporate existing simulation models into Tioga recipes. These simulation models are often computationally intensive and produce large amounts of data. They are typically long-running programs which are optimized to run on certain hardware architectures. Instead of requiring our users to re-code their simulation models as registered POSTGRES functions, we would like to allow them to harness their existing code and incorporate existing programs into Tioga. Scientists wish to use recipes to both assemble data that will be used to control an external simulation model as well as to receive data from the simulation model for further processing and visualization by the recipe. Therefore, Tioga must support sending output to and receiving input from external programs. Furthermore, such simulation programs often produce data in a structured manner. For example, global circulation model simulations produce data in regular grids. We would like our recipes to exploit any such semantic constraints in order to make recipe execution as efficient as possible.

These examples illustrate the need to better handle data entry in Tioga. These needs are not addressed well by simply using run-time parameters to ingredients. The browsers in our original model also do not serve well in meeting data entry needs. Browsers handle displaying of data well, but do nothing to provide input to the rest of the recipe. Often the user would like to interact with the browser in such a way that the browser would produce meaningful output. For example, a common operation is to interactive selecting a subset of data of interest. That subset should be passed on as output of the browser to the rest of the recipe.

The examples above demonstrate the need to not only better support data entry, but to better support interfacing with foreign systems in general, both for inputs and outputs. Forms-based screen painters and simulation models are examples of applications which are foreign to Tioga and run outside the control of Tioga. We need to provide gateways to such systems so that existing application programs can be readily integrated for use in Tioga recipes. To this end, we have extended the Tioga data model with the ability to deal with asynchronous inputs and outputs. The next subsection describes how we generalize the concept of browsers into **programs.**

3.3. Programs as modules in Tioga

The original Tioga model specified two kinds of modules: ingredients and browsers. Ingredients are user-defined functions and browsers are programs which read data from a Tioga recipe through a special client/server protocol. In this subsection we generalize the behavior of browsers so that they can write data to a recipe as well as read data from one. We call generalized browsers **programs.** In addition, we specify several different **behaviors** that programs and ingredients can exhibit. These behaviors are used for optimizing the execution of Tioga recipes.

Like ingredients, Tioga program modules have typed inputs and outputs, specified at program definition time. Type-checking is done for edges to programs just as it is for ingredients. While ingredients are executed in the same process as the DBMS, programs are executed in separate operating system processes. They may also be distributed and executed on different machines. This implies that they may be arbitrarily out of synchronization with the rest of the recipe. This generates buffering requirements to be discussed below. Also, since programs run in separate processes, they do not have a function call interface on input or output; rather, they use a client/server protocol to communicate with the recipe executor.

A program with no inputs is a data producer which we call a **hand,** because it hands data to a recipe for processing. A program that has no outputs is a data consumer which is called a **browser,** because it consumes data, e.g. by visualizing it on the screen. In general, a program is a browser/hand combination. That is, it can be both a producer and a consumer, acting like a browser on input and a hand on output.

A hand has **behavior** characteristics that indicate to the Tioga recipe executor how it produces data. Because many applications produce data in a regular or structured fashion, behavior specification serve as hints to the recipe executor for optimization purposes. There are three possible behavior characteristics of hands. First, a hand can produce a **stream** of objects for the rest of the recipe to consume. This stream is a sequence of ordered pairs:

(sequence#, object_of_type_O)

The sequence number is incremented each time an object is produced. All the objects in a stream must be of the same type. This stream is produced asynchronously until the hand terminates.

The second behavior that is possible for a hand is **constrained.** In this case the hand produces a collection of triples:

(sequence#, location, object_of_type_O)

Location is an multi-dimensional polyhedron which indicates the desired position of the object in an N-dimensional application-specific coordinate space. In the degenerate case, the location can be a single point. A constrained hand adheres to certain constraints as it produces data. At any given time, the hand may produce a special triple:

(sequence#, location, CONSTRAINT)

where the third element of the triple is a keyword "CONSTRAINT". The location in this special triple is used to specify multi-dimensional regions where the hand has completed producing data. Once a hand sends a constraint message, it guarantees that it will not, at any time in the future, produce additional data in the regions specified in the constraint. Global circulation models are examples of constrained hands because they produce data in grids. Once a grid is fully populated with data by the model, the simulation cannot produce any additional data objects whose locations are contained in that grid. The simulation can then send a contraint message indicating that it is "done" with that grid area. This semantic hint gives the recipe execution engine useful information for optimizing the buffering and caching of the output of hands.

The last type of behavior available for hands is **arbitrary.** In this case triples are produced as above; however, no constraint messages are ever produced. There is no restriction on the locations of subsequent objects. An arbitrary hand is a special case of the constrained hand with a null constraint region. In summary, a hand can be (S)tream, (C)onstrained, or (A)rbitrary.

