

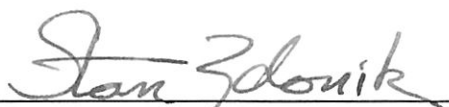
# Implementing a Persistent Query Graphical User Interface for a Streaming Database

Robin Yan

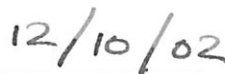
Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the  
Department of Computer Science at Brown University

A handwritten signature in cursive script, reading "Stan Zdonik", written over a horizontal line.

Signature (Prof. Stan Zdonik, Project Advisor)

A handwritten date "12/10/02" written in a simple, slightly slanted font.

Date

BROWN UNIVERSITY  
Department of Computer Science  
Master's Project

"Implementing a Persistent Query Graphical User Interface for a  
Streaming Database"

by

Robin Yan

# Implementing a Persistent Query Mechanism for a Streaming Database

Robin Yan

December 2002

## 1 Introduction

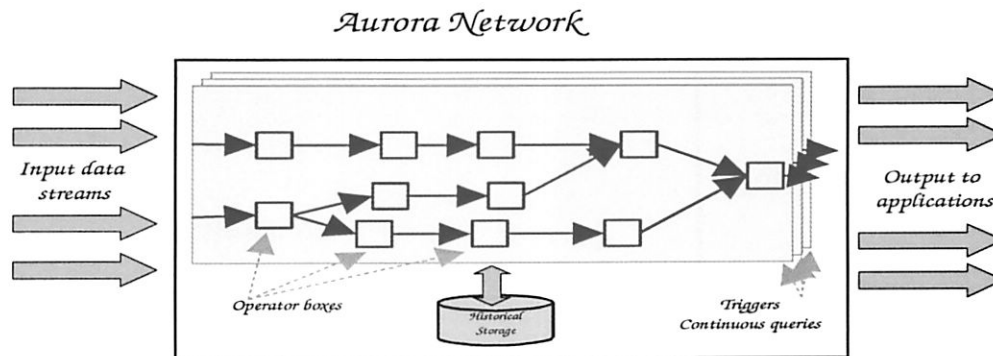
During the spring of 2002, members of database groups from Brown University, MIT, and Brandeis University began collaborating to develop a new streaming database system called Aurora (<http://www.cs.brown.edu/research/aurora/>). This master's project is an ongoing effort to develop a graphical user interface that supports several requirements of Aurora, and to enhance the ease of designing and maintaining persistent queries over streams.

In Aurora, queries are designed and considered persistent, and the focus for the interface is to provide the user with the capability of easy design, understanding, and maintenance of a query. In order to visualize these queries more easily, it was decided that a visual graph-like representation of relational algebra would be a good way to implement and maintain queries.

The user interface was an aggregate effort of six Brown University students – undergraduates Jodi Rodgers, Charlie Kim, Adam Singer, and Matt Hatoun, and graduates Robin Yan and Jeong-Hyon Hwang. Development began in mid-March of 2002 and is an ongoing process. However, as of September 2002, a completed version of the GUI was implemented for the end-of-summer Aurora demonstration.

## 2 Overview of the Aurora system

This new paradigm of streaming data presents a completely different way of considering the task of data management. Because of the intrinsic differences between temporary, streaming data, and static, stored database relations, the task of understanding and processing streaming data is also very different.



Unlike traditional database systems, a streaming database such as Aurora views its data as temporary and its queries as persistent and continuous. Whereas traditional database management systems typically involve one system, or the database, there are three top-level entities in Aurora -- the streaming data sources, the Aurora system, and the applications. The data sources may, for example, be composed of multiple sensors that detect information and continually feed data into the Aurora system. Such sensors, for example, could be highway traffic monitors, physiological sensors for patients in hospitals, or environmental sensors in an intelligent building. In each of these cases, data is continually being streamed. The Aurora system processes the data using the pre-defined persistent query, and then feeds it into applications such as monitors.

