The Tioga-2 Database Visualization Environment

Alexander Aiken*, Jolly Chen, Mark Lin, Mybrid Spalding,
Michael Stonebraker and Allison Woodruff

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
email: tioga@postgres.berkeley.edu

Abstract. This paper reports on user experience with Tioga, a DBMS-centric visualization tool developed at Berkeley. Based on this experience, we have designed Tioga-2 as a direct manipulation system that is more powerful and much easier to program. We present a detailed design of the revised system together with an extensive example of its application. We also give a progress report on a Tioga-2 implementation.

1 Introduction

Database system performance—as measured by either processing speed or the quantity of data that can be managed—has grown by an order of magnitude in recent years, making increasingly sophisticated applications feasible on ever-larger data sets. However, database query languages have changed relatively little and are difficult for non-experts to use. The vast majority of database users are unable to customize applications to their own needs, let alone develop their own custom applications. Thus, at present the expanding capabilities of database systems can be exploited fully only by expert programmers. Making databases easier to use and program, and thereby more accessible, is an important issue today and will become more important as database technology becomes faster, cheaper, and more powerful [11].

This paper reports on the design of Tioga-2, a new database visualization environment. We use the term "visualization environment" rather than "programming environment" to emphasize that most programming operations in Tioga-2 are performed by manipulating graphical representations of either programs or data. Tioga-2 is based on a small set of primitive operations for transforming data and its visualization. These primitives have been chosen carefully to have clear, simple semantics and to be composable. Thus, Tioga-2 users can build sophisticated applications—or modify existing applications—by successive composition of the primitives. We believe that by providing a small set of general "building blocks", minimum language syntax, and immediate feedback on the effect of incremental program modifications, Tioga-2 makes it much easier for database users to develop database applications.

* This research was sponsored by NSF under grants IRI-9400773 and IRI-9411334.

Tioga-2 has not been designed in a vacuum. Previously, we reported on the design and implementation of Tioga, a visualization system that is coupled closely with the POSTGRES DBMS [12]. The design of Tioga-2 has been influenced heavily by what we learned from user experiences with Tioga and a companion commercial product, Illustra Object-Knowledge, based on the same ideas. In the rest of this introduction, we first discuss the problems and lessons from Tioga and then outline our solution to those problems in Tioga-2.

1.1 Tioga

Tioga adopts the "boxes-and-arrows" programming paradigm popularized by AVS [13], Data Explorer [7], and Khoros [9]. Every box is a user-defined function, which has been registered with POSTGRES. A programmer constructs a Tioga program using a drag and drop editor to move and connect boxes on the screen.

Every Tioga program has a designated *viewer* connected to the output of a specified box. The viewer provides the user with a two-dimensional *canvas* onto which the programmer places renderable objects. In addition, the viewer provides a *pan* feature whereby the user can "fly over" the canvas viewing areas of interest. Furthermore, the user can *zoom* into areas of the canvas to see more detail. Zoom is a powerful construct, as it supports so-called *drill-down*—the ability to change the visual representation of data. For example, a state map of the United States could become a county map upon suitable zooming. In addition, we specified but never implemented the features of multiple viewers, viewers within viewers, cloning of viewers, slaving of viewers, and wormholes [14].

Experience with Tioga and Illustra Object-Knowledge can be summarized as follows:

1. Programmer model
   Tioga is based on the idea that an expert programmer constructs POSTGRES user-defined functions (boxes) and that a second programmer uses an editor to "wire up" visualizations. In this way, Tioga implements a "big programmer / little programmer" environment.
   It has been sufficiently hard to construct boxes-and-arrows programs that the little programmer must, in fact, be a big programmer. The key problem is that simplifying the specification of control logic through a boxes-and-arrows notation does not simplify programming sufficiently. For example, to construct Tioga applications, the little programmer must understand locating objects on a canvas and turning objects into graphical representations. It turns out that even expert programmers find these tasks difficult. As a result, little programmers have not been able to program in Tioga because it is not nearly easy enough to use.
2. Programming environment
   Tioga has the familiar notion of building a program, compiling it, and then running the compiled result. Novices have difficulty learning how to program effectively in this paradigm. For example, if nothing appears on the screen,

then there is a "bug" in the program. Bugs are hard for the programmer to find because Tioga provides a viewer only for the final result; it is not possible to place a viewer on any edge in a diagram to visualize the data that is flowing along that edge.

3. Expressive power
   As a result of trying to provide a simple programming model, Tioga is in some ways oversimplified. To select only a single example, because every box must be a user-defined function, a box has a single output, which must be of a specific type. This makes it difficult to implement functionality of the form:

   if condition **then** deliver data to box i **else** deliver data to box j

1.2 Tioga-2: Guiding Principles

Based on our experiences, we are redesigning Tioga completely from scratch, and the result is Tioga-2, described in this paper. We begin with the principles that have guided the redesign.

Much of the problem with the original Tioga system is that there is no way to specify some aspects of a visualization except via ordinary statement- and expression-oriented programming. Learning to write procedural code is a high hurdle for non-programmers, and some visualization aspects—such as writing functions to position data in a multi–dimensional space—are difficult even for expert programmers.

There is an alternative way to specify data visualizations. Non-programmers intuitively understand how to specify desired computations "by example"—by manipulating sample data. Instead of writing in a standard programming syntax, the programmer begins with very simple displays of data and composes them directly on the screen to construct elaborate visualizations. In moving from the boxes-and-arrows notation of Tioga to the direct manipulation programming paradigm of Tioga-2, we have identified a number of principles we believe to be important to a usable, flexible, and powerful direct manipulation visualization system:

1. Every result of a user action has a valid visual representation.
   All data types constructible by Tioga-2 programs have a well-defined screen representation. As such, the programmer obtains immediate visual feedback on the effect of any change to a Tioga-2 program and can visually inspect intermediate results. This principle facilitates debugging activities and solves problem (2) noted above.
2. Programming is incremental.
   Visualizations are constructed incrementally by successive composition of a small number of simple primitives. Combined with the ability to visualize results of incremental changes immediately, we believe that we can empower the little programmer to construct Tioga-2 programs. In Tioga-2, there is no distinction between constructing a program, modifying an existing program, and using an existing program.

3. To the extent possible, programming is specified visually by direct manipulation of visualized data.
   A boxes-and-arrows representation of the user's program is available and must be used for certain operations. However, considerable programming is done by direct manipulation of the screen without reference to this data structure.
4. Every operation has a clear, well-specified semantics.
   Unlike many previous direct manipulation systems there is no inference procedure to synthesize a program from a user's examples [4]. Instead, every Tioga-2 operation has a straightforward, unambiguous meaning as a step in a program.
5. Retain the "big programmer/little programmer" model.
   We recognize that there are computations that cannot be specified in Tioga-2. For example, while Tioga-2 has the equivalent of an if-then-else construct, it does not have arbitrary recursion. Thus, we expect that big programmers will construct additional Tioga-2 boxes as in the original Tioga system.

The remainder of this paper is organized as follows. We begin in Section 2 with a quick tour of the structure of Tioga-2. This section introduces terminology and notation used throughout the paper. Section 3 presents the user's view of Tioga-2, the user interface. The description of Tioga-2 programming begins in Section 4 with the primitive operations for editing boxes-and-arrows diagrams and performing standard database operations. Section 5 presents primitives for defining visualizations of database relations. Section 6 describes three sets of primitives for defining alternative views of data and connections between related data: (a) *drill down*, in which a user moves from a coarse visualization (e.g., a state map) to a more refined visualization of the same data (e.g., a county map), (b) *wormholes*, in which a user can move from a visualization of one data set to a visualization of a different data set, and (c) *rear view mirrors*, which allow users to keep track of "where they came from" (i.e., wormholes through which they have travelled). Section 7 continues with mechanisms to link multiple visualizations together. Section 8 discusses database updates. A progress report on an implementation of Tioga-2 together is given in Section 9; this section also covers a few additional features we have found either necessary or convenient to add based on early experience with Tioga-2. Finally, Sections 10 and 11 cover future and related work respectively, and Section 12 presents a few conclusions.