A browser is a program which takes input from the recipe but does not produce any output. In analogous fashion to hands, a browser can have (S), (C), or (A) behavior. If a browser is stream-oriented, it requests records in ascending sequence number by always asking for the next available data record. A browser can also have constrained behavior where it requests objects by location. The locations requested are guaranteed to obey some constraint specified by the browser. Once a constraint region is given by the browser, no further requests can be made for data in that region. Lastly, a browser can have arbitrary behavior and request records with arbitrary locations. The original browser protocol described in [STON93] specified that browsers could request data objects that were located within arbitrary multi-dimensional polyhedrons. This behavior is an example of arbitrary browser behavior.

Finally, we also associate behaviors with Tioga ingredients. A ingredient can have (S) behavior, in which case it does not produce a location field, or it can have (C) or (A) behavior, in which case locations are produced as part of the output. In summary, the new Tioga model associates with each program and ingredient three possible input and output behaviors. These behavior specifications are used by the Tioga executor to determine buffering and caching requirements at various places in a recipe.

Our extended model addresses the needs listed in the previous subsection. A forms-based package is now a Tioga hand that delivers data to the rest of the recipe. A satellite data feed is another example of a hand, which can have one of several behaviors. It can simply produce a stream of records and have (S)tream behavior. Alternatively, it can produce values in regular grids and specify (C)onstrained behavior. Our model also supports interfaces to foreign systems such as simulation models. A gateway is a browser/hand combination that accepts inputs from upstream[2] portions of the recipe, communicates with an external program, and asynchronously produces output for downstream portions of the recipe.

3.4. Run-time parameters

In Tioga, unconnected ingredient inputs are considered parameters to be supplied by the user at run-time. For some recipes, run-time parameters are initial values supplied by the user at the start of executing a recipe, and are not changed thereafter. In many other cases, however, the user would like to change run-time parameters and see the effects on the running recipe. There are several possible definitions of semantics for run-time parameter changes. We have chosen the following semantics:

> If a run-time parameter P of ingredient I is changed at time T_c from V_{old} to V_{new}, all data requested by browsers downstream of I at times $t > T_c$ will be produced *as if* the parameter $P = V_{new}$ since time t_0.

Time t_0 is defined to be the start of the current recipe execution session. We call this *as if* semantics.

[2] A is "downstream" from B if A directly takes output from B or takes output from a box that is downstream from B. Similarly, B is "upstream" from A if A is downstream from B.

Here is an example of what our semantics imply. Consider a recipe which consists of a hand which produces a stream of frames from a black and white movie, a colorizing ingredient that can perform computer-generated colorization of frames, and a browser which has a video-recorder-like interface, namely, it can move forwards and backwards through the movie. The colorizing ingredient has a boolean run-time parameter which indicates whether colorization should occur or not. In this example, the browser is (A) behavior since it can move to any arbitrary location forwards and backwards as well as request streams of frames. The application-specific coordinate space in this case is one-dimensional, with the dimension being frame number. Suppose that the run-time parameter is initially set to false. The movie being displayed by the VCR-browser will be in monochrome. Our run-time semantics dictate that if the user should change the run-time parameter from false to true, the VCR browser will display a colorized movie. Note two important implications here. First, our semantics dictate that if the VCR was in play mode, i.e. issuing requests to fetch the next frame, the movie would continue seamlessly from the same frame, but now in color. Second, if the user should fetch arbitrary previous frames, i.e. rewind the VCR interface, those movie frames would also be colorized. Our semantics dictate that changing a parameter gives the effect of that parameter having always been assigned the new value since the start of recipe execution.

To handle the actual process of entering run-time parameters, we introduce a new type of Tioga box called a **parameter generator**. Whenever a parameter generator is connected to an input of an ingredient, that input is considered a run-time parameter. Any ingredient inputs which are left unconnected by the user are automatically assigned parameter generators by the recipe editor. A parameter generator is neither a normal ingredient nor a regular hand. Because the user can change a parameter at any time, parameter generators run asynchronously, interacting with the user as necessary. Generators are not ingredients because ingredients run synchronously. Parameter generators are also not hands for two reasons. First, the last run-time parameter entered to an ingredient is always available to the ingredient. That is, whenever the ingredient needs to run again, the last parameter value entered by the user is always used. This differs from hands which cause downstream ingredients to block while waiting for input. Second, the semantics of run-time parameter changes differ dramatically from data produced from a hand. The recipe execution system must detect and handle each change of a run-time parameter appropriately so as to ensure the semantics we described above. Finally, making parameter generators explicit instead of simply leaving ingredient inputs dangling gives us an additional benefit. By explicitly representing parameter generators as boxes, we enable the users to share parameter values. If a same parameter generator is connected to inputs of different ingredients, any change of parameter is made simultaneously to all affected ingredients.