Unlike traditional database management systems where data are permanently stored in a disk and are created, updated, removed, and retrieved by multiple queries, Aurora processes temporary, incoming streams of data from sensors via an unchanging and persistent query. One of the more important goals of this system is to behave optimally in situations where there is an excess of incoming data or where processor resources are scarce. In order to handle several types of extreme circumstances, Aurora employs complex and sophisticated algorithms for scheduling operators for processing, quality of service monitoring, and dropping tuples of data. In addition, a sophisticated storage manager is in place for temporarily storing tuples queued up for processing at various operators. Suitable applications include monitoring physiological conditions of patients in a hospital, monitoring the physical location of books and inventory in a library, and monitoring and analyzing streams of data generated by the stock market. On the output side of Aurora, the applications could be computers that detect events and take appropriate actions.

As in traditional database systems, data are represented as tuples of multiple fields. These tuples may be operated on by traditional database query operators, or boxes, such as SELECT, JOIN, GROUP BY, and SORT. However, the nature of stream data, as opposed to non-streaming relational data, results in specific parameters for particular operators. For example, because the data is theoretically streaming infinitely, it is impossible to define blocking operators such as JOIN or aggregate operator such as COUNT and AVERAGE without having also to define time-

windows within which the operators may process the streams. On the same note, new operators such as SLIDE, RESTREAM and TUMBLE are introduced in order to modify the stream-specific properties of the data or to shift tuples downstream.

A connected system of boxes is considered a network diagram, or a graph diagram. For Aurora, loops within the network are not permitted. Instead, to simulate loops, the user may route one of the outputs of the Aurora query into one of the inputs. Therefore, it is necessary for the graphical user interface to accommodate all of these requirements.

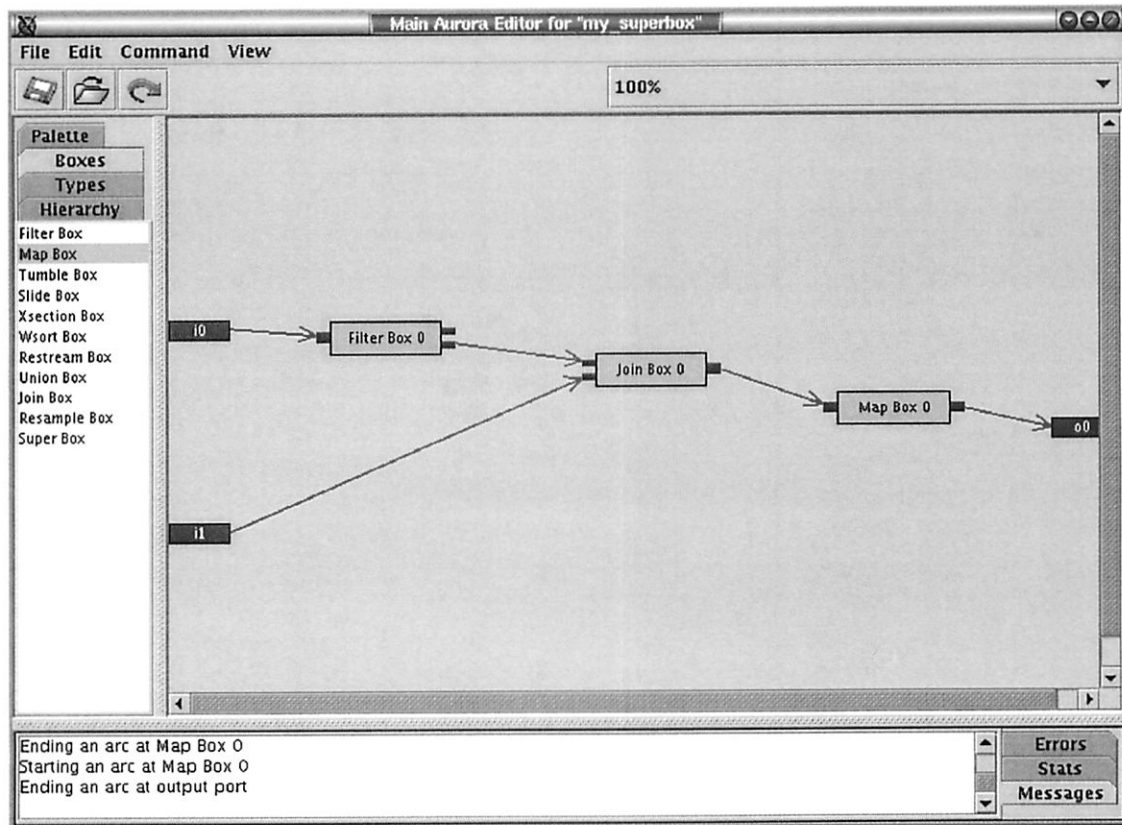
### 3 The role and requirements of the GUI