2 The Model

Before presenting the Tioga-2 system in detail, we define some basic terminology and concepts. The reader may wish to skim this section on a first reading.

Tioga-2 programs are represented by dataflow graphs with boxes and arrows. A *box* is a primitive procedure with some number of *inputs* and *outputs* (see Figure 1). The output of one box may be connected to the input of another box by an *edge* (also called an *arrow*). Box inputs and outputs are typed and

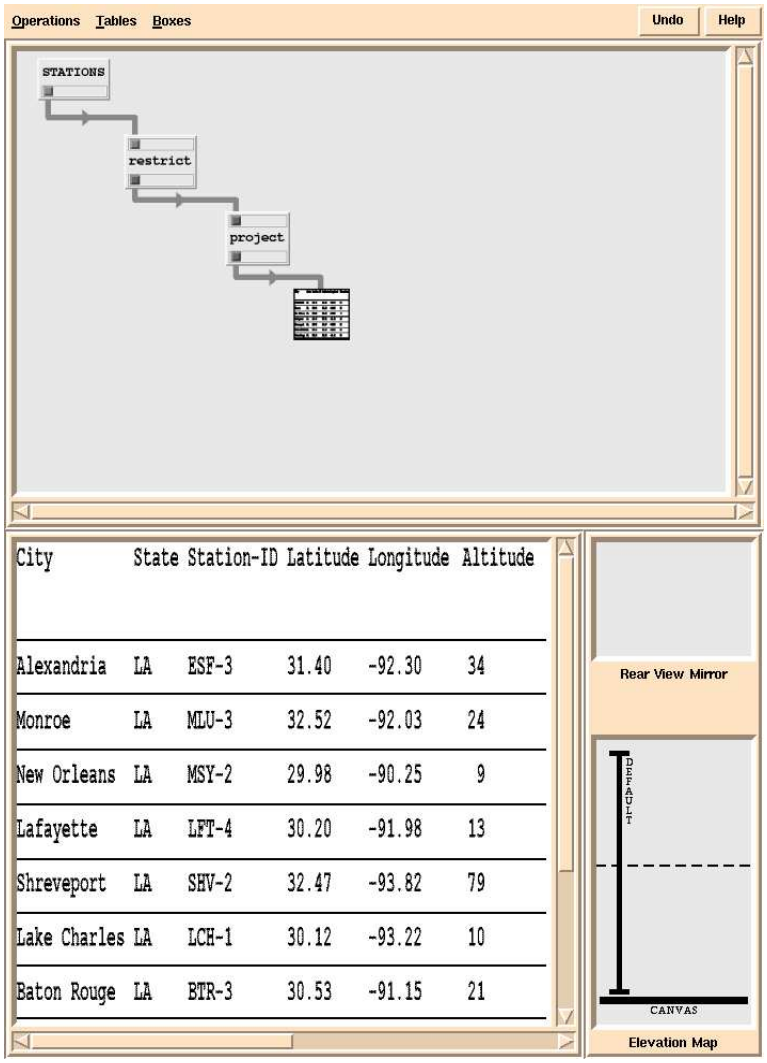

Fig. 1. Weather stations in Louisiana.

edges connect outputs to inputs of compatible types. Any attempt to connect an output to an input of incompatible type is a type error. Tioga-2 programs have dataflow semantics; when data is present on all of a box's inputs, the box can "fire", producing results on one or more outputs. Execution is lazy, evaluating only what is required to produce the demanded visualization.

A box input or output may be a scalar value (e.g., a runtime parameter supplied by the user) or a *displayable*, described below. Displayables define visualizations. Most Tioga-2 boxes compute displayable outputs from one or more

displayable inputs. Tioga-2 has three displayable types: *extended relations*, *composites*, and *groups*.

The first displayable type is an extended database relation R. In Tioga-2, the visualization of R is defined by R's attributes. Intuitively, R "knows" how to display itself. We assume an object-relational DBMS in which a relation has stored attributes as well as methods defining additional attributes. For each tuple t of R, *location attributes* define the position of t on the screen and a *display attribute* defines the screen representation of t. Tioga-2 visualizations are constructed "tuple-wise"—the visualization R is the sum of the visualizations of each tuple of R.

Every visualization has at least the two screen dimensions and a representation for every tuple. Therefore, an extended relation has at least x and y location attributes, corresponding to the two screen dimensions, and an attribute *display*. A relation R may have additional location attributes; the *dimension* of R is the number of R's location attributes. A relation may also have multiple display attributes defining alternative representations of the data. We adopt a uniform notation and write $t.l$ to denote attribute l of tuple t, whether l is a stored or computed attribute. We stress that the location and display attributes used to define visualizations are computed attributes and are not stored in the database.

The second displayable type is a composite of relations $C = \mathit{Composite}(R_1, \ldots, R_n)$. A composite visualization is the overlay of the composite's components—the visualizations are simply superimposed. Thus, composites provide a way to combine visualizations of different relations in the same viewing space.

The third type of displayable is a group of composites $G = \mathit{Group}(C_1, \ldots, C_n)$. A group visualization is the visualization of each of the composites C_i arranged either side-by-side, top-to-bottom, or in a tabular fashion according to the user's specification. Groups allow visualizations of different viewing spaces to be combined. To render $\mathit{Group}(C_1, \ldots, C_n)$, a viewer displays each of the C_i in the specified layout. The viewer has a position for each of the n displayables—the user may independently pan and zoom in each of the grouped visualizations.

In summary, there are three kinds of displayable types, described as follows:

$$
\begin{aligned}
G &= \mathit{Group}(C_1, \ldots, C_n) \\
C &= \mathit{Composite}(R_1, \ldots, R_n) \\
R &= \text{relations with attributes } x, y, \mathit{display}
\end{aligned}
$$

Many Tioga-2 operations presented in subsequent sections are defined only for R or C inputs. To make programming easier, Tioga-2 extends such operations to work on "higher" types. For example, the **Restrict** operation filters a relation; it is a box that takes an R input and produces an R output. Given a group G input to **Restrict**, Tioga-2 asks the user for the composite within the group, and the relation within that composite, to which the **Restrict** applies. After applying the **Restrict** to the selected relation, Tioga-2 reassembles the composite and the group in the obvious way. This is all done graphically with point-and-click operations, so that the user need not be aware explicitly of how **Restrict** is overloaded to work on group and composite displayables.

Displayable types are translated into screen output by *viewer* boxes. If an n-dimensional relation R is the input to a viewer, then the viewer has an $n+1$-dimensional *position* specifying the location of the viewer for each of the n dimensions and the *elevation*. The user controls the position by panning in the n viewing dimensions and by zooming, which changes the elevation, moving the user "closer to" or "further from" the data. A viewer displays the x and y dimensions of R on the 2-D canvas; the remaining $n-2$ dimensions are available as sliders. If R has location attributes $x, y, l_1, \ldots, l_{n-2}$ each tuple t of R is rendered by drawing $t.display$ at position $\langle t.x, t.y, t.l_1, \ldots, t.l_{n-2} \rangle$ in n-space. Because a visualization space may be larger than the canvas, the viewer filters tuples to the ranges specified by the sliders for dimensions $l_1, \ldots, l_{n-2}$, filters tuples to the visible area on the screen for dimensions x and y, and then renders the tuples' *display* attribute to the screen.

3 User Interface

The Tioga-2 user interface contains several main windows. All may be visible on the screen or iconified. There is a single user interface both for building and for using programs, but a user browsing a previously constructed visualization will not require all of the windows available. A screen dump of the interface is shown in Figure 1. The user interface windows are: a *program* window, containing a boxes-and-arrows representation of a Tioga-2 program, a *canvas* window for each viewer in the current program, and a *menu bar* containing the pull-down menus to invoke primitive operations.

A canvas window shows data visible in a viewer at the current position. In addition, each canvas window includes: a *rear view mirror*, zero or more *slider* bars, an *elevation map*, and an *elevation control* (a dashed line through the elevation map).

The menu bar includes menus of all operations, tables, and boxes available, an *undo* button to undo the last operation performed, and a *help* button.

A Tioga-2 program is constructed incrementally by applying program editing operations to the program window (thereby modifying the boxes-and-arrows diagram) and rendering and/or drill down operations to a canvas window (thereby making modifications via direct manipulation). At any stage in the construction of a program the current result is displayed on all non-iconified canvases.