3.5. Ingredients and side-effects

In Tioga, ingredients are classified as either **functional** or **non-functional.** A functional ingredient is one which always produces the same output when given the same inputs and does not have any side-effects. In contrast, a non-functional ingredient is one which may produce different outputs when given the same inputs. In addition, a non-functional ingredient is free to produce any kind of side-effect. Since ingredients can be implemented in C and may be arbitrarily complex, the recipe system has no way in general of automatically detecting whether an ingredient is functional or not. The user must specify this to the system at ingredient definition time. In a recipe, actual updates to the database are performed through non-functional ingredients with side-effects of writing to database tables. Retrievals and computations, on the other hand, can be carried out with functional ingredients.

We distinguish between functional and non-functional ingredients in order to execute recipes more efficiently. Functional ingredients can take maximal advantage of function caches. Any given input can be looked up in the cache. The function only needs to be run if the result for that input is not found in the cache. Non-functional ingredients, on the other hand, not only cannot be cached, but may also force other ingredients to be re-executed as well.

4. RECIPE EXECUTION

4.1. Introduction

Our approach to recipe execution is to divide a recipe into several pieces called **structures.** Structures help to identify pieces of the recipes that may profit from demand-driven execution and pieces that must be executed in a data-driven style. We also explore how the behavior specifications of Tioga ingredients and program modules are used in determining buffering policies. In addition, we show how the presence of parameter generators affects recipe execution. Finally, we examine some general optimization techniques.

4.2. Recipe structures

Recipes can be divided into smaller units called **structures** with the following marking algorithm.

[STEP 1]

Begin at a hand and mark all ingredients downstream from the hand with the identity of that hand. Stop marking a particular branch of the recipe when a browser is reached on that branch. In the course of marking ingredients, some ingredients may already have markings. This occurs if an ingredient has more than one hand upstream from it. In that case, the multiple hands that are upstream from that ingredient are grouped into the same structure.

After step 1, some ingredients in the recipe will have markings of identities of hands and some ingredients will remain unmarked. Ingredients which are marked with the name of a hand are precisely those ingredients which are affected by any data produced by that hand. Ingredient which are unmarked are those which are not downstream of any hands. If there are ingredients that are downstream of multiple hands, then the marks of those hands become equivalent.

[STEP 2]

We begin at each browser and mark all the ingredients upstream from the browser with the identity of the browser, stopping if any ingredient along the branch already has a hand marking. If an ingredient already has a browser marking, that means that ingredient has another browser downstream. We nonetheless mark that ingredient again. In contrast to step 1, we do not make the browser marks equivalent, and do not group the browsers into one structure.

Step 2 of the algorithm identifies ingredients which are upstream from a browser but are not downstream from a hand.

[STEP 3]

At this point, there may still be some ingredients which are unmarked. These "dangling" portions of the recipe are handled as follows. Any unmarked ingredient which has no inputs is assigned a virtual hand. This is necessary in order to force execution of that ingredient and any unmarked ingredients downstream from it. The virtual hand is designated as having stream (S) behavior. Similarly, any unmarked ingredient which has no output is assigned an virtual browser which makes virtual requests of that ingredient. The virtual browser is considered to request records sequentially, i.e. (S) behavior. Given the inclusion of virtual hands and virtual browsers, we retain any existing marks and repeat steps 1 and 2 for the newly assigned virtual hands and virtual browsers. After those two steps are repeated, no unmarked ingredients remain.

We have now divided the recipe into separate structures. There are two kinds of structures: **hand structures** and **browser structures**. A hand structure is the collection of all hands, ingredients, and browsers with the same marking. A browser structure contains a browser and all ingredients that are marked with the identity of that browser. Since a program can be a hand on output and a browser on input, these two kinds of structures can coexist in the same recipe, connected by a browser/hand. Since an ingredient may have multiple browsers downstream from it, it is also possible for browser structures to overlap. Figure 2 shows examples of recipe structures.

In Figure 2, the original recipe is shown on the left. The marking algorithm would proceed as follows. First, the browser/hand combination is identified as a hand on output. All of its downstream ingredients will be marked with the identity of $Hand_1$. Ingredients E, G, and $Browser_2$ would be so marked. Note that Ingredient F is not marked at this point because it is not downstream of $Hand_1$. Since there are no more hands in the recipe, we proceed to step 2 of our algorithm. We mark backwards from the browsers. From $Browser_1$, we mark ingredients C, A, and B. Ingredient D is not marked at this point because it is not upstream from $Browser_1$. From $Browser_2$ we do no markings because it is already marked with a hand marking. We are now at step 3 of our algorithm. We identify the unmarked ingredients as D and F. F is assigned a virtual hand on input and D is assigned a virtual browser on output. We repeat step 1. The virtual hand marks F, then G, but since G is already marked with the identity of $Hand_1$, the virtual hand and $Hand_1$ markings are equivalent. Ingredients $Hand_1$, E, F, G, and $Browser_2$ now belong to the same hand structure. We repeat step 2 and mark ingredients D and B with the identity of the virtual browser. At this point, no unmarked ingredients remain. The recipe has been divided into structures, as shown in the right side of Figure 2. This example recipe consists of one hand structure and two browser structures. Browser structure 1 and the hand structure overlap in the browser/hand combination. Browser structures 1 and 2 overlap in ingredient B.

