Aurora: A Data Stream Management System

D. Abadi[†], D. Carney[§], U. Cetintemel[§], M. Cherniack[†], C. Convey[§], C. Erwin[§], E. Galvez[†], M. Hatoun[§], J. Hwang[§], A. Maskey[†], A. Rasin[§], A. Singer[§], M. Stonebraker[±], N. Tatbul[§], Y. Xing[§], R. Yan[§], S. Zdonik[§]

[†] *Brandeis University*
[§] *Brown University*
[±] *M.I.T.*

Abstract

The *Aurora* system [1] is an experimental data stream management system with a fully functional prototype. It includes both a graphical development environment, and a runtime system.

We propose to demonstrate the Aurora system with its development environment and runtime system, with several example monitoring applications developed in consultation with defense, financial, and natural science communities. We will also demonstrate the effect of various system alternatives on various workloads. For example, we will show how different scheduling algorithms affect tuple latency and internal queue lengths. We will use some of our visualization tools to accomplish this.

Data Stream Management

Aurora is a *data stream management system* for *monitoring applications*. Streams are continuous data feeds from such sources as sensors, satellites and stock feeds. Monitoring applications track the data from numerous streams, filtering them for signs of abnormal activity and processing them for purposes of aggregation, reduction and correlation. The management requirements for monitoring applications differ profoundly from those satisfied by a traditional DBMS:

- A traditional DBMS assumes a passive model where most data processing results from humans issuing transactions and queries. Data stream management requires a more active approach, monitoring data feeds from unpredictable external sources (e.g., sensors) and alerting humans when abnormal activity is detected.
- A traditional DBMS manages data that is currently in its tables. Data stream management often requires processing data that is bounded by some finite window of values, and not over an unbounded past.
- A traditional DBMS provides exact answers to exact queries, and is blind to real-time deadlines. Data stream management often must respond to real-time deadlines (e.g., military applications monitoring positions of enemy platforms) and therefore must often provide reasonable approximations to queries.
- A traditional query processor optimizes all queries in the same way (typically focusing on response time). A stream data manager benefits from application specific optimization criteria (QoS).
- A traditional DBMS assumes pull-based queries to be the norm. Push-based data processing is the norm for a data stream management system.

A Brief Summary of Aurora

Aurora has been designed to deal with very large numbers of data streams. Users build queries out of a small set of *operators* (a.k.a. *boxes*). The current implementation provides a user interface for tapping into pre-existing inputs and network flows and for wiring boxes together to produces answers at the outputs. While it is certainly possible to accept input as declarative queries, we feel that for a very large number of such queries, the process of common sub-expression elimination is too difficult. An example of an Aurora network is given in Screen Shot 1.

A *simple stream* is a potentially infinite sequence of tuples that all have the same stream ID. An arc carries multiple simple streams. This is important so that simple streams can be added and deleted from the system without having to modify the basic network. A query, then, is a sub-network that ends at a single output and includes an arbitrary number of inputs. Boxes can connect to multiple downstream boxes. All such path splits carry identical tuples. Multiple streams can be merged since some box types accept more than one input (e.g., *Join*, *Union*). We do not allow any cycles in an operator network.

Each output is supplied with a Quality of Service (QoS) specification. Currently, QoS is captured by three functions (1) a latency graph, (2) a value-based graph, and (3) a loss-tolerance graph. The latency graph indicates how utility drops as an answer is delayed. The value-based graph shows which values of the output space are most important. The loss-tolerance graph is a simple way to describe how averse the application is to approximate answers.

Tuples arrive at the input and are queued for processing. A *scheduler* selects a box with waiting tuples and executes that box on one or more of the input tuples. The output tuples of a box are queued at the input of the next box in sequence. In this way, tuples make their way from the inputs to the outputs. If the system is overloaded, QoS is adversely affected. In this case, we invoke a load shedder to strategically eliminate

Aurora supports persistent storage in two different ways. First, when box queues consume more storage than available RAM, the system will spill tuples that are less likely to be needed soon to secondary storage. Second, ad hoc queries can be connected to (and disconnected from) any arc for which a *connection point* has been defined. A connection point stores a historical portion of a stream that has flowed on the arc. For example, one could define a connection point as *the last hour's worth of data that has been seen on a given arc*. Any ad hoc query that connects to a connection point has access to the full stored history as well as any additional data that flows past while the query is connected.