It was decided that the Aurora queries would be represented as a visual, graph-like representation of a decomposed relational algebra query instead of a text-based, typed SQL-like query. While it is entirely possible to write a text-based SQL-like query for streaming data, doing so has particular disadvantages due to the persistent nature of Aurora queries. While text-based queries can be written quickly, they do not have the clarity that the tree-like (or graph-like) structure representation of a relational algebra query has. Because the queries are persistent, infrequently written, but perhaps frequently—modified locally, a graph-like visual representation has clear advantages. Furthermore, the visualization of the graph-like structure of a query allows for easy local modification of operators, while a text-based query would require searching through the query string for the modification points. The visual representation also has advantages in clarity as the query increases in size. It allows for copy and paste operations and properties of encapsulation that facilitates query maintenance.

In addition to the disadvantages of in clarity and modification difficulties of text-based SQL queries, the graph-like representation also yields better results for individual users. Because there may be multiple independent queries authored by several individuals for different purposes running in Aurora, the interface would have to run on a client machines separately from the Aurora server. This would enable multiple query developers to simultaneously work on separate query graphs in their own developer spaces called “super boxes”. With the anticipated potential of Aurora processing approximately 10,000 operator boxes, the clarity that this design decision offers is unparalleled.

#### 3.1 Structure using boxes, arcs, and ports

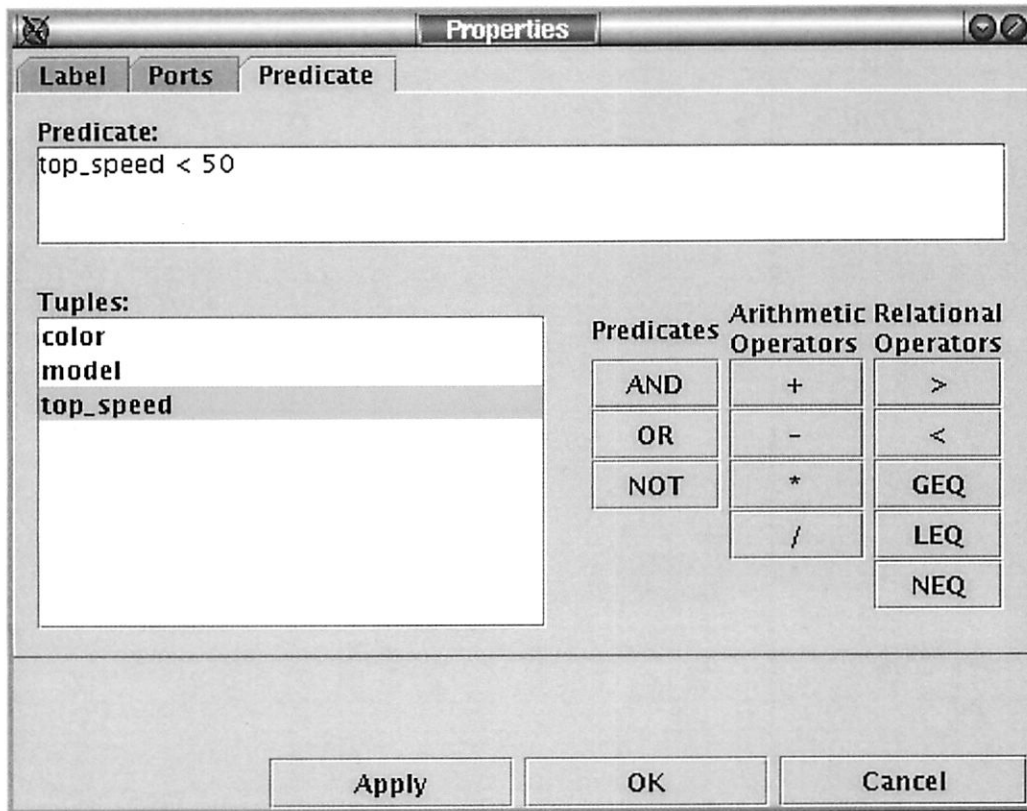
In order to represent the query graph, a system of boxes, ports, and arcs is used. Boxes represent the operators. Each box has a set of input ports and output ports. For example, a FILTER (otherwise known as SELECT) box has one input port which receives incoming tuples, and two output ports -- one output port for tuples that have been filtered through during runtime, and another output port for tuples that have not been filtered through during runtime. On the other hand, a JOIN box has two input ports -- each input port for each type of incoming tuple -- and one output port. These boxes are visually connected by arcs. In the GUI, to add a new operator to the query graph, the user may drag a list entry from the boxes tab into the editor workspace.