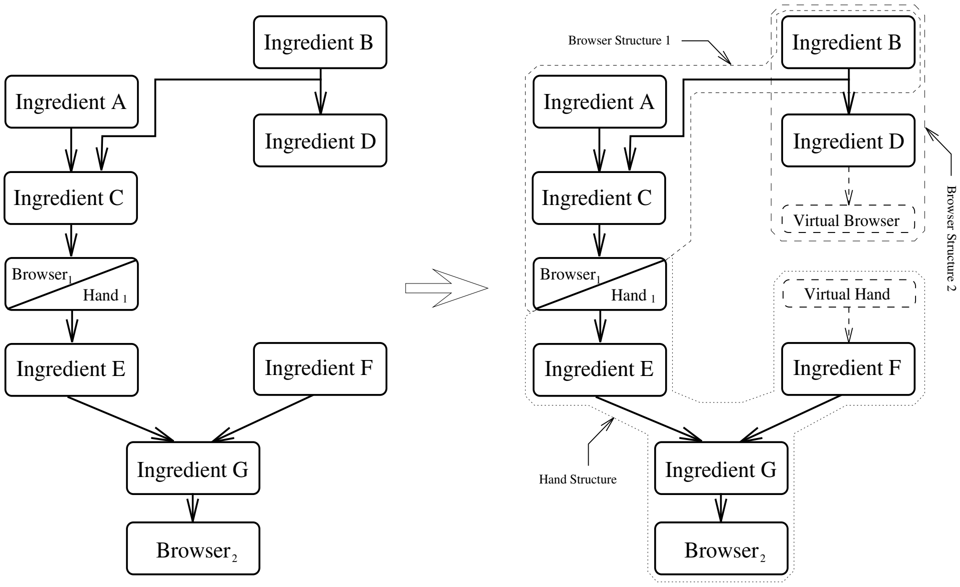

Figure 2: Examples of recipe structures

4.3. Executing hand structures

To execute hand structures, we adopt an eager evaluation strategy. This means that as data is produced by a hand, the data is "pushed" through the recipe and the downstream ingredients execute as much as possible. To accomplish this, the Tioga run-time system provides the following:

- A speed-matching buffer is added on the output of each hand in the recipe. This output buffer holds data from the hand until downstream ingredients can consume it. Whenever the output buffer of a hand is non-empty, the recipe execution engine attempts to execute the ingredient downstream from the hand with the next data record in the buffer. Because the hand is asynchronous, it may produce data at a higher rate than that which can be processed by downstream ingredients. Speed-matching buffers can be kept in virtual memory or managed in the database as persistent queues[BERN90].

- Input buffers are allocated for any ingredient with multiple inputs if that ingredient is downstream from multiple hands. Since hands are asynchronous, all the inputs for the ingredient may not be ready at the same time. The ingredient cannot execute until all inputs have arrived. Since the upstream hands are independent, inputs may be accumulating at varying rates right before the ingredient in question. Input buffers are allocated to handle this. These input buffers before ingredients are in addition to speed-matching buffers allocated for the outputs of hands.

- Input buffers are allocated for browsers. Hand structures typically end with an ingredient connected to a browser. In this case, the output of the last ingredient must be buffered in the database so that the browser protocol can be run efficiently.

Being careful about when to buffer and how much data to buffer is critical for many scientific applications because the data sets involved are often very large. The buffering requirements for each browser in a hand structure are determined by the following three different dimensions:

1) How much data must be buffered before the browser request can be satisfied?

2) When can data be evicted from the buffer?

3) What data structures are needed to support efficient access to the buffer?

The amount of data that must be buffered before a browser request can be satisfied is dependent on the input behavior of the browser and the output behavior of the immediately preceding ingredient. Recall that browsers and ingredients can have behaviors of (S)tream, (C)onstrained or (A)rbitrary. Table 1 shows the matrix of ingredient output behaviors versus browser input behaviors and the implications for how much data to buffer before filling an input request.

In Table 1, **none** indicates that no data needs to be buffered before a browser request can be satisfied. This results from the (S)tream behavior of the browser. Because the browser is only requesting the next record, any new data record available can immediately satisfy the request. Note that a function with (S) behavior cannot be profitably connected to a browser exhibiting (C) or (A) behavior. The asterisks in the table denote meaningless possibilities. In those cases, browser requests can never be satisfied because the output from the ingredient contains no locations.