Since a canvas may be much larger than the available screen real estate, we allow the user to change the viewer's position, altering the area visible in the viewer. Scroll bars control panning in the screen dimensions x and y; canvas slider bars control panning in any remaining dimensions. The elevation control allows the user to drill down into data displayed on the screen. Elevation maps are an interface for programming drill down (Section 6).

4 Program and Data Management Operations

This section discusses the operations available in the program window and Tioga-2's database operations. These operations allow the incremental construction of a boxes-and-arrows program specifying data the user wishes to visualize. Operations for constructing visualizations themselves are discussed beginning in Section 5.

We use the following example to illustrate Tioga-2 programming. An agricultural specialist wishes to construct a visualization of temperature and precipitation data for various sites in Louisiana. The data is stored in two relations: *Stations*, which contains a tuple describing each weather station, and *Observations*, which contains all observations (e.g., date, time, conditions) from all stations. The data covers all of North America and contains a great deal of information besides temperature and precipitation.

As a first step toward constructing a temperature and precipitation visualization for Louisiana, the user limits the *Stations* relation to the stations of interest. For every relation known to the Tioga-2 system there is a box of the same name that takes no inputs and produces as output the tuples of the relation. Beginning with the *Stations* box, the user incrementally adds boxes to perform standard database operations such as restricting the data to tuples satisfying a predicate (e.g., stations in Louisiana) and projecting out unneeded fields (e.g., date of construction). Figure 1 shows a boxes-and-arrows diagram and canvas. The last box in Figure 1 is a viewer, which in this case displays data using a default two-dimensional table format. The user can also inspect any of the partial results. If the user discovers that any step produces unexpected results, he can inspect, delete, and replace boxes as necessary to fix the program.

For convenience, the operations in this section are subdivided into operations that manipulate program structure and database operations.

4.1 Program Operations

This group of primitives permits the initialization, loading, and saving of programs, as well as the deletion, insertion, and connection of boxes into an existing program. There are also primitives that provide familiar language abstractions analogous to procedures and macros. The operations are listed in Figure 2; we briefly discuss the most interesting.

If the user clicks on one or more edges in the current program, **Apply Box** gives the user a menu of all boxes whose inputs match the types of the selected edges. This is a shorthand way to identify those boxes in the database that could possibly take the indicated edges as input.

A design principle of Tioga-2 is that every operation preserves a visual representation. The thesis is that users are most likely to understand their programs and recognize errors if the results of every small, incremental change can be visualized and inspected. Deleting boxes from a program is dangerous, because inputs of other boxes may be left dangling and, therefore, their results unavailable for visualization. To preserve the property that "everything is always visualizable",

arbitrary box deletions are not allowed in Tioga-2. A box may deleted if it has no outputs connected to other boxes (in which case no box inputs are left dangling), or if it has a single input and output of the same type (in which case the system connects the deleted box's predecessor to its successor). A box may also be **Replaced** by another box with compatible types.

A **T** box simply passes its input unchanged to both outputs, and allows another box, for example a viewer, to be connected to the **T**.

Encapsulate permits the user to define new boxes. The user specifies a portion of the program to be encapsulated by drawing a closed curve around a region of the program. Edges cut by the curve are the inputs and outputs of the new box. The new box may be used like any other primitive box.

Encapsulated boxes may also be parameterized to create something akin to a macro or (more accurately) a higher-order function. The user draws additional closed areas within the program region to be encapsulated. These areas become "holes"—they are not included in the encapsulated box, and edges cut by a hole are unconnected. To use an encapsulated box with holes, the user must specify a box—with compatible types—that can be plugged into each hole.

Operation	*Effect*
New Program	Erase the program canvas.
Add Program	Add a named program to the program canvas.
Load Program	Shorthand for **New Program** followed by **Add Program**.
Save Program	Save the current program in the database.
Apply Box	*see discussion*
Delete Box	*see discussion*
Replace Box	Replace one box by a different box with compatible types.
T	Add a T-node to a designated edge.
Encapsulate	*see discussion*

Fig. 2. Operations that manipulate the boxes-and-arrows diagram.

4.2 Database Operations

The primitives in this group provide database operations, which are listed in Figure 3. Each operation adds a new box to the program. The type of the introduced box is indicated in Figure 3. Note that all input/output types are R. As discussed in Section 2, these operations are extended to apply to composite (C) and group (G) types as well.

As mentioned above, the **Add Table** operation adds a new "source" box to the current program. The box is named for a table in the database and has a single output edge. The parameters of many Tioga-2 operations can be specified in several ways; usually there is at least one textual and one graphical method. For example, the user may specify the table to add to the program by either

typing the name or selecting it from a menu of available relations. Note that **Add Table** is a special case of **Apply Box** with zero inputs.

A **Restrict** box filters its input, retaining only tuples that satisfy a restriction predicate. The user is prompted for the predicate to be applied. A **Sample** box produces a random subset of an input relation on its output. Each input is retained with a user-specified probability. **Sample** is useful for improving interactive response by reducing the size of data sets to be processed.

Operation	*Box Type*	*Effect*
Add Table	$\emptyset \rightarrow R$	Add the box producing a specified relation as output.
Project	$R \rightarrow R'$	Standard database projection; user is prompted for fields.
Restrict	$R \rightarrow R$	Filter a relation to tuples satisfying a predicate.
Sample	$R \rightarrow R$	Randomly sample a relation.
Join	$R \times R' \rightarrow R''$	Standard join of two relations; user is prompted for join predicate.

Fig. 3. Operations on relations.

The result of applying these operations is to iteratively build up a boxes-and-arrows program in the program window. We now turn to the visualization of the result of such programs.

5 Rendering Operations

The previous section has indicated how a Tioga-2 program can be built to retrieve complex computations (relations) from the database. Now we must deal with two additional questions:

- How are tuples positioned on the canvas?
- How are tuples rendered as screen pixels?

As discussed in Section 2, these questions are addressed by location attributes specifying the position of tuples in n-space and display attributes that specify tuples' screen representations. This section describes location and display attributes, default displays, and their associated operations.

5.1 Location and Display Attributes

Figure 4 shows a visualization of the Louisiana weather station data produced by the diagram shown in Figure 1. Each station in the state is represented by one tuple in the relation. The visualization shows a circle and the name of each station at its *(longitude, latitude)* coordinate. To position representations

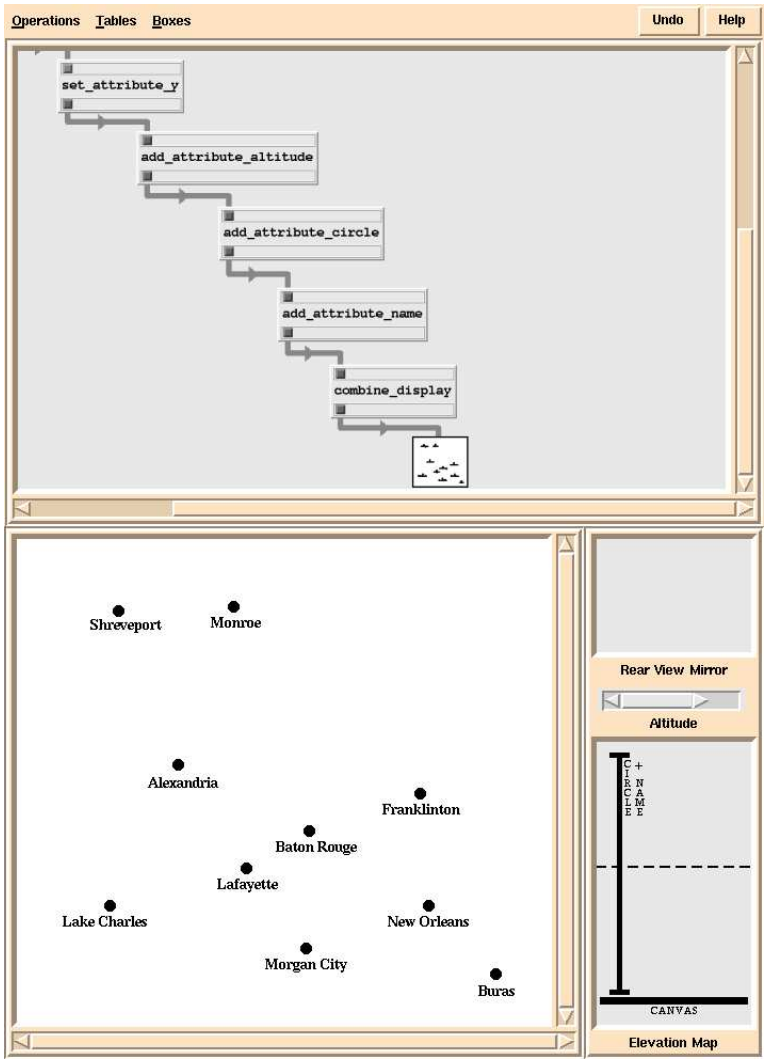

Fig. 4. A visualization of weather station locations.