In order to create an arc, the user needs only to click on an operator's output port or an input port node and drag the arc end to the input port of another box or an output port node. Otherwise, all other mouse events are interpreted as arc movement or box selection/movement operations. In the GUI all input ports of boxes except for union may not have more than one incoming arc, and arcs may not originate from output port nodes or input ports of boxes. In the above diagram, the query graph is conceptually drawn from the left of the editor workspace towards the right.

In order to submit multiple independent queries to Aurora, independent query graphs with their own input source stream nodes and output application nodes may be designed.

### 3.2 Support for modifiers



The visualization properties of the query mechanism provide another advantage over the text-based version. Because each operator must have its own modifier, the user needs to right click on the appropriate box and raise the box properties dialog in order to specify the modifier for the operator. However, in the text-based version, the WHERE predicate for a SELECT is typically not written right after the SELECT. While this may increase the clarity of the SQL statement for English speakers, it can be extremely confusing in large nested queries. In the GUI, the modifier for each operator must be written specifically in accordance with the operator grammar. Each operator box has its own specific modifier definition dialog. A FILTER box, as shown above, requires a predicate returning a boolean for its modifier. A MAP box, on the other hand, requires a list of functions given the input tuple attributes.

### 3.4 Source stream boxes and application boxes

In order to connect the query graph to the source streams and the applications, the GUI also provides input boxes and output boxes. On the very left side of the editor workspace, input boxes, or source streams, may be added via a command item in the menubar. The output boxes, or the applications, are placed on the very right of the editor workspace.



## **4 Design and Implementation Decisions**

### **4.1 Java/Swing/Petal**

Because the GUI is meant to run as a client application communicating with the server-side Aurora system, it was decided that Java would be an appropriate language to use. Initially it was decided that the GUI would be run as an applet on the client, communicating with an application server or a servlet engine such as Jakarta Tomcat. The GUI would first be developed as an application running locally on the Aurora server, and then transformed into an applet. As a result, the GUI would be executable from different platforms. However, during the latter stages of development it was decided that Jeong-Hyon Hwang's remote object model could be a good tool to use instead of an applet. Currently, the GUI has run successfully on the Solaris Unix, Debian Linux, and Microsoft Windows XP platforms.

Another advantage of using Java for the GUI was a tool called Petal, written by Brown University professor Steve Reiss. Petal is a graphical user interface tool that allows construction of graphical nodes and arcs. It was written in Java Swing. Accordingly, it was also decided that Java Swing would be the package to use.