Table entries with **some** and **all** represent cases where either some amount of data or all data records must be buffered before a browser request can be satisfied. When the browser behavior is (C) or (A), it is requesting data in multi-dimensional regions. If the ingredient has (C) behavior, then it periodically specifies a constraint region in which all data records have already been produced. If the region requested by the browser is contained in the constraint region of the ingredient, then no additional buffering is needed. The browser's request can be filled immediately since all data in the region has been generated and saved in the buffer. However, if the region requested by the browser is not contained in the constraint region of the ingredient, then the input request cannot be satisfied immediately. The ingredient may still be producing data in the region of request. Immediately satisfying the browser request would result in an incomplete answer. The data produced by the ingredient must be buffered until a larger constraint region is specified by the ingredient. That is why some data must be buffered when the ingredient behavior is (C). "Some" data is the data produced by the ingredient until a large enough constraint region is specified by the ingredient. When the ingredient behavior is (A), no constraint region is ever specified by the ingredient. Therefore, the browser can never be sure it has the complete data set for the given input request unless the ingredient has run to completion. In that case, all data must be buffered before any browser requests can be satisfied.

Note that regardless of the None, Some, and All classification all data produced by an ingredient is initially inserted into the buffer. This is because the browser is asynchronous and may request data at a different rate than the ingredient is producing it. Data is always buffered initially so no data is ever lost before being used. The buffering requirements serve to determine how much data must be available in the buffer before a browser request can be satisfied. After a browser request is satisfied, some data records may be evicted from the buffer. The eviction policy for buffers is discussed below.

Table 2 indicates when data can be evicted from the buffer preceding a browser. If the browser has (S) behavior, records can be evicted immediately after they are read. Since the browser can never request previously seen records, there is no need to keep the records in the buffer after they are read. If the browser has (C) behavior, then some data records may be evicted from the buffer once a constraint is specified by the browser. Recall that the constraint specifies the region from which the browser will no longer request data. Therefore, we can safely evict from the buffer any records with locations in the constraint region. Lastly, in the case of the browser with (A) behavior, we may never safely evict records from the buffer because the browser may at a later point request the same data again.

The buffer before a browser can be simply implemented as a data base table. If the browser is of (S) behavior, a simple queue suffices. In the cases where the browser can request data by regions in N-space, we can build indices on the table using spatial access methods like R-trees to facilitate efficient access. R-trees can also be useful in selecting records to evict since eviction is based on containment in N-dimensional constraint regions.

		Browser behavior		
		S	C	A
Ingredient behavior	S	none	*	*
	C	none	some	some
	A	none	all	all

Table 1: Amount of data that must be buffered before a browser request can be met

		Browser behavior		
		S	C	A
Ingredient behavior	S	on read	*	*
	C	on read	after constraint	never
	A	on read	after constraint	never

Table 2: Policy for evicting data from the buffer before a browser

4.4. Executing browser structures

Browser structures contain ingredients and browsers, but no hands. Therefore, the execution of browser structures is driven by asynchronous browsers. As a result, there are two possible processing strategies for browser structure. Browser structures can always be executed using eager evaluation just like hand structures. If a browser structure contains only functional ingredients, a lazy evaluation strategy may be employed.

To perform eager evaluation on a browser structure the Tioga run-time system executes ingredients in downstream order. Each ingredient is run to completion and the results of the ingredient immediately preceding a browser are buffered. Operationally, eager evaluation for browser structures is similar to eager evaluation for hand structures. The only difference is that in hand structures there may be ingredients with multiple hands upstream that cause additional buffering needs. In browser structures, there are no such asynchronous data sources.

If a browser structure contains only functional ingredients, then we may use the alternate strategy of lazy evaluation. Lazy evaluation works by combining all the ingredients in a browser structure into a single **mega-ingredient** by performing query modification. Any two connected ingredients can be combined as follows. Two POSTQUEL ingredients can be combined into a single one by assuming that the second query is a command defined on the view that is materialized by the first ingredient. Standard query modification, as discussed in [STON75], correctly combines the ingredients. If the second ingredient is a C function, then the ingredients can be combined by adding the C function to the target list of the POSTQUEL ingredient. If the first ingredient is also written in C, then combine the two functions into a single ingredient by cascading them as in function composition. Ingredients can be combined in the above ways until there remains only a single ingredient connected to the browser.

Now, if the browser exhibits (C) or (A) behavior, it is requesting data records by location. The rewritten mega-ingredient is also producing objects with locations so we can modify the query to add the requested subset as an additional qualification. The results from the query can be inserted into the buffer before the browser. As is the case for hand structures, we can build spatial indices on the buffer to fetch the result efficiently. In general, any browser command fetches data that is partly a lookup from the buffer and partly computed by an appropriately modified query. In the event the browser requests data with (S) behavior, the mega-ingredient is simply executed with a cursor interface. It can be run incrementally, producing objects as the browser requires them. In this case, there is no need to buffer the results at all since the results can be discarded after use.

We choose to permit lazy evaluation only for browser structures with functional ingredients to avoid ambiguous semantics. In the case where a browser structure contains non-functional ingredients, the entire structure becomes non-functional. Since in lazy evaluation, the mega-ingredient is executed on a demand-driven basis, different request patterns may result in different answers given. We disallow this case so as to avoid scenarios where different processing strategies produce different outcomes.

Since ingredients may have multiple browsers downstream, it is possible for browser structures to overlap. In those circumstances, query modification of the ingredients is less beneficial since the mega-ingredients for each structure would include functions corresponding to the repeated ingredients.