of tuples on the screen, relations have location attributes. Every relation must have x and y location attributes to specify the x and y dimensions of a 2-D canvas; in Figure 4, the x dimension is longitude and the y dimension is latitude. There may be additional location attributes, which specify slider dimensions. In Figure 4, there is a slider dimension *Altitude*. By setting the range of altitude values that are visible using the slider, the user can see any appropriate subset of the stations. Location attributes are represented by floating point numbers.

Tioga-2 requires that every relation have at least one *display* attribute. A

display attribute is a list of primitive *drawable* objects. Intuitively, a viewer renders a tuple by simply painting each drawable in its *display* attribute on the screen. In Figure 4, the display attribute is a list containing the text of the name of the station and a circle. There may be additional display attributes to provide alternative visualizations of the data.

The primitive drawables include: point, line, rectangle, circle, polygon, text, and viewer. Each primitive drawable has an *offset*, a *color*, and a *style*. The offset gives a position relative to the location attributes of the tuple; thus, multiple drawables need not be stacked directly one atop the other. In Figure 4, the name is positioned below the circle. Of the primitives listed above, all but viewers are standard primitives for graphics hardware. Viewers are used to implement wormholes (Section 6). The list of primitive drawables is preliminary and more may be added in the future.

In Tioga-2, every relation is augmented with location and display attributes. Actually computing the values of these attributes should be avoided except where necessary. As discussed in Section 2, display and location attributes, along with any other "extra" attributes, are specified by functions of the base tuple.

5.2 Defaults

To guarantee that boxes produce relations with initial valid displays, Tioga-2 provides default location and display attributes. There is a default display for each atomic type (i.e., each type of a column of a relation). The default display for a relation renders each field in the tuple, side by side, using the default display for each column type to produce a screen representation. The default space has two dimensions: the x-location is 0 and the y-location is the sequence number of the tuple. Typically, the default attributes define a display consisting of a sequence of tuples in ASCII. The major relational DBMS vendors all have so-called *terminal monitors*, which produce a display of this form for the result of any possible query.

Just as the user may incrementally modify the data management operations to change the data to be visualized, so may the user incrementally modify the location and display attributes of a relation to change the visualization. Initially, every **Add Table** operation introduces a box that produces a relation with the default display and location. The user may then incrementally modify the defaults, or replace them altogether, by adding boxes to the diagram or by manipulating data on the canvas. In Figure 4, the default viewer of Figure 1 has been changed by modifying location functions (to associate *(longitude,latitude)* with (x, y) canvas coordinates) and the display function (changed to the combination of station name and a circle).

5.3 Operations

In the remainder of this section we discuss the operations for modifying location and display attributes listed in Figure 5. Most of these operations apply to all attributes, not just location or display attributes.

Operation	*Box Type*	*Effect*
Add Attribute	$R \rightarrow R'$	Add an attribute to a relation; user is prompted for definition.
Remove Attribute	$R \rightarrow R'$	Remove an attribute; cannot remove attributes x, y, or *display*.
Set Attribute	$R \rightarrow R'$	Change the value of an existing attribute.
Swap Attributes	$R \rightarrow R'$	Interchange two attributes.
Scale Attribute	$R \rightarrow R'$	Multiply numerical attribute by a number.
Translate Attribute	$R \rightarrow R'$	Add a number to a numerical attribute.
Combine Displays	$R \rightarrow R'$	Combine two display attributes.

Fig. 5. Location and display operations.

The user may add new attributes, including new location and display attributes. Adding a location attribute adds a new dimension to the visualization. Adding a display attribute creates an alternative visualization of the data. **Add Attribute** prompts for the type and definition of the new attribute; the definition may depend only on other attributes of the relation. **Set Attribute** changes the type and definition of an existing attribute.

In both **Add** and **Set Attribute**, an attribute's definition may be given in a general query language. However, the preferred method is to begin with a very simple definition (e.g., a copy of another field, or a single primitive drawable) and then refine it using the other operations.

Swap Attributes is handy for interchanging two dimensions (two location attributes), thereby "rotating" the canvas, or interchanging the display attribute with one of the alternative displays, thereby changing the visualization of the data.

Scale and **Translate Attribute** are defined only for numeric fields. These operations are convenient shorthands for more complex **Set Attribute** commands. **Scale** and **Translate** are useful for changing location attributes, thereby scaling or translating dimensions of a visualization.

Combine Display is the mechanism for combining primitive drawables to form more complex displays. The user positions the displays on top of one another graphically to establish the relative position; alternatively, an explicit offset of one display to the other can be entered. The combined display becomes a new display attribute. The user may combine any of the display attributes of the relation. In Figure 4, a circle display has been combined with a text display showing the name of the station.

6 Drill Down

Drill down allows users to see more details in data of interest. There are two distinct, useful notions of drill down. The first provides a more refined view of the same data in the same visualization space (e.g., switching from a state to a

Operation	*Box Type*
Set Range	$R \rightarrow R$
Overlay	$Composite(R_1, \ldots, R_n) \times Composite(R_{n+1}, \ldots, R_m) \rightarrow$ $Composite(R_1, \ldots, R_m)$
Shuffle	$Composite(R_1, \ldots, R_{i-1}, R_i, R_{i+1}, \ldots) \rightarrow$ $Composite(R_i, R_1, \ldots, R_{i-1}, R_{i+1}, \ldots)$

Fig. 6. Primitives for drill down.

county map). The second allows movement between one space and a different, but semantically related, space (e.g., after finding a weather station, switch to look at its temperature/precipitation data).

Two mechanisms provide drill down in Tioga-2. First, the user can specify that additional detail about screen objects becomes available as the user zooms in. Second, we have a notion of *wormholes*, by which a user can move from one canvas to another canvas.

6.1 Additional Detail

The first form of drill down is defined as operations on relations R and composites C. There are three operations:

- **Set Range**
  This operation specifies the maximum and minimum elevations at which a relation's display is defined. Outside of this range, the relation contributes nothing to the canvas.
- **Overlay**
  Two composites may be overlaid. The relative position of one overlay to another may be given either by an explicit n-dimensional offset, or by dragging one canvas over the other. If the component displays are defined with different elevation ranges, then it is possible to program drill down by having the displayable at the lower elevation provide a specialization of the displayable at the higher elevation.
- **Shuffle**
  It may be desirable to change the drawing order of the relations within a composite. The **Shuffle** command moves a relation to the "top" of the drawing order.

Figure 7 illustrates overlay and setting ranges. Weather stations are now shown together with a map of Louisiana; this is achieved by overlaying the map (derived from a relation of lines defining the map) with the result of Figure 4. In addition, a third display is overlaid to give less detail at higher elevations. This display shows only a circle at the station's location. The programmer has set the ranges of the two weather station displays so that station names disappear at high elevations, where they would be illegible. The range of the Lousiana map is all elevations (the default).