### **4.2 Sleepycat database**

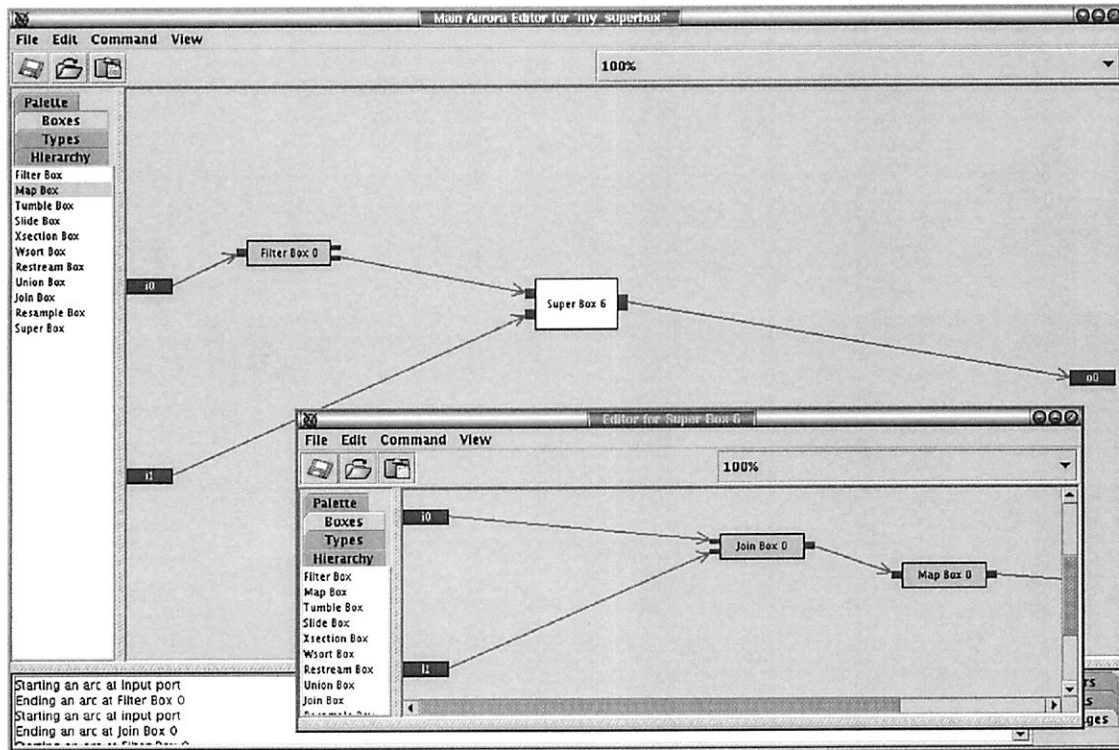
In order to store and retrieve information of the boxes, it was decided that a Sleepycat (Berkeley DB) database would be used. The light-weight, persistent storage properties of Sleepycat were necessary for the GUI to store information for the server-side Aurora to process. Other benefits of using Sleepycat were its ODBC-like network protocols as well as its ability to store variable-length records.

## **5 Features of the GUI**

Besides a simple box and arc diagramming tool, the GUI contains several features that enhance the efficacy and the process of defining a network. It is capable of recursively encapsulating a group of boxes into one superbox and displaying the result as a graph of superboxes. As a cut-and-paste feature, the GUI is also able to save pre-defined boxes into a list and copy them multiple times back into the workspace. Other features include input support for the Quality of Service (QoS) monitor, as well as a workload generator.



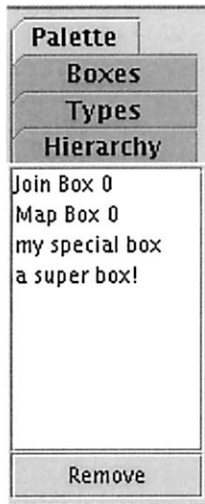
## 5.1 Superboxes



For the purposes of modular design, the GUI allows the user to select a group of boxes and combine them into a superbox using a command in the menu bar. Another method is to drag a superbox list item from the boxes tab list, open up the superbox window by double-clicking the superbox, and adding boxes or arcs within that window. As a result of either operation, the group of boxes is removed from the diagram and replaced by one superbox. When double-clicked, the superbox opens up a new window displaying only the group of boxes that were selected beforehand. As a result, a superbox can be viewed in two different ways – first, as a box node which may be connected to other boxes, and second, in the window where all of its internals are displayed. If there were arcs coming into and coming out of the group of selected boxes, the GUI would automatically generate arcs coming into and out of the superbox. These superboxes may be created recursively.

Conceptually, the root query network is also a superbox. Along with boxes and arcs, it also contains its own input and output port nodes as all other superboxes. In the future, with several box and arc query designers working on the same diagram, enhancements such as password-protected or access-privileged superboxes may be necessary. As a result, it would be possible for two network designers to work in separate, individual superboxes simultaneously. This would be useful in the situation where several query designers are simultaneously maintaining and modifying independent superboxes of the query graph.