4.5. Executing structures with run-time parameters

If a recipe structure contains parameter generators, i.e. contains ingredients with run-time parameters, then recipe execution needs to be modified slightly from our discussion in the previous subsections. Given that a run-time parameter P of ingredient I has changed at time T_c, our semantics dictate that browsers downstream from I must see results *as if* the parameter P has always had the newly changed value. We consider the separate cases of structures being executed with eager evaluation and structures being executed with lazy evaluation.

To achieve *as if* semantics for eagerly evaluated structures, we need to "replay" the structure with the new value of the parameter P. If all the ingredients upstream of I are functional, then we can avoid actual re-execution of them by buffering all the inputs to I. The buffered inputs are sufficient because functional ingredients upstream of I would produce the same result if executed again with the same inputs. Ingredients downstream of I would need to be re-executed since I itself may be producing different values. If the ingredients upstream of I are non-functional, we cannot simply use the buffered results since a second execution of a non-functional ingredient may produce a different result. Instead, we must actually re-execute those ingredients. To accurately achieve the *as if* semantics, we must not only re-execute those ingredients but re-execute them with respect to time t_0. Fortunately, POSTGRES already supports exactly this functionality because it can handle **time travel** queries with its no-overwrite storage manager[STON91]. With POSTGRES's time travel capability, we are able to execute queries with respect to a time in the past. Therefore, on run-time parameter changes, we simply re-execute those ingredients as historical queries.

For browser structures that are executed using lazy evaluation, run-time parameter changes are easy to handle. No replay of ingredients is necessary since subsequent browser requests can be translated into evaluations of the rewritten mega-ingredient with new parameter values set. We simply need to invalidate the data buffered right before the browser.

4.6. General optimizations

In general, we can optimize ingredient execution with the use of **function caches**. A function cache stores the output of a registered POSTGRES function so that if the function is invoked with the same inputs, the result can be looked up instead of re-computed. Every functional ingredient in Tioga is a candidate for a function cache. There is no benefit from caching the outputs of non-functional ingredients.

In [STON93], we discussed techniques for **coalescing** any two ingredients into a single ingredient. This tactic results in a query plan for the coalesced ingredient that is more efficient than the combination of the individual plans for the two ingredients. However, the cost of coalescing is that the opportunity to buffer between the two ingredients disappears. If a run time parameter to the second ingredient changes, then the entire combined ingredient must be recomputed, whereas without coalescing, only the second ingredient needs recomputation. In [STON93], there is a lengthy discussion of the optimization problem surrounding coalescing and buffering. The extended Tioga model discussed here does not compromise this analysis. When an ingredient is downstream from multiple hands, then there is a substantial efficiency tax to coalescing the ingredient along both upstream links. As such, coalescing should not occur for marked ingredients that have multiple hands upstream.

Additional efficiency can be obtained by moving recipe evalution directly inside the DBMS. The query execution engine would need to be extended to directly handle a mega-plan of query nodes. It would also have to be able to manage invocation of and communication with outside processes as it executes Tioga program modules.

5. TRANSACTIONS IN TIOGA

5.1. Introduction

Traditional transaction semantics are not appropriate for Tioga because recipes differ from traditional database applications in two ways. First, recipes are directed graphs. Traditional transactions deal exclusively with linear sequences of database commands. Those commands are executed serially, and each command can see the effects of the previous commands. In a graph of ingredients, however, concurrent executions are possible, subject to the ordering constraints of the directed edges. Visibility semantics for a graph also differ from the linear case, and is discussed below.

A second difference between recipes and traditional transactions is that recipes may take a long time to run. As a result, a recipe may be more susceptible to failures in its lifetime. Under traditional transaction semantics, failure atomicity requires that the entire transaction be rolled back. That would not be desirable for recipe execution. Also, since a recipe is long-running, it may access many tables in the course of execution.

We propose a new transaction model to accommodate Tioga's requirements. The goal of this model is to exploit the concurrency inherent in recipe graphs while supporting long-running transactions. To achieve this, we relax failure atomicity by dividing recipe execution into a sequence of subtransactions. We introduce techniques to control concurrency both within a subtransaction and between subtransactions.

5.2. Tioga subtransactions

A Tioga subtransaction is the single execution of every ingredient in a recipe. A subtransaction consists of the ingredient executions caused by a group of inputs, one from each hand or virtual hand. Because recipes are directed graphs, concurrent executions of ingredients can occur within a subtransaction. In other words, Tioga subtransactions generalize the concept of transactions from linear sequences of commands to two-dimensional graphs of commands.

The execution of a Tioga recipe is a sequence of subtransactions, each of which is atomic, consistent, and durable. Subtransactions are ordered based on the time of their inputs. Subtransactions are not completely isolated from each other; within a recipe execution, subtransactions that commit must commit in order. A subtransaction may commit if a sequentially earlier subtransaction aborts, but a subtransaction may not commit while a sequentially earlier subtransaction is processing and has not yet committed. These semantics allow a long-running recipe to proceed even if one subtransaction fails. Figure 3 shows the sequence of subtransactions for the execution of a five-box recipe. The execution of the recipe results in an ordered sequence of recipe subtransactions shown in the numbered dotted rectangles.