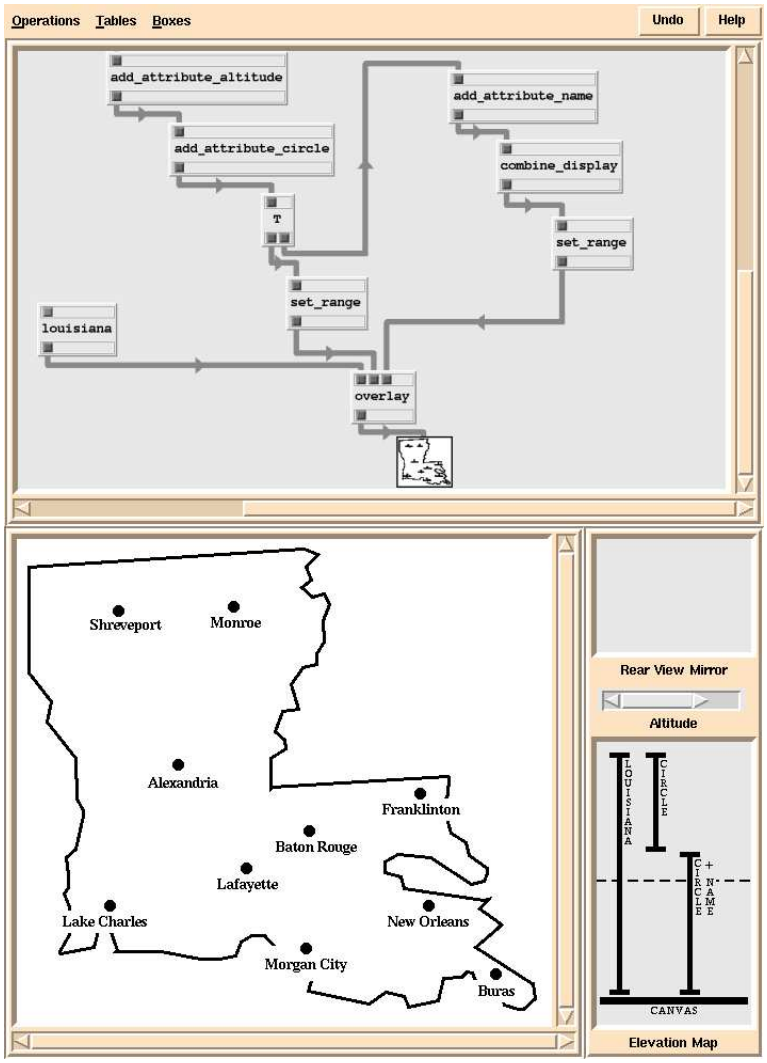

Fig. 7. Overlaid displays with restricted ranges.

There is a small difficulty with the overlay in Figure 7. The visualization of the state map of Louisiana has no *Altitude* dimension, and such a dimension makes no sense for a flat map. However, the composite has an *Altitude* slider; how are changes in Altitude to be interpreted for the Louisiana map? If the user attempts to overlay relations with different dimensions, Tioga-2 warns about the mismatch. If the user wishes, the underlying relations are treated as invariant in the "extra" dimensions. This achieves the desired effect in Figure 7: the user can change the *Altitude* slider to see different subsets of the stations, but the

Louisiana map remains in place for reference.

The *elevation map* is a bar-chart display of the maximum/minimum elevations and drawing order of all elements of a composite on the current canvas (see Figure 7). The elevation map can be manipulated directly by the user to adjust the ranges and drawing order of overlaid relations. For a group displayable, a viewer shows an elevation map for only one member of the group at a time. In this case, the user can explicitly cycle through all of the elevation maps.

6.2 Wormholes

It is often desirable to associate objects in one visualization space directly with objects in a different visualization space. A *wormhole* is a viewer onto another canvas, i.e., what is visible inside a wormhole is a point on another canvas from some elevation. Figure 8 shows an example application of wormholes. Upon zooming into an individual station s, a wormhole appears (achieved by a combination of modifying display functions and overlaying and setting ranges) that takes the user to a canvas displaying temperature data for each station as a function of time. The user is initially positioned viewing the data for station s.

Providing wormholes is technically straightforward. Viewers are primitive drawable objects; thus, Tioga-2 programs may produce displays containing viewers (wormholes). A viewer drawable requires several parameters, including the size for the viewer, a destination canvas, the elevation from which the canvas is viewed, and the initial location; the user defines these values as part of the display attribute. As with any drawables, wormholes can be overlaid with other drawables. In Figure 8, the axes labels are the result of overlaying text at an offset from the wormhole (for brevity, these boxes are not shown).

When a user zooms in on a wormhole and reaches zero elevation he passes through the wormhole and moves from the original canvas to the destination canvas. Needless to say, the user can pan and zoom on this second canvas, as well as move to a third canvas. After changing canvases several times, there is a definite possibility the user will get lost. For this reason, we introduce the notion of a rear view mirror.

6.3 Rear View Mirrors

For each canvas, we introduce an additional window called a *rear view mirror*. This window shows the "bottom side" of the canvas through which the user last moved. Hence, immediately after going through a wormhole, the user is looking down at a new canvas from some specific elevation and is at negative ground level for the canvas he just left. As the user descends toward the new canvas, the distance from the previous canvas increases. In Figure 9, the rear view mirror shows that the user came through the wormhole at New Orleans in Figure 8.

Every Tioga-2 displayable has a minimum and maximum elevation. If both are positive, then the viewer only shows objects on the top side of the canvas. If the minimum and maximum elevations are both negative, then the viewer places objects only on the underside of the canvas, and they are visible only in

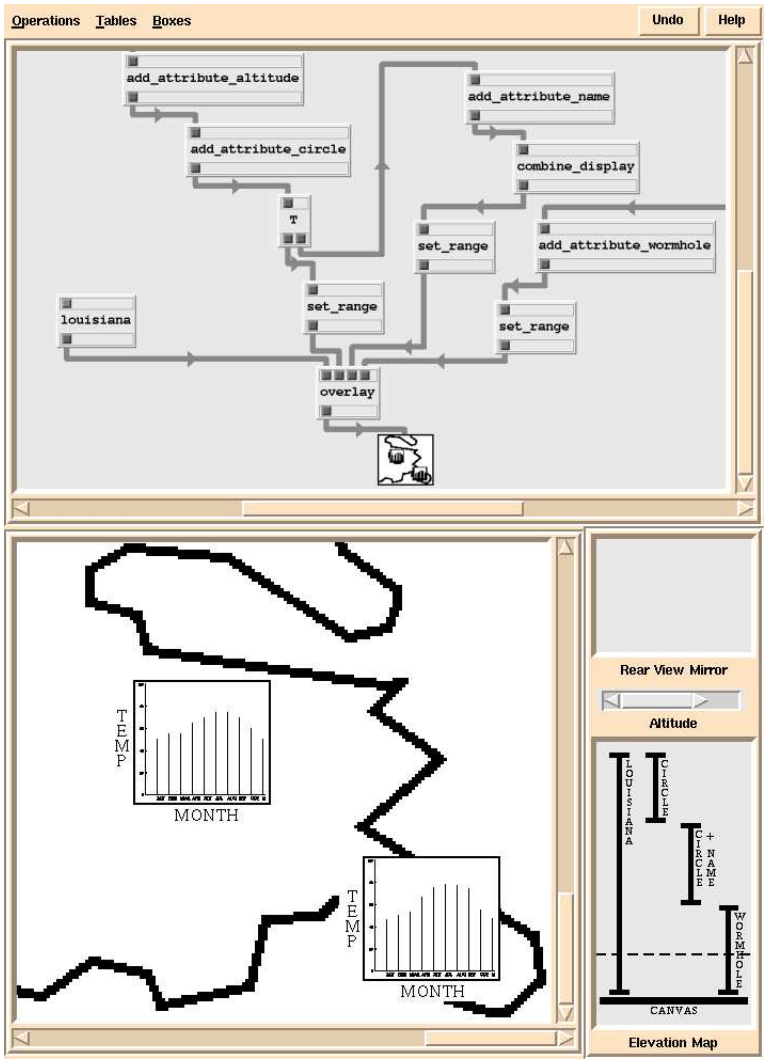

Fig. 8. A visualization with wormholes.

the rear view mirror after the user proceeds through a wormhole. If the minimum elevation is negative and the maximum is positive, then the objects can be seen on both sides of the canvas. Thus, the programmer can create overlays in such a way that the top side and the underside of the canvas both have meaning. One is visible from above in the viewer window and one is visible from below in the rear view mirror.