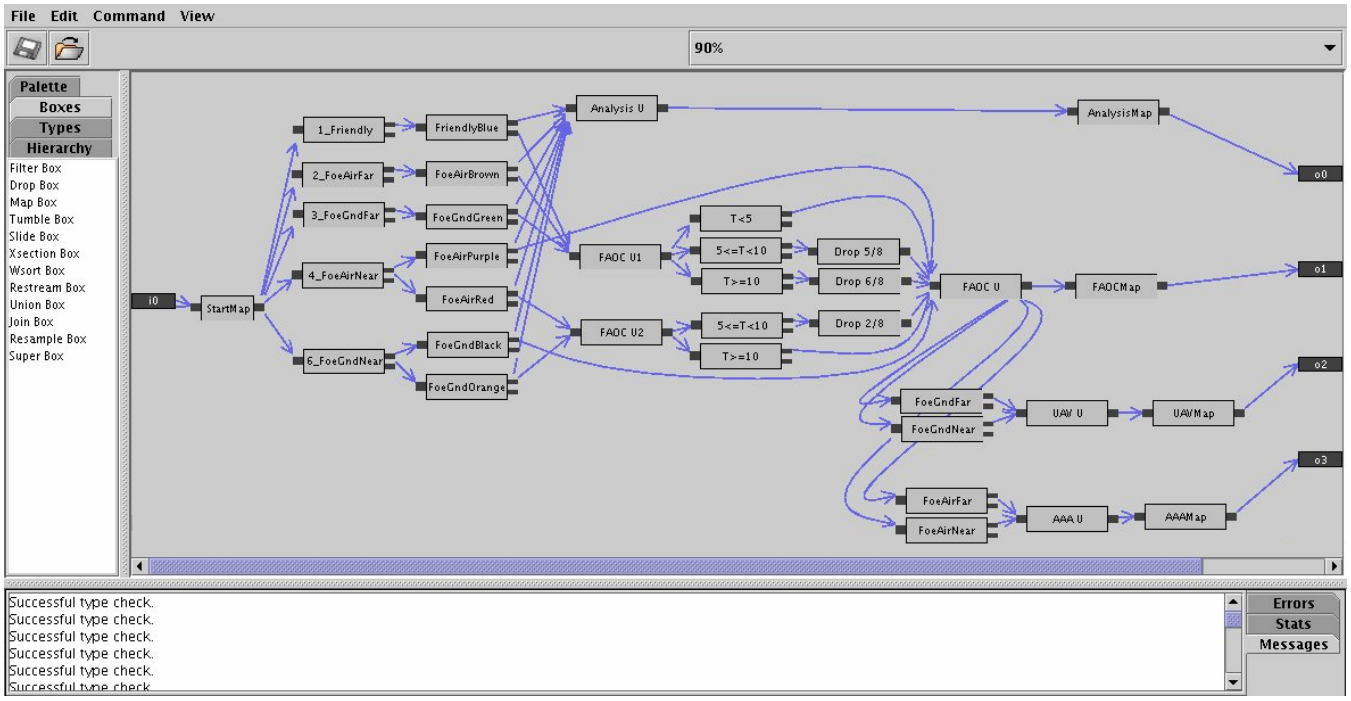

Figure 1: Aurora's development GUI

The Aurora System

The Aurora system consists of the following components:

- A Java-based GUI development environment, where tuple structures and Aurora flow networks are defined. See Figure 1.
- A server that executes an Aurora network. The inputs and outputs of the Aurora server are streams of tuples, delivered over TCP/IP sockets.
- A Java-based GUI performance monitor that shows the quality of service being provided by the server at a given moment. See Figure 3.

All applications must provide the following to Aurora:

- A TCP/IP-based interface supplying data streams to the Aurora server.
- A set of persistent queries (applications) that describes the processing that the Aurora server is to perform on the input streams. These queries are authored in the Aurora development environment.
- A set of Quality of Service (QoS) specifications (one per application) that specify application-specific performance and quality requirements of the Aurora system. For example, a QoS specification can describe the quality of a tuple returned as a query result as a function of latency, accuracy or the values it contains. QoS is specified in the Aurora development environment.
- A set of applications awaiting the query results (streams of tuples) emitted by the Aurora server. In our prototypes, applications present Aurora output in some human-friendly form. See Figure 2 for an example.

Novel Features

Stream-oriented query operators

Traditional DBMS query languages (e.g., SQL) are insufficient for processing infinite streams. Aurora introduces a novel set of operators specifically tailored towards stream processing. For example, Aurora includes a number of *window*-based operators that act on finite moving windows over input streams.

QoS specifications

QoS specifications not only serve as specifications of desired system behavior – they also serve to drive policies for scheduling and load shedding (described below). The overall goal of the Aurora system is to maximize overall quality of service from all applications.