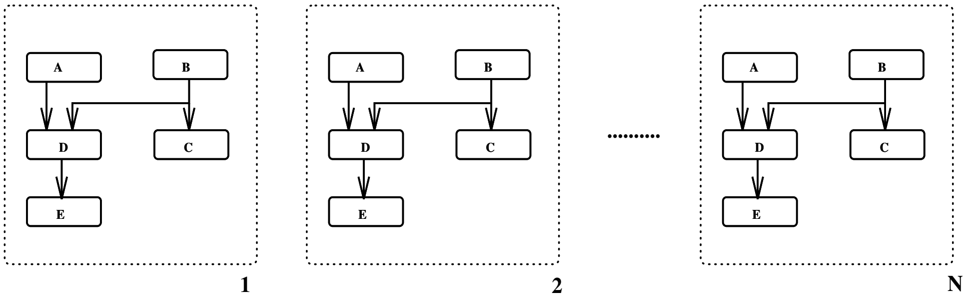

Figure 3: A sequence of recipe subtransactions

Since the edges of the subtransaction graph correspond directly to dataflow edges in the recipe, they represent dependencies between outputs and inputs of connected ingredients. Because no ingredient can execute before its inputs arrive, the edges of the subtransaction graph impose an ordering on the execution of ingredients within a subtransaction. An ingredient cannot execute before an upstream ingredient does.

Visibility for ingredients within a subtransaction is based on topological ordering. An ingredient I_1 in a subtransaction can see the effects of ingredient I_2 in the same subtransaction if there is a path from I_2 to I_1 in the subtransaction directed graph. In other words, an ingredient can see the effects of upstream ingredients in the same subtransaction. For the recipe in Figure 3, in a given subtransaction, commands in ingredient D can see the effects of commands in ingredients A and B, but ingredient B cannot see the effects of ingredient A. Since ingredients A and B do not see each other's changes, we require that A and B see the same database state regardless of how A and B are interleaved in execution.

5.3. Concurrency control for recipe subtransactions

There are two concurrency control issues for executing a recipe. The first is controlling concurrency within a subtransaction. The second is controlling concurrency between subtransactions.

To control concurrency within a subtransaction, we order the execution of ingredients within a subtransaction graph. The edges in the subtransaction graph corresponding to dataflow edges in the recipe already impose some ordering constraints. In general, these edges do not define a total ordering, so concurrent execution of ingredients is possible within a subtransaction. When accessing shared data, concurrently executing ingredients may conflict. To detect conflicts, we require ingredients to **predeclare** their read and write sets[BERN80]. Using these predeclarations, we resolve conflicts by adding ordering edges to our subtransaction graph. Ordering edges are added between ingredients with conflicts unless they are already constrained in their execution order. Referring to Figure 3, ordering edges would not be added between ingredients A and E because, within a subtransaction, ingredient A is already constrained to execute prior to ingredient E. On the other hand, an ordering edge could be added between A and C because there is no ordering constraint between them to start with. Once the ordering edges are added, any topological ordering of the subtransaction graph is a correct interleaving of ingredient executions.

To control concurrency between subtransactions, we use dynamic locking. Dynamic locking enables subtransactions to be interleaved with other traditional transactions as well as subtransactions from other recipes. Our semantics require that subtransactions commit in order. The conservative approach to satisfying this requirement is to execute subtransactions serially. A subtransaction does not begin until the previous subtransaction completes. We propose an aggressive approach based on starting a subtransaction before its previous subtransaction completes.

By starting subtransactions early, we gain concurrency but face two potential hazards. The first is that the later subtransaction may complete before the earlier one, violating our semantics. To prevent this, our recipe executor blocks a subtransaction from committing until its previous subtransaction completes. The other hazard is that the execution of later subtransaction may pass the execution of the previous subtransaction so as to cause an inconsistent retrieval or a lost update. To avoid this, we define ordering edges between subtransactions using predeclarations. We add ordering edges between ingredients that have read/write or write/write conflicts. We resolve conflicts in favor of the sequentially earlier subtransaction. By adding ordering edges between subtransactions, we create one large graph of ingredients for the entire recipe execution. Because we only permit edges from sequentially earlier subtransactions to sequentially later subtransactions, the overall graph, like the subtransaction graphs, is acyclic. Any topological ordering of this larger graph is now a correct execution interleaving. To run multiple subtransactions of a recipe concurrently, locks held by the earlier subtransactions are inherited by later subtransactions as they start.