A natural use of the rear view mirror is to illuminate the wormholes back to the canvas from which the user came to this canvas. In this way, the user can

"find the way home" if he is lost. As such, the rear view mirror is a generalization of the notion of "back" in a hypertext system.

7 Additional Operations

This section discusses the remaining Tioga-2 features, with the exception of updates (Section 8) and a few user-interface features discussed in Section 9. *Slaving* constrains two viewers to move together. *Magnifying glasses* provide hierarchical viewers (viewers within viewers). As discussed below, magnifying glasses are quite different from wormholes. *Stitch* and *replicate* produce group displays. Slaving and magnifying glasses are operations on viewers, while stitch and replicate are operations on displayables.

7.1 Slaving

Two viewers may be *slaved* together, in which case the system maintains the relative offset between the two viewers. When a viewer is deleted, all of its slaving relationships are also deleted. Slaving is only defined for two viewers with the same dimensions.

7.2 Magnifying Glasses

A user may create a *magnifying glass* by placing a viewer inside another viewer. Typically, a user places a copy of the current viewer inside itself and then zooms the inner viewer to magnify what is in the outer viewer. A magnifying glass must have the same dimensions as its containing viewer. The inner and outer viewers may be slaved; magnifying glasses may also be deleted.

A simple technique for correlating temperature and precipitation uses a magnifying glass in Figure 9. The user begins with a temperature vs. time display. The underlying relation that is being visualized has more information—in particular, the precipitation data—that is not being utilized. An alternative display attribute shows precipitation vs. time (the boxes defining the precipitation display are not shown). By creating a magnifying glass using this alternative display, the user sees the precipitation data for points underneath the magnifying glass. In Figure 9, the magnifying glass is realized by making the precipitation display the *display* attribute (done by the **Swap Attribute** box) and then viewing the result.

7.3 Stitch

Any number of composites can be *stitched* to form a group displayable. Groups can be displayed side-by-side, arranged vertically, or laid out in a tabular fashion. If the user performs a window operation on one of the group members, such as moving the window on the screen or iconifying it, then the same operation is performed on the other members. Zooming and panning is defined for each of

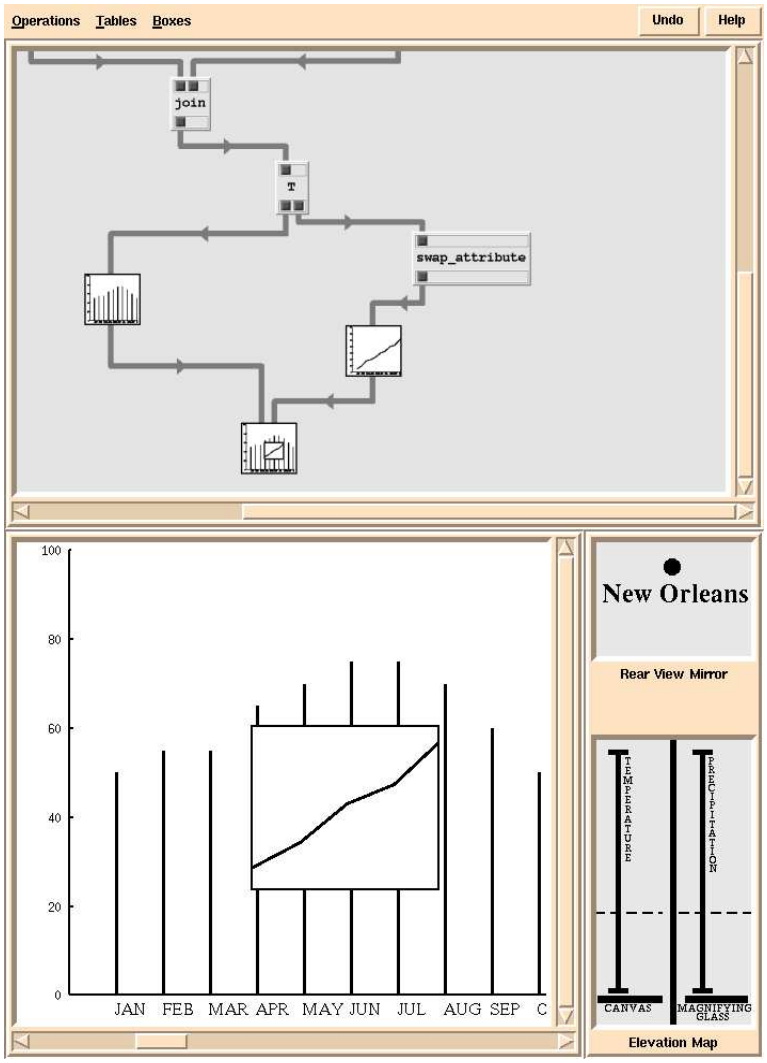

Fig. 9. Using a magnifying glass.

the constituent displays. That is, there is a separate focus for all components, as well as separate x, y, slider, and zoom dimensions. Components may be slaved.

In Figure 10, a display showing temperature vs. time is stitched to a display showing precipitation vs. time. The precipitation display is slaved to the temperature display, so that whenever the user changes the date range under temperature, the precipitation display changes to display the same date range.

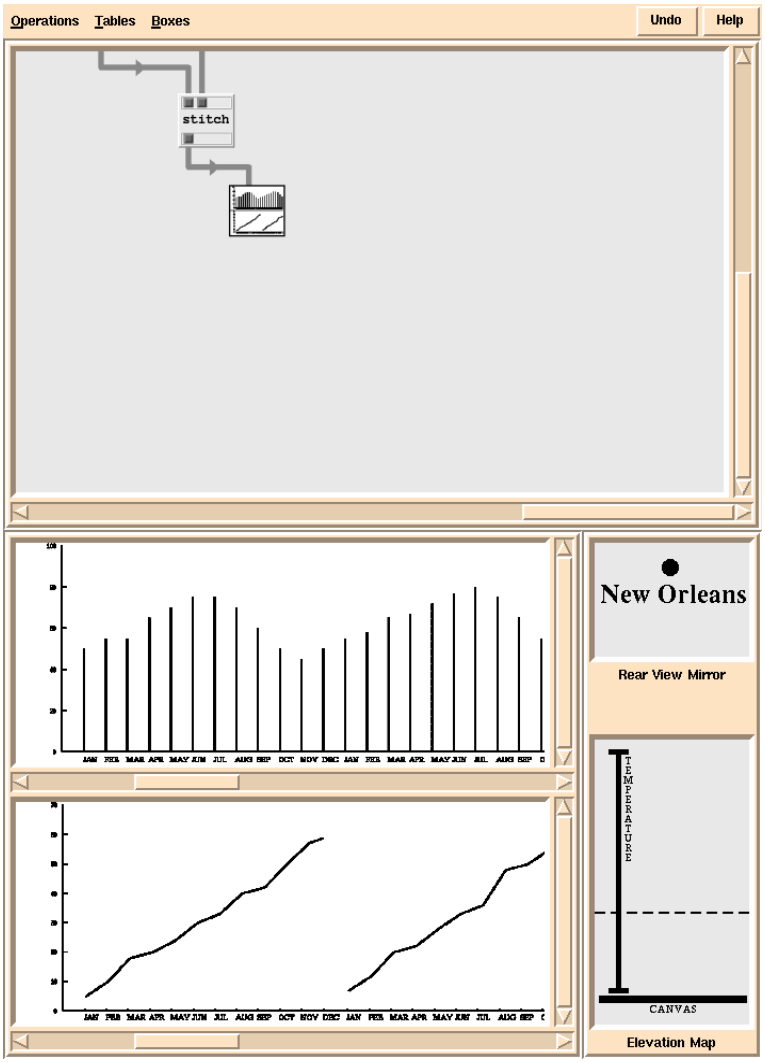

Fig. 10. An example of stitched viewers.

7.4 Replicate

A relation can be *replicated* by specifying a partition. Replicated displays for each partition are stitched together into a group. The user must specify the area to be given to each display and the initial point of focus.