Load shedding

Traditional RDBMS'es are designed to produce correct results regardless of the time required to produce them. Such a goal is ill-suited to many of the applications we've seen for stream processing. For many stream applications, it's better to discard a fraction of the data than to process all of it in an unacceptably long amount of time. Aurora's scheduler uses the QoS specifications for an application to decide when and how records should be dropped, which can actually *increase* overall QoS.

Real-time Scheduling

We have developed several scheduling algorithms that reduce scheduler and box invocation overheads as well as the use of disk. We do this by scheduling more than one tuple at a time (i.e., trains) through more than one box at a time (i.e., superboxes).

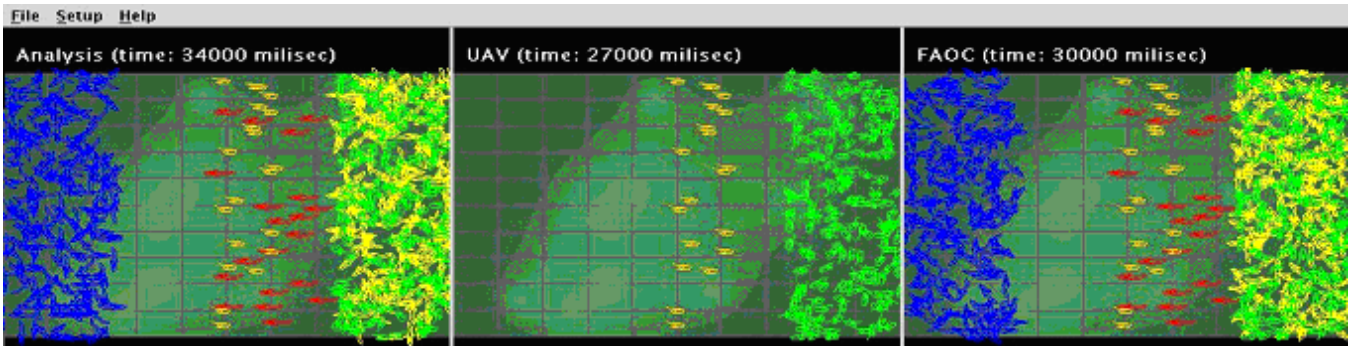

Figure 2: Mitre application output

Storage Management

Our storage manager is designed for storing ordered queues of tuples instead of sets of tuples (relations). It also combines the storage of push-based queues with pull-based access to history.

Demonstration Details

Our demonstration will include the illustration of multiple query specifications using the development environment GUI, execution of some example Aurora applications (described below), and use of Aurora performance monitoring tools to demonstrate system internals and QoS functions.

Tracking Application

We have been working with Mitre Corporation on a military Command and Control application. It involves the intelligent dissemination of enemy positions on the battlefield to various ground stations. Each ground station has a different role and thus requires different information with different freshness requirements. Each ground station specifies the relative importance and the minimum update frequency for each kind of data.

In this application, bandwidth is very limited. During periods of high stress, the data rates can swamp the communication links. Aurora's job is to selectively discard or delay less important data so that the more important data can get through in a timely fashion.

The demonstration receives simulated position data from a variety of sources and regarding a variety of objects (e.g., tanks, airplanes). The purpose of the application is threefold:

1. To allow military commanders to quickly understand the current state of the theatre. Different personnel need different subsets of the information to do their respective jobs. See Figure 2.
2. To alert personnel when a certain event has occurred. For instance, when at least five enemy soldiers have crossed a particular line on the map.
3. To show that some information is delayed in order to service high priority items. In the interface of Figure 2, some of the icons will move (update) frequently, and others will seem to jump infrequently.

The Aurora prototype plus a data visualizer handles all these needs. Figure 2 shows three different displays, where each display is intended for a different user. Note that the parts of the Aurora network that produce the data for the three different displays are given different processing priorities, leading to one display being refreshed more often than another. This is en example of Aurora's QoS system at work. I.e., it's more important for the general to know when an attack is imminent, than for the refuelling coordinator to know about every time a fuel truck moves.

Toxicity Monitoring Application

This application collects respiratory data from a set of fish, as well as attributes of the surrounding water. The fish are confined to small cages, and carbon blocks on either side of their gills can detect small currents that are generated by muscle motion. This current can be used to reproduce the motion of the gills. Erratic gill motion can indicate the presence of toxins in the water. When several fish have sufficiently abnormal readings, an alarm is sounded, indicating that the water supply may have been compromised.

We will demonstrate the use of Aurora to monitor the data for interesting patterns. Some patterns will be determined by referencing the current streams with stored reference streams. We expect that we could actually receive live feeds from some fish arrays in Texas if we can get an Internet connection at the conference.