Figure 4 shows an example of ordering edges between subtransactions. The dashed edges represent ordering edges. In this case, ingredients A and C have conflicts. Within a subtransaction, the ordering edge goes from A to C, meaning C must wait for A to complete. Between subtransactions, we always resolve the conflicts in favor of the earlier subtransaction. Thus, an ordering edge is directed from C of subtransaction i to A of subtransaction $i+1$. In the example in Figure 4, a possible interleaving order is one which executes ingredient A of subtransaction $i+1$ after ingredient C of subtransaction i even before subtransaction i has fully completed execution.

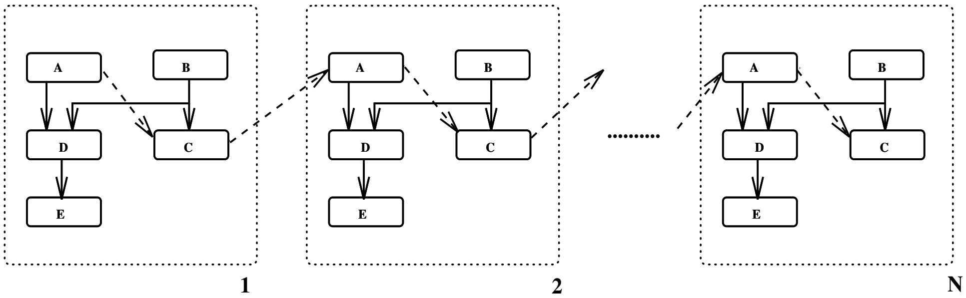

Figure 4: Inter-subtransaction concurrency control

6. CONCLUSIONS

In this paper, we showed how the a graphical query system can be extended into a complete database application building environment. We added the ability to interface with external programs in Tioga recipes. By doing so, many types of data entry applications can be expressed as recipes. We also characterized the behavior of programs as (S)tream-oriented, (C)onstrained, or (A)rbitrary, and made use of that characterization for optimizing recipe execution. We clearly defined the semantics of run-time parameter changes and show how they affect recipe execution strategies.

In order to optimize recipe execution, we presented a marking algorithm to divide a recipe into hand and browser structures. We showed how structures, run-time parameters, and behaviors affect execution and buffering strategies. We explained how both eager and lazy evaluation strategies can be carried out.

Finally, we established clear transaction semantics for recipe execution. A recipe execution is a sequence of subtransactions, each of which is a directed graph of database commands. We discussed techniques for controlling concurrency both within a subtransaction and between subtransactions.

At the current time, we have developed a prototype Tioga system that can execute recipes containing hands, browsers, and ingredients. We have constructed a number of recipes that demonstrate actual application needs of our users. Currently, our recipe execution engine uses an eager evaluation strategy and conservative concurrency control for recipe execution. For our second prototype, we intend to move to internal DBMS evaluation of recipes rather than an external supervisor.

7. ACKNOWLEDGEMENTS

We would like to thank Jeff Sidell, Alan Su and Allison Woodruff for their contributions to this work in both discussion and implementation.

REFERENCES

[BERN80] Bernstein, P.A., Shipman, D.W, and Rothnie, J.B., "Concurrency Control in a System for Distributed Databases (SDD-1)," ACM-TODS, March 1980.

[BERN89] Bernstein, P.A., et al., "Future Directions in DBMS research," SIGMOD Record, March 1989.

[BERN90] Bernstein, P.A., et al., "Implementing Recoverable Requests Using Queues," Proc. 1990 ACM-SIGMOD Conference, Atlantic City, NJ, May 1990.

[DOZI92] Dozier, J., "How Sequoia 2000 Addresses Issues in Data and Information Systems for Global Change," Sequoia 2000 Technical Report 92/14, University of California, Berkeley, August 1992.

[LUCA92] Lucas, B. et al., "An Architecture for a Scientific Visualization System," Proc. 1992 IEEE Visualization Conference, Boston, MA, October 1992.

[RASU92] Rasure, J. and Young, M., "An Open Environment for Image Processing Software Development," Proceedings of 1992 SPIE Symposium on Electronic Image Processing, February 1992.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference, San Jose, CA, May 1975.

[STON91] Stonebraker, M. and Kemnitz, G., "The POSTGRES Next-Generation Database Management System," Communications of the ACM, October 1991.

[STON92] Stonebraker, M. and Dozier, J., "SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research," SEQUOIA 2000 Technical Report 91/1, Electronics Research Lab, University of California, Berkeley, March 1992.

[STON93] Stonebraker, M., et al., "Tioga: Providing Data Management for Scientific Visualization Applications," Proc. 1993 VLDB Conference, Dublin, Ireland, August 1993.

[STON93a] Stonebraker, M., et al., "The Sequoia 2000 Architecture and Implementation Strategy," Sequoia 2000 Technical Report 93/23, University of California, Berkeley, April 1993.

[STON93b] Stonebraker, M., et al., "DBMS Research at a Crossroads: The Vienna Update," Proc. 19th VLDB Conference, Dublin, Ireland, August 1993.

[UPSO89] Upson, C., et al., "The Application Visualization System," IEEE Computer Graphics and Applications, July 1989.