## 5.2 Palette



As a mechanism similar to copy-and-paste, a palette was provided so that a user may predefine any box and its modifiers, and then add it to the palette. Any box may be added to the palette by right-clicking on the box and selecting the "add to palette" menu-item on the popup menu. The palette then saves all user-related information regarding the box, such as its name, description, modifiers, and ports. Hence, if the usage of that box is required numerous times, it is possible to create boxes from the palette multiple times. In addition, if the user desires to save a group of boxes, he/she may aggregate those boxes into a superbox and save that superbox into the palette. In order to remove items from the palette, the user must first select the palette item to remove, and then click on the "remove" button within the palette tab, or select the "remove current palette item" from the command menu of the menubar.

## 5.3 Hierarchy-tree view



Because of the recursive properties of superboxes, another view within the left-side toolbox pane was added which supports the hierarchical view of superboxes. This hierarchy displays the list of superboxes and their contained boxes as a JTree. From the hierarchy, it is possible to view all boxes that exist. With this view, it is possible to view, modify or delete any box located in an arbitrary superbox without having to traverse different superbox windows. The popup menu of a box may be opened by selecting a box in the JTree and right-clicking. All normal pop-up menu operations may then proceed from this point.

## 5.4 QoS support

	X values	Y values
Qos values 0	34	24
Qos values 1	-32	6.664
Qos values 2	56	0
Qos values 3	0	0
Qos values 4	0	0

In the Aurora server, a quality of service monitor has several responsibilities, such as monitoring throughput, monitoring the number of tuples waiting in a queue in each arc before being processed by a box. In the root superbox, properties dialogs may be opened up for output port nodes. Within these properties dialog boxes, it is possible to edit and adjust the quality of service values.

The quality of service menu allows the user to specify multiple QoS graphs. Each QoS graph contains text fields for specifying the number of QoS points to define and a QoS type. Once the number of QoS points is entered, the panel creates the appropriate number of QoS values for input.

## 5.5 Workload generator support

Because Aurora is currently a prototype system, there are no sources for streaming information. As a result, a stream generating program has been written, for which the GUI has built-in support. Each input port node has a properties dialog which may specify workload generation parameters. The parameters may specify the rate distribution of the incoming aggregate streams, the mean value of the distribution, the number of tuples to pass through the input port, the number of streams, and the variance.

Furthermore, the workload generator GUI is capable of communicating with the workload generator using the TCP/IP protocol. As a result, during the runtime of Aurora, the GUI is capable of updating and changing the parameters of the workload

generator. After setting the workload generator specifications, the user can select the “start workload generator” menu item from the command menu. To change workload generation parameters, the input port node specifications must be changed accordingly and then the “update workload generator” menu item must be selected from the “command” menu.