The partitioning predicate is specified by giving a collection of predicates in the underlying query language or an enumerated type. For example, the specification may be that replication is tabular, with predicates `salary` $\leq$ `5000` and `salary` $>$ `5000` in the horizontal dimension and the enumerated type

`department` in the vertical dimension.

In Figure 11, a viewer showing temperature vs. time and precipitation vs. time has been replicated to show records for years prior to 1990 and after 1990 separately. This example motivates the need for operator overloading discussed in Section 2. Because **Replicate** partitions a relation, it takes an R as input and produces multiple R's as output. However, in this example the display is a G type (a group of two displays). Thus, before the replication can be performed, the user must specify the relation. When the user selects **Replicate**, the system prompts the user for the group component on which the replication is to be done.

8 Updates in Tioga-2

Tioga-2 is oriented toward browsing a database. As such, we expect users to wander around a canvas and possibly notice things they wish to update. For example, the quantity on hand of specific items could appear on a canvas. The user would find an item of interest and then wish to order a certain number of the item, thereby decreasing the quantity on hand. The user could also notice data errors and simply wish to fix them. As a result, we focus on providing an update capability that allows specific screen objects to be updated in the database. We do not consider general SQL update statements in Tioga-2.

For each primitive type, the type definer is required to implement a default display function that is used by Tioga-2 to render tuples containing this type. Similarly, we require the type definer to write a second *update* function that enables Tioga-2 to provide updates for instances of the type that appear on the screen. When a user clicks on a screen object, the Tioga-2 run time system activates a generic update procedure, passing it the tuple corresponding to the screen object. The function engages a dialog with the user to construct a new tuple—using the primitive update functions for the fields—and then perform an SQL update to install the new value in the database. This machinery is all encapsulated within the update function itself.

When the user customizes a visualization, he can replace the default update command with one of his own choosing, if he so desires. In this way, he can make an update system with a desired "look and feel".

9 Implementation

In this section we discuss briefly how the design is evolving to address issues encountered during implementation. The changes described here result from observations about how users progressively render data in a multi-dimensional space. The changes include a paint program window to provide more intuitive rendering and two new object types for displaying objects that are not associated with database data.

The current implementation of Tioga-2 is being developed on DEC Alpha workstations using Postgres95 for the database engine, Tcl/Tk for the boxes-and-arrows editor, and OpenGL for the 3D graphic visualization.

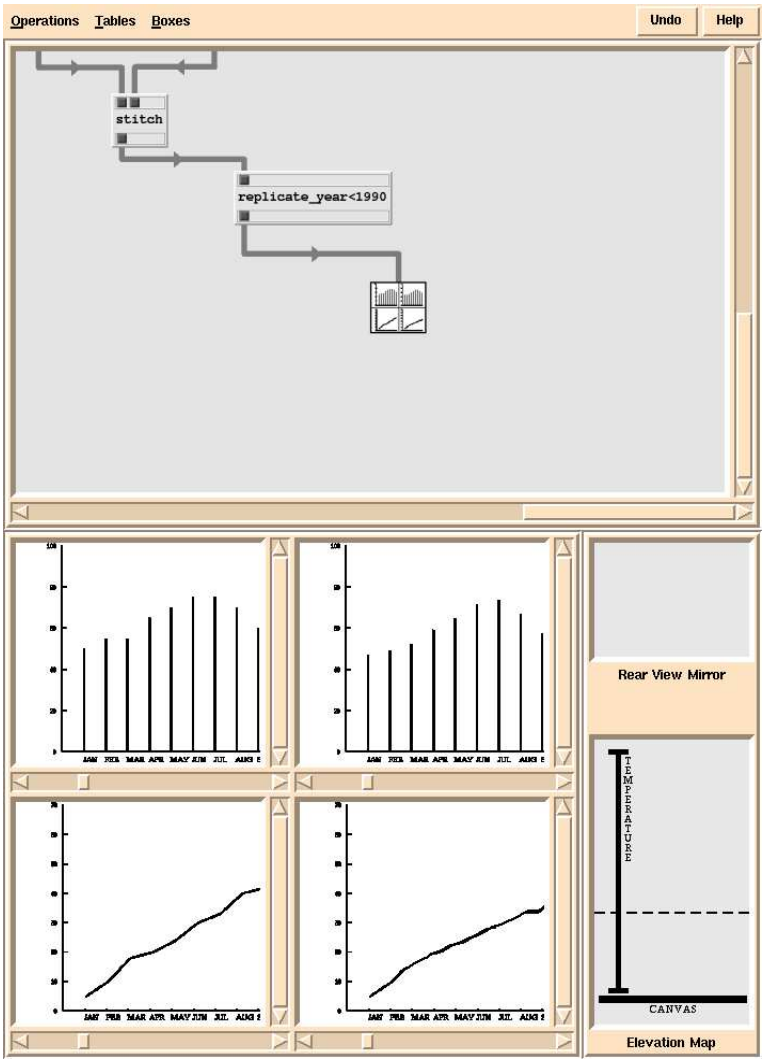

Fig. 11. A replicated viewer.

9.1 Overall Design

Tioga-2 is currently being constructed as four major modules:

- a boxes-and-arrows editor for data programming (Section 4),
- a new paint program window for rendering display attributes and specifying wormholes (Sections 5 and 6.2),
- a composite and group editor for additional operations (Section 7),
- and a menu bar.

The boxes-and-arrows editor is very similar to the one reported in [12] and is not discussed here. The paint program implementation is largely complete, with the exception of wormholes, and is described in Sections 9.2–9.5. The composite and group editor is not yet implemented and is not discussed.

9.2 The Paint Program

To make rendering intuitive for the user of Tioga-2, an interface similar to those in *paint programs* has been constructed. This window has a palette of displayable primitive objects (point, line, rectangle, circle, polygon, and text) on the left side of the screen.[2] Like conventional paint programs, a displayable primitive is rendered by selecting that primitive from the paint palette and then placing it in the canvas window.

The paint program includes a window that shows tuples from the visualized relation in the default format, constructed by converting all objects to a textual representation. Postgres95 requires such a function for every valid type and Tioga-2 simply uses it. Note that this display is in addition to any visualization the user constructs; thus, the Tioga-2 programmer can see both the visualization and attribute values of sample tuples simultaneously. Access to the actual data being visualized helps users quickly interpret unexpected results of incremental changes to a visualization. For example, suppose that weather station data is visualized using a rectangle whose height is set to an attribute representing average annual rainfall. If the relation is filtered to include only stations in the world's driest areas, then the height of each rectangle may be zero. Upon noticing that all displayed rectangles are, in fact, lines, the Tioga-2 programmer instantly can check the attribute values in the tuple window to confirm that the corresponding attribute values indeed produce this result.

9.3 Displayable Objects

The paint program can also draw displayables that are not associated with any tuple. These displayables are useful for "trim" such as borders, titles, company logos, etc. Semantically, these displayables are objects associated with an overlay. We introduce two such types of objects: *static* and *sticky* (a term borrowed from the Pad project [8]).

An example static object is a scale for the (x, y) dimensions with tick marks on the axes. These objects are static because they have constant position in the (x, y) dimensions. Panning and zooming of a static object has the same visual effect as panning and zooming in the rendered data.

An example sticky object is a window title. The object sticks to a particular position in the window and does not have an (x, y) position. As such, panning has no effect on a sticky object. Zooming on a sticky object produces a screen representation so long as the sticky object's overlay is visible at a particular viewing elevation.

[2] Thus, the appearance of the current interface has evolved to look somewhat different from the screenshots shown in this paper.

9.4 Dimensia Disorientation

Early in our implementation efforts we noticed an unanticipated problem: certain operations could leave the Tioga-2 user suddenly visualizing a region with no data in it, resulting in a blank screen. The most important case arises when the user changes the dimensions of the visualization space. For example, suppose that employee tuples are being viewed and the fields are `salary`, `name` and `age`. Further suppose that the tuple (`10000`, `john`, `18`) appears on the screen and that the x dimension is set to the `salary attribute` (i.e., the x location of the viewer is approximately 10,000). Now suppose that the programmer changes the x dimension to `age`. In all likelihood the tuple disappears from the viewer—in fact, all data disappears from the viewer—and the Tioga-2 programmer suffers from *dimensia disorientation*.