Financial Application

This application monitors streams of stock quotes. As with the military application, the user can both view a digest of the information, as well as be alerted when interesting events occur. In particular, we have been working with Fidelity on several problems that they have ranging from intelligent routing of trades to fraud detection on transaction data.

GUI Development Environment

We'll demonstrate the development environment by modifying the military application (Figure 1). The editing may include adding new arcs/boxes, deleting existing boxes/arcs, and modifying the parameters of existing boxes. The results of the modification will be shown in the application's output GUI (Figure 2).

Performance Monitoring Tools

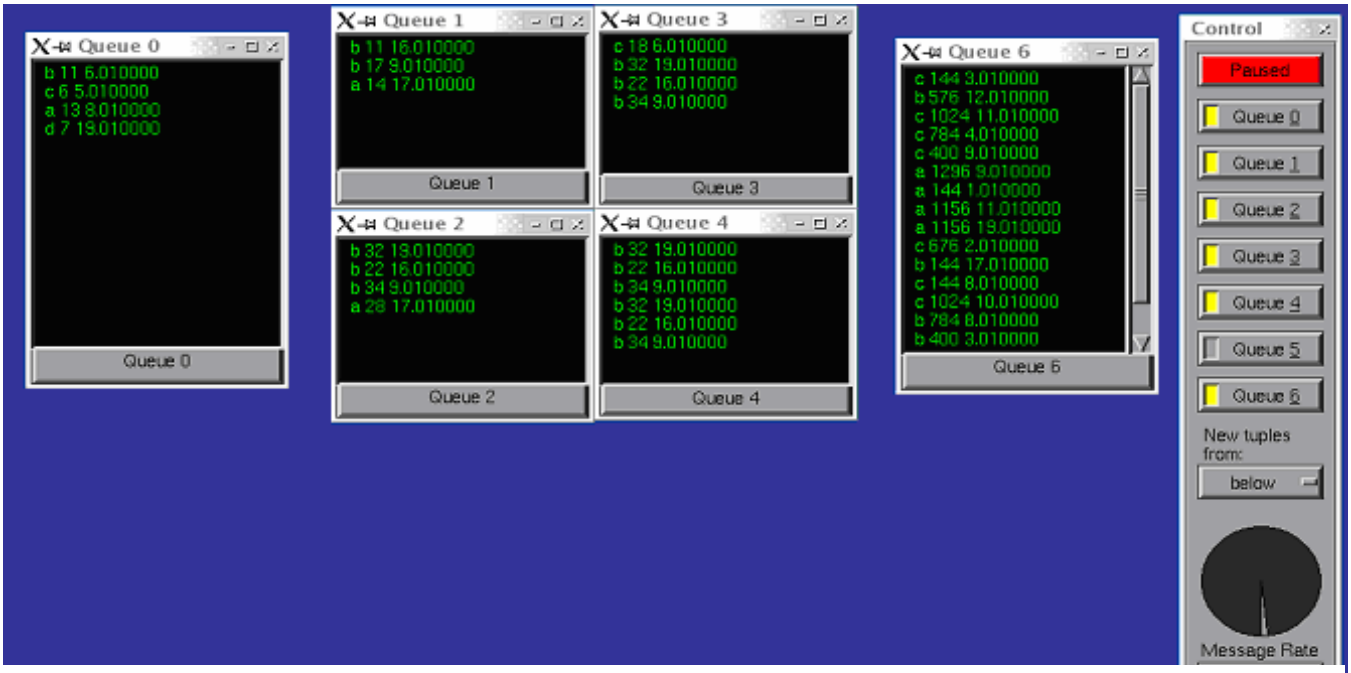

Figure 3: Aurora's Performance Monitoring Tools and Simulation Input Pacing Tool

Aurora's performance monitoring tools are useful for showing how much data is in various parts of the system and its flow rate, as well as the QoS level that Aurora is able to achieve. We'll show how varying the rate of input data flow affects the Aurora network's internal queue contents as well as the overall delivered QoS.

Impact

Aurora, whose design was presented at VLDB 2002 [1], has been the focus of positive attention from the industrial, scientific, and academic sectors. We've developed a prototype Aurora application for a defense company that easily solved a problem that the company considered difficult. We're working with partners in the financial and natural science fields to develop prototype Aurora applications for those sectors as well.

REFERENCES

[1] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul and Stan Zdonik, *Monitoring Streams – A New Class of Data Management Applications,* Proceedings of Very Large Databases (VLDB), Hong Kong, China, August, 2002.