## 6 Types, type checking, and type inference

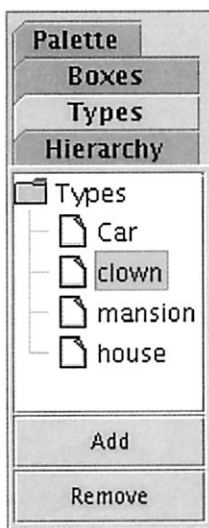
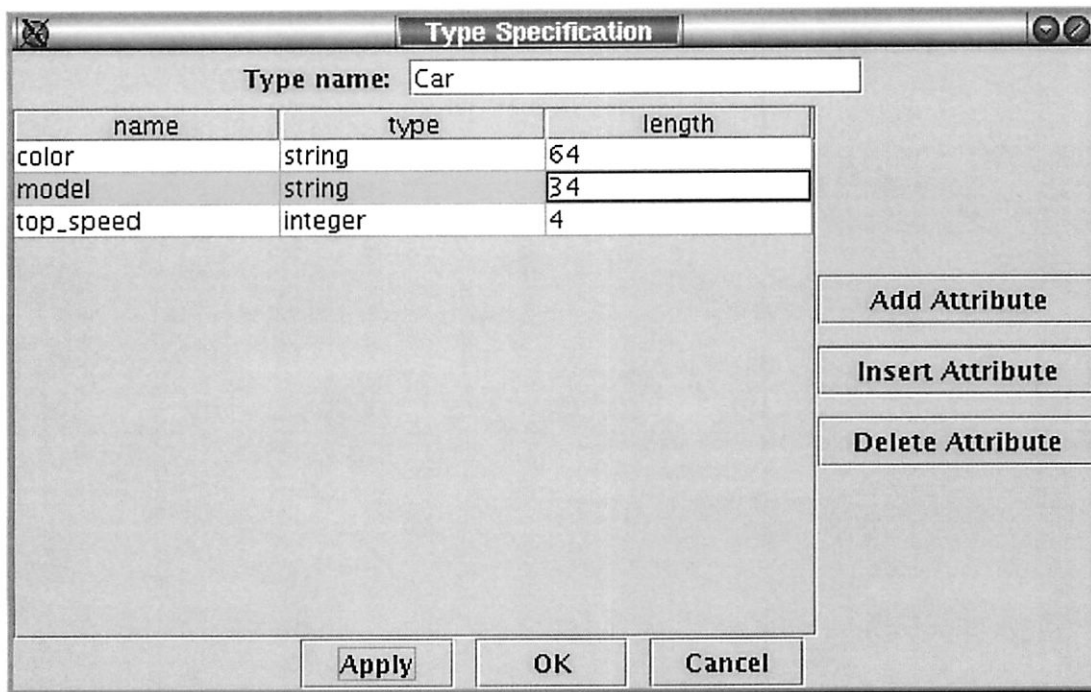
The screenshot shows a 'Properties' dialog box with a 'Source Stream Specification' tab. The 'Use workload generator' checkbox is checked. The 'Rate Distribution' dropdown is set to 'Periodic'. The 'Number Of Tuples' text box contains '1', and the 'Number Of Streams' text box contains '0'. There are buttons for 'Rate Options' and 'Variance Options' on the right side of the input fields. At the bottom are 'Apply', 'OK', and 'Cancel' buttons.

In order to represent the schema of the tuples, the notion of types was built into the system. Abstractly, there is a representation of primitive types, such as *integer*, *float*, *double*, and *string*. While most primitive types are of fixed length, some primitive types such as *string* have user-defined lengths. The types handled by Aurora consist of ordered compositions of primitive types. For simplicity's sake, Aurora does not handle nested composite types.

### 6.1 Server-side representation of types

Server-side Aurora regards tuples as byte arrays. Within the modifier of a box, the server-side expects fixed-length attributes in the tuple description. These attributes include the argument index, the offset, the size, and the typeId (or the primitive type) of the attribute field. To reference a type attribute in modifiers, all that is required is a sequence of numbers delimited by a specified character that correspond to the tuple attributes.

## 6.2 UI representation of types



The user interface representation of types is quite different from the server-side representation. Internally, types are composed of primitive types. They may be unnamed (inferred) or named (explicitly defined) types. For named types, the GUI provides an interface for type creation and maintenance. Each named type is independent of other named types – as long as the type names are unique, each type may have the same attributes as any other type. Type compatibility, however, is defined on a per-attribute basis.

Unnamed types, on the other hand, are not explicitly defined in the type manager, but created as a result of type inference of boxes. Type inference occurs when a modifier for a box is defined and the parser returns an outgoing type.

In order to define a modifier for a box, all input ports of that box must typically be associated with a type. This is necessary because a modifier often contains field names in its modifiers, and field names cannot be known without having their types defined. Furthermore, it is necessary to define type names in the input port node for incoming source streams.

One key advantage to having named types is that unconnected boxes can have defined predicates. This enhances the capability of the palette to store more specific and useful predefined boxes. As a result of having defined input types, box modifiers may also be defined, because type attribute names exist for reference in the modifier.

Inferred types, on the other hand, allow the user to connect boxes without having to separately specify outgoing types for every box. This allows the creation of networks to require fewer clicks and thus be faster and more efficient.