To allow the user to keep the focus of a visualization in an area of interest when performing dimension operations, we have added *sticky tuple mode* to Tioga-2. This mode ensures that a particular tuple remains on the screen when the dimensions of the visualization space are altered.

9.5 Painting Displayables

A Tioga-2 visualization of a relation is the sum of the visualizations of each tuple of the relation. Each time the programmer modifies the display it is potentially necessary to recalculate the visualization of each tuple of the relation. This is especially expensive when the user is actively modifying the visualization instead of simply browsing—in this case the underlying relation and the desired visualization may both change.

To make visual programming as interactive as possible, we have added *one tuple mode*, a restriction of sticky tuple mode. In this mode, only the single sticky tuple appears on the screen and, therefore, the screen can be painted without access to the database. The user edits the visualization of the single example tuple to his liking and then switches to viewing the entire relation to confirm that the visualization is as desired.

The sticky tuple is specified by the programmer by selecting an example tuple in the default data window and then clicking on a `One Tuple Mode` button in the paint program.

10 Future Work

Tioga-2 raises several interesting issues that we plan to address in future work. A few of these problems are discussed briefly in this section.

10.1 Caching Data vs. Caching Graphics

Tioga-2 is designed for visualizing large databases, and thus not all data can be held in memory at any one time. For this reason, and because browsing has

locality (i.e., panning and zooming move to nearby points in the viewing space), caching both the database data that is being graphically represented as well as the actual graphical representation appears to be beneficial. However, given that there is limited space available for all caches, any space used for caching data is not available for caching graphics and vice versa. This is not a trivial optimization problem because the graphical representation is typically much larger than the data representation. As a result the graphical coverage will likely be much smaller than the data coverage; on the other hand, fast response time to panning operations is only possible with a graphical cache. We expect to explore these multi-cache issues in detail.

10.2 Sampling

When programming a visualization from a very large data base, it may be desirable to construct a visualization for a random sample of the data. In this way, the programmer can move from initial rendering to a final product on a small data set. Only when he is satisfied with the result should he move to execution on the complete data set. Hence, a possible extension (or alternative) to Tioga-2's "one tuple mode" is the seamless integration of random sampling.

10.3 Clutter

In many cases data is very non-uniform when placed on the canvas. For example, if the population of the United States is rendered at the (x, y) coordinates of each citizen's home address, then the spacing appropriate in Montana yields incredible clutter in New York City. Conversely, a spacing appropriate in New York City places people much too far apart in Montana. We plan to search for solutions to the problem of intelligently displaying non-uniform or "cluttered" data.

10.4 Foreign Systems

It is possible that a Tioga-2 application would entail some browsing of a canvas, along with the display of reports, spreadsheets, and forms. How Tioga-2 should interact with other subsystems, such as spreadsheets, is a topic for future investigation.

11 Related Work

While developing browsers for exploring data is a relatively new research area, the literature is already substantial. This section surveys a cross-section of related work.

As discussed in Section 1, Tioga-2 retains the boxes-and-arrows notation for programs originally developed for dataflow languages and popularized for visualization by AVS [13], Data Explorer [7], and Khoros [9]. These systems are

similar to Tioga in their reliance on simplifying programming by using dataflow graphs. Thus, these systems share Tioga's basic problem that boxes-and-arrows notation alone does not simplify programming sufficiently for novice programmers (see Section 1.1). *Weaves* is another boxes-and-arrows system [3]. Weaves are intended to support visual programming, so the boxes-and-arrows program is itself the only visualization of interest. An extension of weaves supports limited drill down [5].

Many browsing systems are based on a "paradigm". A classic example is the Fisheye interface, which magnifies data in the center of focus to a greater degree than data at the periphery [10]. Another example is Magic Lenses, which provides a set of primitive lenses (windows akin to our magnifying glasses) that can be placed over data and over each other to modify a visualization [1]. While we find paradigms appealing, we suspect a flaw in the assumption that the space of possible visualizations can or must be greatly restricted in advance.[3] In our experience, paradigms serve a class of users well and frustrate users with other applications. To be generally useful—as Tioga-2 aims to be—it is important that users be able to construct arbitrary *ad hoc* visualizations of their own, even inventing their own paradigms if necessary. In short, visualizations should be as programmable as possible.

A different approach has been taken by the ambitious Pad project [8]. In Pad, all data lives on a two-dimensional plane. As in our system, every entity (an object in Pad, a tuple in Tioga-2) has a position and "knows" how to draw itself. Pad also provides facilities for overlay and drill down that are in some ways richer than the facilities in Tioga-2. Pad allows a very large class of visualizations to be built. However, Pad is not end-user programmable; it is designed as a toolkit for expert programmers and provides a traditional programming interface.

Within the area of browsers for databases, the work of Krishnamurthy and Zloof on Rendering By Example (RBE) is closest to our own. In particular, RBE shares our view on the importance of a system that is both highly programmable and easy to program [6]. RBE provides a more declarative programming interface than Tioga-2, but RBE can construct a much less general class of visualizations.

Finally, a database-centric visualization system raises the issue of how browsing queries are implemented with tolerable performance. This question is beyond the scope of this paper; the interested reader is referred to [2] for related work on the optimization and efficient implementation of browsing queries.

12 Conclusions

We are now hard at work implementing Tioga-2. An initial version of the system is functional, and we expect to have a complete prototype by summer 1996. We plan to systematically test the implementation on little programmers to ascertain whether it lives up to its goals.

[3] In fairness, Magic Lenses is not intended strictly as a browsing paradigm, but as a general user interface paradigm.

References

1. E. Bier, M. Stone, K. Pier, W. Buxton, and T. DeRose. Toolglass and magic lenses: The see-through interface. In *Proc. of SIGGRAPH 1993*, pages 73–80, Anaheim, CA, August 1993.
2. J. Chen. Optimizing interactive browsing queries. Unpublished manuscript, University of California, Berkeley, June 1995.
3. P. Cox, M. Gorlick, and R. Razouk. Using weaves for software construction and analysis. In *Proc. of the 13th International Conference on Software Engineering*, pages 23–34, Austin, TX, May 1991.
4. Allen Cypher. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.
5. M. Gorlick and A. Quilici. Visual programming-in-the-large versus visual programming-in-the-small. In *Proc. of the IEEE Symposium on Visual Languages*, pages 137–144, St. Louis, MO, October 1994.
6. R. Krishnamurthy and M. Zloof. RBE: Rendering by example. In *Proc. of the 11th International Conference on Data Engineeering*, pages 288–297, Taipei, Taiwan, March 1995.
7. B. Lucas, G.D. Abram, N.S. Collins, D.A. Epstein, et al. An architecture for a scientific visualization system. In *Proc. of the IEEE Visualization Conference*, pages 107–114, Boston, MA, October 1992.
8. K. Perlin and D. Fox. Pad: An alternative approach to the computer interface. In *Proc. of SIGGRAPH*, pages 57–64, Anaheim, CA, August 1993.
9. J. Rasure and M. Young. An open environment for image processing software development. In *Proc. of the SPIE Symposium on Electronic Image Processing*, pages 300–310, San Jose, CA, February 1992.
10. M. Sarkar and M.H. Brown. Graphical fisheye views. *Communications of the ACM*, pages 73–84, December 1994.
11. M. Stonebraker, R. Agrawal, U. Dayal, E. Neuhold, and A. Reuter. DBMS research at a crossroads: The Vienna update. In *Proc. of the 19th International Conference on Very Large Data Bases*, pages 688–692, Dublin, Ireland, August 1993.
12. M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *Proc. of the 19th International Conference on Very Large Data Bases*, pages 25–38, Dublin, Ireland, August 1993.
13. C. Upson et al. The application visualization system. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.
14. A. Woodruff, P. Wisnovsky, C. Taylor, M. Stonebraker, C. Paxson, J. Chen, and A. Aiken. Zooming and tunneling in Tioga: Supporting navigation in multidimensional space. In *Proc. of the IEEE Symposium on Visual Languages*, pages 191–193, St. Louis, MO, October 1994.

This article was processed using the LaTeX macro package with LLNCS style