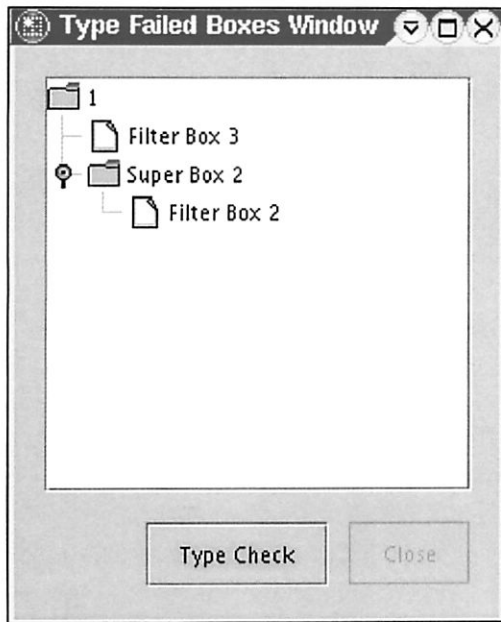
### **6.3 Properties of type inference, checking, and predicate parsing**

One of the major features of the GUI is the ability to infer and check types in the query graph. Similar to a compile-time of a program, the GUI has the capability to walk through the graph of operators and check to see if the types are consistent. Once the modifiers of all the boxes are defined and the types are consistent, the Aurora system may then properly run the query.

In order to fully define a box, there are some prerequisites. As discussed above, the modifier for a box may only be defined if all of the input ports of a box are properly typed. This may occur if the user explicitly sets the type for the port, or if the port is connected by a previous box whose outgoing type is known. Once the modifier for the box is defined, a parser then checks the validity of the modifier string. If the modifier string is properly formatted, then the parser infers the outgoing type of the box.

In order to implement type checking, a depth-first search starting at all of the independent operator graph roots through all of the boxes was implemented. This process invokes the parser on all of the boxes and produces their output types if possible. The search then proceeds to the next boxes and accordingly sets the input types of the following boxes, if they are set to infer types. This search can transparently traverse input and output port nodes of container superboxes. As it defines the output types of boxes, it sets the color of the box to grey if the output type is defined, light yellow if the output type is null. As a result, if all of the boxes in a diagram are grey, the user knows that the whole box-arc network is ready to be saved into the database to run the server-side Aurora system.





A type checking error may occur in two scenarios: (1) if the input type of a box is set to a defined type, which results in a conflict with the type it would be inferred with, or (2) if the modifier of a box is defined with a particular type in mind and the output type of the preceding box is incompatible. When a type-checking error occurs, the GUI opens up a "Type Check Failed Boxes" window (as shown on the left) which displays the boxes containing the error and the superboxes that contain them in hierarchy form. The "close" button of this window is only enabled when there are no more boxes with type check errors.

There are several subtleties involved with type-checking. First, if the user successfully defines a valid network, and then proceeds to modify or delete a type or to modify the type of a box, he may inadvertently cause the previously-valid query graph to be invalid. Second, if a box is inferring types and if the input arc has been deleted, the box may lose its input type. Third, some operators such as map may create their own types, and the modifier dialog and the parser must handle the existence of these new types.

As a result of the type-checking subtleties that occur during query-graph manipulation, it was decided that once a modifier for a box has been defined, it will remain even if the input port node types are null. However, if the input port nodes are non-null and incompatible with the modifier, the modifier will be erased and set to the empty string. Furthermore, because many connected boxes could possibly contain the empty string for their modifiers, empty-string modifier boxes do not show up in the types-failed boxes window.

Because inferred types are created anew during every single network type check, the system must keep track of equivalent, previously inferred types so as not to use an excess amount of objects. These inferred types are implicitly stored in the model but not displayed in the named-types tabbed pane. They are also stored and loaded from the Sleepycat database.

## 7 Future Work

Future work for the Aurora includes connection points and ad-hoc queries, where modifications at arcs can be made to the operator network during runtime. The Aurora user interface, as a result, must be able to allow the addition and removal of boxes at arc points, and to provide a mechanism for referring to historical data. Aurora will also be able to support user-defined functions, in order to allow the network administrator to define scripts for processing of special aggregates and other



functions.

On the GUI side, the implementation of the client/server side model must be completed. Furthermore, because non-referenced inferred types are not deleted from the Sleepycat database, the GUI will also require the garbage collection of these types.

Other future considerations include security, user-specific access to particular superboxes, and a concurrent-version control of network modification where read and read-write access is handled appropriately to prevent network merging problems.

## References

- [1] D. Carney, U. Cetintemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002
- [2] J. Hwang. Implementation of a Distributed Aurora GUI Environment. May 15, 2002